GPU Acceleration of Numerical Weather Prediction

John Michalakes
National Center for Atmospheric Research
Boulder, CO
michalak@ucar.edu

Manish Vachharajani
University of Colorado
Boulder, CO
manishv@colorado.edu

Abstract—**Weather and climate prediction software has enjoyed the benefits of exponentially increasing processor power for almost 50 years. Even with the advent of large-scale parallelism in weather models, much of the performance increase has come from increasing processor speed rather than increased parallelism. This free ride is nearly over. Recent results also indicate that simply increasing the use of large-scale parallelism will prove ineffective for many scenarios. We present an alternative method of scaling model performance by exploiting emerging architectures using the fine-grain parallelism once used in vector machines. The paper shows the promise of this approach by demonstrating a 20× speedup for a computationally intensive portion of the Weather Research and Forecast (WRF) model on an NVIDIA 8800 GTX Graphics Processing Unit (GPU). We expect an overall 1.3× speedup from this change alone.**

I. Introduction

Exponentially increasing processor power has fueled fifty years of continuous improvement in weather and climate prediction through larger and longer simulations, higher-resolutions, and more sophisticated treatment of physical processes. Even with the advent of large-scale parallelism in the 1990s, much of the performance increase has come from underlying processor improvements. No developer intervention was necessary. However, this free ride is over [3]. To continue the historic rate of application performance increase into petascale (10^{15} floating point operations per second), developers must adapt software models.

Large Clusters. A popular proposal is to expose orders of magnitude more parallelism in weather and climate models to leverage clusters with hundreds or thousands of nodes. A recent Gordon Bell finalist weather simulation [9] used a record 15-thousand IBM Blue Gene processors but only by greatly increasing problem size to 2-billion cells covering an entire hemisphere at a 5km resolution, yielding only weak performance scaling. Ultra-large simulations stress many other aspects of the system, including output bandwidth, analysis, and visualization. Moreover, Ultra-large problem sizes are ineffective for applications that need strong-scaling, e.g. real-time forecasting and climate, where faster time-to-solution is paramount.

To continue performance scaling, architectures must exploit abundant fine-grained parallelism in weather and climate models, not just large-scale coarse-grain parallelism. This paper shows that low-cost commodity graphics coprocessors (GPUs) can improve the performance of a widely used community weather model.

GPU-based Computing. Today, graphics processing units (GPUs) are a low-cost, low-power (watts per flop), very high performance alternative to conventional microprocessors. For example, NVIDIA's 8800 GTX, with a theoretical peak 520 GFLOPS and dissipating 150 watts, costs $500. This is an order of magnitude faster than CPUs, and GPU performance has been increasing at a rate of 2.5× to 3.0× annually, compared with 1.4× CPUs [7].

GPUs exploit data-parallelism in graphics code, allowing GPU manufacturers to:

> ...spend their growing transistor budgets on additional parallel execution units; the effect of this is additional floating point operations per clock. In contrast, the Pentium4 is designed to execute a sequential program. Adding math units to the processor is unlikely to improve performance since there is not enough parallelism detectable in the instruction stream to keep these extra units occupied [4].

Weather and climate models have much fine-grained data parallelism, which was exploited by vector processors [10] and the SIMD supercomputers of the 1990s [5], [6]. Today, most compute-cycles for weather modeling come from large microprocessor-based clusters, which are unable to exploit parallelism much finer than one *s*ubdomain, i.e., the geographic region(s) allocated to one processor. Fine-grain parallelism is wasted because CPUs lack the memory-bandwidth and functional units needed to exploit it.

Graphics processors, like earlier SIMD systems, are designed to exploit massive fine-grain parallelism. Unlike old SIMD systems, all memory is not assumed to be low-latency. GPUs introduce layers of concurrency between data-parallel threads with fast context switching to hide memory latency. In other words, when one *warp* of threads is stalled another can be quickly swapped in. GPUs also have large, dedicated, read/write and read-only memories to provide the bandwidth needed for high floating-point compute rates. Using GPUs for numerical weather prediction (NWP) raises several questions.

- Which weather model modules involve the most computation?
- How much speedup can GPU co-processing deliver?
- What are the prospects for improving overall model performance?
- How easy is developing a GPU-accelerated module?
- Can modules be efficient and yet portable?
- What hardware and software improvements are needed to fully exploit GPUs and other coprocessors for NWP?

In this work, we begin to answer these questions by selecting a computationally intensive module from the Weather Research and Forecast (WRF) model [11], adapting this Fortran model to run on NVIDIA's 8800 GTX GPU using their Compute Unified Device Architecture (CUDA), and then assessing performance improvement for the module itself and the projected overall performance improvement. We compare these improvements to overheads such as data transfer and re-engineering costs. We also uncover and explore common domain-specific program abstractions that may be exploited in the form of directives or language constructs to simplify the task of converting large sections of the model to run on the GPU.

II. Approach

To demonstrate the promise of GPU computing for NWP, we ported a computationally intensive WRF physics module, then validated, benchmarked, and compared its GPU performance to that of the original module on 3 conventional processors.

The Weather Research and Forecast model is a state-of-the-art non-hydrostatic NWP model maintained and supported by the National Center for Atmospheric Research. First released in 2000, WRF is now the most widely used community weather forecast and research model in the world [1]. Atmospheric models may use double (64-bit) precision but since WRF's WRF fluid dynamics core uses explicit

[1]See http:www.wrf-model.org

finite-difference approximation, it requires only single (32-bit) floating point precision. Physics, other modules that represent non-CFD atmospheric processes, are also single-precision.

WRF Single Moment 5-tracer (WSM5) [8] microphysics represents condensation, fallout of various types of precipitation, and related thermodynamic effects of latent heat release. WSM5 is only 0.4 percent of the WRF source code but consumes a quarter of total run time on a single processor.

WRF represents the atmosphere over a geographic region using a 3-dimensional grid. Two horizontal dimensions x,y are over an equally spaced Cartesian coordinate system; the third dimension z is over vertical levels of the atmosphere in a pressure-based terrain-following coordinate. WSM5 is a type of "column physics" in which computation proceeds along z for each vertical stack of grid cells over a given coordinate in x,y. The memory footprint is large with 40 single-precision floating point variables per cell. Depending on the state of the atmosphere, WSM5 involves an average 2400 floating point multiply-equivalent operations per cell per invocation. This relatively high computational density arises from WSM5's heavy use of Fortran intrinsics (sqrt, log, exp).

The NVIDIA GPU. The target GPU for this investigation is the NVIDIA 8800 GTX, which comprises 128 SIMD "stream processors" operating at 1.35 GHz. Theoretical peak is 520 gigaflops [1]. Eight physical stream processors work together as a SIMD unit called a multiprocessor and there are 16 multiprocessors in the 8800 GTX. All multiprocessors have access to 768 MB of multiported DDR SDRAM. Accesses to this device memory are high-latency operations taking hundreds of cycles. Each multiprocessor has a local 16 kB thread-shared "scratchpad" memory, with a 2-cycle access time, and a local register file.

Stream processors are not programmed directly; rather, one writes a CUDA *kernel* for the GPU. Each kernel consists of a collection of threads arranged into blocks and grids. Each grid is a group of blocks, each block is a group of threads. Conceptually, each block is bound to a virtual multiprocessor; the hardware will time-share the multiprocessor amongst blocks provided that the hardware has enough memory resources to satisfy all the block requirements within the physical multiprocessor. In general, the more threads per block, the better the performance because the hardware can hide memory latencies. The only caveat is that a kernel should have enough blocks to simultaneously utilize all the multiprocessors in a given NVIDIA GPU, 16 in the case of the 8800 GTX. Generally, one should have at least 32 threads per block and 16 blocks for a minimum of 512 threads. However, hundreds of threads per block are typically recommended [2].

For WSM5, a thread-per-column decomposition yields 4,118 threads for the Storm of the Century (SOC) workload [2]. Unfortunately, GPU memory size works against large numbers of threads per block. The memory footprint for a column in the SOC benchmark is 4320 bytes per column. With 32 threads per 16kB block, we have only 3 columns worth of space available to a thread. Data that does not fit in the fast shared memory must be stored in the slower DRAM device memory. Therefore, care must be taken allocating shared memory for arrays that are reused the most. Section III describes how these limitations affect performance.

Code translation. WSM5 is a 1500 line Fortran90 module in the WRF community software distribution. We manually converted the WSM5 module into a CUDA kernel using a few prototype lan-

[2]See Section III for a description of this workload.

guage extensions that we developed to aid in the process of managing the GPU memories. Rewriting in C (CUDA is C-based) requires converting globally addressed multi-dimensional arrays to locally addressed single-dimensional arrays with explicitly managed indexing. Furthermore, arrays need to be declared and indexed differently depending on whether they are arguments or local arrays and whether they are accessed from device memory or from thread-shared memory. Only a few of the three-dimensional arrays accessed by WSM5 will fit into thread-shared memory. Furthermore, movement of data into and out of this memory is managed explicitly [3]. A manual analysis of definition-use chains in the WSM5 code indicated that arrays storing various moisture tracers were most reused so these were copied into shared memory at the beginning and then copied back out into device memory at the end of the kernel. We made no further attempt to optimize memory accesses; this could be a fruitful avenue of research.

The task of converting the code from Fortran to CUDA C was simplified with the development of several simple directives and a simple Perl-based preprocessor/translator to help define dimensionality and memory residency of an array. Subsequently, the programmer can write purely column-oriented CUDA C code using only the vertical index (k) to index the array. Figure 1(a) shows a (simplified) section of original Fortran and Figure 1(b) shows the corresponding C using our directives.

The `_def_` directives tell the preprocessor that all three arrays are three-dimensional and passed in as arguments to the routine, but that `q` should be copied into fast thread-memory at the beginning of the routine and copied back at the end. All array references need only the vertical index `k`, even though `q` is being accessed from thread-shared memory while `t` and `den` are stored in device memory.

The CUDA compiler sees code similar to that in Figure 2. `S3` and `D3` are macros that expand into indexing expressions for the three-dimensional arrays in shared and device memory, respectively. The `copy_up_memory` directive expands to declare and copy into

[3]Arguably, this explicit management is a performance advantage compared to CPU caches

```
1  DO j = jts, jte
2   DO k = kts, kte
3    DO i = its, ite
4     IF (t(i,k,j) .GT. t0c) THEN
5       Q(i,k,j) = T(i,k,j) *
6                  DEN( i,k,j )
7     ENDIF
8    ENDDO
9   ENDDO
10 ENDDO
```

(a) Fortran

```
1  //_def_ arg ikj:q,t,den
2  //_def_ copy_up_memory ikj:q
3   [...]
4  for (k = kps-1; k <= kpe-1; k++) {
5    if (t[k] > t0c) {
6      q[k] = t[k] * den[k] ;
7    }
8  }
9   [...]
10 //_def_ copy_down_memory ikj:q
```

(b) CUDA C

Fig. 1. Simplified code fragment for WSM5.

```
1  __shared__ float * q_s; int k;
2   [...]
3  for(k=kps-1;k<kpe;k++) {
4    q_s[S3(ti,k,tj)]=q[D3(ti,k,tj)]; }
5
6   [...]
7  for ( k = kps-1 ; k <= kpe-1 ; k++ ) {
8    if ( t[k] > t0c ) {
9      q_s[S3(ti,k,tj)] =
10       t[D3(ti,k,tj)] *
11       den[D3(ti,k,tj)] ;
12   }
13 }
14  [...]
15 for(k=kps-1;k<kpe;k++) {
16   q[D3(ti,k,tj)]=q_s[S3(ti,k,tj)]; }
```

Fig. 2. WSM5 CUDA C Code after Processing Directives.

`q_s`, a shared memory version of `q`. The `copy_down_memory` directive expands into a reverse copy of `q` at the end. The macros, directives, and source translation preprocessor used in this work are part of a more comprehensive application domain-specific set of translations under development.

III. Results

This section presents initial validation and benchmark results comparing the original WSM5 code with the GPU version. Development and testing was done standalone (outside the WRF model) on a Linux workstation with a 2.80 GHz Pentium-D CPU and an NVIDIA 8800 GTX GPU coprocessor. The original code was compiled with gfortran/gcc 4.3.0 and -O3 optimization. The GPU implementation was compiled using the NVIDIA nvcc compiler (release 1.0, V0.2.1221). The original code was also compiled and benchmarked on an IBM Power5 1.9 GHz processor using XLF (IBM Fortran) version 10.1 with -O4 -qhot optimization and an Intel Xeon 3.0 GHz quad core under Mac OS/X using pgf90 (Portland Group Inc.) version 7.0-7 64-bit and with -O2 -fast -fastsse optimization.

Storm of the Century Test-case. We chose the well-tested, relatively small, and easy to work with Eastern United States January 2000 "Storm of the Century" (SOC). The SOC domain is 115-thousand cells covering an atmospheric grid 71 by 58 cells at a horizontal resolution of 30km and with 27 vertical levels. At the WSM5 microphysics call-site, the WRF model was modified to save off the WSM5 input arguments to files at each time step and was then run for a short time after a reasonable spin-up period to generate ten sample sets of input. We built a standalone WSM5 test driver to validate and debug the CUDA C code as well as benchmark against the original Fortran code. For the tests, this driver read one input data set and then invoked the original WSM5 routine or the GPU implementation.

Validation We took considerable care to ensure the GPU implementation produced correct output with respect to the original WSM5 routine. For NWP, bit-for-bit floating point agreement never occurs between different processors, compilers, and libraries. The perturbation for the NVIDIA GPU was even more significant [2]:

- Square root and division are non-standard-compliant,
- Dynamically configurable rounding mode is not supported,
- Denormalized source operands are treated as zero, and
- Underflow is flushed to zero.

Validation and debugging was performed using difference plots – color contour plots of the point-wise arithmetic difference between output from the original code and the GPU implementation. Small differences in a "snow"-like random distribution pattern were assumed to be from round-off. Validating using double (64-bit) floating point precision with the CUDA emulator on the Pentium host was also helpful.

Performance. Calls to the WSM5 implementations were measured using UNIX gettimeofday.[4] A second set of timers measured host-GPU data transfer costs. Six sets of runs were conducted to measure cost of WSM5 on the CPUs and the GPU. On the host Pentium, average cost per invocation was was 0.935 seconds ($\sigma = 1.932$ milliseconds). The Power5 averaged 0.213 seconds ($\sigma = 1.458$ milliseconds) and the Xeon, 0.236 seconds ($\sigma = 1.315$ milliseconds). On the NVIDIA GPU, average cost was 0.042 seconds ($\sigma = 0.074$ milliseconds); 0.055 seconds ($\sigma = 0.266$ milliseconds) including data transfer between the host and GPU memories. Figure 3 shows the average floating point rates for the

[4]Timers around the GPU implementation also included a call to cudaThreadSynchronize after the call to WSM5.

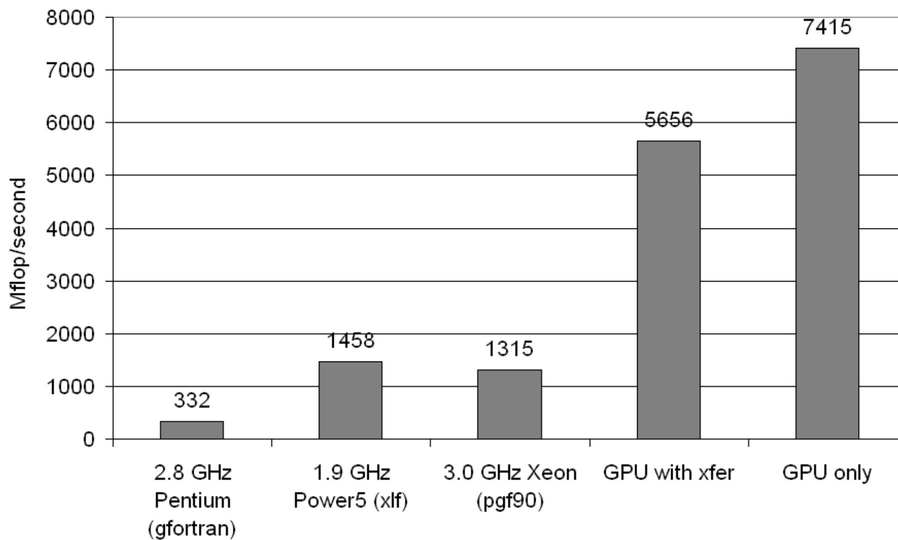

Fig. 3. Performance on Pentium, Power5, Xeon, and GPU.

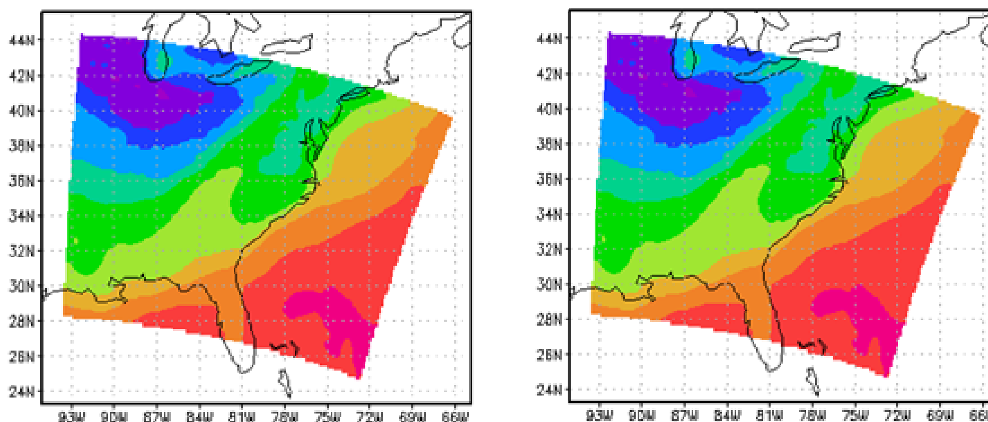

Fig. 4. WSM5 potential temperature from the original (left) and GPU code (right).

Pentium-D, Power5, Xeon and GPU, both with and without data transfer overhead. Data transfer cost was measured only on the host Pentium D system and would likely differ on other systems. The operation count was obtained using the hardware performance monitor on the Power5 system.

At this writing, almost no effort has gone into optimizing GPU performance. Nevertheless, initial results are extremely encouraging, showing a better than 17× GPU speed advantage relative to the host CPU, including data transfer. Meteorological output from the CPU and GPU versions was visually indistinguishable (Figure 4).

IV. Conclusion

For numerical weather prediction and climate modeling, exclusive focus on large-scale parallelism on clusters neglects vast quantities of fine-grained parallelism. This limits NWP and climate to weak-scaling, hindering science that requires faster turn-around for fixed-size simulations. This paper shows that low-cost/high flops-per-watt

GPUs can exploit fine-grain parallelism and help restore *strong scaling* for scientific problems at petascale.

With this work, we have demonstrated that a modest investment in programming effort for GPUs yields an order of magnitude performance improvement for a small but performance critical module the widely used WRF weather model. Only about one percent of GPU performance was realized but these are initial results; little optimization effort has been put into GPU code. Despite this limitation, porting just this one package still provides significant overall benefit: the 5× to 20× increase in WSM5 performance translates into 1.25× - 1.3× increase in total application performance (Amdahl's law limits the total increase to 1.3×). A 1.25× improvement in model performance from a few months effort is rare.

Though 1.3× is clearly not enough to support strong scaling, the initial result is still promising. Moving more computation into the GPU will yield equivalent performance from smaller more efficient clusters. Furthermore, planned improvements in GPU speed, host proximity, and programmability will allow WRF and other highly data-parallel weather and climate models to execute almost entirely on the GPU.

REFERENCES

[1] *Technical Brief: NVIDIA GeForce 8800 GPU Architecture Overview*. Santa Clara, California, November 2006.

[2] *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide Version 0.8*. Santa Clara, California, February 2007.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[4] I. Buck. *Stream Computing for Graphics Hardware*. PhD thesis, Department of Computer Science, Stanford University, Palo Alto, California, United States, September 2006.

[5] J. Dukowicz, R. D. Smith, and R. Malone. A reformulation and implementation of the bryan-cox-semtner ocean model on the connection machine. *Atmos. Ocean. Tech.*, 10:195–208, 1993.

[6] S. W. Hammond, R. D. Loft, J. M. Dennis, and R. K. Sato. Implementation and performance issues of a massively parallel atmospheric model. *Parallel Computing*, 21:1593–1619, 1995.

[7] B. Himawan and M. Vachharajani. Deconstructing Hardware Usage for General Purpose Computation on GPUs. *Fifth Annual Workshop on Duplicating, Deconstructing, and Debunking (in conjunction with ISCA-33)*, 2006.

[8] S. Hong, J. Dudhia, and S. Chen. A Revised Approach to Ice Microphysical Processes for the Bulk Parameterization of Clouds and Precipitation. *Monthly Weather Review*, 132(1):103–120.

[9] J. Michalakes, J. Hacker, R. Loft, M. O. McCracken, A. Snavely, N. Wright, T. Spelce, B. Gorda, and B. Walkup. Wrf nature run. In *proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–6, 2007.

[10] S. Shingu, H. Takahara, H. Fuchigami, M. Yamada, Y. Tsuda, W. Ohfuchi, Y. Sasaki, K. Kobayashi, T. Hagiwara, S. ichi Habata, M. Yokokawa, H. Itoh, and K. Otsuka. A 26.58 tflops global atmospheric simulation with the spectral transform method on the earth simulator. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–19, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[11] W. C. Skamarock, J. B. Klemp, J. Dudhia, D. O. Gill, D. M. Barker, W. Wang, and J. G. Powers. A description of the advanced research WRF version 2. Technical Report NCAR/TN-468+STR, National Center for Atmospheric Research, January 2007.