GPU Acceleration of the Advanced Regional Prediction System (ARPS)

Benjamin A. Whetstone*†
Computer Science
Coastal Carolina University
P.O. Box 261954
Conway, SC 29528-6054
bawhetst@coastal.edu

Varavut Limpasuvan
Chemistry and Physics
Coastal Carolina University
var@coastal.edu

D. Brian Larkins
Computer Science
Coastal Carolina University
blarkins@coastal.edu

ABSTRACT

A recent breakthrough in high-performance computing is the application of general-purpose graphics processing units (GPGPUs) to scientific computing. This paper presents a case study of a project to reformulate mathematical computations within a weather model, the Advanced Regional Prediction System (ARPS), to work with GPGPU hardware. This hardware typically consists of hundreds of simple processors, compared to the conventional central processing units (CPUs) with fewer than eight processors (as found in most computers). While GPGPUs are extremely powerful, their usage requires specialized programming to achieve their full potential. As a forecasting tool, ARPS is capable of producing very high-resolution weather simulations. Using ARPS to simulate detailed atmospheric disturbances necessitates the use of large-scale distributed-memory parallel computing clusters. The adaptation of a critical numerical kernel within ARPS for GPGPUs resulted in a six-fold speedup over the CPU version. These optimizations dramatically reduce simulation time, thereby leading to faster weather predictions that may benefit society as well as the research community.

Keywords

GPU, weather forecasting, finite differences method, ARPS

1. INTRODUCTION

The devastating impact of violent weather systems has motivated the development of more advanced and comprehensive numerical weather prediction models. Improvements made to these models and the computational systems that run them provide disaster planners with better and faster predictions that can potentially save lives. Weather modeling relies heavily on iterative, computationally intensive differential equation solvers that are amenable to concurrent execution, such as multi-grid methods, finite element analysis, and finite difference methods (FDM) [1, 8, 9].

The application of these techniques necessitates the use of large-scale distributed-memory parallel computing clusters. Using weather models with these systems has been the accepted practice and is well understood. Recently, General-Purpose Graphical Processing Units (GPGPUs) have provided an opportunity for significant performance improvements. However, their usage requires specialized programming and a deep architectural understanding to realize their full potential. Several solver-based techniques used in parallel weather modeling have been successfully applied to work with GPGPU hardware [2, 3, 9].

Created at the University of Oklahoma, the Advanced Regional Prediction System (ARPS) has been developed as a parallel stormscale atmospheric modeling framework since 1995 [7]. ARPS relies on a set of finite-difference method kernels to provide realtime data analysis and assimilation. ARPS has been adapted to work on large distributed-memory systems and its performance has been studied and optimized extensively for these architectures [10]. To be of practical usefulness, a balance must exist between the model's ability to produce high-resolution comprehensive simulations against the run-time (wall clock) of the simulation process. Detailed and accurate predictions that are generated too late to be of practical use are no better than fast predictions of limited realism. Therefore, any improvement in raw program performance has the impact of shifting this balance, allowing models to be run faster, at finer scale, and with better accuracy – all within a reasonable computational timeframe.

In this paper, we describe the application of the numerical finite-difference method kernels used within the ARPS framework to utilize GPGPU hardware. Our approach is based on the insights that (1) the prior work studying the performance bottlenecks within ARPS would be useful in determining opportunities for GPGPU acceleration and (2) that the numerical kernels within ARPS were viable candidates for the GPU platform.

This work makes the following contributions: First, we detail the analysis of the ARPS kernels with respect to issues impacting GPGPU system architecture. Second, we describe the implementation of GPGPU kernels and discuss

*Corresponding author.
†Undergraduate Student.

ACM SE '14, March 28–29, 2014, Kennesaw, GA, USA.

http://dx.doi.org/10.1145/2638404.2735458

$$\frac{\partial(\bar{\rho} * u)}{\partial t} = -[ADV(u)] - \frac{\partial(p' - \alpha Div)}{\partial x} + (\bar{\rho} f v - \bar{\rho}\tilde{f} w) + D_u,$$

$$\frac{\partial(\bar{\rho} * v)}{\partial t} = -[ADV(v)] - \frac{\partial(p' - \alpha Div)}{\partial y} + \bar{\rho} f u + D_v,$$

$$\frac{\partial(\bar{\rho} * w)}{\partial t} = -[ADV(w)] - \frac{\partial(p' - \alpha Div)}{\partial z} + \bar{\rho} B + \bar{\rho}\tilde{f} u + D_w,$$

$$\frac{\partial(\bar{\rho} * \theta')}{\partial t} = -[ADV(\theta')] - \bar{\rho} w \frac{\partial \bar{\theta}'}{\partial z} + D_\theta + S_\theta,$$

$$\frac{\partial p'}{\partial t} = -\left[u\frac{\partial(p')}{\partial x} + v\frac{\partial(p')}{\partial y} + w\frac{\partial(p')}{\partial z}\right] + \bar{\rho} g w - \bar{p} C_s^2 \left[\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}\right],$$

$$\frac{\partial(\bar{\rho} * q_\psi)}{\partial t} = -[ADV(q_\psi)] + \frac{\partial(\bar{\rho} v_q q_\psi)}{\partial z} + D_\theta + S_\theta,$$

where

$$Div = \left[\frac{\partial \bar{\rho} u}{\partial x} + \frac{\partial \bar{\rho} v}{\partial y} + \frac{\partial \bar{\rho} w}{\partial z}\right]$$

and

$$ADV() = \bar{\rho} u \frac{\partial()}{\partial x} + \bar{\rho} v \frac{\partial()}{\partial y} + \bar{\rho} w \frac{\partial()}{\partial z}.$$

Figure 1: Predictive Atmospheric Parameter Equations used within ARPS

the mitigation of challenges (e.g. low temporal locality) that arose. Lastly, we provide an experimental validation of our approach resulting in a speedup of just over six times the CPU-only implementation.

2. BACKGROUND

2.1 ARPS Climate Modeling System

The Advanced Regional Prediction System (ARPS) is a numerical modeling suite that has been developed by the Center for Analysis and Prediction of Storms at the University of Oklahoma. Currently in its fifth major revision, ARPS is a widely-used simulation tool for both basic scientific research as well as operational numerical weather prediction [8]. ARPS was originally implemented in FORTRAN 77 and has since been updated to use FORTRAN 90. It has been extensively adapted to operate on distributed-memory parallel computers using both MPI and PVM.

The ARPS model uses the equations of state for an atmosphere containing water constituents, shown in Figure 1. These equations provide values for u, v, w, θ', p', and q_ψ which correspond, respectively, to the x, y, and z components of the velocity, potential temperature and pressure perturbations, and the six categories of water constituents (water vapor, cloud water, rainwater, cloud ice, snow and hail). These predictive equations are solved using finite-difference methods within the simulation as detailed in [8].

The equations in Fig. 1 are solved using finite difference methods as applied to an offset Arakawa C-grid using a mode-splitting time integration, comprising both large timestep and small timestep integrations (to minimize sound waves). The equations themselves are represented in a curvilinear coordinate system that is projected onto a plane tangent to or intersecting the earth's surface (e.g. Lambert and Mercator projection). Solving for the desired parameters using these equations consumes a substantial amount of the runtime for a typical ARPS simulation.

The solutions for the key predictor parameters are derived using the above equations in conjunction with sets of large two- and three-dimensional arrays corresponding to the x, y, and z dimensions of the grid size (resolution and scale), specified by the user in the simulation domain configuration. The large number of equations, in conjunction with the quantity of terms in each equation leads to unique challenges when considering GPU-based acceleration.

2.2 Related Work

The applicability of GPGPU hardware to scientific computing problems has been well studied in the literature. Previous work has shown successful application of GPU acceleration to common numerical techniques such as multigrid, FEM, and FDM [2, 3, 9].

Within weather models, GPU acceleration has been extensively applied to the prominent modeling tool Weather and Research Forecast (WRF) as described in [5, 6]. While GPU acceleration has never been applied to the ARPS modeler, it has been the subject of much analysis and parallel optimization for use on shared-memory SMP and distributed-memory clusters, as described in [7, 8].

3. ANALYSIS

3.1 ARPS Analysis

The ARPS system is a very mature, highly optimized scientific simulation code, comprising nearly half of a million lines of FORTRAN 77 and FORTRAN 90 code for the core program. The core numerical kernels within ARPS have been optimized for both efficient sequential and parallel execution. While these previous efforts make it difficult to find new performance improvements, prior work has focused on optimizing for traditional CPU-based systems. These activities helped us identify the performance hotspots within ARPS in conjunction with our own profiling and timing analysis.

ARPS utilizes both an atmospheric model (relying on the differential equations listed above) as well as a two-layer soil model and several other model components computed during the modeling process. Preliminary performance analysis showed that of the entire ARPS simulation pipeline, the largest single contributor to overall run-time performance was the differential equation solver for the atmospheric parameters, at approximately 25% of total run-time.

The computational structure of the FDM solver code consists of a sequence of twenty-five distinct loops operating on a number of multidimensional arrays. In total, there are 31 distinct similarly-sized three-dimensional arrays defined by the grid size specified for simulation. These arrays are replicated over every MPI process participating in the computation. A typical resolution used is 5-20 km in the horizontal direction and 0.5km in the vertical. There are another nine two-dimensional arrays and a number of scalar values used throughout the process. In practice, real-world problem sizes are roughly 300×300 grid points in the x and y dimensions and 100 points in the z dimension. Research runs typically use larger problem sizes of 1000 points in each

```
tema = (dtsml1*tacoef)**2 * wpprt*g/cpdcv
temb = (dtsml1*tacoef)**2 * dzinv

DO k=3,nz-2
  DO j=1,ny-1
    DO i=1,nx-1
      pk = tem4(i,j,k)*tema*pbzi(i,j,k)
      nk = tem4(i,j,k)*temb*rstwi(i,j,k)

      rhostru(i,j,k)= (-nk+pk) * (tem1(i,j,k-1)
                      + tem2(i,j,k-1))
      rhostrw(i,j,k)= ( nk+pk)*(tem1(i,j,k   )
                      - tem2(i,j,k))
      rhostrv(i,j,k)= 1
          +nk*(tem1(i,j,k)+tem2(i,j,k)
              -tem1(i,j,k-1)+tem2(i,j,k-1))
          +pk*(tem1(i,j,k)+tem2(i,j,k)
              +tem1(i,j,k-1)-tem2(i,j,k-1))
    END DO
  END DO
END DO
```

Listing 1: Partial ARPS 3D Solver Kernel ($loop_{14}$)

of the x, y, and z domains which results in a single array requiring 3.8 GB for a single-precision realization and twice that for double-precision data.

Of the loops used in the solver, one-quarter of the time spent inside the solver code was within a single kernel loop block, $loop_{14}$, which is shown in Listing 1. As the other loops within the solver code follow a very similar structure, we chose to isolate this kernel as the principal candidate for GPGPU acceleration. The remainder of this paper will focus on the optimization of this specific kernel as these results should be directly applicable to both the remaining loops within the solver as well as a number of other computationally intensive kernels throughout ARPS.

3.2 Data Locality

The innermost loop of the kernel of interest ($loop_{14}$), is comprised of several computations involving eleven distinct array elements in eight arrays for each step. For large runs (900^3 grid points), the total memory accessed during this step is approximately 21.7 GB in each MPI process. The FORTRAN code for this loop is shown in Listing 1.

An examination of the loop code reveals that each iteration of the loop consists of eight multiplication operations and twelve add/subtract operations (excluding array index calculations). Further analysis reveals that of the nine arrays involved in this computation, seven have no temporal locality whatsoever. The two remaining arrays (`tem1` and `tem2`) have somewhat better locality, in that each element is accessed six times (three as $\texttt{tem}_{ijk}$ and three as $\texttt{tem}_{ijk-1}$) over the execution of the nested loops.

The relatively small amount of computation and large number of array elements needed for each loop iteration lead to very poor temporal locality. This mismatch in computation versus memory communication is the principal challenge in accelerating performance for the ARPS kernel and is exacerbated by the need to transfer the array data to and from the GPU for computation. The remaining kernels within the solver code also exhibit similarly poor locality properties. The loop structure of $loop_{14}$ exhibits good spatial locality, but it must be handled carefully within the context of the GPGPU architecture as is discussed below.

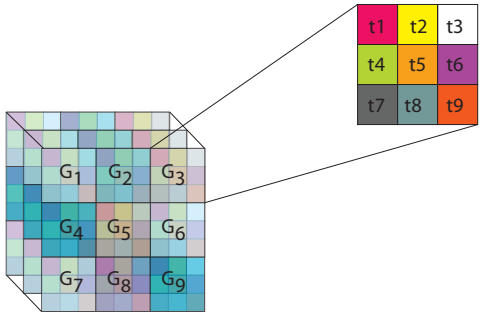

Figure 2: Initial data decomposition and thread mapping for ARPS FDM kernel.

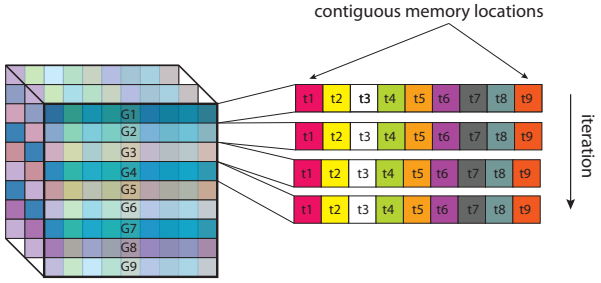

Figure 3: Coalesced memory transfer layout and thread mapping.

4. IMPLEMENTATION

Adapting loop-based numerical kernels for execution on GPU hardware requires concern for many architectural nuances and constraints not needed for conventional CPU-based programming. GPUs are typically connected via the PCIe bus and require explicit memory copy operations prior to the concurrent execution of a numerical kernel. Obtaining the highest performance from the GPU hardware entails balancing multiple dimensions of constraints, including the GPU memory hierarchy, register usage, and data access patterns [4]. In this section we describe several transformations made to port the ARPS kernel to a version that is able to extract meaningful benefit from GPU acceleration. To accomplish this, the FORTRAN-based ARPS kernel needed to be ported to CUDA, the programming model for executing code on NVIDIA GPGPUs.

4.1 Computation Mapping

Several factors must be taken into consideration when transforming a sequential loop that executes on the CPU into a parallel loop that executes on the GPU. First, the sequential nested loop structure must be mapped to the GPU threads that will concurrently execute the kernel code. Second, the data access pattern must be adapted to work within the constraints of the memory limitations and transfer conventions used on the GPU.

CUDA kernel tasks are organized in terms of *threads* and *blocks*, with a thread being the basic logical unit of computation and a block being a collection of threads. Kernel

invocation requires the specification of parameters for block size (threads per block) and a total number of blocks to be allocated. The way that CUDA threads are scheduled and are executed on the GPU are heavily dependent on assigning appropriate values for the block mapping of threads and can have a dramatic impact on performance [4].

The parallel execution of a GPU kernel happens in a hierarchical manner, where the iteration space of the computation may be mapped onto multidimensional blocks each consisting of a fixed number of threads. The initial implementation mapped each block to a single two-dimensional slice of the arrays, which would be processed by multiple groups of threads working on each block. An example of this initial approach is shown in Figure 2, where each plane of the array is processed by a 3×3 group of threads.

A problem that arises with the ARPS FDM kernels is that the arrays for single-process CPU implementation are too large to reside within the GPU global memory. If the data set that is being processed within the loop is greater than the available memory, the loops are tiled along the x-dimension in order to fit on the GPU. Tiling introduces some additional complexity in the kernel, both in additional operations to compute the tiled indices and in handling the partially filled final tile(s).

An additional constraint on CUDA-based GPU kernels is that DMA memory transfers are not legal from pageable memory. As a result, pinned memory is required for high-performance memory transfers – especially important given the memory-bound nature of the ARPS kernels. The use of pinned memory within ARPS can be problematic given the large arrays involved when the simulation domain is large. If the arrays are too large to be allocated into non-pageable memory blocks, then they are simply allocated from main memory and copied into pinned buffers prior to transfer to the GPU device.

4.2 Optimization

Our initial results with the tiled GPU version described above demonstrated only a modest speedup compared to the CPU implementation. Further analysis revealed several opportunities to improve our implementation and achieve higher performance.

4.2.1 Maximizing Resource Usage

When launched, each block of threads is mapped onto streaming multiprocessors (SM). Each SM consists of a variable number of streaming processors (SP) or CUDA cores, depending on the particular GPU in use. Each core on the SP is responsible for the execution of a single thread. For architectural reasons, threads within a block are scheduled in groups of 32 threads called *warps*. Specifying a block size that is not a multiple of warp size results in idle SP cores for any partially filled warps. There are maximum limits for each of the number of blocks, warps, and threads per SM.

There is also an architecture-specified maximum number of warps that can be scheduled to an SM. For recent NVIDIA GPUs (CUDA compute capability 3.5+), the limit is 64 warps per SM. If the number of warps scheduled on a single SM is below this limit, then it will not be fully utilized. For example, if 32 warps are in use on a single SM, then it is operating at only half its full capacity or is at 50% occupancy. The maximum number of threads per SM is $max_{SM} = max\,warps \times 32\ threads/warp$. There are also

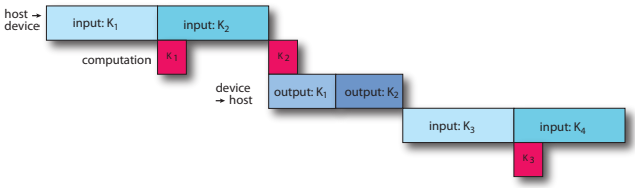

Figure 4: Computation and communication overlap for GPU kernels.

limitations on the number of blocks per SM as well, all leading to a multidimensional constraint problem in determining how to maximize GPU occupancy. In order to achieve maximum occupancy within an SM, the threads per block parameter must be set so that the threads per SM limit is not exceeded, the blocks per SM limit is not exceeded, and the warps per SM limit is matched.

Maximum occupancy may be achieved at multiple block sizes. For example, block sizes of 128, 256, 512, and 1024 all allow for a total of 2048 threads per SM, which is the maximum amount possible. Therefore, additional factors other than SM occupancy must be considered when attempting to find the optimal block size for a given algorithm. For the ARPS FDM kernel, if the z-dimension is not a multiple of block size, then some threads in the block will not be utilized for computation when processing the end of a row. Because ARPS has a varying input size, it is best to optimize block size so that the amount of inactive threads during end of row computation is minimized in the general case. The number of inactive threads during end of row computation can be expressed as $(\lceil \frac{z}{block_size} \rceil \times blocksize) - z$. By using a smaller block size, the potential number of inactive threads decreases in the general case. Selecting a block sizes smaller than this will eventually reach the blocks per SM limit and reduce the utilization of the SM.

For the ARPS algorithm being studied, it was important to maximize SM occupancy. Therefore, the only block sizes considered were 128, 256, 512, and 1024 due to the fact that these block sizes allow for the maximum amount of thread occupancy per SM. Of these choices, a block size of 128 was selected in order to decrease the potential number of inactive threads during end of row computation.

4.2.2 Memory Coalescing

Our initial task decomposition suffered from a poor memory access pattern given the constraints of the GPU architecture. Recall that each block of threads corresponds to a plane in a three-dimensional array. Consider the example decomposition shown in Figure 2. Each thread group consists of 9 threads, iterating over nine 3×3 sections. The GPU hardware automatically coalesces contiguous memory accesses up to the size of a warp. With a row-major layout, this results in three coalesced memory accesses: $read\{t_1, t_2, t_3\}$, $read\{t_4, t_5, t_6\}$, and $read\{t_7, t_8, t_9\}$. The thread mapping was reorganized into the pattern shown in Figure 3. By restructuring from a two-dimensional thread mapping to a one-dimensional (row-based) map the kernel is able to coalesce the maximum amount of array accesses:
$(read\{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9\})$.

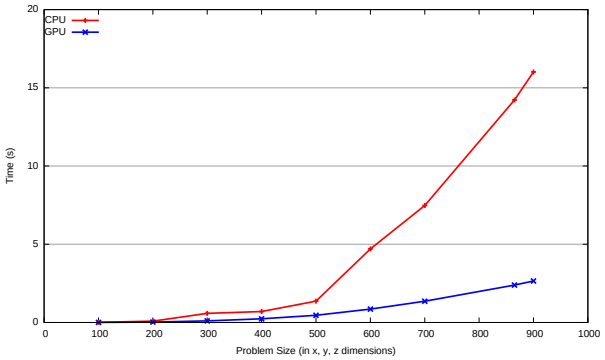

Figure 5: Runtimes of CPU and GPU versions

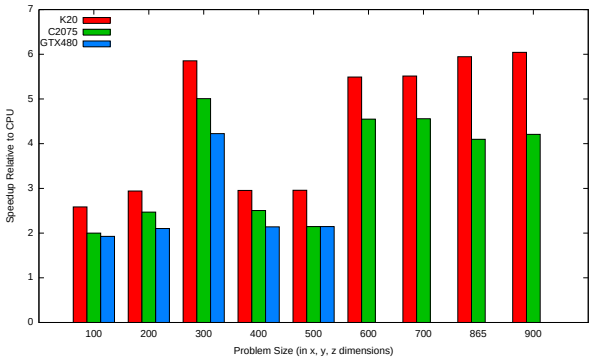

Figure 6: Speedup of various GPUs

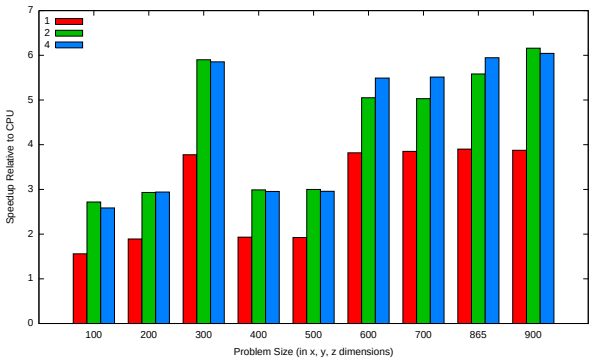

Figure 7: Speedup of varying concurrent kernels

4.2.3 *Asynchronous Memory Transfers*

CUDA supports asynchronous memory transfers which permit computation to occur simultaneously with host-device memory transfers. This overlap hides some of the penalty from the decreased throughput associated with host-device memory transfers since the memory transfer and computation occur in parallel. In this context, the concept of asynchronous memory transfers is similar to instruction pipelining. The disadvantages of asynchronous memory transfers are similar to pipelining. Stalls or bubbles may form due to data dependencies. If the data to be operated on is not in GPU memory, the computation must wait until that data is in memory to begin.

Communication and computation overlap can be implemented by using *concurrent kernels*. It is possible to split data flow from one kernel into two or more separate invocations of the same kernel, as shown in Figure 4. Adding additional sources of data flow allows one or more kernels to compute while host-device memory transfers are occurring. The benefit of this optimization is dependent on the specific hardware architecture of the GPU used. Some GPUs may have separate copy engines for *host* $\rightarrow$ *device* and *device* $\rightarrow$ *host* transfers. Additionally, kernel completion signals may be delayed, which can reduce overlap due to the queuing up of *device* $\rightarrow$ *host* transfers. Modern CUDA GPU cards (CUDA compute capability 3.5+) have hardware support to reduce the impact of kernel launch order and memory transfer interleaving.

Given that the ARPS algorithm requires a large amount of data per computation, has very little temporal locality, and is memory bound, the use of asynchronous memory transfers is paramount for optimizing the ARPS algorithm to run on a GPU.

5. EVALUATION

5.1 Experimental Setup

The experiments were performed on three separate CUDA-capable GPU computer configurations:

The Stampede cluster at the Texas Advanced Computing Center (TACC) contains CUDA capable nodes that consist of 2 quad-core Intel Xeon processors at 2.7 GHz, 32 GB of system memory, and an NVIDIA K20 GPU. The K20 is a CUDA compute capability 3.5 GPU with 8 GB memory.

Additional testing was performed on two CUDA workstations owned by Coastal Carolina University . The first has 2 Intel Xeon processors at 2.2 GHz, 64 GB of system memory, and 2 NVIDIA Tesla C2075 GPUs. The C2075 is a CUDA compute capability 2.0 GPU with 6 GB memory.

The second has one Intel Xeon processor at 3.6 GHz, 8 GB of system memory, and 1 NVIDIA GTX 480 GPU. The GTX 480 is a CUDA compute capability 2.0 GPU with 1.5 GB memory.

The development environment consisted of the GNU GCC host-code compiler and the NVIDIA `nvcc` compiler toolchain. The TACC Stampede cluster codes were compiled with GCC 4.4.7 and CUDA 5.0. The machines local to CCU used the GCC 4.8.2 and CUDA 5.5 compilers.

Experiments were run using a CUDA implementation of $loop_{14}$ with varying input resolutions from $100 \times 100 \times 100$ to $900 \times 900 \times 900$. The input consists of 6 three-dimensional single-precision floating point arrays and the output consists of 3 three-dimensional single-precision floating point arrays. The total input size ranges from 19.07 MB to 13.58 GB, and the total output size ranges from 11.44 MB to 8.15 GB.

5.2 Experimental Results

The CPU and GPU versions of the FDM solver algorithm previously described were run with a threads per block parameter of 128 and four concurrent kernel invocations per call. These results are shown in Figure 5. The results show that the GPU implementation is significantly faster than the CPU version when the input resolution is greater than 500^3. A speedup of 2.6$\times$ faster than the CPU-only version was achieved at an input resolution of 100^3. This amount of

speedup on a relatively small input size shows that CUDA may also be beneficial to use on models with small problem sizes. The maximum speedup achieved over the CPU version was 6× at the 900^3 resolution. As resolution is increased, the runtimes of the CPU-only version trend upward at a much higher rate than the GPU version. This indicates that the GPU version has greater scalability as compared to the CPU version. These results show that a significant speedup can be obtained by running on the GPU – even for memory-bound computations.

Next, the experiment was run on three different NVIDIA GPUs: a K20, a C2075, and a GTX 480. The computer hosting the GTX 480 was only equipped with 8 GB of system memory, and was not able to allocate enough pinned memory above the 500^3 input resolution. The results from this experiment can be seen in Figure 6. As shown, the K20 had the highest maximum speedup at 6×. The maximum speedup for the C2075 was 5×, and for the GTX 480, it was 4.2×. Near the edge of the graph, as input increases, the disparity between K20 speedup and C2075 speedup increases. This greater scaling trend is to be expected as the architecture of the K20 has improved greatly over the Tesla 207X series for GPGPU computations.

Then, the experiment was run with three different implementations of the CUDA code: 1 CUDA kernel per call, 2 concurrent CUDA kernels per call, and 4 concurrent CUDA kernels per call. The results are shown in Figure 7. The single kernel version was only able to achieve a maximum speedup of 3.9× faster than the CPU version. The 2 and 4 kernel versions were able to achieve 6.1× and 6× speedups respectively. This disparity illustrates the importance of utilizing multiple kernels to hide the slow memory transfers. The multi-kernel versions are also trending upwards at the end of the graph while the single kernel version remains level at 3.9×. This indicates that the multiple kernel versions are more scalable than the single kernel version.

6. CONCLUSION

Providing detailed and timely regional weather predictions is an important task which can reduce the expense of weather-related damage and help save lives. This paper has described the acceleration of the ARPS weather modeling tool using GPGPU programming techniques. The adaptation of a portion of the finite difference kernel to the GPU has resulted in up to a six-fold improvement in the program performance.

We believe that this work has several promising leads and we intend to pursue research on this application. While our focus was on a portion of the FDM solver module, we believe that this work is applicable to a large number of the remaining kernels used throughout the ARPS code. Preliminary analysis yields a large number of numeric kernels within ARPS that share a similar structure to those optimized in this paper.

In the future, we aim to investigate additional approaches in mitigating the poor temporal locality exhibited in these kernels. Since some of the FDM solver loops share intermediate arrays, techniques such as loop fusion may be helpful in improving the communication/computation balance and therefore result in better overlap.

7. ACKNOWLEDGEMENTS

This work was supported internally at Coastal Carolina University through the Research Enhancement Grant program sponsored by the Office of the Provost. We would also like to thank the NVIDIA Corporation for its generous donation of hardware for this project. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575.

8. REFERENCES

[1] R. Bermejo, J. Carpio, J. Diaz, and P. Sastre. A Finite Element Algorithm of a Nonlinear Diffusive Climate Energy Balance Model. In A. Camacho, J. Diaz, and J. Fernandez, editors, *Earth Sciences and Mathematics*, Pageoph Topical Volumes, pages 1025–1047. Birkhauser Basel, 2008.

[2] M. Geveler, D. Ribbrock, D. Göddeke, P. Zajac, and S. Turek. Efficient Finite Element Geometric Multigrid Solvers for Unstructured Grids on GPUs . Technical report, Fakultät für Mathematik, TU Dortmund, Jan. 2011. Ergebnisberichte des Instituts für Angewandte Mathematik, Nummer 419.

[3] G. A. Gravvanis, C. K. Filelis-Papadopoulos, and K. M. Giannoutakis. Solving Finite Difference Linear Systems on GPUs: CUDA based Parallel Explicit Preconditioned Biconjugate Conjugate Gradient type Methods. *The Journal of Supercomputing*, 61(3):590–604, 2012.

[4] D. B. Kirk and W.-M. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[5] J. Michalakes and M. Vachharajani. GPU acceleration of numerical weather prediction. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–7, 2008.

[6] J. Mielikainen, B. Huang, H. Huang, and M. Goldberg. Improved GPU/CUDA Based Parallel Weather and Research Forecast (WRF) Single Moment 5-Class (WSM5) Cloud Microphysics. *Selected Topics in Applied Earth Observations and Remote Sensing, IEEE Journal of*, 5(4):1256–1265, 2012.

[7] A. Sathye, G. Bassett, K. Droegemeier, M. Xue, and K. Brewster. Experiences using high performance computing for operational storm scale weather prediction. *Concurrency - Practice and Experience*, 8(10):731–740, 1996.

[8] A. Sathye, M. Xue, G. Bassett, and K. Droegemeier. Parallel weather modeling with the advanced regional prediction system. *Parallel Computing*, 23(14):2243–2256, 1997.

[9] Y. Wang, S. Chang, and J. Cogan. Application of a Multigrid Method to a Mass-Consistent Diagnostic Wind Model. *Journal of Applied Meteorology*, 44(7):1078–1089, July 2005.

[10] M. Xue, D. Wang, J. Gao, K. Brewster, and K. Droegemeier. The Advanced Regional Prediction System (ARPS), storm-scale numerical weather prediction and data assimilation. *Meteorology and Atmospheric Physics*, 82:139–170, 2003.