Computational Particle Dynamic Simulations on Multicore Processors (CPDMu)

Final Report – Phase I

DOE SBIR 2009 Topic 39a

(*Simulation of Engineering Problems*)

Principal Investigator: **Mark S. Schmalz**

UltraHiNet, LLC
709 SW 80th Blvd
Gainesville FL 32607-6523

Project Summary/Abstract

Company Name: UltraHiNet LLC

Project Title: Computational Particle Dynamic Simulations on Multicore Processors

Principal Investigator: Mark S. Schmalz, Ph.D.

Topic No./Subtopic: DOE SBIR 2009 Topic 39a

Statement of Problem: Department of Energy has many legacy codes for simulation of computational particle dynamics and computational fluid dynamics applications that are designed to run on sequential processors and are not easily parallelized. Emerging high-performance computing architectures employ massively parallel multicore architectures (e.g., graphics processing units) to increase throughput. Parallelization of legacy simulation codes is a high priority, to achieve compatibility, efficiency, accuracy, and extensibility.

General Statement of Solution: A legacy simulation application designed for implementation on mainly-sequential processors has been represented as a graph *G*. Mathematical transformations, applied to *G*, produce a graph representation $\underline{G}$ for a high-performance architecture. Key computational and data movement kernels of the application were analyzed/optimized for parallel execution using the mapping $G \rightarrow \underline{G}$, which can be performed semi-automatically. This approach is widely applicable to many types of high-performance computing systems, such as graphics processing units or clusters comprised of nodes that contain one or more such units.

Phase I Accomplishments: Phase I research decomposed/profiled computational particle dynamics simulation code for rocket fuel combustion into low and high computational cost regions (respectively, mainly sequential and mainly parallel kernels), with analysis of space and time complexity. Using the research team's expertise in algorithm-to-architecture mappings, the high-cost kernels were transformed, parallelized, and implemented on Nvidia Fermi GPUs. Measured speedups (GPU with respect to single-core CPU) were approximately 20-32X for realistic model parameters, without final optimization. Error analysis showed no loss of computational accuracy.

Commercial Applications and Other Benefits. The proposed research will constitute a breakthrough in solution of problems related to efficient parallel computation of particle and fluid dynamics simulations. These problems occur throughout DOE, military and commercial sectors: the potential payoff is high. We plan to license or sell the solution to contractors for military and domestic applications such as disaster simulation (aerodynamic and hydrodynamic), Government agencies (hydrological and environmental simulations), and medical applications (e.g., in tomographic image reconstruction).

Keywords. High-performance Computing, Graphic Processing Unit, Fluid/Particle Simulation

Summary for Members of Congress. Department of Energy has many simulation codes that

Table of Contents

Section/Subsection	Page
1. Significance, Background, and Results of Phase I Research	**4**
1.1. Significance, Background, and Overview	**4**
1.1.1. Significance of Problem	4
1.1.2. Scope and Relevance of Phase I Work	4
1.1.3. Discussion of Phase I Work	6
1.2. Benefits of UHN's Phase I Research	**15**
1.2.1. Parallel Implementation of Legacy Simulation Codes	16
1.2.2. Ready Optimization of Parallel Implementations	16
1.2.3. Explore Techniques of Legacy Code Parallelization for New Architectures	16
1.3. Degree to Which Phase I Has Demonstrated Technical Feasibility	**17**
1.3.1. Conformance to Proposed Phase I Technical Objectives	17
1.3.2. Phase I Test Results	18
2. Facilities and Equipment	**23**
3. Consultants	**23**
4. Bibliography and References Cited	**23**

1. Significance, Background and Results of Phase I Research

1.1 Significance, Background, and Overview

UltraHiNet, LLC (UHN) is pleased to submit this Final Report pursuant to Department of Energy (DOE) SBIR 2009 Topic 39a, entitled *Computation of Engineering Problems*. In Phase I of this SBIR effort, we profiled, analyzed, parallelized and tested key computational kernels of grid-based algorithms such as computational particle dynamics (CPD) codes for simulation on multicore graphics processing unit (GPU) processors and clusters of multicore GPUs.

The remainder of this subsection contains an overview of the significance of the problem (Section 1.1.1). We overview the approach that we have successfully developed in Phase I (Section 1.1.2), then discuss details of our Phase I approach and Phase I results (Section 1.1.3).

1.1.1. Significance of Problem. Computational Particle Dynamics, together with the related area of Computational Fluid Dynamics (CFD), is an important foundational technology for a wide variety of dynamic particle- and field-based simulations of interest to DOE, DOD, and NASA, to name but a few of our prior or current research sponsors. Example applications include but are not limited to DOE's interests in simulation supporting combustion design and engineering, weapons design and damage reduction, DOD and NASA interests in aircraft/spacecraft design, aerodynamic simulation, and medical imaging (e.g., tomography), and numerous other applications.

In practice, the mapping of DOE's CPD and CFD simulation applications to parallel architectures, such as GPUs or clusters of CPU-GPU nodes, is not straightforward in the sense that mapping a pointwise image operation to a synchronously parallel SIMD mesh is realized. In practice, many realistic CPD/CFD simulation problems have irregular grids whose size and resolution can change locally or globally [Liao96]. Further, in adaptive simulation applications, the grid, population and density of particles, and/or field boundaries can change with time [Rav10]. In realistic simulation problems, the external simulation constraints and/or output resolution can also change. Together with the problems of real-time interpolation between cells, these types of simulations (e.g., regular or irregular grid, mesh, or particle-in-cell) challenge current understanding of parallelization using existing algorithm-to-architecture mapping theory.

In response to this problem, our research team has developed in Phase I an approach that successfully maps (for example) key computational kernels of CFD/CPD simulation applications to parallel architectures such as GPUs, or to clusters comprised of nodes having a CPU that hosts multiple GPUs. In terms of applicability to DOE's interest in energy generation and control stated in the solicitation, these simulation problems are found extensively throughout the energy sector – in design of combustion or fusion based power plants, turbine design, performance modeling, etc. In each of these cases, the use of fixed or variable model parameters or configurations will be addressed in terms of our parallelization approach that was proven successful in Phase I.

1.1.2. Scope and Relevance of Phase I Work. In Phase I, the UHN research team analyzed, parallelized, and tested computational kernels of CPD codes provided by our consultant Dr. S. Balachandar, and parallelized these kernels successfully. Our research, development, test, and analysis is based on our research team's extensive proven expertise in mathematical analysis and implementation of problems and solutions for *parallel computation of grid- or mesh-based processing algorithms*, *parallel vector and array processing algorithms*, including but not limited to *computational particle dynamics*, *computational fluid dynamics*, *image and signal processing*, and *high-performance simulation*.

Relevance of Phase I Parallel Implementation for CPD/CFD Simulation. Our Phase I research pertains directly to DOE's energy-related goals of designing efficient combustion and power generation facilities. In combustion simulation, highly efficient, accurate parallel processing is required for computing CPD simulation codes and related CFD simulation codes, with physical fidelity and user-friendly interactivity. Existing CPD/CFD simulation codes have been primarily written for Central Processing Units (CPUs), with calculations performed in serial or parallel fashion on single or multiple CPUs. When these legacy CFD/CPD simulation codes are ported to a parallel architecture by merely inputting them (for example) to an optimizing compiler, the results often exhibit disappointingly poor performance in comparison with the target architecture's advertised peak performance. Thus, manual optimization (local as well as global) is required, often requiring much effort and public expense.

Fortunately, video gaming chip manufacturers have developed computational hardware and software for life-like visualization, as well as high-bandwidth parallel arithmetic computation. Such Graphics Processing Units (GPUs) can achieve peak computational throughput of gigaflops to teraflops, with sustained processing rates currently in the tens to hundreds of gigaflops. GPUs can also be clustered into massively parallel supercomputers with hundreds of energy-efficient nodes comprised of a CPU controlling multiple GPUs. Although impressive speedups have been realized by running CPD codes on GPUs versus a single CPU, significant challenges remain for multicore GPU implementation. Prior to and during our Phase I research, we determined that such challenges include but are not limited to efficient data movement, mapping of unstructured data to mesh grids, effective mapping of grid structures to cores in the GPU, resolution of data and control dependencies in local mesh processing or iterative mesh refinement, multiprocess collaboration and sharing of data, as well as uncertainty quantification.

Parallel architectures such as GPUs have only recently been available to the public. Despite the concerted development of parallel computing over the past four decades, the programming of parallel architectures has remained challenging due to the lack of a unifying model for parallel algorithm design and implementation. In contrast, sequential (von Neumann) architectures have for over four decades exploited a unifying model that facilitates porting of legacy sequential codes to other sequential architectures. Unfortunately, the porting of legacy codes to a parallel architecture is fraught with difficulty since there exists no machine-independent programming interface (e.g., middleware) for parallel machines to achieve efficiencies comparable to hand-coding in languages supported by a target (parallel) architecture. Even worse, when a legacy application is manually converted (often at great expense) to run on a parallel architecture, it may not run efficiently (i.e., the investment may be lost) when ported to the next generation parallel architecture with a different programming model or interface.

Overview of Parallel Problems. In order to provide a conceptual and theoretical basis for mapping legacy CPD/CFD simulation algorithms to parallel architectures, in Phase I we first viewed these applications as inherently parallel problems. This perspective allowed us to characterize parallel simulation problems in terms of the following five general categories [Fox88], each of which covers a broad range of applications:

- *Synchronous* problems are data parallel, with the restriction that the time dependence of each datum is computed by the same operations. Algorithmically, as well as in natural SIMD implementations, synchronous problems are synchronized microscopically at each processor

clock cycle. Examples of such problems include pointwise arithmetic on two images, matrix multiplication, and other algorithms that are often popular in academia [Chou92].

- *Loosely Synchronous* problems (LSPs) are also data parallel, but constraints are relaxed slightly so that different data points can be evolved with different algorithms. Points are often linked irregularly, in a data-dependent manner – such problems are often termed "irregular". For example, such problems can represent macroscopic physical processes that evolve due to interactions between irregular homogeneous objects in a synchronized manner. Thus, loosely synchronous problems are spatially regular but temporally irregular – CPD and CFD simulations tend to be good examples of this problem type.
- *Asynchronous* problems are irregular in space and time, and thus are not necessarily supportive of general methods for parallelization. Some run well with imposed time synchronization, others run well with functional decomposition, and some have never run well on massively parallel machines [Fox91a].
- *Embarassingly Parallel* problems are totally disconnected in space and time, and do not need synchronization except in the final stage where results are collected. These problems can run on either SIMD or MIMD hardware, but are not within the focus of this proposal.
- *Loosely Synchronous Complex* problems are an asynchronous collection of loosely synchronous problems, for example, command and control applications. Each constituent task is synchronous or loosely synchronous, and can be parallelized with an asynchronous expert system coordinating interactions between tasks [Chou92].

In terms of practical implementation, we have found that the architecture and organization of commercially-available GPUs (e.g., Nvidia Tesla and Fermi processors) somewhat supports the parallelization of loosely synchronous problems (LSPs) such as solver, interpolation, and migration kernels of CPD and CFD simulations. Further, Nvidia's CUDA software tools can be utilized to enhance the efficiency of parallelization for carefully designed implementations of LSPs, provided that the CUDA tools are properly configured. We have extensively researched these design and configuration problems in our Phase I effort.

Additionally, over the past several decades, we have developed heuristic and algorithmic techniques for mapping applications within the preceding taxonomy of parallel problems to various parallel architectures [Heyw92a,b;Liao96;OR97;Rank90a-e;Rank91a-b]. In particular, a follow-on effort should concentrate on the GPU cluster implementation of several subclasses of loosely synchronous problems (LSPs). These LSPs could comprise the parallel sections of the CPD/CFD applications that would thus be implement on target architectures with a high degree of parallelism (e.g., multicore CPU, GPU, and clusters of CPU-GPU nodes). In particular, the sequential portions of the applications would be implemented on CPUs, which would support efficient implementation of the entire CPD or CFD simulation application. In a follow-on effort, we would also like to investigate load sharing between the CPU and GPUs.

Accordingly, we next overview how these prior (e.g., Phase I) developments relate to target multicore CPUs, GPUs, and CPU-GPU clusters.

1.1.3. Discussion of Phase I Work. In prior research, and in particular our Phase I research and development effort, we have found that loosely synchronous problems (LSPs) can be divided into a sequence of concurrent computational phases that are appropriate for parallel (e.g., GPU) based implementation. This development is important to DOE's CPD applications, as an understanding of LSP problem subtypes directly supports an understanding and correct

implementation of the corresponding algorithm-to-architecture mappings that can support parallelization of LSP problems such as CPD/CFD simulations.

1.1.3.1. LSP Problem Subtypes. Differences between subclasses of LSPs manifest primarily in how the phases are separated and when intra-phase computational and communication patterns are set. To illustrate this concept, we next present an overview of LSP subtypes, with brief examples.

LSP Subclass 1: Static Single-Phase Computations. A static single phase computation consists of a single concurrent computational phase, which may be executed repeatedly without change. Examples include iterative solvers using sparse matrix-vector operations [Saad86] and explicit unstructured mesh-based fluid dynamic calculations [Whit90]. The key challenge for efficient implementation is partitioning the data and computation to minimize data movement latencies while balancing load. This partitioning then dictates the program's synchronization and communication requirements, which must also be computed by the proposed *CPDMu* middleware because the computational pattern is set at runtime (as a result, this cannot be done by the Nvidia CUDA compiler). Reducing the overhead of these middleware calls, by reducing information re-use and call frequency as well as latency, is vital for efficient implementation.

For example, in some CPD/CFD applications there is a straightforward relationship between the way distributed arrays are partitioned and the way work is partitioned. Figure 1 depicts a sparse matrix-vector multiplication operation. The integer array *col* represents the sparsity structure of the matrix. Loop *S1* sweeps over the sparse matrix rows, while loop *S2* sweeps over its columns and calculates the inner product. If the sparse matrix-vector multiplication in Figure 1 is computed repeatedly, then it is reasonable to partition x and y between processors in a conforming manner. In such a problem, we follow the convention of carrying out computational work associated with computing a value for distributed array element $y(i)$ on the processor onto which $y(t)$ is mapped [Das91a].

```
S1    for i = 1 to m do
S2    { for j = 1 to n do
          y(i) = y(i) + a(i,j) · x(col(i,j)) }
```

Figure 1. Sparse matrix-vector multiplication operation, for example, in an iterative mesh solver for CPD or CFD problems.

As another example of single static-phase computations, we have found that there are common cases where assignment of distributed array elements (and/or work) to processors (e.g., threads or cores in a GPU) cannot be coupled as straightforwardly as in the preceding example of matrix-vector multiplication. For instance, Figure 2 depicts a loop that sweeps over the edges of a mesh – here, we indirectly index array x on the right hand side of *S3* while also indirectly indexing array y on the left hand side of *S4* and *S5*. Here, it can be advantageous to assign each loop iteration to a single processor, thus avoiding recalculation or communication of values for the variable *flux*, since $y(n1)$ and $y(n2)$ appear on the left hand sides of statements. As a result, we have found it important to separately develop partitioning methods for distributed array elements and loop iterations.

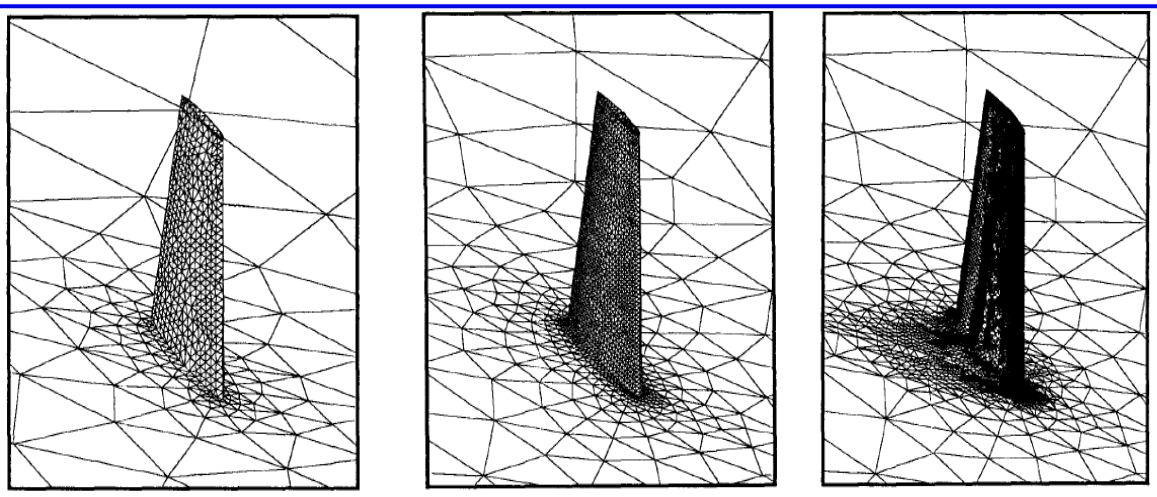
```
      for i = 1,N do
S1    { nl = nde(i, 1)
S2      n2 = nde(i, 2)
S3      flux = f(x(nl), x(n2))
S4      y(nl) = y(nl) + flux
        ssy(~2) = y(~2) – flux }
```

Figure 2. Computation of flux, where two loops sweep over the rows and columns of a sparse matrix. Flux across a mesh edge is calculated, which involves flow variables stored in array x, and accumulation of flux in array y.

LSP Subclass 2: Static Multi-Phase Computations. A static multi-phase computation consists

of a series of dissimilar loosely synchronous components. These applications usually have several parallelizable loops involving multiple distributed arrays. For simplicity, we herein consider only the case where each individual phase is a static single phase computation as defined previously. Importantly for a practical implementation of variable or adaptive grid CPD and CFD simulations, examples of static multi-phase computations include unstructured multigrids [Duff86], parallelized sparse triangular solver [Edels], particle-in-cell codes [Fox91b,c], and vortex blob calculations [Fox90].

As with static single-phase computations, the key implementational problem is partitioning computations and data, but this partitioning is more complicated because interfaces between phases must be considered. Synchronization and communication requirements are similarly complicated by the presence of multiple phases. As before, this partitioning must be performed at runtime, for example, by *incremental* runtime routines [Koel90] that would take advantage of information computed at run time.



(a) (b) (c)

Figure 3. Unstructured multigrid: (a) coarse grid, (b) refined grid, (c) adaptive grid.

1: Sweep over coarse mesh
for *i* = *1,Ncoarse* do
{ for *j* = 1, *Kcoarse* do
yc(*i*) = *yc*(*i*) + *ac*(*i,j*) * *xc*(*ic*(*i,j*)) }

2: Transfer data from coarse mesh to fine mesh
for *i* = *1,Nfine* do
{ for *j* = 1, *Ninterpf*(*i*) do
xf(*i*) = *xf*(*i*) + *weightf*(*i,j*) * *yc(interpf*(*i,j*)) }

3: Sweep over fine mesh
for *i* = 1, *Nfine* do
{ for *j* = 1, *Kfine* do
yf(*i*) = *yf*(*i*) + *af*(*i,j*) * *xf*(*if*(*i,j*)) }

4: Transfer data from fine to coarse mesh
for *i* = *1,Ncoarse* do
{ for *j* = 1, *Ninterpc*(*i*) do
xc(*i*) = *xc*(*i*) + *weightc*(*i,j*) *
yf(interpc(*i,j*))}

Figure 4. Example oversimplified multiple mesh computation, where *xc,yc* represent coarse mesh variables, and *xf,yf* fine mesh variables. Typically, such computations are performed iteratively.

To better understand this technique, we describe an unstructured multigrid application that illustrates some implementational complexities of this subclass. Unstructured multigrid codes [Duff86] perform mesh relaxation over individual, increasingly refined meshes M_1, …, M_n. Figure 3a,b depicts two refinement levels from a fluid dynamics code we have parallelized [Chou92]. Both grids represent the same physical geometry; Figure 3b is more refined.

This algorithm alternates between sweeping over each mesh and moving data between meshes, per the code in Figure 4. The meshes M_1, …, M_n should be partitioned so that (1) each sweep over each mesh M_i minimizes interprocessor communication, (2) the computation for sweeping over each mesh should exhibit good load balance, and (3) interpolations and projections should require modest amounts of data movement. Note that the grids in Figure 3 were partitioned per [Fox90b] with good results, but many other partitioning methods could be used.

LSP Subclass 3: Adaptive Irregular Computations. Adaptive irregular computations feature a loosely synchronous computation executed repeatedly, where the data access pattern changes between iterations. Such changes may be gradual, reflecting adiabatic changes in the physical domain; or abrupt, reflecting additions to a data structure. With respect to possible future development, we note that particle dynamics applications often exhibit gradual changes because interactions between particles are implemented by neighbor lists, which change as particles move [Fox91a]. Adaptive PDE solvers often exemplify abrupt changes, as discussed below. The key problem in implementing these algorithms is quick reaction to data structural changes, although physical and numerical properties of these algorithms typically guarantee that large-scale data restructuring is only needed infrequently.

```
for k_c = 1 to K do
{ sweep over U_c
  flag region of U_c that should be refined.
  if flagged region is not empty, then
  {  modify shape of U_r
     interpolate boundary values for U_r from U_c
     for k_r = 1 to K_r do
     {  sweep over U_c }
   inject values of U_r into U_c } }
```

Figure 5. Adaptive version of two-mesh algorithm, where coarse mesh U_c covers entire domain, refined mesh U_r covers the "active" domain partition, and the location, shape and size of the refined mesh are subject to change.

Adaptive irregular algorithms are useful (for example) in solving Euler and Navier-Stokes problems that arise in computational fluid dynamics. In these algorithms, mesh refinement is realized in partitions of a computational domain where additional resolution may be required [Liew89,Lu91]. The grid in Figure 3c exemplifies an adaptive refinement of the grid in Figure 3b, where the initial mesh-point distribution is determined from the geometry of the object around which flow is simulated. Adaptive mesh refinement is achieved by adding new points in regions of large flow gradients – a simple example of this algorithm is given in Figure 5, where remapping is performed before the inner loop is executed.

LSP Subclass 4: Implicit Multiphase Loosely Synchronous Computations. An implicit multiphase computation contains irregular dependencies between iterations. Thus far, our discussion of irregular problems has focused on a sequence of clearly demarcated computational phases. In contrast, a number of grid- and mesh-based problems have inter-iteration dependencies that might initially seem to inhibit parallelization. Such data dependency patterns are known only at run time, but can be fully predicted before a program enters the loop or loops where irregularities occur. Figure 6 exemplifies the back substitution phase of sparse matrix factorization, which is similar to solving sparse triangular systems of linear equations arising from ILU preconditioning methods [Mav91a,b]. Another example of this class is the tree generation phase of adaptive fast multipole algorithms for particle dynamics simulation [Mir88,Das91b] .

A key problem in implementing these algorithms is detection and exploitation of opportunities for partial parallelization. For example, in the code shown in Figure 7, one can realize many different simultaneous row substitutions. Although the sparsity structure determines which row

substitutions can be performed concurrently, this information is available at runtime only. In a possible future research effort for such problems, we could carry out a form of runtime preprocessing with the goal of defining a sequence of loosely synchronous computational phases. We know that this is feasible, since for bus-based shared-memory multiprocessors, we have demonstrated that it is possible to integrate runtime parallelization with compilers [Baden91]. We anticipate that it would be possible to link scalable run time parallelization with compilers targeted at multicore GPUs (e.g., CUDA-C), and in Phase I have analyzed preliminary techniques.

A more difficult problem is runtime aggregation of work and data. When performing computations such as sparse triangular solves or sparse direct factorizations [Baden91] our run time preprocessing techniques can determine the *number and content* of concurrent computational phases that comprise a computation. We call this process *runtime aggregation or runtime tiling.* A variety of numerical algorithms can support runtime tiling for multiprocessor and vector computers [Saad86,Salm90], and could be adapted to support runtime tiling on multicore CPUs and GPUs.

```
for i = l, N do
{  y(i) = rhs(i)
   for j = ija(i), ija(i+1) – 1 do
   { y(i) = y(i) - a(j) * y(col(j))  } }
```

Figure 6. Implicit multiphase problem: sparse triangular solve on the unit diagonal.

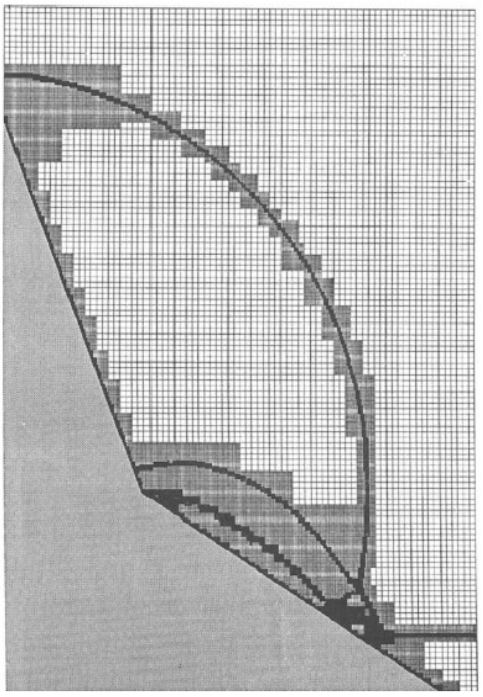


Figure 7. Mesh for calculating interaction of planar shock wave with a double wedge.

LSP Subclass 5: Static and Dynamic Structured Computations. These problems comprise highly structured computations on collections of subdomains that are coupled irregularly. The computations on each subdomain are often highly structured, but the computational relationship between subdomains is known at run time only. Further, the relationship between subdomains often changes dynamically during a computation. In the previous four LSP subclasses, we dealt with irregularly coupled "points", whereas static and dynamic structured problems feature collections of nontrivial structures. Examples of such problems include the adaptive mesh method described below and a combined fluid dynamics and particle simulation implemented by Edelsohn [Edels]. The key to realizing efficiency on these problems is to aggressively apply optimizations to the regular subproblems, which can thus be implemented with reduced overhead. Also, the larger granularity of coupled subproblems can be exploited to reduce preprocessing overhead and memory requirements [Simon91]

An example of this subclass of LSPs is shock profiling [Simon91], where one solves a partial differential equation in the presence of a shock, computing the profile (detailed shape) of the shock. Resolution constraints imply that a highly refined grid must represent the neighborhood of the shock. Initially, one computes the solution on a coarse mesh, then applies an error estimator to determine regions that will be covered by a refined mesh. An example mesh from this two-level refinement is shown in Figure 7, where the solution is time-dependent.

Time-marching on the refined mesh is performed by taking many (e.g. 100) time steps on the *refined* mesh for a single *coarse-grid* time step. The refined mesh is dynamic in its location, shape, and size, so the relationship between coarse and refined meshes will change during simulation execution. As a result, the structures of the computations and data movement operations change with time, giving rise to a nonuniform communication pattern due to data sharing between grids. This example also generalizes to a full structured adaptive multigrid.

Summary of Our Prior and Phase I Work in Loosely Synchronous Problems. We have found that time-dependent or iterative loosely synchronous computations in CPD and CFD simulations (fixed or variable grid, fixed or variable particle population, etc.) can exhibit a wide range of dynamic behaviors that can be represented by the following three categories:

(A) data dependency pattern is static and does not change between iterations;

(B) data dependency pattern is modified on occasions but between changes, the dependency pattern remains static for many iterations;

(C) data dependency pattern changes every iteration.

Problems in Category A would occur either in the class of static, single phase loosely synchronous computations (Subclass LSP-1) or as static, multiple phase loosely synchronous computations (LSP-2). In contrast, problems in Categories B and C would be classified as unstructured adaptive problems (LSP-3 and LSP-4) or structured adaptive problems (LSP-5). It is also useful to categorize irregular problems encountered in CPD and CFD simulation, by whether or not a given *iteration or time step* is composed of multiple, dissimilar loosely synchronous computational phases. In such cases, one often must partition a problem to account for all of the computational phases in an iteration. There are also some issues related to partitioning and run time aggregation [Saad86,Salm90,Walk90] that can influence implementational performance.

For example, Dr. Ranka has previously made extensions to Fortran D, in order to facilitate specification of partitioning strategies and irregular meshes. In Phase I research, we determined that enhancements of these techniques could be used to perform the following middleware tasks in a more advanced prototype:

(1) indicate which loops in a program are to be considered when partitioning distributed arrays;

(2) allow users to force the selection of a particular partitioning strategy;

(3) allow users to assert that a given set of loop dependencies can or cannot change when the loop is iteratively invoked; and

(4) allow users to specify granularity with which CPU and GPU parallelism is to be exploited.

In [Koel90], we proposed extensions (and developed run time support) that fulfill the first two of the above mentioned goals. There is also a need for development of new data structures targeted towards problems in which highly structured computations on a set of subdomains are coupled in an irregular manner. In particular, we are interested in representing structured adaptive problems in which subdomains are coupled by irregular tree dependency structures.

In [Agra93], portable runtime support for static single and multiphase problems, and for static structured problems, is described that is oriented towards distributed memory MIMD architectures. The runtime support for static single and multiphase problems has also been ported to SIMD architectures. Our Phase I work has indicated the utility of adapting these proven innovations to develop appropriate runtime support for Adaptive Irregular Computations, Implicit Multiphase Loosely Synchronous Computations, and Dynamic Structured Problems

targeted towards multicore GPU architectures and clusters of nodes, where each node is comprised of a CPU that controls multiple GPUs. Salient hardware technology is described as follows.

1.1.3.2. Multicore Graphics Processing Units. Multicore GPUs were developed to meet the expanding computational needs of computer graphics rendering algorithms. The broad, rapid growth in sophistication of graphics applications such as computer games has resulted in the availability of GPUs that have hundreds of processors and peak performance approximating a teraflop and that sell for hundreds of dollars to a few thousand dollars. Although GPUs are optimized for graphics calculations, their low cost per gigaflop has motivated significant research into their efficient use for non-graphics applications. This effort promises great potential longevity, as the widespread use of GPUs for computer gaming ensures a large, long-lasting installed base. Additionally, unlike traditional (expensive) supercomputers having a very high development cost borne by a relatively small user group (e.g., national laboratories and defense establishments), GPUs are backed by a large, growing domestic gaming industry. This makes it more likely that GPU architectures will remain affordable (lower cost per unit due to large market), technologically viable (ever-expanding need for graphics speed and realism) and will continue to evolve (applications drive technology, and vice versa).

Although GPUs have low cost per peak gigaflop, obtaining near-peak performance requires careful programming of application codes. In practice, GPU programming is complicated by (a) lack of a unifying model for parallel programming – especially for GPU architectures, which continue to evolve; (b) a sometimes-problematic memory hierarchy, e.g., device memory, shared memory, constant cache, texture cache, and registers, each having different latency, and (c) partitioning of the available scalar processors or cores into groups called *streaming multiprocessors* (as shown in Figure 8a).

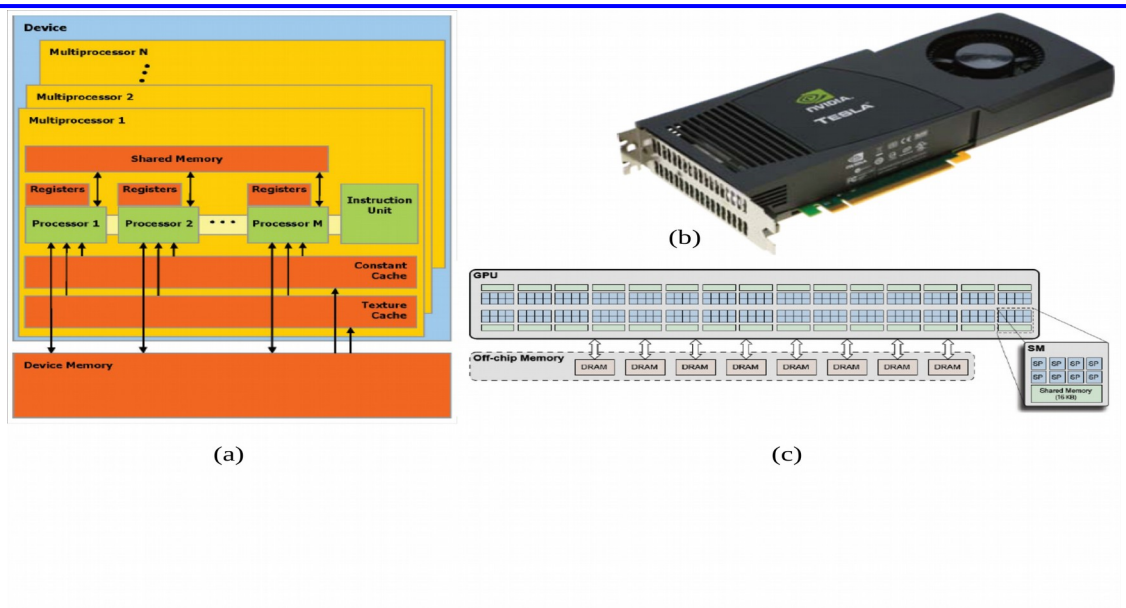


Figure 8. Nvidia GPU: (a) CUDA architecture, (b) Tesla C1050 GPU architecture, and (c) Tesla PCIex16 card (www.nvidia.com).

In our Phase I research, we surveyed different types of GPUs. Our requirements development and survey of available computing hardware led us to focus on Nvidia's Tesla series of GPUs, in

particular, the C1060 (a.k.a. Tesla, as shown in Figure 8b-c) and T2050 (Fermi), which conform to Nvidia's CUDA architecture specification. In the Phase I effort, our programs were developed with partial assistance of CUDA-FORTRAN, CUDA-C, and CUDA-C++ compiler support. Further performance gains resulted from hand optimization of the legacy codes.

GPU Hardware. NVIDIA's Tesla C1060 GPU, Figure 8, is an example of NVIDIA's general purpose parallel computing architecture CUDA (Compute Unified Driver Architecture) [Sati09]. Figure 8c is a simplified version of Figure 8a with $N = 30$ and $M = 8$. The C1060 comprises 30 streaming multiprocessors (SMs) and each SM comprises 8 scalar processors (SPs), 16KB of on-chip shared memory, and 16,384 32-bit registers. Each SP has its own integer and single-precision floating point units. Each SM has 1 double-precision floating-point unit and 2 single-precision transcendental function (special function, SF) units that are shared by the 8 SPs in the SM. The 240 SPs of a Tesla C1060 share 4GB of on-chip memory referred to as device or global memory [CUDA10]. A C1060 has a peak performance of 933 GFlops of single-precision floating-point operations and 78 GFlops of double-precision operations. The peak of 933GFlops is for the case when Multiply-Add (MADD) instructions are dual-issued with special function (SF) instructions. In the absence of SF instructions, the peak is 622GFlops (MADDs only) [Tesla10]. The C1060 consumes 188W of power.

The architecture of the NVIDIA Tesla C2050 (Fermi) corresponds to Figure 8 with $N = 14$ SMs and $M = 32$ SPs per SM. So, a C2050 has a total of 448 SPs or cores. Although each SP of a C2050 has its own integer, single- and double-precision units, the 32 SPs of an SM share 4 single-precision transcendental function units. An SM has 64KB of on-chip memory that can be configured as 48KB of shared memory with 16KB of L1 cache (default setting) or as 16KB of shared memory with 48KB of L1 cache [CUDA10]. Additionally, there are 32K 32-bit registers per SM and 3GB of on-chip device/global memory that is shared by all 14 SMs. The peak performance of a C2050 is 1,288 GFlops (1.288 TFlops) of single-precision operations and 515 GFlops of double-precision operations, which requires that MADDs and SF instructions be dual-issued. When there are MADDs alone, the peak single-precision rate is 1.03GFlops. In Nvidia-speak, the C1060 has compute capability 1.3 while the compute capability of the C2050 is 2.0.

As the C2050's power consumption is 238W, the ratio of power consumption to peak single-precision throughput is 0.2W/GFlop for the C1060 and 0.18W/GFlop for the C2050. For double-precision operations we have 2.4W/GFlops for the C1060 and 0.46W/GFlop for the C2050. A Tesla GPU is packaged as a double-wide PCIe card (Figure 8c); with an appropriately large motherboard and power supply, up to 4 GPU cards can be installed on one motherboard.

In Phase I, we focused on optimizing selected kernels of the *CONSIF* (COmputational Navier-Stokes equation Solver with Immersed boundary technique – Florida) CPD application provided by our consultant Dr. S. Balachandar for a single Nvidia Fermi GPU. We next describe the supporting CUDA programming model.

GPU Programming Model. At a high level of abstraction, a GPU uses the master-slave programming model [Won89] in which the GPU operates as a slave under the control of a master or host processor. In our Phase I experimental set up, the master or host is a 3.33 GHz six-core Intel Core i7-980X Extreme Processor, and the GPU is an 1.15 GHz, 448-core Nvidia Fermi C2050. Programming in the master-slave model requires a master program to be written that runs on the master processor (Xeon). The master (1) sends data to the slave(s) (in Phase I, a single C2050 GPU), (2) invokes a kernel or function that runs on the slave(s) which processes the returned data, and (3) receives results from the slave (GPU). This process of sending data to

the slave, executing a slave kernel, and receiving the computed results may be repeated several times by the master program. In CUDA, the host/master and GPU/slave codes may be written in C. CUDA provides extensions to C to allow for data transfer to/from device memory and for kernel/slave code to access registers, shared memory, and device memory.

At another level, GPUs use the single instruction multiple thread (SIMT) programming model whereby a GPU accomplishes a computational task using thousands of lightweight threads. The threads are grouped into *blocks*, which are organized as a *grid*. While a block of threads may be 1-, 2-, or 3-dimensional, the grid of blocks may only be 1- or 2-dimensional. Kernel invocation requires specification of the block and grid dimensions with any kernel parameters, as illustrated below for a matrix multiply kernel *MatrixMultiply* with parameters a, b, and c that point to the start of the row-major representation of *n* x *n* matrices and the

kernel computes $\mathbf{c} = \mathbf{a} \cdot \mathbf{b}$:

```
MatrixMultiply <<< GridDimensions, BlockDimensions >>> (a,b,c,n)
```

A GPU has a *block scheduler* that dynamically assigns thread blocks to SMs. Since all threads of a thread block are assigned to one SM, these threads may communicate with one another via the shared memory of an SM. Further, the resources needed by a block of threads (e.g., registers and shared memory) should be sufficiently small that a block can be run on an SM. The block scheduler assigns more than 1 block to run concurrently on an SM when the combined resources needed by the assigned blocks does not exceed the resources available to an SM. However, since CUDA provides no mechanism to specify a subset of blocks that are to be co-scheduled on an SM, threads of different blocks can communicate only via the device memory. Thus, the optimization of inter-block communication should be a key objective in a follow-on effort.

Returning to our GPU programming and execution model, once a block is assigned to an SM, its threads are scheduled to execute on the SM's SPs by the SM's warp scheduler. The warp scheduler divides the threads of the blocks assigned to an SM into warps of 32 consecutively indexed threads from the same block. Multidimensional thread indexes are serialized in row-major order for partitioning into warps. Thus, a block of 128 threads is partitioned into 4 warps. Every thread assigned to an SM has its own instruction counter and set of registers. The warp scheduler selects a warp of *ready threads* for execution. If the instruction counters for all threads in the selected warp are identical, all 32 threads execute in 1 step. On a GPU with compute capability 2.0, each SM has 32 cores and so all 32 threads can perform their common instruction in parallel – provided that this common instruction is an integer or floating point operation. (Unfortunately, this does not work with I/O instructions.) In contrast, on a GPU with compute capability 1.3, each SM has 8 SPs and so a warp can execute the common instruction for only 8 threads in parallel. Hence, when the compute capability is 1.3, the GPU requires 4 rounds of parallel execution to execute the common instruction for all 32 threads of a warp. When the instruction counters of the threads of a warp are not the same, the GPU executes the different instructions serially. Note that instruction counters may become different as the result of data dependent conditional branches in a kernel [CUDA10].

Given the GPU's complex memory hierarchy and small local memories, the hiding of memory latency is key to obtaining good performance. At best, an SM's warp scheduler can hide much of the 400 to 600 cycle latency of device-memory access by executing warps that are ready to do arithmetic operations while other warps wait for device-memory accesses to complete. Thus, in many cases, the performance of code that makes many accesses to device memory can often be improved by optimizing it to increase the number of warps scheduled on an SM. This

optimization could involve increasing the number of threads per block and/or reducing the shared memory and register utilization of a block, to enable the scheduling of a larger number of blocks on an SM.

Summary of Our Prior and Phase I Work in Parallel GPU Implementation. For more than three decades, our research team has developed and published algorithm-to-architecture mapping techniques for parallel and real-time computation of image and signal processing algorithms, including fluid dynamics, mechanical engineering simulation, military applications such as automated target recognition (ATR), medical applications such as tomography, multi-look image reconstruction for persistent surveillance, computer vision, and many other applications. We have mapped such algorithms to a wide variety of parallel processors, including massively parallel meshes, MIMD arrays, SPMD networks of workstations, high-performance embedded processors, FPGAs, cellular automata, multicore GPUs and CPUs. We thus have the technological expertise to address the proposed effort of mapping CPD algorithms to parallel multicore GPU processors or clusters of GPUs.

In Phase I, we developed techniques for mapping key computational kernels of the *CONSIF* CPD application to a single Nvidia Fermi GPU. This has allowed us to investigate the data movement, computation, and communication modes of the Fermi and its CUDA support software, in the context of a realistic CPD application. The CPD application and its kernels can be run in fixed- or variable-grid, fixed- or variable-population modes, with different levels of computational precision. As discussed in detail in Section 1.2, we have achieved speedups in the range of 15X to 25X for realistically-parameterized kernels, without loss of computational accuracy with respect to the reference (single-core CPU) implementation.

In summary, we achieved objectives of CPD/CFD application analysis, implementation, test, and verification by first decomposing each application in its entirety to portray operations, data structures, and data flows. The analysis of cost for each application and its constituent computational kernels driven by the target algorithm's computational requirements determines the computational work and time complexity. Similarly, an algorithm's data structure organization and data flows, analyzed with respect to a target architecture's memory map and bus structure, determine the memory transformations that are required to optimize I/O cost and avoid I/O bottlenecks. Because the gap between processor performance and bus throughput continues to grow as register cycle rate outstrips channel bandwidth, we have found in Phase I that I/O cost primarily impacts performance in array-based applications such as CPD computations.

1.2. Benefits of UHN's Phase I Research

There is a rapidly growing need for a broad range of scientific and engineering application software that can be utilized on terascale and petascale computer systems. Although few commercial vendors are addressing this need, UltraHiNet LLC (UHN) has developed advanced technology for porting applications to a wide variety of High Performance Computing (HPC) platforms, including graphics processing units (GPUs) and clusters of nodes comprised of a CPU hosting multiple GPUs. Our approaches involve complexity and scaling analyses, algorithm redesign, software re-architecting, software porting, and software testing, in order to fine-tune and optimize the use of applications on these massively parallel supercomputing systems.

Of particular interest to this proposal are algorithms that represent

CPD/CFD simulation applications of interest to DOE. For example, grid based simulations of applications that are key to energy production and control, such as computational particle dynamics in power generation and rocket combustion. ***If grid-based algorithms could be readily and automatically ported to a wide range of current and foreseeable computing platforms, then the pace of technology advancement could be increased, and considerable public expense could be alleviated.***

Multicore GPUs in modern computers and gaming consoles can currently achieve terascale throughput, and can be clustered in massively parallel supercomputers with thousands of energy-efficient GPUs. Our research aims to minimize the size of the required GPU cluster through the development of highly efficient mapping technology (CPDMu) that will automatically produce accurate, highly efficient architecture-tailrored GPU implementations of applications such as the CPD kernels. ***Because our methodology is mathematically sound, it can be adapted to yield more general algorithm-to-architecture mappings for a wider class of DOE grid- and mesh-based simulation applications.*** Anticipated benefits will include the following:

1.2.1. Parallel Implementation of Legacy Simulation Codes. Computational particle dynamics and computational fluid dynamics simulations are frequently represented as large assemblies of legacy code. For example, DOE applications (e.g., weapons design, simulation, and analysis; nuclear fusion power generation research and development), NASA applications (e.g., rocket fuel burning (solid or liquid); sub-sonic to hypersonic aerodynamics simulation supporting advanced aircraft/spacecraft design), domestic applications such as law enforcement (crowd behavior modeling) or environmental hazard dispersion – each of these applications areas is represented by extensive legacy codes developed for implementation on CPUs.

In practice, the parallelization and optimization of these legacy simulation codes for GPU clusters would significantly increase the accessibility of these simulations to researchers and engineers. In practice, we foresee that this increased accessibility would be due to (a) shorter runtimes resulting from implementation on fast, commercially-available parallel processors, which could lead to faster design iterations; (b) wider availability of commercialized CPD/CFD simulations on parallel architecture such as GPUs; and (c) a longer useful life for each simulation, given more effective updating/porting of simulation codes to emerging or novel parallel architectures.

1.2.2. Ready Optimization of Parallel Implementations. The current practice of manually implementing and manually optimizing legacy simulation codes for each parallel target architecture is potentially very expensive (consumptive of public resources such as time, human effort, and money) and fraught with potential pitfalls such as unforeseen obsolescence. For instance, if a legacy code is hand-implemented and manually-optimized for a parallel architecture (A_1), then the architecture is replaced several years later by a different, more efficient architecture (A_2), the manual implementation and optimization effort expended for A_1 must be repeated for A_2. Given the speed at which computer technology development progresses, it is unsurprising that manual implementation and optimization are wasteful processes.

Our Phase I results have made it possible to develop and enhance middleware and software prototypes that instantiate techniques for mapping CFD/CPD algorithms to parallel architectures such as GPUs and clusters of multiple nodes, where each node has a CPU that controls one or more GPUs. In future, these results are expected to support further development, implementation, and commercialization of semi-automatic algorithm-to-architecture mapping techniques and software that would significantly reduce the cost and effort associated with

manual implementation and optimization of such mappings.

1.2.3. Explore Techniques of Legacy Code Parallelization for New Architectures. As mentioned previously, there is currently no unifying model for parallel programming. As a result, when a new parallel architecture is introduced that is different from previous architectures, efforts to parallelize codes for the new architecture usually begin afresh, with little re-use of previous techniques. This leads to significant amounts of re-work that are somewhat repetitive from one architecture to another, and incurs problems cited in Section 1.2.1 and 1.2.2.

Our Phase I results have allowed us to investigate means for parallelizing on GPU clusters the basic types of operations typically found in CPD/CFD simulation codes, such as pointwise operations between arrays, vector-matrix and matrix-matrix multiplication, inner products, matrix row/column operations, etc. This could support further development of a more general system for parallelizing a wide variety of legacy codes on GPUs and GPU clusters. Additionally, this aspect would help pioneer new areas of theoretical and applied research in algorithm-to-architecture parallel mapping, and is designed to make HPC versions of legacy CPD/CFD simulations available for a wide variety of DOE, DOD, and domestic applications, as noted in Section 1.2.1.

1.3. Degree to Which Phase I Has Demonstrated Technical Feasibility

1.3.1. Conformance to Proposed Phase I Technical Objectives. In order to validate and demonstrate the feasibility of the CPDMu prototype, we proposed and achieved the following technical objectives in our Phase I research and development effort:

1. *Design efficient parallel implementations for CPD based combustion simulation algorithms*, based on paradigms discussed in Section 1.

   Result: In Phase I, we designed parallel implementations of key computational kernels of the CPD simulation legacy code (Fortran), for one multicore Nvidia Fermi GPU processor, using the CUDA-C language. These implementations were performance-tested, with example test results given in Section 1.3.2.

2. *Analyze error of each system module*, to determine effect on the computational accuracy of CPD simulation.

   Result: Output error of the parallel CPD kernel implementations as well as the single-core CPU implementation was measured and no significant differences were detected between the CPU and GPU implementations.

3. *Analyze computational cost associated with CPD implementation strategies*, and effect of different data structure size and processor or I/O bus parameters on computational and accuracy performance metrics.

   Result: Prior to producing the parallelized CPD kernels, the CPD computational kernels were analyzed to determine computational and data movement cost. The measured effect (Section 1.3.2) of data structure size variations and particle population variations was correctly predicted by our analysis theory.

4. *Survey GPU software and hardware in support of algorithm-to-architecture mapping analysis and prototype implementation* in Phase I.

   Result: We surveyed the available GPU hardware and support software and determined that the Nvidia Fermi GPU had the most effective storage, computational throughput, and communication, as well as excellent potential for upward growth and compatibility. Thus, we found the Fermi to be the platform of choice for our Phase I demonstrations and tests, as well as possible follow-on efforts in software development and implementation.

5. *Develop a proof-of-principle implementation of a simplified but representative CPD algorithm on a multicore GPU processor* that our analysis (Objectives 3 and 4, above) indicates is the best-performance platform.

   Result: Per Item 1), above, we implemented key computational kernels from the CPD simulation on the Nvidia Fermi GPU, with performance results given in Section 1.3.2.

6. *Design and analyze indicated Phase II enhancements to a production CPD simulation system*, with detailed performance analysis and prototyping of hardware implementation. This will include estimation of cluster size required for fast parallel CPD based combustion simulations under various assumptions about simulated physical parameters.

   Result: We have designed Phase II enhancements that we proposed to DOE, including middleware configuration, graphical user interface(s), mapping of the entire CPD simulation application, and a CFD simulation application (e.g., XGC-1 plasma burning simulation). These applications will be implemented on one or more GPU clusters, where each cluster node is comprised of a multicore CPU that controls multiple Nvidia Fermi GPUs.

1.3.2. Phase I Test Results. As proof-of-principle for Phase I, we firstly developed and analyzed a C version of the Fortran 90 legacy *CONSIF* CPD simulation application obtained from our consultant Dr. S. Balachandar. Secondly, by applying our algorithm-to-architecture mapping transformations to the legacy code, we produced a GPU implementation of the parallel computational kernels (most of the *CONSIF* code is sequential). Thirdly, we measured the kernels' performance on the target GPU. In this section, we present our Phase I test results specific to two such kernels, which are representative of our Phase I work.

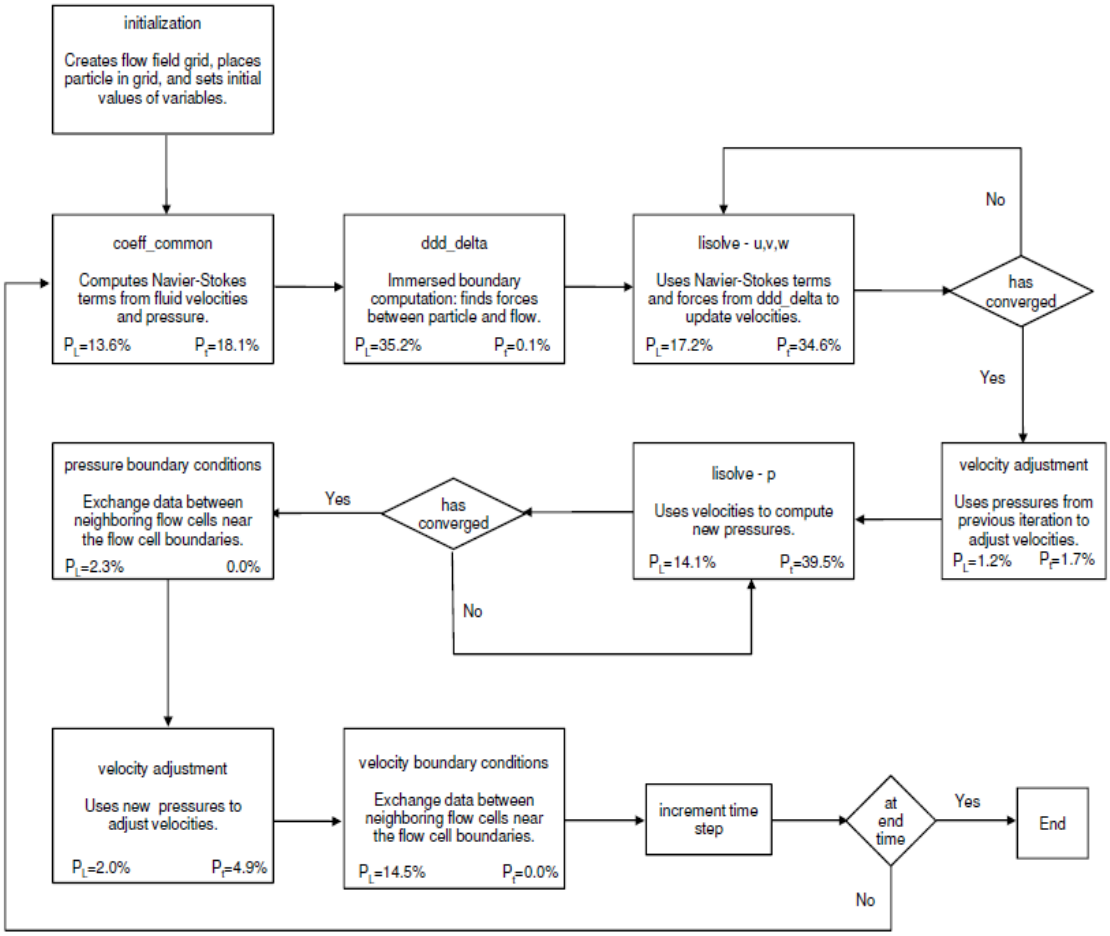


Figure 9. Flowchart of the *CONSIF* CPD application, partitioned into computational kernels and sub-processes including key decision points. Here, P_L and P_t respectively denote percentage of total lines of code occupied by a kernel process, and percentage of total compute time on the CPU. For example, the kernel *velocity adjustment* has P_L = 2.0% (occupies 2 percent of total lines of code) and P_t = 4.9% (compute time is 4.9 percent of total compute time). These preliminary figures are not precise for all test cases, but were used as a baseline to determine which kernels to optimize first.

The Nvidia Fermi GPU described in Section 1 was controlled by an Intel six-core CPU clocked at 3.33 GHz, and running the Linux Ubuntu Version 10.04 operating system. To obtain our reference data, we translated the legacy Fortran 90 code into C, which was compiled on a GNU C Compiler Version 4.4.3, then ran the C code on the aforementioned CPU in single-core mode. A flowchart of the legacy code is given in Figure 9, where P_L denotes percentage of lines of source code, and P_t denotes percentage of total execution time, occupied by a given kernel process (large boxes). Given this approximate runtime profiling information, we gathered additional analytical results using sophisticated profiling procedures described in Section 2 (Work Plan), which could be implemented in a possible follow-on effort. We exploited these measurements and analytical results to transform key computational kernels of the application using our algorithm-to-architecture mapping and parallelization techniques. The resulting CUDA-C implementation was implemented on the GPU with the Nvidia CUDA Toolkit Version 3.2.

Examples of transformed CUDA-C code (for the GPU) are given in Figure 10 (for kernel *ddd_delta*) and Figure 11 (kernel *coeff_common*). Table 1 lists the performance data as a function of program parameters that are independent variables in the simulation. Observe that speedups relative to the single-core CPU range up to 29.5:1 for the *ddd_delta* kernel – this is the naïve case, with very little hand optimization.

FORTRAN version of *ddd_delta*	**CUDA-C** version of *ddd_delta*
DO N=1,NUM_P **DO** L=1,N_L I1 = P_IW(L,N) I2 = P_IE(L,N) J1 = P_JS(L,N) J2 = P_JN(L,N) K1 = P_KB(L,N) K2 = P_KT(L,N) **U_TILDE**(I1:I2,J1:J2,K1:K2) = U_N(I1:I2,J1:J2,K1:K2) + DTIME*(- DPX(I1:I2,J1:J2,K1:K2) + DIFX(I1:I2,J1:J2,K1:K2) - 1.5D0*NLX_N(I1:I2,J1:J2,K1:K2) + 0.5D0*NLX_N1(I1:I2,J1:J2,K1:K2)) * VOL_DV_T(I1:I2,J1:J2,K1:K2) **V_TILDE**(I1:I2,J1:J2,K1:K2) = V_N(I1:I2,J1:J2,K1:K2) + DTIME*(-DPY(I1:I2,J1:J2,K1:K2) + DIFY(I1:I2,J1:J2,K1:K2) - 1.5D0*NLY_N(I1:I2,J1:J2,K1:K2) + 0.5D0*NLY_N1(I1:I2,J1:J2,K1:K2)) * VOL_DV_T(I1:I2,J1:J2,K1:K2) **W_TILDE**(I1:I2,J1:J2,K1:K2) = W_N(I1:I2,J1:J2,K1:K2) + DTIME*(-DPZ(I1:I2,J1:J2,K1:K2) + DIFZ(I1:I2,J1:J2,K1:K2) - 1.5D0*NLZ_N(I1:I2,J1:J2,K1:K2) + 0.5D0*NLZ_N1(I1:I2,J1:J2,K1:K2)) * VOL_DV_T(I1:I2,J1:J2,K1:K2) **UP_TILDE**(L) = SUM(U_TILDE(I1:I2,J1:J2,K1:K2) * DDF(I1:I2,J1:J2,K1:K2) *H**3) **VP_TILDE**(L) = SUM(V_TILDE(I1:I2,J1:J2,K1:K2)*DDF(I1:I2, J1:J2,K1:K2) *H**3) **WP_TILDE**(L) = SUM(W_TILDE(I1:I2,J1:J2,K1:K2)*DDF(I1:I2, J1:J2,K1:K2) *H**3) **ENDDO** **ENDDO**	L = threadIdx.x + blockIdx.x*blockDim.x; N = threadIdx.y + blockIdx.y*blockDim.y; // particle index b = L + N * N_L; I1 = p->P_IW[b]; I2 = p->P_IE[b]; J1 = p->P_JS[b]; J2 = p->P_JN[b]; K1 = p->P_KB[b]; K2 = p->P_KT[b]; nx = NI+1; nxny = nx*(NJ+1); UP_TILDE = 0; VP_TILDE = 0; WP_TILDE = 0; **for**(int k = K1; k <= K2;k++) **for**(int j = J1; j <= J2;j++) { **for**(int i = I1; i <= I2;i++){ a = i+ j*nx + k*nxny ; a1 = i-I1+(j-J1)*4+ (k-K1)*16; **U_TILDE** = p->U_N[a] + p->ddlt_NS_X[a]; **V_TILDE** = p->V_N[a] + p->ddlt_NS_Y[a]; **W_TILDE** = p->W_N[a] + p->ddlt_NS_Z[a]; UP_TILDE+=U_TILDE*DDF[a1]*H_3; VP_TILDE+=V_TILDE*DDF[a1]*H_3; WP_TILDE+=W_TILDE*DDF[a1]*H_ } } }

Figure 10. Legacy and CUDA-C versions of CPD simulation kernel *ddd_delta*, which computes the corresponding required intermediate velocity at the Lagrangian grid point, by summing the intermediate velocities of the surrounding cells, then multiplying by the Dirac delta function computed earlier, times the mesh resolution cubed.

Observe how the application transformation process works in Figure 10. Firstly, the particle and index loops are converted to thread definitions. Thus, we have the Fortran loops

```
DO N=1,NUM_P
DO L=1,N_L
```

being converted to the following CUDA thread definition statements:

```
L = threadIdx.x + blockIdx.x*blockDim.x;
N = threadIdx.y + blockIdx.y*blockDim.y; // particle index
b = L + N * N_L;
```

where the latter statement defines the range of particle index. Subsequently, the range limits *I1*, *I2*, *J1*, *J2*, *K1*, and *K2* are defined in Fortran in terms of arrays and, in CUDA, in terms of pointers to arrays. The variables *U_TILDE*, *V_TILDE*, and *W_TILDE* are computed in Fortran using inherently parallel notation (e.g., U_TILDE(I1:I2,J1:J2,K1:K2) with ranges), while CUDA utilizes loops that cycle over each dimension of the array (e.g., via indices *k*, *j*, and *i*) per thread, resulting in cleaner, more compact code.

In contrast to the code example given in Figure 10, the application transformation process proceeds somewhat differently for the kernel *coeff_common*. As can be seen from the left-hand side of Figure 11, the legacy Fortran code is not well organized, consisting of a long, run-on expression. In contrast, the transformed legacy code, expressed in CUDA-C, is on the right-hand side of Figure 11, and is more clearly structured. Firstly, the limit parameters *nx* and *ny* are defined, then the base thread ID is specified as

```
id = (threadIdx.x+1) + nx*( blockIdx.x+1 + ny*(blockIdx.y+1) )
```

FORTRAN version of *coeff_common*	**CUDA-C** version of *coeff_common*
NLX_N(sx1:ex1,sy1:ey1,sz1:ez1)= DTIME*(DY(sx1:ex1,sy1:ey1,sz1:ez1)*DZ(sx1:ex1,sy1:ey1,sz1:ez1)*(UE(sx1:ex1,sy1:ey 1,sz1:ez1)*0.5D0*(U_N(sx1:ex1,sy1:ey1,sz1:e z1)+U_N(sx1+1:ex1+1,sy1:ey1,sz1:ez1)) - UW(sx1:ex1,sy1:ey1,sz1:ez1)*0.5D0*(U_ N(sx1:ex1,sy1:ey1,sz1:ez1)+U_N(sx1-1:ex1-1, sy1:ey1,sz1:ez1))) + DX(sx1:ex1,sy1:ey1,sz1:ez1)*DZ(sx1:ex1, sy1:ey1,sz1:ez1) * (VN(sx1:ex1,sy1:ey1,sz1:ez1) * 0.5D0 * (U_N(sx1:ex1,sy1:ey1,sz1:ez1)+U_N(sx1:ex1, sy1+1:ey1+1,sz1:ez1))-VS(sx1:ex1,sy1:ey1,sz 1:ez1) *0.5D0* (U_N(sx1:ex1,sy1:ey1,sz1:ez1)+U_N(sx1: ex1,sy1-1:ey1-1,sz1:ez1))) + DX(sx1:ex1,sy1:ey1,sz1:ez1)* DY(sx1:ex1,sy1:ey1,sz1:ez1) *(WT(sx1:ex1,sy1:ey1,sz1:ez1)* 0.5D0*(U_N(sx1:ex1,sy1:ey1,sz1:ez1)+U_	int nx = blockDim.x+2; int ny = gridDim.x+2; int id = (threadIdx.x+1) + nx*(blockIdx.x+1 + ny*(blockIdx.y+1)); int id_px = id + 1; int id_mx = id - 1; int id_py = id + nx; int id_my = id - nx; int id_pz = id + nx*ny; int id_mz = id - nx*ny; double **nlx** = dtime*(dev->DY[id]*dev->DZ[id] * (dev->UE[id]*0.5*(dev->U_N[id] +dev->U_N[id_px]) - dev->UW[id]*0.5*(dev->U_N[id] + dev->U_N[id_mx])) + dev->DX[id]*dev->DZ[id] * (dev->VN[id]*0.5*(dev->U_N[id] +dev->U_N[id_py]) -dev->VS[id]*0.5*(dev->U_N[id]

N(sx1:ex1,sy1:ey1,sz1+1:ez1+1))-WB(sx1:ex1,sy1:ey1,sz1:ez1)*0.5D0* (U_N(sx1:ex1,sy1:ey1,sz1:ez1)+U_N(sx1:ex1,sy1:ey1,sz1-1:ez1-1))))	+dev-U_N[id_my])) + dev->DX[id]*dev->DY[id] * (dev->WT[id]*0.5*(dev->U_N[id]+dev->U_N[id_pz]) -dev->WB[id]*0.5*(dev->U_N[id]+dev->U_N[id_mz])));

Figure 11. Legacy and CUDA-C versions of CPD simulation kernel *coeff_common*, which takes the results of the last fluid dynamics computation, and computes *NLX_N* (a convective term from Navier-Stokes, expressed as $\mathbf{u} \bullet \Delta\mathbf{u}$) which is used in the immersed boundary method.

From the variable *id*, the offsets for indirect indexing denoted by *id_px*, *id_py*, *id_pz*, *id_mx*, *id_my*, and *id_mz* are defined. These offsets are then applied to device memory, e.g.,

```
dev->U_N[id_py]
```

to reference intermediate values in the computation. The result of summing the intermediate parts is multiplied by the time increment *dtime* and is stored in the double-precision variable *dlx*.

From inspection of the right-hand side of Table 1, we see that the simple translation of *coeff_common* shown in Figure 11 can be speeded up on the Fermi processor by a factor of over 31.5:1 for values *ex1* = 100, *ex2* = 50, and *ex3* = 50. In practice, as the problem becomes larger, prior to overrunning device memory, the speedup increases significantly.

In order to successfully address device memory overrun, one can distribute the parallelization problem over multiple GPUs in a cluster. However, this would require more sophisticated optimization of the data movement between levels of the memory hierarchy (i.e., *host* > *device* > *shared memory* > *local memory*). We have developed transformations that support algorithm-to-architecture mapping and optimization for this purpose [Band10], which could eventually support mapping CPD/CFD simulations to GPUs and clusters of CPU-GPU nodes.

Table 1. Measured performance for (left-hand table) simulation kernel *ddd_delta* (see code in Figure 10) and (right-hand table) simulation kernel *coeff_common* (see code in Figure 11), where *NUM_P* (number of particles) and *N_L* (number of loops) are as in the Fortran code, and measured times (Fortran legacy code and CUDA-C code) are in milliseconds. *Speedup* = *Fortran_time* / *CUDA_time*.

ddd_delta

NUM_P	N_L	FORTRAN (msec)	CUDA-C (msec)	Speed up
1	64	2.60E-01	4.00E-01	0.65
1	256	1.10E+00	4.20E-01	2.62
1	512	2.20E+00	4.40E-01	5.00
1	600	2.56E+00	4.60E-01	5.57
4	16	2.80E-01	3.40E-01	0.82
4	64	1.10E+00	4.20E-01	2.62
4	256	4.36E+00	4.60E-01	9.48
4	512	8.76E+00	5.20E-01	16.85
4	600	1.03E+01	5.00E-01	20.64
16	4	2.60E-01	3.00E-01	0.87
16	16	1.08E+00	3.40E-01	3.18
16	64	4.36E+00	4.40E-01	9.91
16	256	1.73E+01	9.40E-01	18.40

coeff_common

ex1	ey1	ez1	FORTRAN (msec)	CUDA (msec)	Speed up
1	1	50	6.40E-03	4.62E-02	0.14
5	1	50	2.01E-02	4.65E-02	0.43
25	1	50	9.35E-02	4.75E-02	1.97
50	1	50	1.79E-01	4.80E-02	3.73
100	1	50	3.47E-01	4.80E-02	7.23
1	25	50	1.49E-01	1.62E-01	0.92
5	25	50	5.21E-01	1.75E-01	2.98
25	25	50	2.24E+00	2.00E-01	11.20
50	25	50	4.35E+00	2.13E-01	20.47
100	25	50	8.57E+00	2.75E-01	31.16
1	50	50	2.81E-01	2.79E-01	1.01
5	50	50	1.43E+00	3.13E-01	4.57
25	50	50	4.86E+00	3.38E-01	14.40
50	50	50	9.11E+00	3.50E-01	26.03
100	50	50	1.73E+01	5.50E-01	31.53

The following equipment was employed in Phase I.

2. Facilities and Equipment

UltraHiNet LLC's computational facilities include a computer with a 12 MB Intel Smart Cache and a six-core Intel Core i7-980X Extreme Processor clocked at 3.33 GHz. This computer hosts an Nvidia C2050 Fermi GPU that has 448 cores and is clocked at 1.15 GHz. The Fermi GPU has 3 GB of memory, with a 64 KB cache, configurable as 48/16 or 16/48 L1 cache. The GPU's shared memory bus has throughput of 144 GB/sec. The external bus is a PCI Express bus with throughput measured at 4,280.6 MB/sec.

Supporting software for Phase I included Ubuntu 10.04 LTS ("Lucid Lynx", by Canonical LTD), Nvidia CUDA Toolkit Version 3.2, and GNU C Compiler Version 4.4.3.

3. Consultants

Dr. Sivaramakrishnan Balachandar, funded by DTRA to develop CPD based simulations of air blast phenomena, and previously funded for development of combustion simulations, was a consultant on the Phase I project. He supplied *CONSIF* code and expertise in CPD-based combustion simulation.

4. Bibliography and References Cited

[AAR00] I. Al-furaih, S. Aluru and S. Ranka. (2000) Parallel Construction of Multidimensional Binary Search Trees, *IEEE Transactions on Parallel and Distributed Systems*, February 2000, pp. 136-148.

[Adam96] N. Adams and K. Shariff. (1996). A high-resolution hybrid compact-ENO scheme for shock-turbulence interaction problems. J. Comput. Phys. 127, 27–51.

[Adam09] M. Adams, S-H. Ku, P. Worley, E. D'Azevedo, J.C. Cummings, C.S. Chang, (2009) Scaling to 150K cores: Recent algorithm and performance engineering developments enabling XGC-1 to run at scale, *J.Phys. Conf.* vol 180, p. 012036.

[Agra93] G. Agrawal, A. Sussman, and J. Saltz. (1993) An Integrated Runtime and Compile-time Approach for Parallelizing Structured and Block Structured Applications, CRPC Tech.Rpt. TR933368, Rice University.

[Alfu97] I. Al-furaih, S. Aluru and S. Ranka, (1997) Practical Algorithms for Selection on Coarse Grain Parallel Computers, *IEEE Transactions on Parallel and Distributed Systems*, August 1997, pp. 803-810.

[Baden91] S. Baden. (1991) Programming Abstractions for Dynamically Partitioning Localized Scientific Calculations Running on Multiprocessors, *SIAM Jrn. Sci. and Stat. Computation*, vol. 12, no. 1.

[Bag02] P. Bagchi and S. Balachandar, (2002) Steady Planar Straining Flow past a Rigid Sphere at Moderate Reynolds Number, *Journal of Fluid Mechanics*, Vol. 466, pp. 365-407.

[Bag03] P. Bagchi and S. Balachandar, (2003) Inertial and viscous forces on a rigid sphere in straining flows at moderate Reynolds numbers. *Journal of Fluid Mechanics* 481, 105-148.

[Bala95] S. Balachandar, D. A. Yuen, D. M. Reuteler and G. S. Lauer, (1995) Viscous dissipation in three-dimensional convection with temperature-dependent viscosity. *Science* 267, 1150-1153.

[Bala01] S. Balachandar, J.D. Buckmaster, and M. Short, (2001) The generation of axial vorticity in solid-propellant rocket-motor flows, *Journal of Fluid Mechanics*, Vol 429, pp.283-305.

[Bala08] S. Balay, Buschelman K, Eijkhout V, Gropp W D, Kaushik D, Knepley M G, McInnes L C, Smith B F and Zhang H (2008) PETSc users manual TR ANL-95/11 - Revision 3.0.0 Argonne National Laboratory.

[Band10] S. Bandopadhyaya and S. Sahni. (2010) "GRS - GPU Radix Sort For Multifield Records", *Proceedings of the IEEE International Conference on High Performance Computing (HiPC)*.

[Bow01] K. Bowers, (2001) Accelerating a Particle-In-Cell simulation using a hybrid counting sort, *J.Comput.Phys*, vol. 173, p.393.

[Bozk93] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. (1993) *Compiling HPF for Distributed memory MIMD computers, Impact of Compilation Technology on Computer Architecture*. David Lilja and Peter Bird, eds. Kluwer Academic, Norwell, MA, October 1993, pp. 191-217.

[Cai99] F.M. Caimi, M.S. Schmalz, and G.X. Ritter, (1999) Mapping of image compression transforms to reconfigurable processors: Simulation and analysis. *Proceedings SPIE*, Vol. 3814, pp.98-114.

[Chan09] C.S. Chang, S. Ku, P.H. Diamond, Z. Lin, S. Parker, T.S. Hahm and N. Samatova, (2009) Compressed ITG turbulence in diverted Tokamak edge, *Phys. Plasmas* Vol. 16, pp. 056108.

[Chou92] A. Choudhary, G.C. Fox, S. Ranka, S. Hiranandani, K. Kennedy, C. Koelbel, and J. Saltz. (1992) Software Support for Irregular and Loosely Synchronous Problems, Tech. Report CRPC-TR92258, Rice University.

[Chou93] A. Choudhary, G. C. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, S. Ranka, and C-W. Tseng. (1993) Unified Compilation of Fortran 77D and 90D, *ACM Letters on Prog. Lang. and Systems*, Vol. 2, pp. 95-114.

[Cook05] A. Cook and W. Cabot, (2005). Hyperviscosity for shock-turbulence interaction. J. Comput. Phys. 203, 379–385.

[Cook07] A. Cook, (2007). Artificial fluid properties for large-eddy simulation for compressible turbulent mixing. Phys. Fluids 19, 055103.

[CUDA10] NVIDIA CUDA Programming Guide, Version 3.0, 2010, (http://developer.nvidia.com/object/gpucomputing.html).

[Das91a] R. Das, J. Saltz, and H. Berryman. (1991) Manual for Parti Runtime Primitives, Revision 1, Int.Rpt. 91-17, ICASE.

[Das91b] R. Das, P. Ponnusamy, J. Saltz, and D. Mavriplis. (1991) Distributed Memory Compiler Methods for Irregular Problems: Data Copy Re-use and Runtime Partitioning, in *Compilers and Runtime Software for Scalable Multiprocessors* (J. Saltz and P. Mehrota, Editors), Amsterdam: Elsevier.

[Decy95] V. Decyk, (1995) Skeleton PIC codes for parallel computers, *Computer Phys. Comm*. Vol. 87, pp.87–94.

[Des02] P.E. Des-Jardins, M.R. Baer, R.L. Bell & E.S. Hertel. (2002) Towards numerical simulation of shock induced combustion using probability density function approaches. Tech. Rep. SAND2002-2175.

[DOE1] Science Based Nuclear Energy Systems Enabled by Advanced Modeling and Simulation at the Extreme Scale, Workshop Summary, Washington, DC, May 11-12, 2009.

[DOE2] Basic Energy Sciences Workshop on Basic Research Needs for Advanced Nuclear Energy Systems, Report of the Basic Energy Sciences Workshop, July 31-August 3, 2006.

[DOE3] Workshop on Simulation and Modeling for Advanced Nuclear Energy Systems, August 15-17, 2006.

[DOE4] Report of the Nuclear Physics and Related Computational Science R&D for Advanced Fuel Cycles Workshop, Bethesda, MD, August 10-12, 2006.

[Duff86] I.S. Duff and J.K. Reid. (1986) *Direct Methods for Sparse Matrices*, New York: Oxford University Press.

[Edels] D.J. Edelsohn. Hierarchical Tree Structures as Adaptive Meshes" SCCS Report 193, Syracuse Univ. (USA).

[Fer02] J. Ferry and S. Balachandar, (2002) Equilibrium expansion for the Eulerian velocity of small particles, *Powder Technology*, Vol. 125, pp.131-139.

[Fer05] J. Ferry and S. Balachandar, (2005) Equilibrium Eulerian approach for predicting the thermal field of a dispersion of small particles. *International Journal of Heat and Mass Transfer* 48, 681-689.

[Fox88] G.C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. (1988) *Solving Problems on Concurrent Computers*, Prentice-Hall, Englewood Cliffs NJ.

[Fox90a] G.C. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. (1990) "Fortran D Language Specification", Department of Computer Science Technical Report 90-141, Rice University.

[Fox90b] G.C. Fox. (1990) "Hardware and Software Architectures for Irregular Problems Architectures", Invited Talk at ICASE Workshop on Unstructured Scientific Computations on Scalable Multiprocessors, Nagshead NC.

[Fox91a] G.C. Fox. (1991) "Architecture of Problems and Portable Parallel Software Systems", in *Proc. Supercomp '91*.

[Fox91b] G.C. Fox. (1991) "Parallel Problem Architectures and Their Implications for Parallel Software Systems", DARPA Workshop, Providence RI, February 1991.

[Fox91c] G.C. Fox. (1991) "Fortran D as a Portable Software System for Parallel Computers", Presentation at Supercomputing USA/Pacific 91, Santa Clara, CA.

[Gai99] D. Gaitonde and M. Visbal, (1999). Further development of a Navier-Stokes solution procedure based on higher-order formulas, AIAA Paper 99-0557.

[God05] D. Goddeke, R. Strzodka, and S. Turek (2005), "Accelerating Double Precision FEM Simulations with GPUs," In *Proceedings of ASIM 2005 - 18th Symposium on Simulation Technique*.

[GPGPU] GPGPU -- XGC-1-Wiki, the free XGC-1 reference, http://www.XGC-1-online.com/Wiki/GPGPU.

[Gram01] A. Grama, V. Kumar, S. Ranka and V. Singh (2001) Architecture Independent Analysis of Parallel Programs, *Proc. 2001 International Conference on Computational Science*, Vol. 2, pp.599-608.

[Gior97] R. Giorgi, C. A. Prete, G. Prina, and L. Ricciardi, (1997) Trace Factory Generating Workloads for Trace-Driven Simulation of Shared-Bus Multiprocessors," *IEEE Concurrency*, Vol. 5, No. 4, (1997), pp. 54-68.

[Hahm88] T.S. Hahm, (1988) Nonlinear Gyrokinetic Equations for Tokamak Microturbulence, *Phys. Fluids* Vol. 31, p. 2670.

[Har04] M.J. Harris, (2004) Fast Fluid Dynamics Simulation on the GPU, Chapter 38, Excerpted from GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics, Addison-Wesley.

[Has09] A. Haselbacher, J. Chao, and S. Balachandar. (2009) A Massively Parallel Multi-Block Hybrid Compact-WENO Scheme for Compressible Flows (submitted to Journal of Computational Physics).

[Heyw92a] T. Heywood and S. Ranka. (1992) A Practical Hierarchical Model of Parallel Computation: Binary Tree and FFT Graph Algorithms, *Journal of Parallel and Distributed Computing*, November 1992, 233--249.

[Heyw92b] T. Heywood and S. Ranka. (1992) A Practical Hierarchical Model of Parallel Computation: The Model, *Journal of Parallel and Distributed Computing*, November 1992, vol. 3, pp. 212--232.

[Hig02] N. Higham, (2002) *Accuracy and Stability of Numerical Algorithms*, Siam Press, Second Edition.

[Hofs05] H. Hofstee, (2005) Power Efficient Processor Architecture and Cell Processor, *Proceedings of the 11th Int'l Symposium on HPC Architecture*.

[Koel90] C. Koelbel and P. Mehrota. (1990) Compiling Global Name-Space Loops for Distributed Execution, in *IEEE Transacations on Parallel Distributed Systems*, vol. 2, no. 4.

[Lad03] F. Ladeinde, X. Cai, M. Visbal, and D. Gaitonde, (2003). Parallel implementation of curvilinear high-order formulas. *Int. J. Comp. Fluid Dyn.* Vol. 17, 467–485.

[Lee83] W.W. Lee, (1983) Nonlinear gyrokinetic equations *Phys. Fluids*, Vol. 26, pp.556-562.

[Lee87] W.W. Lee, (1987) Gyrokinetic particle simulation model, *J. Comput. Phys.*, Vol.72, p.243.

[Lee88] W.W. Lee and W.M.Tang, (1988) Gyrokinetic particle simulation of ion temperature gradient instabilities, *Phys. Fluids*, Vol. 31 pp. 612-624.

[Liao96] W. Liao, C. Ou and S. Ranka. (1996) Dynamic alignment and distribution of irregularly coupled data arrays for scalable parallelization of particle-in-cell problems, *Proc.IPPS '96, Par. Proc. Symp.*, pp. 57-61.

[Liew89] P.C. Liewer and V.K. Decyk. (1989) A General Concurrent Algorithm for Plasma Particle-in-Cell Simulation Codes, *Journal of Computational Physics*, vol. 85, no. 2, pp. 302-322.

[Lin98] Z. Lin, Hahm T.S., Lee W.W., Tang W.M. and White R.B. (1998) Turbulent transport reduction by zonal flows: Massively parallel simulations, *Science* Vol. 5384, pp. 1835-1837.

[Liu90] L. Liu and J. Peir: (1990) A Performance Evaluation Methodology for Coupled Multiple Supercomputers. *Proc. 1990 International Conference on Parallel Processing*, vol. I, pp. 198-202.

[Lu91] L.C. Lu and M.C. Chen. (1991) Parallelizing Loops with Indirect Array References or Pointers, in *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*.

[Mav91a] D.J. Mavriplis. (1991) Three-dimensional Unstructured Multigrid for the Euler Equations, in *Proceedings of the AIAA 10th Computational Fluid Dynamics Conference*.

[Mav91b] D.J. Mavriplis. (1991) Unstructured and Adaptive Mesh Generation for High Reynolds Number Viscous Flows, ICASE Technical Report 91-25.

[Mir88] R. Mirchandaney, J.H. Saltz, R.M. Smith, D.M. Nicol and K. Crowley. (1988) Principles of Runtime Support for Parallel Processors, *Proceedings of the 1988 ACM Int'l. Conf. Supercomputing*, pp. 140-152.

[Nev05] W.M. Nevins, G.W. Hammett, A.M. Dimits, W. Dorland, D.E. Shumaker, (2005) Discrete particle noise in particle-in-cell simulations of plasma microturbulence, *Phys. Plasmas*, vol. 12, p.122305.

[Nyl07] L. Nyland, (2007) N-body on the GPU, presented at AstroGPU Conference, Princeton, NJ (ftp://ftp.cs.unc.edu/pub/publications/techreports/04-032.pdf).

[OGR95] C. W. Ou, M. Gunwani, and S. Ranka. (1995) Architecture-Independent Locality-Improving Transformations of Computational Graphs Embedded in k-Dimensions, *Proceedings of the 1995 International Conference on Supercomputing*, pp. 295--298.

[OR97] C. W. Ou and S. Ranka. (1997) Parallel Incremental Graph Partitioning, *IEEE Transactions on Parallel and Distributed Systems*, Aug 1997, pp. 884-896.

[Pas00] A. Pascarelli, U. Piomelli, and G. Candler, (2000). Multi-block large-eddy simulations of turbulent boundary layers. J. Comput. Phys. 157, 256–279.

[Pham06] M.V. Pham, F. Plourde, S.D. Kim, and S. Balachandar, (2006) Large-eddy simulation of a pure thermal plume under rotating conditions. *Physics of Fluids* 18, 015101.

[Pod07] V. Podlozhnyuk, Histogram calculation in CUDA, 2007. http://www.nvidia.com/object/cuda_sample_data-parallel.html)

[Pov98] A. Povitsky, (1998). Parallel directionally split solver based on reformulation of pipelined Thomas algorithm, NASA Technical Report, NASA/CR-1998-208733.

[Prze05] V. Przebinda, J. Cary, (2005) Some improvements in PIC performance through sorting, caching, and dynamic load balancing, U. Colorado.

[Rank90a] S. Ranka and S. Sahni. (1990) *Hypercube Algorithms with Applications to Image Processing and Pattern Recognition*, Springer Verlag, June 1990.

[Rank90b] S. Ranka and S. Sahni. (1991) Image Template Matching on MIMD Hypercube Multicomputers, *Journal of Parallel and Distributed Computing*, September 1990, vol. 10, No. 1, pp. 79--84.

[Rank90c] S. Ranka and S. Sahni. (1991) String Editing on a Hypercube Multicomputer, *Journal of Parallel and Distributed Computing*, vol. 9, pp. 411--418.

[Rank90d] S. Ranka and S. Sahni. (1991) Convolution on an SIMD Mesh Connected Multicomputer, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 12, pp. 315--318.

[Rank90e] S. Ranka and S. Sahni. (1991) Odd-Even Shifts on SIMD Hypercubes, IEEE Transactions on Parallel and Distributed Systems, vol. 1, pp. 77--82.

[Rank91] S. Ranka and S. Sahni. (1991) Image Transformations on Hypercube and Mesh Multicomputers, *Parallel Architectures and Algorithms for Image Understanding*. Prasanna Kumar, ed. Academic Pr., pp. 227-248.

[Rank91a] S. Ranka and S. Sahni. (1991) Efficient Serial and Parallel Algorithms for Median Filtering, *IEEE Transactions on Acoustics, Speech and Signal Processing*, June 1991, vol. 39, No. 8, pp. 1462--1466.

[Rank91b] S. Ranka and S. Sahni.(1991) Clustering on a Hypercube Multicomputer, *IEEE Tr. Par.Distrib. Sys*, vol. 2, pp. 129-137.

[Rav10] G. Ravunnikutty, R. G. Joseph, S. Ranka, E. D'Azevedo, and S. Klasky. Efficient GPU implementation for Particle in Cell algorithm, submitted to IPDPS 2011 (1 Oct 2010).

[Roc89] F. Rocca, C. Cafforio, and C. Prati, (1989) Synthetic aperture radar: A new application for wave equation techniques. *Geophysical Prospecting*, vol. 37, pp. 809–830.

[Saad86] Y. Saad. (1986) Communication Complexity of the Gaussian Elimination Algorithm on Multiprocessors, Linear Algebra Applications, vol. 77, pp. 315-340.

[Salm90] J.K. Salmon. (1990) "Parallel Hierarchical N-Body Methods", Tech.Report CRPC-90-14, CA Inst.Technol.

[Sati09] N. Satish, M. Harris, and M. Garland. (2009) Designing Efficient Sorting Algorithms for Manycore GPUs, in *Proc. IEEE 2009 International Parallel and Distributed Processing Symposium*.

[Sch97] M.S. Schmalz. (1997) Automated analysis and prediction of accuracy and performance in ATR algorithms, *Proceedings SPIE* vol. 3079, pp. 249-272.

[Sen07] T. Sengupta, A. Dipankar, and A., Rao, (2007). A new compact scheme for parallel computing using domain decomposition. *J. Comput. Phys*. Vol. 220, pp. 654–677.

[Shan97a] R. Shankar and S. Ranka. (1997) Random Data Accesses on a Coarse-grained Parallel Machine I. One-to-one Mappings, *Journal of Parallel and Distributed Computing*, July 1997, pp. 14-23.

[Shan97b] R. Shankar and S. Ranka. (1997) Random Data Accesses on a Coarse-grained Parallel Machine II. One-to-many and Many-to-one Mappings, *Journal of Parallel and Distrib. Computing*, July 1997, pp. 24-34.

[Shen06] Y. Shen, G. Yang, and Z. Gao, (2006). High-resolution finite compact difference schemes for hyperbolic conservation laws. J. Comput. Phys. 216, 114–137.

[Shew96] J.R. Shewchuk, (1996) Applied Computational Geometry: Towards Geometric Engineering (*Lecture Notes in Computer Science* vol 1148) ed Lin M C and Manocha D (Springer-Verlag) pp 203222

[Simon91] H. Simon. (1991) Partitioning of Unstructured Mesh Problems for Parallel Processing, in *Proc. Conf. on Parallel Methods on Large Scale Structural Analysis and Physics Applications*, Pergamon Press.

[Slaw07a] M. Slawinska, et al. (2007) Enhancing Portability of HPC Applications across High-end Computing Platforms, *IEEE International*, March 26-30, 2007, pp.1-8, (http://www.dcl.mathcs.emory.edu/downloads/ /hwb/papers/hcw07.pdf).

[Slaw07b] J. Slawinski, M. Slawinska and V. Sunderam. (2007) Porting Transformations for HPC Applications, in *Proc. 21st Intl. Parallel & Distrib. Proc. Symp.*, (http://www.dcl.mathcs.emory.edu/downloads /hwb/papers/pdcs07.pdf).

[Sor08] T. Sorensen, T. Schaeffter, K. Noe, M. Hansen, (2008) Accelerating the nonequispaced fast fourier transform on commodity graphics hardware, *IEEE Transactions on Medical Imaging*, Vol. 27, pp.538–547.

[StanIP] G. Stantchev, D. Juba, W. Dorland, A. Varshney, High-performance computation and visualization of plasma turbulence on graphics processors, *Computing in Science and Engineering* (2008).

[Ston07] J.E. Stone, J.C. Phillips, P.L. Freddolino, D.J. Hardy, L.G. Trabuco, K. Schulten, (2007) Accelerating molecular modeling applications with graphics processors, *J. Comput. Chemistry*, Vol. 28, pp. 2618–2640.

[Sun96] X.-H. Sun, and S. Moitra, (1996). A fast parallel tridiagonal algorithm for a class of XGC-1 applications, NASA Tech. Paper 3585.

[Swe98] M. Sweat and Wilson, J.N. (1998) Overview of AIM: Supporting computer vision on heterogeneous high-performance computing systems. *Proceedings SPIE*, Vol. 3452, pp. 81ff.

[Tay08] Z. A. Taylor, M. Cheng, S. and Ourselin, (2008) High Speed Non-linear Finite Element Analysis for Surgical Simulation Using Graphics Processing Units, *IEEE Transactions on Medical Imaging*, vol. 27, no. 5, pp 650-663.

[Tesla10] Nvidia Tesla (2010) http://wapedia.mobi/en/NVIDIA Tesla.

[Walk90] D.W. Walker. (1990) "Characterizing the Parallel Performance of a Large-Scale, Particle-in-Cell Plasma Simulation Code", *Concurrency: Practice and Experience*, vol. 2, no. 4, Chichester UK: John Wiley.

[Whit90] D.L. Whitaker, D.C. Slack, and R.W. Walters. (1990) Solution Algorithms for the Two-Dimensional Euler Equations on Unstructured Meshes, in *Proc. of the AIAA 28th Aerospace Sciences Mtg*, Reno NV.

[Won89] Y. Won and S. Sahni. (1989) Hypercube-to-Host Sorting, *Jrn. of Supercomputing*, vol. 3, pp. 41-61.

[Yee99] H. Yee, N., Sandham, and M. Djomehri, (1999). An artificial nonlinear diffusivity method for supersonic reacting flows with shocks. *J. Comput. Phys.* Vol. 150, pp. 199–239.

[Zeng05] L. Zeng, S., Balachandar, and P. Fischer, (2005) Wall-induced forces on a rigid sphere at finite Re. *Journal of Fluid Mechanics* 536, 1-25.

[Zho98] X. Zhong, (1998). High-order finite-difference schemes for numerical simulation of hypersonic boundary-layer transition. *J. Comput. Phys.* Vol. 144, pp. 662–709.

[Zhou07] Q. Zhou, Z. Yao, F. He, and M., Shen, (2007). A new family of high-order compact upwind difference schemes with good spectral resolution. *J. Comput. Phys.* Vol. 227, pp. 1306–1339.