Parallel Processing Letters

GPU ACCELERATION OF NUMERICAL WEATHER PREDICTION

JOHN MICHALAKES
National Center for Atmospheric Research
*Boulder, Colorado**

MANISH VACHHARAJANI
University of Colorado at Boulder
Boulder, Colorado

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 nearly 8× speedup for a computationally intensive portion of the Weather Research and Forecast (WRF) model on a variety of NVIDIA Graphics Processing Units (GPU). This change alone speeds up the whole weather model by 1.23×.

Keywords: graphics processing units, weather modeling, high-performance computing, CUDA

1. 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 [4]. 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 [10] used a record

*The National Center for Atmospheric Research is sponsored by the National Science Foundation.

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 [8].

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 [5].

Weather and climate models have much fine-grained data parallelism, which was exploited by vector processors [11] and the SIMD supercomputers of the 1990s [6,7]. 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 [12], adapting this Fortran code to run on NVIDIA's Compute Unified Device Architecture (CUDA). Performance improvement is then assessed and analyzed across a variety of CUDA-compatible GPUs including the 8800 GTX, GTX 200, and Quadro 5600. 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.

2. 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 a number of 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 [a]. Atmospheric models may use double (64-bit) precision but since WRF's fluid dynamics core uses explicit 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) [9] 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 pressure-based terrain-following coordinates. 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

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

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 GPUs. We evaluate a variety of GPUs for this investigation: The NVIDIA 8800 GTX, the NVIDIA Quadro 5600, and a pre-release of the NVIDIA GTX 200. The architecture of these GPUs is similar; below we discuss the 8800 GTX. Later, architectural differences are highlighted as needed to explain varying results. The NVIDIA 8800 GTX 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 "scratch-pad" 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 a workload based on a standard WRF forecast case, a U.S. East Coast January 24-25, 2000 Winter Storm workload [b]. Unfortunately, GPU memory size works against large numbers of threads per block. The memory footprint for a column in the benchmark is 4320 bytes per column. With 32 threads per 16kB block, we have only several 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 3 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 language 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-

[b] http://www4.ncsu.edu/~nwsfo/storage/cases/20000125. See Section 3 for a description of this workload.

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 [c]. A manual analysis of definition-use chains in the WSM5 code indicated that arrays storing various moisture tracers were most reused so in one implementation of the WSM5 kernel these were copied into shared memory at the beginning of the kernel and then copied back out into device memory at the end. 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.

```
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.

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

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.

```
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.

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 `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.

3. Results

This section presents initial validation and performance results for the GPU version of the WSM5 microphysics code running standalone and within the full application. Development and testing was done using a Linux workstation, first with an NVIDIA 8800 GTX 1.35 GHz Ultra GPU and then with a newer GTX 2000 (a pre-release engineering sample of the board). More detailed profiles were later conducted using the 8800 GTX and NVIDIA's 2.0 release of the CUDA SDK and Toolkits. The 2.0 envirornment provides access to GPU hardware event counters, measurements from which are presented below. End-to-end timing benchmarks of the WSM5 kernel running standalone and then in the WRF model as a whole were conducted using a large Infiniband-connected Linux cluster at the National Center for Supercomputing Applications at the University of Illinois comprising 16 nodes, each with two dual-core 2.4 GHz AMD Opteron processors and four 1.35 GHz NVIDIA Quadro 5600

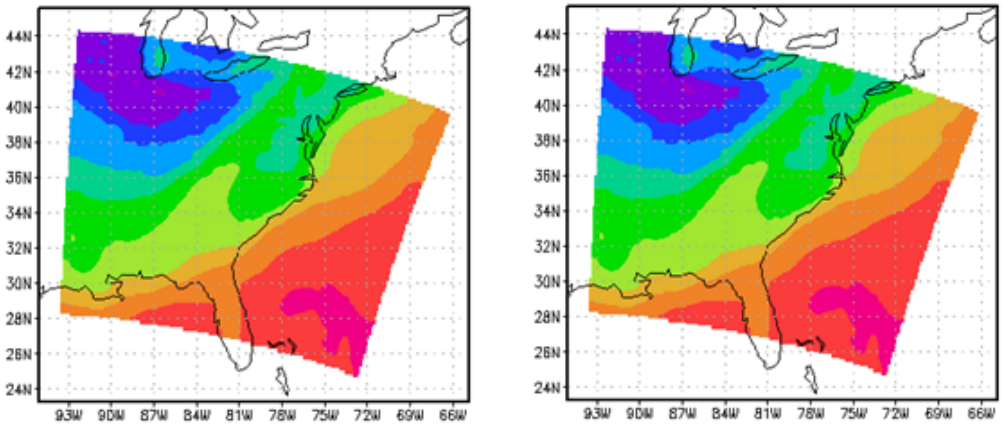

Fig. 3. WSM5 potential temperature from the original (left) and GPU code (right).

GTX GPUs per node (for a total of 64 CPU-GPU pairs). The NCSA cluster included Intel Fortran for compiling the original WRF code. On the NCSA system, GPU compilation was done using the older 1.0 version of the NVIDIA SDK, so detailed profiling was not available for the 5600 GTX.

3.1. *Standalone WSM5*

January 2000 Test-case. We chose the well-tested, relatively small, and easy to work with Eastern United States January 24-25, 2000 winter storm case (Jan00). The Jan00 domain is 115-thousand cells covering an atmospheric grid 74 by 61 cells at a horizontal resolution of 30km and with 27 vertical levels. This workload involves 311 million floating point operations as measured by the IBM Hardware Performance Monitor utility on a Power5 processor. 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. Each invocation of WSM5 for the Jan00 workload involves 10 three-dimension fields of input (4.5 MB) and generates 7 three-dimensional fields of output (3.1 MB), which must be transferred back and forth between CPU and GPU when the GPU version of microphysics is invoked.

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. Meteorological output from the CPU and GPU versions was visually indistinguishable (Figure 3). Validating using double (64-bit) floating point precision with the CUDA emulator on the Pentium host was also helpful.

Benchmark performance.

The standalone WSM5 code was benchmarked with an eye towards uncovering performance sensitivities to controllable parameters, and then attempting to discover the underlying mechanisms responsible for these sensitivities. Obviously, the need to reproduce the output of the original Fortran routine constrained modifying the code itself as a parameter for the benchmark experiments. Rather, a straightforward and more easily modified parameter was used. The factor that was controlled was varying the number of threads per block (and thus the number of blocks per grid) when the kernel was invoked. This affects the number of threads available to supply the GPU with adequate parallelism, but also involves countervailing constraints. More threads increases demand on the limited number of registers (8192 registers for the 8800 and 5600 GTX, double that for the GTX 200) per multiprocessor. This constraint was especially acute for the state-heavy WSM5 kernel. Using nvcc with the -cubin option, it was determined that each thread required between 50 (for the non-copying kernel) and 61 registers (for the copying kernel). Each multiprocessor in the GPU can support only up to 8 active thread blocks at a time, and each block is limited to a maximum of 512 kernels. These limits, especially for registers, had the effect of sharply curtailing the actual number of WSM5 kernel threads that could be active. For example, with 64 threads per block (two warps), the WSM5 kernel has enough parallelism to support 320 threads per multiprocessor; however supplying each thread with 50 registers outstrips the available number by nearly a factor of two, so that only two blocks (128 threads) can be run on each multiprocessor.

Runs were conducted with the standalone WSM5 kernel on a single CPU/GPU pair of the NCSA cluster to measure the performance with the Jan00 workload. Cost per invocation was was measured using UNIX gettimeofday. Timers around the GPU implementation also included a call to cudaThreadSynchronize after the call to WSM5. A second set of timers measured host-GPU data transfer costs. On the host Opteron processor, the average cost per invocation was 192 milliseconds. On the GPU, the average cost per invocation was 25.1 milliseconds. The cost for moving 7.6 MB of input and output data between the host and GPU before and after

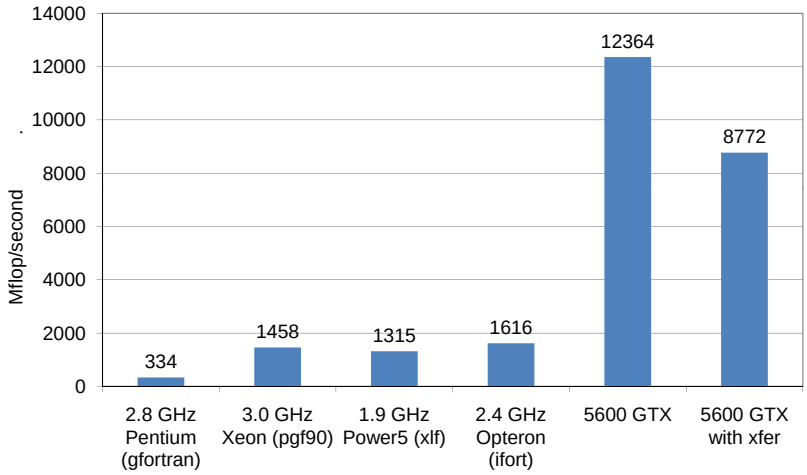

Fig. 4. WSM5 kernel performance on a number of conventional CPUs, including host CPU (2.4 GHz dual-core Opteron) of NCSA cluster qp.ncsa.uiuc.edu, compared with GPU (NVIDIA Quadro 5600) both by itself and effective GPU performance including cost of host-GPU data transfer overhead.

the invocation was about 10.35 milliseconds (730 MB/second), so that the effective GPU speed (including overhead) was 35.4 milliseconds. Benchmark performance expressed as floating point rates is summarized in Figure 4.

Benchmarking the WSM5 standalone kernel over several different models of the NVIDIA GPU revealed that performance was more sensitive to clock rate than to other factors such as size of register file or number of multi-processors per device. Figure 5 shows cost per call for the three devices tested as number of threads per block was varied. Comparing best times, both 1.35 GHz GPUs, the 8800 and the 5600, were faster than the newer 1.08 GHz GTX 200 pre-release, despite its 1.5× more processor cores and double-sized register file. The best cost-per-call time on the GTX 5600 was faster by a factor of 1.24× than the GTX 200 – the approximate factor of difference in their clock rates. The GTX 200 was notable in another respect. The larger internal resources of the GTX 200 allowed up to 320 threads per block, compared with 128 for the 8800 and 5600 devices.

Performance of the kernel showed a marked transition between 64 and 80 threads per block on all three GPUs, suggesting some internal threshold is exceeded at

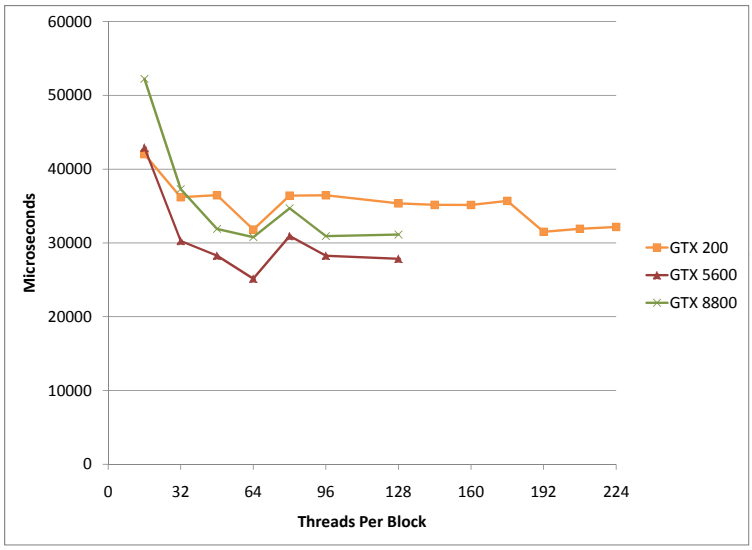

Fig. 5. Cost per call for Jan00 workload on three NVIDIA chip series: 1.35 GHz GTX 8800, 1.35 GHz GTX 5600, and 1.08 GHz GTX 200 (pre-release).

this point. Determining the source of this feature was the subject of additional investigation using the NVIDIA profile utility, discussed below.

GPU Profiles. As mentioned above, CUDA 2.0 provides detailed counts of loads and stores to device memory and local memory, the number of blocks issued, warp occupancy (percentage of maximum warp count in the GPU), time spent in the GPU itself separate from the time spent invoking the kernel, and other quantities [3]. Detailed profiles were collected for the 1.35 GHz 8800 GTX and the pre-release GTX 200 on a Linux workstation with CUDA 2.0. As above, the parameter controlled for the experiments was the number of threads per block for the fixed-size Jan00 benchmark.

One striking initial result from the profiles is the amount of time spent invoking the GPU kernel compared with the time spent on the GPU itself, shown for the GTX 200 in Figure 6. (This cost is different from data transfer overhead, considered elsewhere; rather it is the CPU cost within the interface itself of invoking the kernel on the GPU). As one would expect, the time for the GPU device itself is what is sensitive to varying the number of threads per block. The CPU driver cost is relatively constant, but also dominates the cost of an invocation for this kernel and workload. This suggests that more work could be added to this kernel to amortize the fixed CPU component of the cost, perhaps another physics module from WRF.

Figure 7 shows profiler-collected cost per call to the WSM5 kernel on the 8800

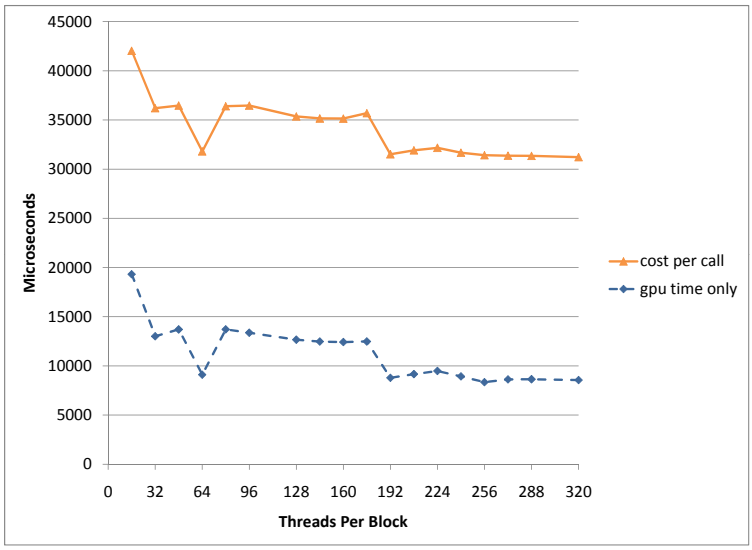

Fig. 6. CUDA 2.0 provides time spent in the GPU itself separate from the overall cost per invocation. Cost per call is dominated by the fixed cost on the CPU, not the GPU (the combined cost per call shown here does not include CPU/GPU data transfer costs).

GTX GPU as the number of threads per block were varied. Occupancy is well-correlated with performance, and is likely causal in terms of the observed cost-per-call for a state-heavy kernel such as WSM5. However, it is important to understand lower-level mechanisms that affect both occupancy and performance. Is the kernel running out of registers or some other resource and spilling to device memory? Is it hitting a limit on the number of blocks per multiprocessor? Some combination of both? The remainder of this section is a summary of the profiles obtained as they pertain to the observed performance plots of the two kernels.

Best performance occurs at 64 threads per block, once each block has two warps to work with, allowing better hiding of stalls for accesses to device memory. Figure 8 shows only a slight decrease in the number of coherent loads and stores to device memory at 64 threads per task. Cost per call increases again at 80 threads per block and does not fall back again until it reaches 192 threads per block, where there is a corresponding *rise* in coherent loads and stores. Loads and stores to shared memory also fall off at 64 thread-per-block where performance is best. There is a steady increase in loads and stores to shared memory above 128 threads-per-block which, nevertheless, appears uncorrelated with performance of the kernel. On the other hand, non-coherent loads and stores – that is, more costly loads and stores that are not contiguous and 16-word aligned – were correlated with the cost per call.

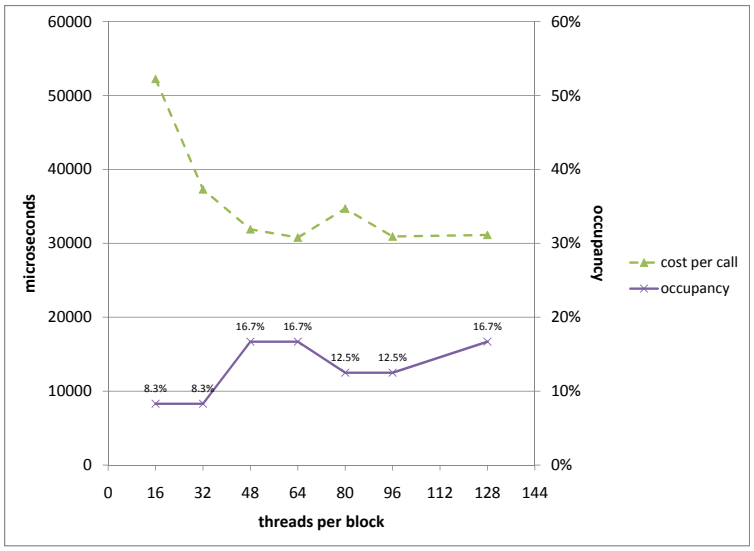

Fig. 7. Cost per invocation and warp occupancy from profiled runs of the Jan00 workload as number of threads per block are varied on 1.35 GHz NVIDIA 5600 GTX for the standalone WSM5 kernel.

The most likely candidate for explaining occupancy and, in turn, cost per call is the value of the CTA_LAUNCHED counter, which counts "the number of executed thread blocks" per Texture Processing Cluster (TPC) – a pair of multiprocessors on the GTX 8800 and GTX 5600, and a triplet of multiprocessors on the GTX 200. Multiplying the value of this counter by the number of threads per block then dividing by the number of multiprocessors per TPC provides the total number of threads launched on one of the multiprocessor chips within the GPU. It is important to recognize, first, that as documentation for the CUDA profiler explains, "the profiler can only target one of the multiprocessors in the GPU" so that "users should not expect the counter values to match the numbers one would get by inspecting kernel code." And second, the threads per multiprocessor value obtained using the CTA_LAUNCHED counter is a total for the entire execution of the kernel; it does not reveal the most relevant factor for performance: the number of *active* thread blocks. Even so, the total number does agree reasonably well with estimates of total threads per multiprocessor based on static analysis. Therefore, it should be possible to derive the number of active threads by combining the available profiler information with static analysis.

Figure 9 shows results from one such synthetic estimate of active threads per multiprocessor. The total number of threads per multiprocessor for a kernel obtained

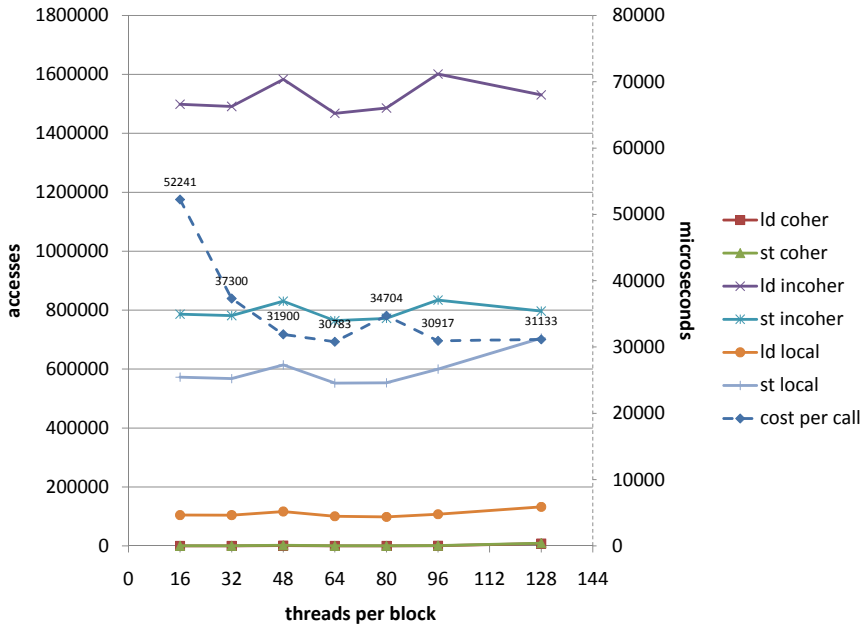

Fig. 8. Access counts for loads and stores from device memory and shared memory as number of threads per block is varied. Profiles gathered on 8800 GTX and NVIDIA SDK 2.0.

from the CTA_LAUNCHED counter is adjusted downward to account for constraints of 8 active blocks per multiprocessor and that there are only 8192 registers. The model assumes that each thread needs 50 registers, as determined using nvcc -cubin. As the number of threads per block is varied, better (lower) cost-per-call (also shown in the figure) should result from increased numbers of active threads per multiprocessor. This is not the case, however, if it is true that 8192 all registers are available for threads. If so, the model shows a *decrease* in active threads per multiprocessor at 64 threads per block, where the kernel performance is *best*. If, instead, the model further assumes that not all registers are available for threads (or that the thread requirements are higher than reported by nvcc -cubin), the model then agrees much better with the observations. Assuming there are only between 7199 and 7999 registers available for threads, the model correctly predicts the bump in elapsed time at 80 threads per block. Assuming this visual correlation between the plots is valid (the actual statistical correlation is not very good, only -0.24 and -0.28 respectively) one may infer that there is some small reserve of registers that are unavailable to satisfy per-thread requirements; or that nvcc's static estimate of per-thread register requirements underrepresents what is actually needed at runtime; or, of course, that this model is missing some other important factors. The last possibility seems most likely and underscores the difficulty of using static analysis or hardware counter data to model effects of multiple internal mechanisms interacting nonlinearly at run time. In this respect at least, the GPU is still quite

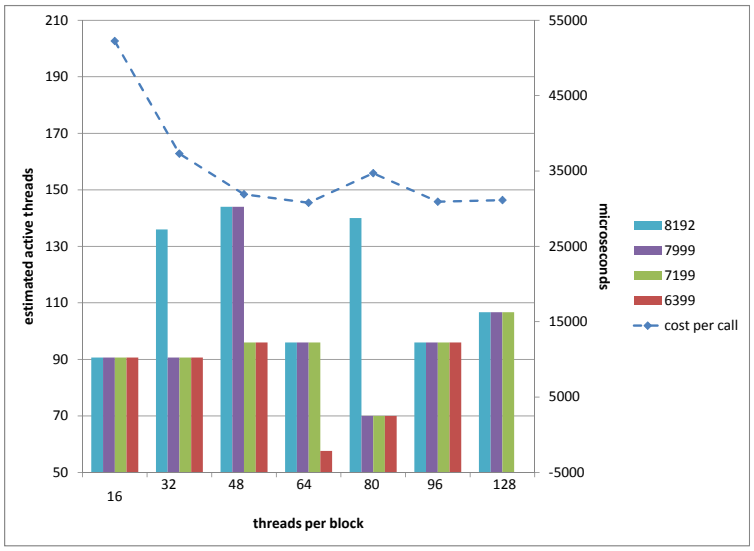

Fig. 9. Estimate of active threads per multiprocessor based on the CUDA profile counter that provides the total over the entire kernel invocation (CTA_LAUNCHED) adjusted for limits on register usage and the number of blocks that can be active at one time on a multiprocessor on the GTX 8800. The dotted line shows cost per call for the WSM5 kernel. Each set of bars represents the estimated active number of threads assuming all 8192 registers are available and than assuming fewer than 8000, 7200, and 6400 registers. The best (negative) correlation between the estimated active registers and the actual observed performance occurs if the model assumes there are between 7200 and 7999 registers available for use by threads.

similar to conventional CPUs. Until the situation improves, the best approach may be some form of auto-tuning where parameter sweeps to find best kernel performance are done case-by-case when a kernel is compiled and run. Regardless, it is reasonable to conclude that exhaustion of available register resources by the state-heavy WSM5 kernel is a key factor affecting its performance on the GPU.

3.2. *Whole code*

The GPU accelerated WSM5 microphysics was incorporated back into the WRF model and then benchmarked for a larger workload, a 12km resolution forecast over the continental United States (CONUS) domain for severe weather events October 24, 2000. This workload is one of the standard WRF cases distributed and maintained by the WRF developers for computational benchmarking over a range of supported platforms.[d]. The grid is 425 by 300 cells with 35 vertical levels. Com-

[d]See: http://www.mmm.ucar.edu/wrf/WG2/bench

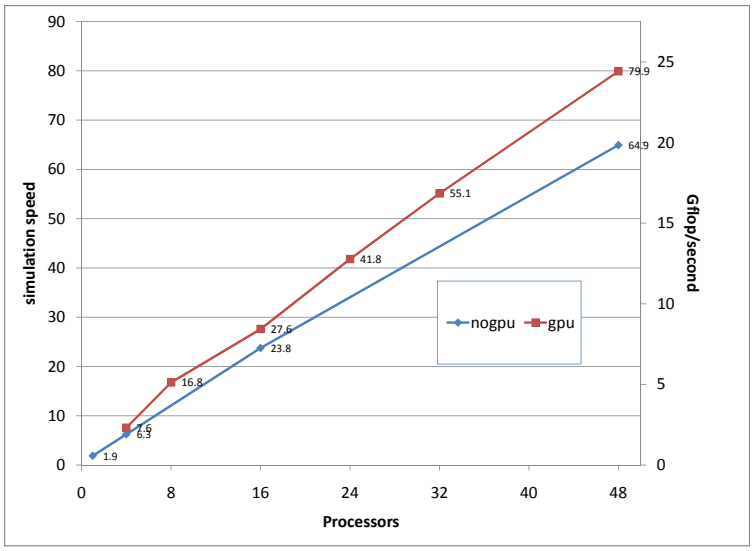

Fig. 10. WRF simulation speed (simulated time per unit of wall time) and floating point rate in Gflop/second with and without GPU-accelerated cloud microphysics.

putational cost is 22 billion floating point operations per 72 second time step. The benchmark is a 3 hour simulation that begins from a restart at the 24 hour mark of the 48 hour simulation period so that moisture fields are fully spun-up – particularly important for a representative microphysics benchmark. GPU accelerated microphysics was incorporated into WRF by adding a call to the CUDA code at the WSM5 call site. The original WSM5 code was also retained so that the original or the CUDA version of the microphysics package was selectable at compile time.

Benchmarks were conducted using the Opteron/GTX 5600 cluster at NCSA, described above. Both versions of the code were compiled using the Intel Fortran compiler version 10.0, and -O3 optimization. The codes were also compiled with MPI to allow both conventional distributed memory parallel scaling and GPU acceleration of each task, up to the number of CPU/GPU pairs on the 16 node Infiniband connected system. Code using the cudaSetDevice() interface to CUDA was added to the WRF model startup routines to ensure that each MPI task had sole access to one of 4 GPUs on each node. Series of runs were conducted with increasing numbers of MPI tasks. Per-timestep timings were collected and floating point rates calculated based on the average cost per time step (22 GFlop) divided by the measured average time per time step. Results for the original and GPU accelerated WRF executables are shown in Figure 10.

Running on 48 MPI tasks and measuring the MPI task with the largest mi-

crophysics load (microphysics is the main source of load imbalance in WRF), the original non-GPU WRF code required 166 seconds, 25.5 seconds (15.4 percent) in microphysics (discounting I/O and initialization time). WRF with GPU-accelerated microphysics ran in 135 seconds and of this, 12.1 seconds (9.0 percent) was spent in microphysics. The time on the GPU itself was only 4.6 seconds (3.4 percent), the remaining 7.5 seconds (5.5 percent) was data movement around the GPU invocation which included both CPU/GPU data transfer but also cost of copies in the Fortran code itself for array subsetting around the call to the WSM5 driver.

The per-task workload for the CONUS case with 48 tasks is roughly the same size as the Jan00 standalone workload, both about 100 thousand cells, so that the GPU-only times are roughly the same: 150 invocations of the standalone workload take between 3.5 and 4.5 seconds, depending on tasks per block and copying versus non-copying kernel selections. The data transfer cost is much larger for the whole-code benchmark because of the extra copying for array subsetting mentioned above. This could be eliminated by rewriting the WSM5 kernel, which currently expects fully contiguous stride-1 vectors without spaces for halos or boundaries, but at a potential cost of introducing more non-coherent accesses to GPU device memory.

One surprising result of the whole code WRF benchmarks was that the *rest* of the model ran 1.16 times faster with GPU microphysics. On 48 tasks the non-microphysics parts of WRF took 122 seconds with GPU-accelerated microphysics versus 141 seconds without (again measuring the task with heaviest microphysics load). This effect is not as yet explained but may result from fortuitous cache-loading associated with the copying around the call to GPU microphysics or the reduction in load imbalance associated with the microphysics. The GPU microphysics was still subject to an about 20 percent load imbalance ($1-mean/max$ over tasks) compared with 24 percent for the non-GPU code. However, the effect of the imbalance is less because the microphysics takes a smaller percentage of the run time.

4. 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 almost an order of magnitude performance improvement for a small but performance-critical module of 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 7.7× increase in WSM5 performance translates into 1.23× increase in total application

performance (Amdahl's law limits the total increase to 1.3×). A 1.23× improvement in model performance from a few months performance tuning effort is rare. Based on these results, the GPU-accelerated WSM5 microphysics developed here was included as an option in WRF Version 3.0, released to the community in May 2008, making WRF the first GPU-accelerated weather model.

Though 1.23× 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. Thus, the next step for this work applying GPUs to NWP will be to address modules within WRF dynamics which, unlike microphysics and other column physics modules, include horizontal data dependencies for finite differencing and horizontal interpolation. Here, the inability to synchronize blocks of threads running on current-generation GPUs will be an obstacle, since synchronization, when needed, will require multiple kernel invocations. On the other hand, stencil operations in WRF dynamics routines such as advection and diffusion will be able to make use of other levels of the GPU memory hierarchy such as the read-only two-dimensional texture cache. The results presented in this paper show significantly increasing the capacity of the GPU's fast memory resources (e.g. registers and thread-shared memory) will benefit applications with large memory footprints such as NWP. Bandwidth to device memory, currently under 90 GB/second for a theoretical half a peak teraflop, may also become a limiter unless sufficient reuse is available in the application and supported by the device's memory hierarchy. At present, it does not appear such costly non-graphics related improvements are on any GPU manufacturer's near term priority list.

With respect to the ease with which NWP kernels can be programmed and optimized for to an architecture designed for graphics, NVIDIA has made considerable strides toward extending the GPU architecture to more general purpose high-end scientific computing. At present considerable manual effort is required to first translate NWP kernels from Fortran to C/C++ before they can be adapted to CUDA. NVIDIA has stated that CUDA Fortran is on the company's road map for later in 2008. The profiling/hardware performance monitoring currently available in the new SDK provides some basic information needed to fully understand program behavior. It remains difficult, however, to construct a reliable model of the various internal mechanisms and constraints that combine in non-linear fashion to affect performance. Exposing additional hardware performance data will provide significant benefits for understanding and optimizing GPU performance.

Acknowledgments

The authors gratefully acknowledge Sumit Gupta, Jonathan Cohen, Timothy Murry, Lars Nyland, Brent Oster, and others at NVIDIA Corp. for access to GPU

hardware and technical advice. This work was partially supported by the National Center for Supercomputing Applications which made available the NCSA's experimental GPU cluster. Also thanks to Profs. Wen-mei Hwu and John Stone for access and Jeremy Enos for support using the NCSA's cluster.

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] *CUDA Profiler Documentation (distributed with SDK 2.0 beta)*. Santa Clara, California, June 2008.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.