CUDA Parallelization of a 2-D Compressible, Non-Hydrostatic Atmospheric Model

Matthew R. Norman

December 5, 2009

1 Motivation

1.1 Scaling Current Architectures

The field of computational fluid dynamics (CFD) is a very computationally demanding field. All things considered, the time discretization has the greatest effect on parallel scalability, and there are generally two options: implicit and explicit. In the atmosphere, because the maximum Mach number is generally about 1/2, we have the luxury of choosing either an implicit or explicit method. Without getting into the mathematics, the maximum stable time step of an explicit method is limited by the following condition: $\Delta t < C\Delta x/u_{max}$ where u_{max} is the maximum wave speed supported by the equations set and C is a constant that is usually close to one. Implicit methods do not suffer the same restriction. So the rule of thumb is that implicit methods require more computation per time step but take longer time steps. Clearly this is a trade-off.

When a geophysical scientist refers to scalability, he or she is usually referring to throughput scaling. In other words, if a model at 100 km grid spacing takes 5 simulated years per day (SYPD), that is 5 model years simulated per wallclock day, then running at 25km resolution should also take 5 SYPD after distributing the model over more processing elements (PEs). What makes this particularly difficult in CFD is that the algorithmic complexity is never purely spatial, and one cannot spread PEs across the time domain. So throughput does not scale (in the Gustafson's scaling sense) asymptotically with spatial refinement (at least with current algorithms), and it likely never will.

Implicit methods rely on solving large (usually pretty sparse) linear systems iteratively. Usually, a Krylov-type method is used such as the General Minimized Residual (GMRes) method because it handles any general matrix without assumptions on eigenvalue spread or symmetry. For sparse matrices, the algorithmic complexity is generally the same order as explicit methods. However, as the number of grid points grows, the condition number of the matrix also usually grows. Therefore, more iterations are required to converge to some desired relative residual. Also, as resolution is increased, the multi-scale nature of the flow is also greatly increased which makes effective pre-conditioning more difficult (requiring even more iterations to converge). GMRes can only perform but so many iterations before the algorithmic complexity grows to order n^2 (where n is the total number of model grid cells). To avoid this, the method is "restarted" which is a costly procedure. Therefore, the amount of work per time step is asymptotically super-linear as the number of grid cells increases. This makes the method hard to scale.

Also, implicit methods require global communication patterns which are expensive in parallel compared to explicit methods. Assuming the time step can be held constant as the spatial grid is refined and assuming the work per time step is linearly proportional to the number of grid cells, a factor of m refinement in space requires a factor of m^3 more computation (in 3 spatial dimensions). Therefore, if a grid spacing of 300 km runs effectively on 256 processors, 150 km grid spacing will require 2048 processors assuming perfect speed-up. However, due to the heavy global communication requirements of implicit methods, this scaling will be highly sub-linear, even for Gustafson scaling. Also, for physical time scale reasons, the time step

cannot be held constant asymptotically. As shown before, the work per time step is not proportional to the number of grid cells but grows superlinearly.

One may think, then, that for that atmosphere, explicit methods would scale much better. However, they are limited by the constraint that the maximum stable time step is proportional to the grid spacing. Because of the local communication requirements in parallel, an intelligent algorithm should scale in space very well (assuming the data layout lines up with the network topology well). However, because of the time step constraints, a factor of m refinement tin space now requires a factor of m^4 more computation (in 3 spatial dimensions). Therefore, though explicit methods scale well spatially, they can never scale asymptotically in time (and therefore throughput). One can alleviate the time step constraint some, and the method I am using in this study is intended for eventual extension to larger time steps. This incurs some extra communication, but the larger time step would outweigh that in the overall throughput. However, this is not an asymptotic solution but a temporary boost in throughput.

The conclusion asymptotically is, therefore, dismal because neither method scales in throughput as spatial refinements are made. Multiscale methods such as adaptive mesh refinement (AMR) offer some hope, but they also have aspects that are very difficult to scale. For instance, in explicit AMR, a factor of 2 refinement will use a factor of 2 smaller time step and is iterated while larger cells are held constant. The processors allocated to larger cells do nothing while the smaller cells are iterated which leads to load balancing difficulties. Remapping the processors each time step requires global communication, and the only option for a single model time step is to go implicit which was discussed above. I think that ultimately, if atmospheric models are to scale well into and past exascale computing, a breakthrough must take place in implicit methods such that computational work scales linearly with the total number of grid cells, and somehow communication costs must be reduced significantly. Otherwise, we'll simply need some new tricks to see feasible throughput in refined climate runs.

However, considering the "required" climate throughput of 5 SYPD (I don't know how rigid this truly is), the constant of algorithmic complexity can be manipulated effectively for explicit methods over very large numbers of processors for the time being. For instance, there is a very low communication method called spectral element (an approximation to finite element) which is giving promising results at impressively high resolutions near 14km using $\mathcal{O}\left(10^5\right)$ processors. Strong scaling can be obtained via more efficient algorithms with lower communication and computational requirements. It can also be done with hardware improvements. Also, I/O, storage capacity, and data analysis constraints will likely dominate before the explicit computational throughput barrier is reached. A particularly enlightening presentation[1] by Dr. Rich Loft at the National Center for Atmospheric Research shows some insights regarding the human brain as defining a level of efficiency for a parallel (petascale) computer. My main doctoral research motivation here at NCSU is to develop an explicit algorithm with a competitive time step and minimal parallel communication requirements.

1.2 GPGPUs as Alternative Architectures

Viewing the limitations of current architectures, I had considerable interest in testing out other architectures such as General Purpose Graphical Processing Units (GPGPUs). Given that they are designed for flop-intensive algorithms with low communication / synchronization requirements, there is great competitive potential. This paper describes the GPGPU parallelization of a serial 2-D, non-hydrostatic, compressible atmospheric model using Nvidia's CUDA language. For a single GPU, after initialization, there is no need for any DMA transfers between device and host because all computation can be done on the device via kernels. Therefore, the synchronization between time steps is automatic and very cheap. However, for MPI domain decomposition across multiple GPUs, the boundaries must be exchanged each time step which can be quite constraining for high latency and/or low bandwidth networks.

The goal of the project was to achieve an overall 64x speed-up over a well-tuned CPU application using multiple GPUs connected via MPI. The reason for a 64x speed-up was that a factor of 4 increase in resolution requires a factor of $4^3 = 64$ speed-up (halving in two spatial dimensions and in time) in order to complete

[1] `http://www.cgd.ucar.edu/cms/pel/asp2008/6-Loft-Petascale.pdf`

the simulation in the same amount of time. I compute speed-up indirectly because I don't have time to wait days for a CPU job to finish. Therefore, I compare the wallclock time of a given CPU experiment with the wallclock time of a larger GPU experiment by extrapolating the CPU wallclock time by the known increase in compute requirements. If a CPU job takes T_{CPU} seconds to complete and a GPU job refined spatially by a factor of r takes T_{GPU} seconds to complete, the speed-up is computed as $S = r^3 T_{CPU}/T_{GPU}$. This is probably optimistic for the CPU because caching issues would seemingly become worse with larger problem size. Also, periodic output is also included in the wall clock times which also give preference to the CPU because of smaller file sizes. Therefore, estimates of speed-up are conservative.

2 Model Description

The model in this study is a 2-D non-hydrostatic, inviscid, fully compressible atmospheric model. Mathematically, it is a numerical approximation to a set of four conservation laws (a special type of partial differential equation) conserving mass, momentum and entropy in two dimensions with a gravity source term. The approximation is performed with the finite volume method in which cell means are updated in increments of time called time steps based on interface fluxes between cells. Computing the fluxes accounts for nearly all of the computational effort in the model.

Fluxes are computed with a new method I have developed during my doctorate research called the characteristics-based flux-form semi-Lagrangian method. The method has some advantages over conventional finite volume methods for meteorology in that only one exchange of boundary information is needed per time step (rather than multiple swaps per time step). The mathematical details are relatively unimportant computationally except to say that there is no if/then logic in the implementation. They WENO (Weighted Essentially Non-Oscillatory) interpolation is fairly compute intensive, and much of this is hidden by the GPU acceleration.

The experiment I had originally chosen for this study did not visually show the advantages of added resolution terribly well. Therefore, I have created a different experiment that is more visually revealing. In a thermally neutral and initially hydrostatic atmosphere, a cold bubble is placed and the top of the domain and a warm bubble is placed at the bottom. As the simulation progresses, they are buoyantly propelled towards one another, and after collision, turbulence ensues. It is the increasing resolution of smaller and smaller scale rotors that shows up in the temperature plots. Regardless, the actual computations have no bearing on the required flops.

2.1 Porting to C

The model was originally implemented in Fortran 90, and some modifications had to be made for an effective C implementation. First, the multiple dimensional arrays were lined into single dimensional arrays for easier manipulation on the GPUs, and C macro functions were used for easier indexing. Also, the order of loop nesting was reversed for row major storage. I took out many of the model parameters and turned them into macro defined constants for the sake of efficiency. For sake of a fair comparison between CPU and GPU implementations, I included OpenMP parallel for pragmas before the outer loop of each computationally intensive section, and the speed-up was very close to linear. I placed a macro switch between single and double precision.

2.2 Porting to CUDA

With the code ported to C, the CUDA implementation was relatively quick. First, before the main computational loop, space is allocated on the GPU device for state variables, fluxes, and basic states. Memory requirement for $m \times n$ cells is $8mn + 4m + 6n + 4$. After data initialization, it is transferred via DMA to the device. Six subroutines were then converted into CUDA kernels, and device memory pointers to state variables, fluxes, and basic states are passed to them. They are: `flux_x`, `update_x`, `flux_z`, `update_z`, `source`, and `boundaries`. Most of the computation occurs in the flux routines, but the others must be

turned into kernels even if they didn't give speed-ups because the data must remain resident in the device. Repetitive DMA transfers would be detrimental to overall efficiency.

To actually transform the subroutines into kernels, the for loops were transformed as follows:

```
for (i = GS; i < NXC+GS+1; i++) {
    for (j = GS; j < NZC+GS; j++) {
```

was converted to

```
ii = blockIdx.x*blockDim.x+threadIdx.x;
jj = blockIdx.y*blockDim.y+threadIdx.y;
for (i = GS+ii; i < NXC+GS+1; i+=gridDim.x*blockDim.x) {
    for (j = GS+jj; j < NZC+GS; j+=gridDim.y*blockDim.y) {
```

I also placed unroll pragmas before any of the loops with constant loop size which increased the speed-up. On the GTX 280 devices, there are 16K registers available per block. I tested limiting the registers per thread to 128 to support 128 threads per block, and I tested a maximum of 64 registers per thread to support 256 threads per block. It turns out that even though limiting to 64 registers reduces the single-thread speed and increases local memory accesses (which are slow and uncached), having 256 threads is faster. This could only be because of a more effective hiding of slow memory accesses with increased computation.

2.3 CUDA + MPI

Once the GPU kernels were successfully implemented, the next step was to introduce MPI communication on the host side to tether multiple GPUs for further speed-up. The modifications were only in three places. First, the initial setup was altered to divide and scatter the data. Next, the output routine was altered to DMA the information to the host and then gather the data before output. Finally, the `boundaries` routine was turned back into a regular C function, and internal boundaries are passed with asynchronous sends and receives after DMA to the host. After the boundaries finish, the data is passed via DMA back to the device. Currently, the domain decomposition is implemented to split up in the x direction only. Though this is certainly not the most efficient implementation, it does make the programming very easy for the time being. I placed some wall timers to give a simple profiling of the communication, computation, and output costs.

3 Experimental Setup

To compute the speed-up and other parameters for the GPU and GPU+MPI codes over the CPU code, they are actually run with different numbers of grid cells. The CPU code is run at the standard resolution of 160x80 cells. The GPU is coded to run with over 100,000 threads per kernel call, but the standard resolution only has about 13,000 cells. Therefore, I had to run with 640x320 cells to get enough cells to fill the GPU device. When running with MPI, I ran up to 8 GPUs at a time. In order to successfully fill the devices, I ran the code with 2560x1280 cells which is about 3.3 million cells. The CPU used was an Intel(R) Core(TM)2 CPU 4400, and an Nvidia GTX 280 was the GPU used.

To actually compute the speed-up with different work loads, I use the fact that the work load is proportional to the number of cells and the time step. With $m \times n$ cells at a time step, Δt, the computational cost is $C_0 = \kappa mn\,(T/\Delta t)$ where T is the total simulation time. To refine the grid spacing by a factor of r, the new computational cost is $C_r = r^3 \kappa mn\,(T/\Delta t) = r^3 C_0$ because not only must both dimensions be refined, but the time step must be refined by the same factor. Therefore, the 640x320 cell runs would take 64 times longer than the 160x80 cell runs, and the 2560x1280 cell runs would take 4096 times longer. Therefore, comparing a run refined by a factor of r against the standard resolution on a CPU, the speed-up is computed as $S = r^3 w_{CPU}/w_{GPU}$ where w is the wall time.

Description	Precision	# Cells	Walltime (s)	Speed-up	Comm/Comp	Efficiency
CPU	double	160x80	200	—	—	—
1 GPU	double	640x320	737	17.2	—	—
1 GPU	single	640x320	642	19.8	—	—
2 GPUs	double	2560x1280	29,750	27.5	0.098	0.80
2 GPUs	single	2560x1280	19,180	42.6	0.066	1.08
4 GPUs	double	2560x1280	19,005	42.9	0.38	0.62
4 GPUs	single	2560x1280	11,750	69.4	0.27	0.88
8 GPUs	double	2560x1280	12,659	64.1	0.81	0.47
8 GPUs	single	2560x1280	7,341	110.7	0.52	0.70

Table 1: Simulation results

4 Results

4.1 Parallel Metrics and Discussion

4.1.1 Single GPU

Wall times, speed-ups, communication/computation ratios (for MPI runs), and efficiencies (for MPI runs) are given in the Table 1. To begin interpreting these results, it is most obvious that local and global memory accesses are not being handled efficiently. The single precision time should be half of the double precision if not even less. Clearly, the computation is not successfully hiding memory latencies in single precision. The fact that the double precision barely takes a performance hit is showing that the increased amount of work is masking memory latency. This could indicate that for double precision computations, memory optimizations may be less important.

Note that the CPU costs $113 in a 1,000 bulk unit price, and the GPU runs at about $255. For equal double precision performance, you would need 17 CPUs (with the extremely generous assumption of perfect parallel efficiency). Performance/cost ratio improves by at least a factor of 7.5 and is more likely at least a factor of 10 (assumption of 75% efficiency) with real world parallel overheads. That's a powerful incentive to consider GPUs in the real world. Another benefit of GPUs is that less networking is involved because fewer need to be used to get a desired speed-up. $17 \cdot 8 = 136$ CPUs is a lot more to network than just 8 GPUs, and one could probably use a faster and more expensive networking (such as hypercube) with the GPUs than for CPUs because there are fewer of them to link. This would lead to likely much better GPU efficiency when networked well.

4.1.2 GPU+MPI

Now considering the MPI runs, it appears that the main limitation is network bandwidth. Consider that the computations in double precision take 15% longer than in single precision, and yet the single precision communication / computation ratio (CCR) is 30% less than double precision. The only possible explanation for this is that the single precision simulations are transferring only half of the data. Therefore, an increase in network bandwidth would go a very long way in reducing the communication times. The efficiencies really do not look very good, especially in double precision with less than half for 8 GPUs. I did get the 64x speed-up finally with 8 GPUs in double precision. Notice that for single precision and 2 GPUs, the efficiency shows superscaling. The only possibility I can think of is that the larger problem size and more work per thread hid more of the memory latency. It also didn't hurt that the two nodes communication were adjacent (assuming hostname numbers indicate physical proximity).

The fact that the CCR is less than unity for each of the MPI runs means there is good potential to hide the latency with overlap. One could simply run the internal domain (neglecting the boundaries which depend on communicated data) of a node while communicating data and then run the outer domain border after

communication. This would probably completely hide the communication, and the only penalty would be on the GPU itself because there are fewer cells to spread across the device on the border kernel launches. This would be more difficult to do as more GPUs are used and the number of cells spread across each becomes smaller (in a strong scaling sense).

4.2 How many GPUs should I use for 2560x1280?

To simplify things by removing the communication overhead, with the CCR given, communication takes a proportion of $CCR/(CCR+1)$ of the total wall time. Therefore, the speed-up without communication overhead will be $S^{'} = S/M_C$ where $M_C = 1 - CCR/(CCR+1)$ could be thought of as a communication penalty multiplier. Going from one to eight GPUs without communication has a speed-up of 6.76. The parallel portion (with communication extracted) of the algorithm can be pulled out of Amdahl's law with $p = [n/(n-1)] \cdot [(s-1)/s]$. For eight processors, this is $p = 0.9738$. Using this to obtain $S^{'}$ and dividing by p to get the efficiency, we can get an "ideal" picture of maximum speed-up. With 16, 32, and 64 GPUs, I can get idealized speed-ups of 72%, 55%, and 38%, respectively. So even with a very good networking system, it wouldn't be wise to use more than 16 or 32 GPUs at this resolution before I need to resume Gustafson's scaling to keep the efficiency constant.

4.3 Translation to Real Life

Now it's time for some back of the envelope calculations. A 1 degree grid spacing climate model with 50 vertical levels will have 360x180x50=3.24 million cells. The 2560x1280 resolution 2-D run has 3.28 million cells at a grid spacing of 7.8125 meters and a time step of 0.015625 seconds, and it computes 64,000 time steps in a wall time of 12,659 seconds. The ratio climate model time step to 2-D model time step is the same as the ratio of grid spacings (which is roughly 14,008). So the climate model time step would be 222 seconds, and 64,000 time steps would give a simulation of 164.4 model days. The simulated years per day (SYPD) throughput would, then, be 3.07 SYPD. Keep in mind also that a 3-D model will have 50% more work to do in a time step because of an added dimension of fluxes, making the throughput 2.05 SYPD in double precision. In single precision, it would be 3.27 SYPD. Assuming the network bandwidth were vastly improved to reduce the CCR to 0.05, then I would have a double precision throughput of 3.53 SYPD and a single precision throughput of 4.7 SYPD.

These are, of course, back of the envelope computations, and this is only for the dynamics and doesn't take into account the physics packages which would be difficult to port to GPUs because of extensive use of if/then logic in some of them. But it seems that efficiency would have to be sacrificed in order to get the necessary throughput of 5 SYPD. It is also true that my MPI parallelization is highly inefficient because it decomposes the domain only in one dimension. The CCR is always lower when domain decomposition is performed in multiple dimensions rather than one. Also, there are other optimizations such as extending the algorithm up to a Courant number of 2 (which I hope to do soon), and this would increase the throughput So there does seem to be some good hope in GPUs. The thing to be weary of is that once I begin Gustafson's scaling, the throughput will automatically cut in half because of time stepping constraints. So realistically, I would like to have a solid throughput of around 20 SYPD at 1 degree before problem scaling so that simulating at around 28km would still give the required throughput.

All of this being said, to have a 1 degree climate model running on on 16 or 32 GPUs is very economically tempting because the hardware itself is going to be less than $10K. Also, there are many cards coming out that are even better than the GTX 280 cards, and these may push the envelope on speed-up. I have also read on message boards (no formal confirmation yet) that ATI graphics cards are generally 4x better than Nvidia for double precision flop rates. If this is true, then openCL may be the better option.

Of course, I have somewhat treated single precision as if it were not an option. It shows some strong asymmetries for solutions that should be symmetric, and the placement of some features is very poor. This gives me somehesitance to use single precision in a real model run. It would have to be demonstrated in practice whether or not these asymmetries will adversely affect the overall climate simulation since there are

many other errors in the model that may dominate. If single precision is possible, then memory efficiency will need to be addressed because a boost in single-GPU flops will increase the throughput quite a bit.

5 Future Work

In the future, I would like to try out openCL on ATI graphics cards to see if the double precision support really is superior. Also, I need to change the domain decomposition into a 2-D one to further reduce the CCR. I am currently using fairly simple asynchronous sends and receives for MPI communication of the boundaries, and it would be interesting to see if there is a better option for that. Also, since the computation time is larger than communication time, it would be worthwhile to mask the communication times with some overlap from each node's internal domain which doesn't depend on the boundaries. The algorithm I used for fluxes is intended to extend the time step to any arbitrary (within reason of course) Courant number to increase throughput. I need to finish testing this, and if it is possible, it would be a good implementation to include in the CUDA version of the code. Parallel I/O is a must at some point because eventually, the host memory will overflow if I/O is done serially. Also, the output is one of the serializing aspects that keeps the parallel fraction low in Amdahl's law. The parallel NetCDF library is one option for this. Finally, I would like to see what kind of memory optimizations can be performed in the kernels to increase the flops. These, together, should act to drastically increase the throughput.