An Incremental Approach to Porting Complex Scientific Applications to GPU/CUDA

Javier Delgado^2, João Gazolla^1, Esteban Clua^1, S. Masoud Sadjadi^2

^{1}Instituto de Computação – Universidade Federal Fluminense, Niterói , RJ , Brazil
^{2}College of Engineering and Computing – Florida International Univ. Miami, U.S.A.

{gazolla,esteban}@ic.uff.br, {jdelga06,sadjadi}@fiu.edu

Abstract. *This paper proposes and describes a developed methodology to port complex scientific applications originally written in FORTRAN to the nVidia CUDA. The process was developed and validated by porting an existing FORTRAN weather and forecasting algorithm to a GPU parallel paradigm. We believe that the proposed porting methodology described can be successfully utilized in several other existing scientific applications.*

1. Introduction

Recent work has shown that it is no longer necessary to rely solely on the CPU of a computer to perform all of a program's computations. Graphics Processing Units (GPUs), 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 to CUDA. Since current solutions provide limitations in terms of programming languages, large codebases need to be entirely rewritten in many cases.

In this work, we describe a methodology for easing the process of porting software with a large amount of code to CUDA. The software that we used for our case study is Weather Research and Forecasting (WRF), version 3.0. WRF requires a large amount of computational resources in order to generate useful simulations. Aside from that, in [Michalakes and Vachharajani 2008] the authors describe the need for the fine-grained parallelism, which GPUs can provide, for numerical weather modeling. The module we port as a proof-of-concept of our paradigm is a continuation of the work described in [Michalakes and Vachharajani 2008].

We organize the rest of this paper as follows: In Section II, we provide a summary of the status of GPU-enabling WRF, in Section III, we describe our proposed porting methodology, applied to the WRF problem. In Section IV, we provide results using the GPU-enabled WRF and in Section V, 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 through widely used programming languages. Currently, programming in CUDA is possible using an extension of the C programming language.

2.1. 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 [Michalakes and Dudhia 2004], MM5 [Grell and Dudhia 1994], and Elmer. Most approaches are specific to the target application, yet many applications share certain characteristics. For example, their inputs and outputs usually consist of multi-dimensional arrays consisting of floating point values. By observing these characteristics, and noting the popularity of FORTRAN, a generally-applicable paradigm can be devised in order to save software engineering effort in future works.

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 [Skamarock 2005]. 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 [Iacono 2000], which supports different parallel communication APIs. CUDA enhances WRF's for WRF parallel programming support and performance for significantly faster execution on commodity hardware.

Quantum Espresso is another scientific application with 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 is shared by most modules of WRF.

The *swrad* module of WRF, which is what we ultimately decided to port, consists of two loops around the Cartesian plane. The calculations inside these loops consist of several short loops through one of the dimensions. 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. Without a not domain expert's assistance, understanding the code is challenging. However, porting the calculations and conditional statements is trivial. A porting approach must take this into account.

Porting WRF to CUDA is time consuming since 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 (i.e., it is not necessary to port the entire application to execute exclusively on the 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.

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 Fig. 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

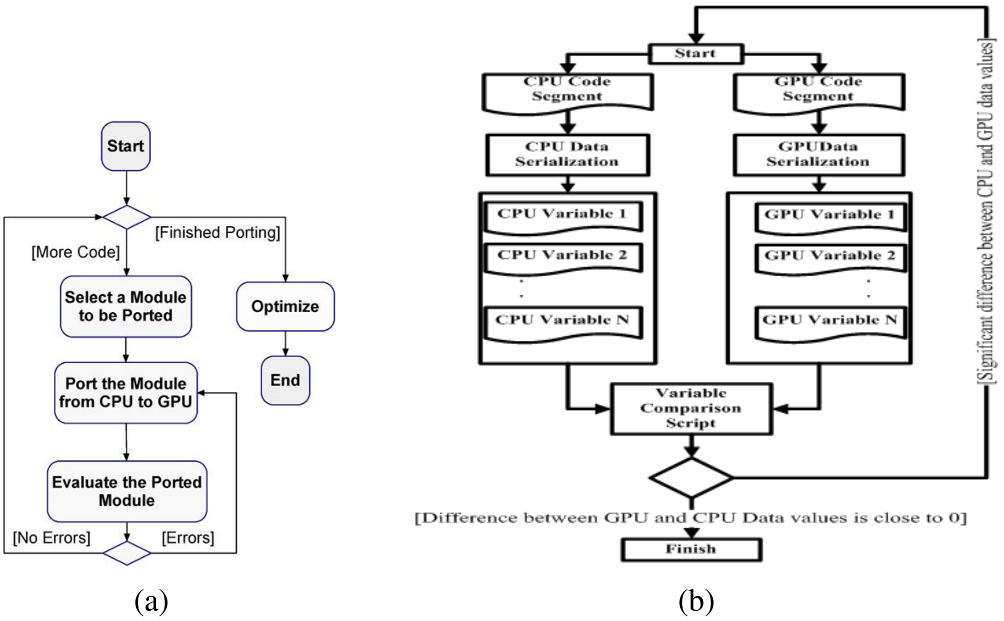

Fig. 1. Overview of the porting methodology used. (a) shows the overall methodology. (b) shows the "Evaluate the Ported Module" procedure.

Several instrumented executions of WRF were run to determine what module to port/optimize. Table 1 shows the computing systems that were utilized for the profiled executions. Table 2 describes temporal and geographical properties of the WRF input domains used, and Fig. 2 shows the percentage of execution time used by the most time-consuming functions, the one presumably benefit most from the added power of GPUs are the cloud microphysics (*wsm52d*) and scalar advections, corresponding to over 25% of the total execution time.

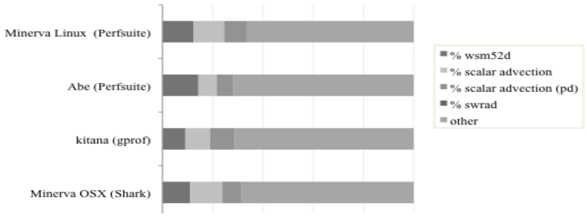

Fig. 2. Breakdown of execution per function in a WRF simulation, using different systems.

Generally, porting the module that takes the most time results in the most speedup. However, this is not always the case. For example, in [Michalakes and Vachharajani 2008] 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.

Table 1. Description of Systems Used

Name	CPU	RAM	GPU	#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
1500x15	1500	1500	15	24
2000x15	2000	2000	15	24

The results shown in Tables 1 and 2 and in Fig. 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 Fig. 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.

3.2. Development Approach

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 complicate 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 after porting to CUDA from ANSI C.

Even porting existing C code to CUDA is difficult. One of the difficulties faced is the fact that 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 pre-allocate it at the GPU. Another factor to consider is the highly-parallel execution model, where there is no guarantee about the order of thread execution.

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 management 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.

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 and actually the indexing doing in C is not used in CUDA; this can result in a lot of wasted effort.

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. The authors of [Michalakes and Vachharajani 2008] addressed some of the problems 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 [Michalakes and Vachharajani 2008]. 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, applying it in general to other applications should be feasible.

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 [Hillesland 2004]. 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.

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 well suited for this. Difference plots show the relation between the difference of values and their means. This is ideal for our application since many parameters, with a wide range of values, need to be plotted.

3.4. Optimization

After ensuring correctness of the code, it is necessary to optimize it. 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 [Ryoo 2008].

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 [Ryoo 2008] provides a general methodology of minimizing the optimization space for CUDA programs, which saves optimization time.

Profiling and Other Computer-assisted 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 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.

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 and applied to WSM-5, as follows.

- **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. In the case of WSM-5, the large amount of input data limits the amount of memory that can be put into shared memory. The SPT preprocessor described in Section 3B allows developers to easily decide which variables should go in the registers and does this.

- **Optimize global memory access**. Programmers can minimize the performance impact of using 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. Array indices in WSM-5 are sized 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.
- **Minimize Transfers** – Communication between the system memory and the PCI Express slot can easily become a bottleneck. 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. In WSM-5, this ratio is approximately 1:350, which is very bad. This is caused by the large amount of input data.
- **Maximize Occupancy for bandwidth-limited codes**. The CUDA profiler revealed that the occupancy was only 0.25. 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, if possible. 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. Results and Discussion

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 WSM-5 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 WSM-5 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 proves that there is room for improvement.

5. Conclusion

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, without having to invest an unreasonable amount of time to do so. 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 application's particular problem of having **a large** data size to computation ratio. **Future work will emphasize improving performance.**

8. Acknowledgment

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 in part by IBM and TeraGrid. Also, the Brazilian authors would like to thank for the support of CAPES, CNPq and FAPERJ, from Brazil.

7. References

Michalakes, J. and Vachharajani, M. (2008) "GPU Acceleration of Numerical Weather Prediction", Parallel Processing Letters. Vol. 18 No. 4. World Scientific. pp. 531-548.

Michalakes, J., Dudhia, J., Gill, D., Henderson, T., Klemp, J., Skamarock, W. and 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, p. 25-29, Reading U.K.

Grell, G.A., Dudhia, J. and Sauffer, D. R. (1994) "Description of the fifth generation Penn State/NCAR Mesoscale Model (MM5)", NCAR Tech. Rep., TN-3981STR, pp. 121, Natl. Cent. for Atmos. Res., Boulder, Colo.

Skamarock, W.C., Klemp , J. B., Dudhia , J., Gill , D. O., Barker , D. M., Wang ,W. and Powers , J. G. (2005) "A Description of the Advanced Research WRF Version 2" NCAR/TN- 468+STR.

Iacono, M.J., Mlawer, E. J., Clough, S. A. and Morcrette, J. J. (2000) "Impact of an improved longwave radiation model, RRTM, on the energy budget and thermodynamic properties of the NCAR Community Climate Model" CCM3, J. Geophys. Res., 105, 14,873–14,890.

Hillesland, K. E. and Lastra, A. (2004) "GPU floating-point paranoia" In GP2 ACM Workshop on General Purpose Computing on Graphics Processors, p.8.

Ryoo, S., Rodrigues, C., Stone, S., Baghsorkhi, S., Ueng, S., Stratton, J., Hwu, W. (2008) "Optimization space pruning for a multithreaded GPU", in: International Symposium on Code Generation and Optimization, CGO.