Journal of Computational Interdisciplinary Sciences (2011) 2(1): 3-11

Printed version ISSN 1983-8409 / Online version ISSN 2177-8833
http://epacis.org

A case study on porting scientific applications to GPU/CUDA

Javier Delgado[1], João Gazolla[2], Esteban Clua[2] and S. Masoud Sadjadi[1]

Manuscript received on August 7, 2010 / accepted on January 15, 2011

ABSTRACT

This paper proposes and describes a methodology developed to port complex scientific applications originally written in FORTRAN to nVidia CUDA. The significance of this lies in the fact that, despite the performance improvement and programmer-friendliness provided by CUDA, it presently lacks support for FORTRAN. The methodology described in this paper addresses this problem using a multiple step process that includes identification of software modules that benefit from being ported, familiarization with the code, porting, optimizing, and verifying the ported code. It was developed and carried out by porting an existing module of a weather forecasting application written in FORTRAN. Using this approach, we obtained a functional prototype of the ported module in approximately 3 months, despite our lack of knowledge of the theory of the weather code. Considering the relevance of this application to other scientific applications also written in FORTRAN, we believe that the proposed porting methodology described can be successfully utilized in several other existing scientific applications.

Keywords: GPU, programming, CUDA, weather modeling.

1 INTRODUCTION

Recent work has shown that it is no longer necessary to rely solely on the CPU of computers to perform all of a programs computations. Graphics Processing Units (GPUs), particularly, with their large number of processors, provide a cost-effective solution for high-performance computing (HPC). As an added benefit, modern programming frameworks, such as nVidia's Compute Unified Device Architecture (CUDA) have made programming on the GPU more straightforward and friendly for programmers of high-level languages with basic parallel programming knowledge. However, exploiting the benefits of the GPU architecture complicates programming. Due to the different programming paradigm of GPUs, it is not trivial for many scientific applications to be ported. Since current solutions provide limitations in terms of programming languages, large codebases need to be entirely rewritten in many cases. Specifically, the current implementation of CUDA only supports the C language. In this work, we describe a methodology for easing the process of porting complex scientific software with a large amount of FORTRAN code to CUDA. The software that we used for our case study is Weather Research and Forecasting (WRF), version 3.0 [6]. WRF requires a large amount of computational resources in order to generate useful simulations. Aside from that, some of the main WRF developers themselves have mentioned the need for fine-grained parallelism in numerical weather modeling applications, which prompted them to take the first steps towards GPU enabling WRF [7]. We continue this work by porting more WRF code and formalizing the porting process.

We organize the rest of this paper as follows: In Section 2, we provide a summary of the status of GPU-enabling WRF, in Section 3, we describe our proposed porting methodology, applied

Correspondence to: Javier Delgado – E-mail: javier.delgado@fiu.edu
[1]College of Engineering and Computing, Florida International University, Miami, FL, USA.
[2]Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brazil.

to the WRF problem. In Section 4, we provide results using the GPU-enabled WRF and in Section 5, we conclude this paper.

2 BACKGROUND AND DOMAIN DESCRIPTION

2.1 GPUs and CUDA

CUDA is a parallel computing architecture developed by nVidia Corporation. CUDA is the computing engine in nVidia graphics processing units (GPUs) that is accessible to software developers. It is currently possible to write CUDA applications using an extension of the C programming language.

CUDA allows supported GPUs to be programmable similar to CPUs. Unlike current CPUs, GPUs consist of a parallel many-core architecture, consisting of dozens of cores. Each core can run thousands of threads simultaneously.

2.2 Application domain description

Many approaches have been investigated in order to parallelize scientific problems written in FORTRAN or C to utilize compute clusters and/or grids. FORTRAN is often preferred for highly mathematical code. Many computationally intensive applications are written at least partially in FORTRAN, e.g. Quantum Espresso, WRF [6], MM5 [2], and Elmer. While each application has a particular purpose, many of them share certain characteristics. For example, their inputs and outputs usually consist of multi-dimensional arrays consisting of floating point values. By observing these characteristics, a generally-applicable paradigm can be devised in order to save software engineering effort in future work. We validate the efficacy of this methodology on a well known weather forecasting application.

One of the scientific applications mentioned above, Quantum Espresso, has its most computationally demanding parts written in FORTRAN. After profiling an example, we noted that the function that consumed the most execution time consisted of 4 input variables and 7 output variables, mostly 2-dimensional arrays of floating point values. This characteristic, i.e. processing on several multidimensional arrays, is shared by WRF, which is the application we focus on for this work. We expect similar characteristics in other scientific applications.

The Weather Research and Forecasting (WRF) application, version 3, is the latest version of the next-generation weather model used by the National Oceanic and Atmospheric Administration (NOAA). Contemporary high-performance compute resources have made accurate and detailed weather simulations possible. However, even though there are a large number of tera- and peta-scale systems available today, gaining access to these systems is still either difficult or expensive to obtain.

2.3 Porting WRF: benefits and implications

The fact that large computational resources based on CPUs are still expensive, along with the need for the fine-grained parallelism, makes WRF an ideal candidate to test this methodology. The module we port as a proof-of-concept of our paradigm is a continuation of the work described in [7]. Quick turn-around time for results is particularly important for weather simulations, since adequate time is needed to prepare for such disastrous situations. Therefore, by working on the WRF port, we can validate our methodology while addressing an important scientific problem.

WRF consists of nearly 200,000 lines of code, of which approximately 20% is generated automatically using a Registry, which is based on a computer aided software engineering (CASE) mechanism [9]. Like most high-performance scientific applications, WRF is flexible in terms of parallelism. It supports MPI and OpenMP in order to allow coarse and fine grain parallelism, respectively. WRF uses a separate, high-level parallelism library called Comm-API [5], which supports different parallel communication APIs. CUDA enhances WRFs parallel programming support and performance for significantly faster execution on commodity hardware.

The largest barrier to porting is the fact that the only language supported by CUDA is C. The fact that there exist many applications like WRF has motivated the development of the proposed methodology introduced in the next section. In addition, the fact that CUDA allows incremental porting of an application from CPU to GPU has reinforced our methodology. The WRF codebase is large but modular, which allows the incremental porting model allowed by CUDA to be utilized for piece-wise porting.

The *swrad* module of WRF, which is what we ultimately decided to port, consists of two loops around the i and j dimensions of the Cartesian plane. The calculations inside these loops consist of several short loops through the k dimension. Inside the short loops are several arithmetic calculations and conditional statements. Most of the processing is done on multi-dimensional arrays of floating point values. Since we are not domain experts, understanding the code is challenging. However, porting the calculations and conditional statements is trivial, albeit error prone. Our approach to porting is not hampe-

red by our lack of domain knowledge since we simply test by probing the values of certain variables during our incremental porting process.

2.4 The need for a formal methodology

We would like to emphasize the point that the the purpose of this paper is to describe a formal methodology for porting scientific programs to CUDA. The specific project of porting WRF to CUDA is not new, but we considered WRF an ideal candidate and thus continued on the work discussed in [7], which introduces many of the ideas of the porting paradigm and problems. As a result, in this work we generalize and formalize their approach. We intend to work on other software applications in the future to further demonstrate the viability of this approach. We note that after the above work was published, two additional modules have also been ported, for a total of three CUDA-enabled modules. Profiling a standard WRF run with various domains revealed that the modules with the longest running time have been ported. The authors mention an additional two modules that would benefit from using GPUs. The five modules listed on their project page are the following: WSM-5 (*wsm5*), Tracer Advection, Chemistry, Shortwave radiation (*swrad*), and Long-wave radiation (*lwrad*). Cloud microphysics is responsible for accurately modeling the effect of rain, ice, and snow as described in [4]. Tracer advection models the transport of atmospheric constituent scalars under the force of wind as described in [9]. The *swrad* and *lwrad* modules model radiative transfer as described in [5]. In the case study described in Section 5, we describe the decision to port the *swrad* module.

3 PROPOSED METHODOLOGY

Our proposed approach involves incrementally porting parts of the code and testing by generating output files containing the values of the variables being modified. Basically, our methodology divides the process of porting into 4 different stages: profiling, development, testing, and optimization. Since porting is performed on a per-module basis, this approach follows an incremental software engineering process.

The overall procedure is depicted in Figure 1. The two versions of the code are separately executed. The output and/or state variables data is written to a file. This data may be in a raw/binary format. In the case of WRF, their values are a binary dump of the FORTRAN variables. These files are passed into a generator that creates text-based output in a uniform format.

3.1 Profiling

Most large programs spend the majority of their time in small sections of code. This is also true of scientific programs. Thus, using a code profiler that tells the user where the program is spending most of its time should be the first step in the porting process. After performing this step, the developer will know what code segment(s) should be ported first.

3.2 Porting the code

Our methodology focuses on porting code that presumably has been tested and is production-ready. Such is the case for the WRF modules we analyzed: since many of these applications have been developed for a long time, the code is presumably robust. As a result, our model needs to effectively test the output of the ported code. However, the limited I/O operations supported by CUDA complicates this process. A partial workaround for this is to first port the code to (CPU-only) ANSI C, test it, and then port to CUDA. This way, much of the testing could be done in the middle stage of the implementation (in which all the code is in C). However, the process of porting the code to CUDA requires significant effort and is prone to error, so additional testing is still required to port to CUDA from ANSI C.

Porting code directly from FORTRAN to CUDA is a time-consuming, error-prone affair. Even porting existing C code to CUDA is difficult. For example, the programmer is not able to access the GPU memory directly; the data must be copied to a temporary variable on the host (i.e. main memory) before their value is read and/or modified. Another problem is that there is no dynamic allocation of memory in GPUs – the programmer must know the amount of data needed for each variable and preallocate it at the GPU. Another factor to consider is the highly-parallel execution model. In CUDA, each kernel (i.e. program running on the GPU) is responsible for many blocks, each consisting of thousands of threads. The launch of a kernel is an asynchronous process in which there is no guarantee about the order of the threads' execution, making it almost impossible to debug on-the-fly.

If porting from FORTRAN to CUDA, the added burden of allocating memory, copying data, updating memory, and retrieving data makes the likelihood of introducing bugs greater. CUDA provides an emulation mode, which causes threads to be serialized. However, the emulation mode will not reveal all bugs, since kernel calls are asynchronous when executed on the GPU. Since there is no support for dynamic memory allocation, the programmer has to focus on porting and memory manage-

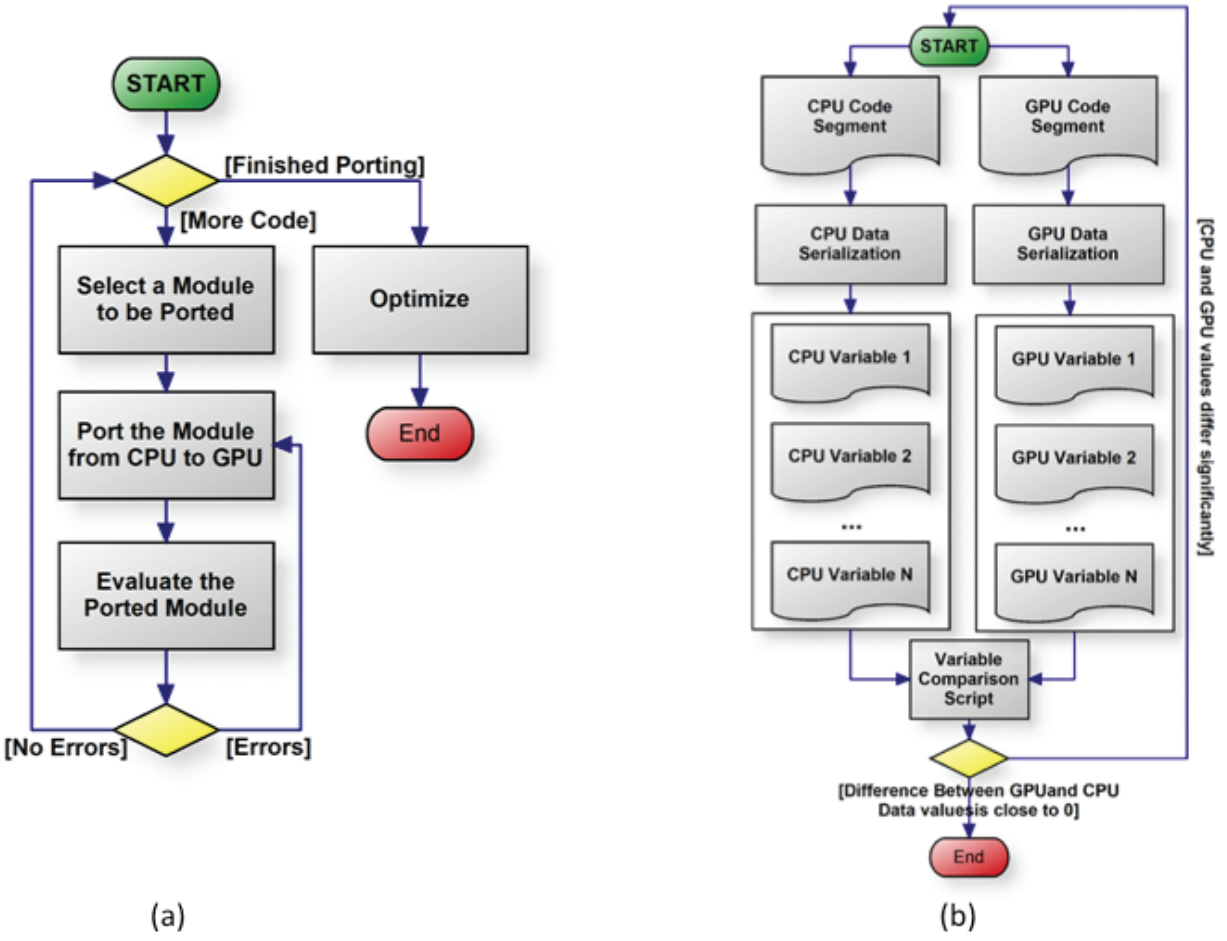

Figure 1 – Overview of the porting methodology used. (a) shows the overall methodology. (b) shows the details of the "Evaluate the Ported Module" block.

ment simultaneously. If the program being ported is written in serial, the parallelization complexity compounds the porting. In the specific case of porting from FORTRAN to C, there are general complexities to be addressed. For example, array indexing is different between the two languages, which makes porting bug-prone, and off-by-one errors do not necessarily reveal themselves in all executions.

Taking the above factors into account, it seems that porting large sets of code directly to CUDA will result in more problems and thus a longer development time than porting the code in two steps. However, many of these problems are faced when porting to CUDA from C as well. Memory allocation, for one, is equally difficult. Also, the difference in array indexing is still a problem; in fact, the indexing of data is being addressed by a different virtual processor within a virtual processor block, so array indexes are determined as a function of the (thread) index within the block corresponding to the active processor. The relation between the thread index and the global index (in main memory) needs to be computed using a complex formula, so indexing is quite complicated in CUDA even for programs written in C. Another major difficulty is that bugs in CUDA programs do not reveal themselves as well as in CPU-run programs. Data in variables may become corrupted without any error or warning being reported. In C, segmentation faults often expose these coding errors.

Our assumption is that development time is reduced if the porting is done directly to CUDA, while mitigating some of the general porting related issues.

3.3 Testing the ported code

To port individual modules that are normally not standalone applications, it is best to implement a standalone version of the module rather than performing an entire simulation to test the single module. To obtain input data for testing, the module can

be modified to print the values of its input variables while performing a full simulation. This output can be directed to a file that can later be used as input data. A test driver may then be developed that executes both versions of the code, with the same input, and compares their output.

The applications we target with this approach typically have large output data sets. Round-off error usually occurs when working with floating point numbers due to the different orders of operations on highly-parallel systems as well as non-standardized floating point rounding specifications. This issue has been particularly common in GPU programming, since GPUs have the added problem of not supporting the IEEE Floating point standard [3]. Therefore, a mechanism for testing the output of the CPU and GPU versions of the code needs to be developed, and simple (bit-wise) comparisons will not work due to round-off error.

Some research and/or application domains have standard metrics of assessing correctness and/or quality. For example, in image processing, Peak Signal to Noise Ratio (PSNR) is typically used. Weather forecasting output has no such metric. WRF includes a tool, *diffwrf*, that is able to compare the final output of two domains, but it does not work on any intermediate output, so it is not useful for testing individual modules. Running an entire simulation to test a single module is not a good solution since it is time-consuming, which increases development time. A more general solution that can be applied to other applications would be beneficial. This makes testing the output of the program itself the optimal solution. Writing the values of output/state variables to a file and using external tools to compare them is an efficient solution for this.

To measure the similarity between the CPU and GPU outputs, text-based and graphical tools can be used. The text-based tools provide a quick quantitative similarity score. However, with large arrays, the statistics calculated and displayed by a text-based tool may be inadequate. In these cases, graphical output is the most revealing. We have found difference plots to be particularly well suited for this. Difference plots show the relation between the difference of values and their means. This is ideal for these kinds of scientific applications since many parameters, with a wide range of values, need to be plotted.

The output generated by the output generator consists of data that can be read by a 2-dimensional array viewer. This format is useful for text-based processing. It can also be used by some graphical tools with little or no modification. For example, the files can be readily used by gnuplot.

3.4 Optimization

After ensuring correctness of the code, it is necessary to optimize it. Optimization in GPU architectures is still a relatively new field of study. The complexity and size of the GPU architecture, particularly the memory hierarchy and distribution of processors makes optimization difficult. Two underlying issues need to be addressed. The first is that the algorithm is efficient. The other is that the runtime configuration (e.g. number of blocks and threads per block) is efficient and matches the target hardware. CUDA has eased the burden by providing profiling tools with their toolkit. Some general optimization schemes have been described in the literature [8].

While every application requires specific tweaks to achieve optimal performance, a general first set of steps can be taken to start the optimization process. We devised a set of code design guidelines and ensured they were met. The approach in [8] provides a general methodology of minimizing the optimization space for CUDA programs, which saves optimization time. Their techniques are applicable to different programs, although they only test on four different ones.

3.4.1 Profiling and other computer-aided analysis

Profiling at the kernel level provides valuable feedback about the resource utilization of the kernel. It is essential for determining bottlenecks and inefficiencies. With multiple-function kernels, profiling quickly reveals the greatest resource consumers. CUDA provides a basic profiler for recording global and local memory usage, number of instructions, number of branches, thread information, and the ratio of CPU-time to GPU-time. Manual analysis with the CUDA debugger can reveal more in-depth information.

Another computer-assisted analysis tool is provided by CUDA by means of intermediate files that are generated when compiling CUDA kernels. There are two particular files of interest. The first is a PTX file, which shows the actual amount of machine instructions and can indicate which memory is being used for some of the variables. The other is the *cubin* file, which gives a rough indication of register usage, as well as shared, and local memory usage.

After compiling the kernel and retaining all of the intermediate data generated by the CUDA compiler, the *cubin* output revealed that the amount of data used is quite small. Just 2,356 bytes of local memory and 224 bytes of shared memory were used per thread block and 60 registers per thread. The Tesla nodes contain 16384 bytes of shared memory and 16,384 regis-

ters. Some of this information is used in the Manual Analysis section that follows.

3.4.2 Manual analysis

In order to determine the optimal number of multiprocessors and blocks per thread, an exhaustive test with multiple configurations was performed. This was possible since the kernel executes quickly (less than 30 minutes were required for the search).

Once the optional runtime configuration was determined, a checklist of optimizations that can be carried out by inspecting source code and analyzing execution time with different inputs was devised.

- **Take advantage of the shared memory.** The CUDA architecture has different kind of memories, including global and shared memory. Access to shared memory is many times faster than global memory. For certain kinds of access it is similar in performance to registers. Due to its relatively small size (e.g. 16 kB per multiprocessor in the GT200 architecture), it must be utilized carefully. Threads on the same multiprocessor (block) can cooperate and access this memory, which allows inter-thread data reuse.

- **Take advantage of the registers.** Registers are the fastest kind of memory on the multi-processor. Registers are only accessible by their corresponding threads. Registers provide a total of 64 kB of memory. If the amount of the shared memory is not enough for an application, it is possible to use register memory instead. However, register memory has the caveat that the data in it cannot be indexed (i.e. arrays cannot be used).

- **Load balancing.** As with any parallel program, it is important that the load is equally balanced amongst the executers.

- **Optimize global memory access.** Global and local memories provide a much larger storage space. Even though they are slower, there are things the programmer can do to minimize the performance impact of using these slower memories. The main technique is to ensure that memory is accessed in a coherent/coalesced manner. When this is done, contiguous memory can be transferred in parallel by different threads. This saves several compute cycles, depending on the type of variable(s). To ensure coalesced access, local memory can be used, which is coalesced by default. All constant-sized arrays are stored in local memory.

- **Minimize message transfers.** Communication between the system memory and the PCI Express slot can easily become a bottleneck. For example, a PCI Express Bus has a speed of 8 GB/s (i.e. it can allocate 2 Giga-words of 4 bytes per second). During one operation, data must be sent from the CPU to the GPU, one operation performed and data must be copied back from the GPU to the CPU. So the 2 Giga-words limit drops to 1 Giga-word. Because thousands of threads may be running, bandwidth is further limited. Basically, it is necessary to ensure as high a ratio as possible of arithmetic operations to memory operations.

- **Maximize occupancy for bandwidth-limited codes.** The CUDA profiler revealed that the occupancy was only 0.25. This leaves a lot of room for improvement. Again, the fact that the amount of data that needs to be transferred is so large seems to be a culprit here. There are two possible solutions to this: one is to delay the transfer of parts of the variables in the kernel. The other, with large codes like WRF, is to perform the transfer while other modules are executing and/or blocking. Both techniques add complexity to the algorithm of the code.

- **Mask latency.** Latency is masked by ensuring multiple threads are available to compute while others are performing I/O.

4 CASE STUDY: APPLYING THE PORTING METHODOLOGY TO WRF

4.1 Profiling

Several instrumented executions of WRF were run to determine what module to port/optimize. To evaluate the performance of the different modules under different execution platforms, three different systems were used. Furthermore, three different WRF inputs domains were used. The systems are shown in Table 1. Table 2 describes temporal and geographical properties of the WRF input domains used. Figure 2 shows the percentage of execution time used by the most time-consuming functions. The results show that the ones that presumably benefit most from the added power of GPUs are the cloud microphysics (*wsm52d*) and scalar advection modules, corresponding to over 25% of the

Table 1 – Description of systems used.

Name	CPU	RAM	GPU	num cores	Clock
Vaia	C2D 2.26 GHz	2 GB	9300M	16	1.1 GHz
Minerva	C2D 2.26 GHz	2 GB	9400M	16	1.1 GHz
Lincoln	Xeon 2.33 GHz	16 GB	Tesla S1070	240	1.5 GHz

Table 2 – Description of WRF domains.

Domain name	Length (km)	Width (km)	Resolution (km)	Simulation time (h)
Jan00	1830	2220	30	12
1500×15	1500	1500	15	24
2000×15	2000	2000	15	24

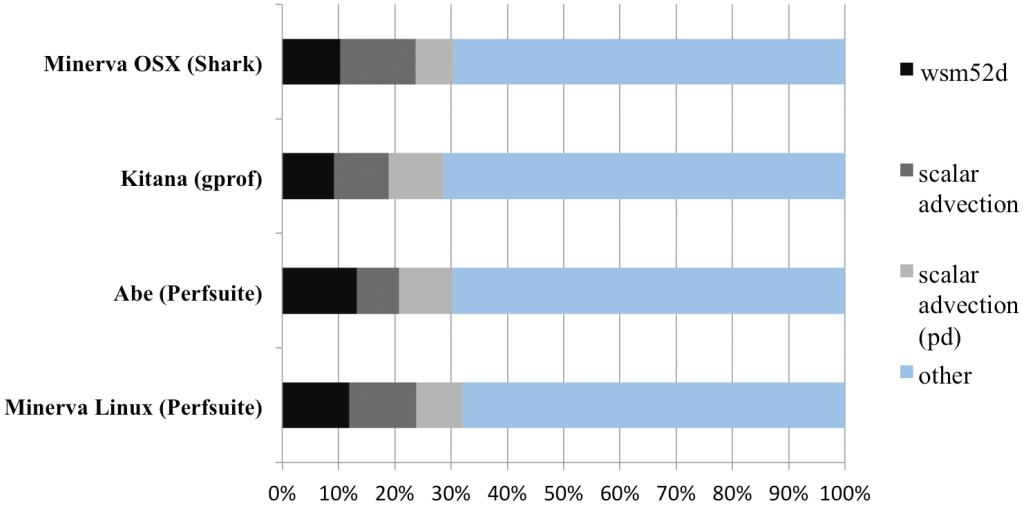

Figure 2 – Breakdown of execution per function in a WRF simulation, using the systems tabulated in Table 1 and the domains tabulated in Table 2.

total execution time. There was already ongoing work for porting these two modules, so we port the *swrad* module, although the *wsm5* module was used for certain optimization steps.

Generally, porting the module that takes the most time results in the most speedup. However, this is not always the case. For example, in [7] the authors found that the overall speedup after porting the WSM5 module of WRF was larger than the individual speedup of the isolated module. Their assessment as to why this happened is because it was the most load-imbalanced module of the application. When executing with 16 nodes of the NCSA Abe cluster, we observed that the percentage of time for the WSM5 module varied between 7.9% and 16.2%. This was considerably more than the next highest varying function, which had a range of 6.5%-11.3%. Note that these values were obtained using the *jan00* domain.

The results shown in Tables 1 and 2, and the diagram shown in Figure 2, assume all other WRF input parameters are constant. WRF allows for a large number of runtime configuration differences corresponding to different meteorological theories. The application profile information in Figure 2 corresponds to executions with the most basic form of short-wave radiation physics and a radiation time step of 30 seconds (which is the recommended value for the input domain used). Since short-wave radiation is listed as one of the modules to port, we did further profiling with different short-wave radiation schemes as well as different radiation periods. We found that this had a significant effect on

the execution time. As a result, in the future it would be beneficial to port the more computationally-intensive radiation schemes.

4.2 Porting the code

The authors of [7] addressed some of the problems of porting by using a framework called "spt" consisting of a preprocessor that processes certain macros in the (ported) code that abstract some of the porting effort. The macros encapsulate the overhead of memory allocation and transfer to and from the GPU. They also encapsulate the array index addressing issue of the CUDA model. The macros and preprocessor are described in more detail in [7]. Among other things, the preprocessor automates the addressing of array indices (of two and three dimensional arrays) at the GPU for variables with a specified compiler directive specified in their declaration. The programmer only needs to use basic array indexing as if a loop existed. This framework eases the porting process and applying it in general to other applications should be feasible.

4.3 Testing the ported code

To test the ported code, we use difference plots that compare the original and CUDA versions of the output variables of *swrad*.

4.4 Optimization

Since our profiled runs showed that the *wsm5* module takes a more significant percentage of execution time than *swrad*, we chose it as the test bed for our optimization approach. We combine computer-assisted and manual analysis to perform the optimization.

For the manual analysis, we used the "checklist" in Section 3.4.2. The first step was ensuring that an optimal memory placement scheme was being used. The large amount of input data used by *wsm5* limits the amount of memory that can be put into shared memory. The SPT preprocessor described in [7] makes placement convenient. The current port, for example, benefits from putting often-used variables in registers. Ensuring coalesced memory access is difficult because array indices in WSM-5 are set based on the size of the physical domain being modeled, which is determined at run time. The workaround to this is to ensure that access to the global variables is coalesced. Output from the CUDA profiler revealed that all global memory was being loaded and stored in a coalesced manner. Possibly the biggest deterrent to performance for *wsm5* is message transfer overhead. The ratio of arithmetic operations to memory operations is approximately 1:350, which is undesirable. This is caused by the large amount of input data. Optimizing this is necessary, but it is beyond the scope of this paper.

5 RESULTS AND DISCUSSION

The project, whose goal was to port a module of WRF to CUDA, was performed in 3 months. This includes trial and error, learning about the software itself, and actually converting the code to CUDA. A significant percentage of this time was spent learning about the code itself as part of a different project. Neither of the developers involved were meteorologists nor domain experts in any way. Reading documents such as [4] gave us a slightly better understanding of the code, but not enough to fully comprehend what it was doing. For this reason, the porting approach relies on using the values of output variables to verify the code. A simple binary compatibility test was not enough due to the floating point issues mentioned earlier, but the difference plot provided an efficient method of ensuring that the output was correct. The resulting paradigm discussed in this paper describes the final methodology that was agreed on.

The fastest overall execution time of the ported *swrad* module on *Minerva* was 9.6ms, compared to 20ms for the CPU version. The performance improvement is good, considering the coding effort put in. However, there is still a large room for improvement. As with the *wsm5* module, the vast majority of the execution time is spent on data transfer – only 0.069ms are spent in the actual kernel computation in the optimal case. Since the variables used for *wsm5* and *swrad* are not the same, the decrease in overall execution time for an entire simulation is just the sum of the time saved from each module. The fact that the performance on the commodity 9400m GPU achieved faster times than the Tesla node makes it clear that there is big room for improvement.

Since a rigorous optimization process is beyond the scope of this paper, we did not look any further into the optimization aspects. Discovering techniques to optimize applications like WRF that require a lot of data transfer would be an interesting future topic to research.

6 FINAL COMMENTS

We have described an approach to porting complex scientific applications to CUDA. The methodology we propose attempts to save development effort by specifying a simple iterative approach to porting that does not require intimate knowledge of the application being ported. By employing it on a module of a well-known weather forecasting application, we were able to speed up the module by more than a factor of two, in a relati-

vely small amount of time. The performance improvement was experienced by virtue of the GPUs relatively large computing power, but there still exists a large amount of potential that is not being exploited due to the chosen applications particular problem of having a large data size to computation ratio. Future work will emphasize the performance optimization aspects of the porting process.

ACKNOWLEDGMENTS

This work was supported in part by the NSF for the support on the PIRE, GCB, and CREST projects (NSF grants OISE-0730065, OCI-0636031, and HRD-0833093), and using Teragrid [1] resources provided by NCSA and in part by IBM. Also, the authors from UFF would like to thank for the support of CAPES, CNPq and FAPERJ, from Brazil.

REFERENCES

[1] CATLETT et al. 2007. TeraGrid: Analysis of Organization, System Architecture, and Middleware Enabling New Types of Applications, HPC and Grids in Action, Ed. Lucio Grandinetti, IOS Press "Advances in Parallel Computing" series.

[2] GRELL GA, DUDHIA J & SAUFFER DR. 1994. Description of the fifth generation Penn State/NCAR Mesoscale Model (MM5). NCAR Tech. Rep., TN-3981STR: 121.

[3] HILLESLAND KE & LASTRA A. 2004. GPU floating-point paranoia. GP2 ACM Workshop on General Purpose Computing on Graphics Processors: 8.

[4] HONG SY, DUDHIA J & CHEN SH. 2004. A revised approach to ice microphysical processes for the bulk parameterization of cloud and precipitation. Mon. Weather Rev., 132: 103–120.

[5] IACONO MJ, MLAWER EJ, CLOUGH SA & MORCRETTE JJ. 2000. Impact of an improved longwave radiation model, RRTM, on the energy budget and thermodynamic properties of the NCAR Community Climate Model. J. Geophys. Res., 105: 14,873–14,890.

[6] MICHALAKES J, DUDHIA J, GILL D, HENDERSON T, KLEMP J, SKAMAROCK WC & WANG W. 2004. The Weather Reseach and Forecast Model: Software Architecture and Performance. In: Proc. 11th ECMWF Workshop on the Use of High Performance Computing In Meteorology: 25–29.

[7] MICHALAKES J & VACHHARAJANI M. 2008. GPU Acceleration of Numerical Weather Prediction. Parallel Processing Letters 18,4: 531–548.

[8] RYOO S, RODRIGUES C, STONE S, BAGHSORKHI S, UENG S, STRATTON J & HWU W. 2008. Optimization space pruning for a multithreaded GPU. International Symposium on Code Generation and Optimization.

[9] SKAMAROCK WC, KLEMP JB, DUDHIA J, GILL DO, BARKER DM, WANG W & POWERS JG. 2005. A Description of the Advanced Research WRF Version 2. NCAR/TN- 468+STR.