ADVANCED NUMERICAL DESIGN SOLVER: ANDSolver

WHITEPAPER

May 2010
Palix Technologies LLC
http://www.palixtech.com
1 Overview of CFD and ANDSolver

1.1 What is CFD?

Computational Fluid Dynamics (CFD) is the use of numerical methods and computers to simulate fluid and gas dynamics problems such as:

- Flow around an airplane
- Municipal water distribution system
- Performance of a race car
- Blood flow through the vascular system
- Shuttle or capsule re-entering the atmosphere

Although all of these problems, and many more not mentioned, cover a wide range of applications the basic underlying computations typically amount to solving systems of equations which express basic laws of physics such as conservation of mass, momentum (Newton’s second law), and energy. Depending on the specific nature of the flow some of these equations may be unnecessary (for example, in some problems conservation of mass and energy become redundant) and in other cases additional equations may be necessary. Practical codes will specialize and handle one or more similar types of computations because the generality to handle such different flows as the water in a pipe and the rarefied gas a space vehicle encounters would lead to a code that has poor performance for all classes of problems.

1.2 CFD Meshes

Because the Partial Differential Equations (PDEs) governing fluid flow do not usually have closed form solutions, it is typical to solve for the flow values at discrete points in space. This can be accomplished by using a mesh to fill the area or volume of interest. Meshes can be categorized as structured, unstructured, or hybrid meshes. In a structured mesh such as Figure 1 the connectivity between quadrilaterals in 2D or hexahedra in 3D is implicit. Neighboring cells and the vertex (or node) indices can be computed as simple offsets.

By contrast Figure 2 is a simple example of a 2D unstructured mesh made up of triangles. Two-dimensional unstructured meshes can contain a mix of triangle and/or quadrilateral elements. The mesh consists of a numbered set of vertices and a list of element connectivities, where the connectivity specifies the vertices that make up a given element. Note that an arbitrary structured mesh can be expressed as an unstructured mesh but not vice versa. There is no simple way of computing neighbors or element vertices as in the case of structured meshes.

A hybrid approach, such as multi-block structured or overset meshes, uses blocks of structured meshes connected in an unstructured fashion as shown by the simple 2D example of an overset mesh in Figure 3. The complexity of a hybrid mesh offers the advantages of mainly working with structured meshes but with the flexibility to connect them in an unstructured fashion to accommodate complex geometries like a Formula 1 car or an airplane in landing configuration with flaps and landing gear extended. Like structured meshes, a multi-block or overset mesh can be expressed or stored as a fully unstructured mesh.
There are advantages and disadvantages to using any of these types of meshes. As already mentioned structured meshes may not offer sufficient flexibility for complicated geometries and require the hybrid approach of multi-block structured or overset meshes. In many cases a fully unstructured mesh may be even easier and more flexible. Because of the need for human intervention and experience in the mesh generation process one often finds that the time to generate a mesh dominates the overall time to do a simulation (days or weeks to generate the mesh and only hours to actually compute the solution). But unstructured meshes aren’t without their disadvantages. Obtaining the same solution quality on an unstructured mesh may require significantly more elements than on a corresponding structured mesh. Then there is the issue of computational efficiency because of the irregular memory access patterns for an unstructured mesh. However, we believe that as the cost of human time continues to increase and the relative cost of computing continues to decrease the trend is clearly in favor of an unstructured mesh approach for CFD.
1.3 Finite Volume Method

Regardless of the type of mesh used, a CFD code basically amounts to keeping track of the amounts of conserved quantities and the transport, or flux, of these quantities. Figure 4 illustrates this using the simple example of a leaky bucket. If we consider an imaginary box around this bucket and keep track of the amount of water crossing the boundaries of the box (both in and out) that will correspond to the change in the amount of water in the bucket because mass is conserved. Now consider a second bucket below the first with its own imaginary box. The water leaking out of the first bucket crosses the boundary into the box around the second bucket and contributes to the change in the amount of water in the second bucket.

This use of these imaginary boxes is called the finite-volume approach and is the basis for many CFD codes (the others being the related finite difference and finite element methods). But instead of having two volumes there will be anywhere from hundreds of thousands to millions or billions of these small volumes depending on the problem. Figure 5 illustrates a typical unstructured mesh of tetrahedrals around an airplane configuration.

The goal of the CFD solver is then to compute the amount of the conserved quantities in each of the finite volumes, using an appropriate method for computing how much of the conserved quantities are crossing the boundaries between volumes (the so called fluxes). Because of the coupled nature of this often non-linear problem an iterative method is typically used. An iterative method essentially consists of making an estimate of the solution; computing the error in the estimate by computing the transport of the conserved quantities and from that the "lost" mass, energy, etc.; and then updating the estimated solution until no conserved quantities are being lost at which point the solution is said to be converged. Depending on the iterative method it may takes hundreds or even thousands of iterations to converge the solution which typically leads to the need for significant computational resources.

ANDSolver solves the so called compressible Euler equations on unstructured meshes of polyhedrals (as seen in Figure 5) using a finite volume approach. The Euler equations model a gas while neglecting viscosity, which is a resistance to shear stress and sometimes referred to as the "thickness" of the fluid. For example, honey has a high viscosity compared to water. All fluids and gases have some viscosity but for many problems this can be neglected and they can be modeled as ideal or inviscid. Compressible means that the density of the gas is allowed to vary. In high speed flow approaching and beyond the speed of sound compressibility effects are often the dominant factor in predicting lift and drag.
1.4 GPU Computing

What distinguishes ANDSolver is the approach it takes for compute resources. The continual increase in CPU speeds has ended due to power and thermal constraints with processors now having multiple cores. Although developing and running applications to use multiple cores is a well established technology in high performance computing for science and engineering applications, a newer trend towards using Graphics Processing Units (GPUs) for computations is emerging. GPUs were originally designed for computing the data associated with geometry and colors on computer screens and have been driven to ever increasing levels of performance: high screen resolutions (lots of pixels), high frame rates, and more realistic images.

The need for high performance and the parallel nature of this problem has led GPU designers to create current designs with hundreds of cores. In the past the power of these hundreds of cores was only available by writing graphics programs using special graphics instructions. However, research in this area has led manufacturers to make available tools for implementing more general software applications that use the GPU not for graphics but for applications such as CFD and others where lots of computations need to be done as fast as possible. In this heterogeneous computing model the GPU serves as a coprocessor or accelerator to the CPU. ANDSolver uses the NVIDIA CUDA technology to run simulations up to 10x faster than using all four cores on an Intel quad core i7 (“Nehalem”) processor.

Despite the tremendous performance gains possible with GPUs, relatively few CFD codes have yet to make use of them and those that do are often not showing compelling performance improvements. Sometimes the overall application runs only 2x faster (or even slower) than a modern multicore CPU. And in fact the performance per watt of power consumption for many of these solutions is actually decreasing due to the higher power consumption of a CPU plus GPU solution. Considering that over its lifetime a typical computer will often cost more to power and cool than the actual purchase price of the hardware, performance per watt is often a key metric which should
be considered in the total cost of ownership.

Why then is ANDSolver different from these other codes? ANDSolver is able to deliver compelling performance gains in pure application speed and in the performance per watt because of two main factors:

- Very careful and detailed algorithm design to accomodate the stringent requirements of the GPU memory system.

- Iterative algorithms that are well suited to the GPU, but which require much more code to be implemented for the GPU itself.

Why the emphasis on memory? Figure 6 shows the relative performance of processors and memory over the past few decades. CPUs have gone to great lengths to bridge the gap between processor and memory performance by introducing instruction and data caches, instruction level parallelism, etc. And although GPUs have taken a bit different approach in terms of hiding memory latency because of their specialization to inherently parallel problems, the fact remains that processor (compute) performance will continue to advance at a much greater rate than memory performance. If we extrapolate out, without any fundamental changes in memory, processors will become infinitely fast relative to memory and performance optimization will become solely an exercise in optimizing data movement.

![Figure 6: Processor - Memory Performance Gap [2]](image)

The GPU was of course originally designed for graphics and the majority of this computation involves computing the individual color for each pixel on the screen. If we think of each pixel as being like a quadrilateral, computing the pixel colors will be extremely similar to the computations on the structured meshes in Figures 1 and 3. There is a very regular, orderly data access pattern with neighbors easily computed by simple offsets of indices. However, the majority of commercial engineering simulation codes like CFD use some form of unstructured mesh with triangles in 2D or tetrahedra in 3D being very common. These types of meshes lead to an irregular and somewhat disorderly data access pattern which is not particularly well suited to the memory system of the GPU. But through a careful process of analyzing the relation between elements and vertices and taking advantage of the huge amount of available processor performance on a GPU, ANDSolver partitions and sorts the mesh in such a way that the data access pattern becomes much more regular and orderly. The preprocessing of the mesh connectivity is a one time step performed just before...
the main computation begins and takes a negligible amount of time but significantly increases the
performance of the overall solver.

Algorithm design for optimizing memory access is further complicated by the number of different memory spaces the developer must take into consideration. Unlike a CPU the memory accesses are under the full and manual control of the developer. Figure 7 shows the many memory spaces on the GPU which in turn is connected to the CPU memory (not shown). Different memory spaces have different scope and access characteristics: some are read only, some are optimized for particular access patterns, etc. Significant gains (or losses) in performance are possible depending on the choice of memory usage.

But a well suited memory access pattern alone is not sufficient to achieve the compelling performance gains of ANDSolver. Many codes use a very simple iterative method that typically includes linearization because it is relatively fast and easy to develop and maintain the code. While these simpler methods are easier for the developer they are not as well suited to the GPU due to the fact that they access a lot of memory but do a rather low amount of actual computation with the data. This metric is typically referred to as arithmetic intensity and GPUs are particularly well suited to algorithms with high arithmetic intensity. ANDSolver has comparatively high arithmetic intensity because significant developer time has gone into computing the full non-linear flux transport computation previously described in conjunction with Figure 4. In fact the GPU and supporting CPU code needed specifically for the GPU implementation of ANDSolver comprise almost 10,000 lines of source code. This seems like a somewhat obvious tradeoff to spend more developer time upfront so that the many users of the application can compute much faster but many code developers simply don’t have the technical experience to do so, or don’t want to. In fact, it’s somewhat to their advantage to not improve the performance of the code too much because it would tend to decrease the number of highly profitable software licenses customers would need to purchase to do their work!

1.5 Performance Results

Figure 8 shows the speedup for the single precision Runge-Kutta implementation of ANDSolver for both a first and second order spatial discretization. The speedup has been normalized with respect to the CPU performance using four cores of a Xeon E5520 processor. CPU parallelism is implemented using the Message Passing Interface (MPI). Although the CPU does scale quite well from 1 to 4 cores, the GPU implementation still shows a 6.5x to almost 10x speedup compared to the CPU.
ANDSolver also supports double precision floating point with both the Runge-Kutta and GM-RES solver. Again the results are shown in Figure 9 for first and second order spatial discretizations with speedup normalized to CPU performance on four cores. Although the speedup is not as high as the single precision implementation we believe it will be significantly improved with future GPU hardware featuring much higher double precision performance.

![Figure 8: Speedup Using Single Precision Floating Point](image1)

![Figure 9: Speedup Using Double Precision Floating Point](image2)

2 References

