NASA/TM—2016-218947

Utilizing GPUs to Accelerate Turbomachinery CFD Codes

Weylin MacCalla
Embry-Riddle Aeronautical University, Daytona Beach, Florida

Sameer Kulkarni
Glenn Research Center, Cleveland, Ohio

January 2016

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program plays a key part in helping NASA maintain this important role.

The NASA STI Program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI Program provides access to the NASA Technical Report Server—Registered (NTRS Reg) and NASA Technical Report Server—Public (NTRS) thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counter-part of peer-reviewed formal professional papers, but has less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., "quick-release" reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at http://www.sti.nasa.gov

- E-mail your question to help@sti.nasa.gov

- Fax your question to the NASA STI Information Desk at 757-864-6500

- Telephone the NASA STI Information Desk at 757-864-9658

- Write to:
  NASA STI Program
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681-2199

NASA/TM—2016-218947

Utilizing GPUs to Accelerate Turbomachinery CFD Codes

Weylin MacCalla
Embry-Riddle Aeronautical University, Daytona Beach, Florida

Sameer Kulkarni
Glenn Research Center, Cleveland, Ohio

National Aeronautics and
Space Administration

Glenn Research Center
Cleveland, Ohio 44135

January 2016

Acknowledgments

The authors would like to thank Dr. Mark Celestina and Mr. Richard Mulac for their insight and advice throughout the course of the project.

Level of Review: This material has been technically reviewed by technical management.

Available from

NASA STI Program
Mail Stop 148
NASA Langley Research Center
Hampton, VA 23681-2199

National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161
703-605-6000

This report is available in electronic form at http://www.sti.nasa.gov/ and http://ntrs.nasa.gov/

Utilizing GPUs to Accelerate Turbomachinery CFD Codes

Weylin MacCalla*
Embry-Riddle Aeronautical University
Daytona Beach, Florida 32114

Sameer Kulkarni
National Aeronautics and Space Administration
Glenn Research Center
Cleveland, Ohio 44135

Abstract

GPU computing has established itself as a way to accelerate parallel codes in the high performance computing world. This work focuses on speeding up APNASA, a legacy CFD code used at NASA Glenn Research Center, while also drawing conclusions about the nature of GPU computing and the requirements to make GPGPU worthwhile on legacy codes. Rewriting and restructuring of the source code was avoided to limit the introduction of new bugs. The code was profiled and investigated for parallelization potential, then OpenACC directives were used to indicate parallel parts of the code. The use of OpenACC directives was not able to reduce the runtime of APNASA on either the NVIDIA Tesla discrete graphics card, or the AMD accelerated processing unit. Additionally, it was found that in order to justify the use of GPGPU, the amount of parallel work being done within a kernel would have to greatly exceed the work being done by any one portion of the APNASA code. It was determined that in order for an application like APNASA to be accelerated on the GPU, it should not be modular in nature, and the parallel portions of the code must contain a large portion of the code's computation time.

Nomenclature

CFD	Computational Fluid Dynamics
CPU	Central Processing Unit
dGPU	Discrete Graphics Processing Unit
GPGPU	General Purpose Computing on the Graphics Processing Unit
GPU	Graphics Processing Unit
Kernel	A portion of code that is run on the GPU

I. Introduction

Graphics processing units (GPUs) are typically used to drive the display of a computer. They are uniquely suited to this because of their ability to run many threads at once. This is ideal for driving a display that is made up of many pixels, which require many threads. This many-threaded ability of the GPU also makes it suitable for general purpose computing. GPUs have been used to speed up many parallel applications that are able to utilize the single instruction multiple thread (SIMT) architecture of the GPU.

*NASA Glenn Research Center, intern from Embry-Riddle Aeronautical University.

Computational fluid dynamics (CFD) codes have been a large part of general purpose computing on GPUs (GPGPU). CFD codes have to do large amounts of computations on large data sets, often in parallel. Researchers have been able to achieve impressive speedups of CFD codes on the GPU relative to CPU runtimes, with some implementations able to achieve 100x decrease in runtime (Ref. 1).

This work focuses on accelerating a turbomachinery CFD code used by researchers in the Turbomachinery and Turboelectric Systems Branch at NASA Glenn Research Center in Cleveland, Ohio. The code, called APNASA (Refs. 2 to 4) is a legacy CFD code that was developed in the late 1980s. It is well validated and researchers are familiar with it. APNASA is a three-dimensional, steady-state, time-average Navier-Stokes code for multistage compressor analysis which solves the average passage system of equations. Because it is used often, it is desired to speed up the runtime of the code to be able to more quickly analyze compressor geometry designs. This work uses the parallel abilities of the GPU in an attempt to speed up APNASA, while also investigating the potential to speed up legacy CFD codes in general.

Two hardware systems are primarily used during the course of this work. The first is equipped with 16 Intel Sandy Bridge cores, and 16 GB of 1600 MHz DDR3 RAM. This system uses an NVIDIA Tesla K40 GPU. The other system has an AMD A10-7850 Kaveri accelerated processing unit, which is composed of four Steamroller CPU cores, and eight Radeon GPU cores. This system is equipped with 8 GB of 1866 MHz DDR3 RAM.

The remainder of this paper is organized as follows: Section II describes the approach to GPU acceleration taken during the work. Sections III and IV discuss the software and hardware used during the project, respectively. Section V examines memory management within GPU computing. Section VI describes the tests performed on the hardware systems, and Section VII discusses the results of those tests. Section VIII describes an investigation to determine the feasibility of GPU computing for the hardware systems used. Section IX discusses an issue in the GPU programming software that was used, and the steps that were taken to try to correct it, followed by conclusions in Section X.

II. Approach

While speeding up legacy codes is desired, re-writing them is not. These codes have the advantage of being well-validated pieces of software, and writing GPU-centric codes could introduce new bugs. Additionally, writing new codes is a time consuming process that is difficult to justify when there is already a tool for the job. In the interest of keeping new bugs from appearing within the code throughout the project, the source code was modified as little as possible, and there was not significant restructuring of the code.

A few methods have been used to accelerate legacy codes using the GPU. These include creating a compiler that re-writes the entire code base in a GPU programming language (Ref. 5), or re-writing and porting sections of the code to the GPU for acceleration (Ref. 6). Creating a compiler is a very time consuming process, and is specific to the formatting of the code. It was determined to be out of the scope of the current effort. Instead, small, compute intensive and highly parallel parts of the code were identified and ported to the GPU for acceleration.

The process that was used to port code to the GPU started with profiling the code. Then, once the most time consuming parts of the source code were found, they were investigated for parallelism. The most parallel parts were ported to the GPU, and then the GPU kernels were optimized. This was an iterative process that was designed to incrementally speed up the code.

III. Software

There were two types of software tools used during the project; profiling software, and GPU programming software. Each are explained in more detail below.

A. Profiling Software

Profiling an application involves timing the running code to determine where the most time is spent. Through the duration of the current effort, three software profiling tools were used, each for different purposes:

GNU Profiling Tool (GPROF) (Ref. 7)

The GNU Profiling tool is part of the GNU compiler package which is freely available under the GNU General Public License. It was used to take the initial profile of the code, and was able to provide subroutine-level results about the most time consuming parts of the code. The information from the GNU Profiler was used to narrow down the search for the sections of code to be offloaded to the GPU.

PGPROF (Ref. 8)

Once the time intensive subroutines were determined using the GNU Profiling Tool, the most parallelizable sections of the subroutine had to be found. PGPROF (distributed by Portland Group) is a line-by-line profiling tool that can be used to determine which lines of code take the most time to execute. This tool was able to point out the areas of the source code that would be most beneficial to be ported to the GPU.

NVIDIA Visual Profiler (NVVP) (Ref. 9)

It was important to be able to track the performance of the GPU code so that it could be optimized further. NVVP was used to visually represent what was happening on the GPU. It was able to map out the time spent transferring data as well as the time spent doing actual computation on the GPU. NVVP also suggested ways to optimize the GPU code that was run, which helped to tune the GPU code once it was created.

B. GPU Programming Software

There were three GPU programming tools that were considered:

CUDA

CUDA is a C-based programming language that works only with NVIDIA GPUs, so it would work with the K40 system, but not the APU system. It is a low-level language that gives the programmer a lot of control over the GPU, however using it would mean manually re-writing sections of code that would be run on the GPU. While CUDA is a powerful language it was not used directly during the project.

OpenCL

Much like CUDA, OpenCL is a low-level language, however it is much more portable than CUDA. OpenCL programs are able to run on most GPUs as well as other types of accelerators, like Intel's Xeon Phi. This portability adds a layer of complexity to the programming process, which makes it more complex to program in than CUDA. It is a difficult process to port existing code to OpenCL because of this complexity. Manually writing OpenCL code would most likely result in significant restructuring, and re-writing many parts of the legacy code, which would not comply with the goals of the project, so OpenCL was not used either.

OpenACC

Unlike OpenCL and CUDA, OpenACC is a high level approach to porting code to the GPU. OpenACC allows the programmer to specify where there is parallelization opportunities in the code through compiler directives, which are formatted comments placed in the source code. Then the compiler

automatically writes OpenCL or CUDA code at compile time. This eliminates the need for multiple source codes, as well as simplifies the process of porting codes to the GPU. What is traded for convenience, though, is access to the low-level capabilities of OpenCL and CUDA. OpenACC takes many of the low level details out of the programmer's hands and does them automatically, such as transferring data to the GPU. It is also able to maintain enough control to allow the programmer to optimize the GPU code. OpenACC is a programming standard with several implementations. The one used throughout the current effort was developed by the Portland Group.

IV. Hardware

There were two computers used in the project, and the main difference between them was the GPU memory architecture. One used the traditional GPU memory architecture, where the CPU and GPU have separate memory spaces. The other used a recently introduced memory architecture, where the CPU and GPU share memory, and have equal access to the entire system RAM. The differences between these memory models are explained in more detail below.

A. NVIDIA Tesla

The NVIDIA Tesla K40 accelerator graphics card was used during the first half of the project. It is a dedicated GPU (dGPU), meaning that it is separated from the CPU physically. The K40 is a high end accelerator that contains 2200 GPU cores, which together are capable of running tens of thousands of concurrent threads. This computational power allows very large kernels to be run on the GPU. This card also has the potential to run multiple kernels at the same time, and transfer data while running kernels. At the time of this writing, the K40 costs $3142.78.

The K40 uses the traditional memory architecture associated with GPUs, where the CPU and GPU do not share memory. This means that in order to do computation on the GPU, data has to be transferred over the PCIe bus to get to the GPU. After GPU computation, the data must be transferred back from the GPU to the CPU to be able to continue serial computation. These data transfers have the potential to introduce bottlenecks in the code that can increase overall runtime rather than reduce it. A simplified diagram of the memory model is shown in Figure 1(a).

B. AMD Accelerated Processing Unit

In addition to the Tesla K40 dGPU, an AMD A10-7850K accelerated processing unit (APU) was used. In an APU, the CPU and GPU are integrated on the same die, and are able to share memory space. Because of the smaller space, the APU's GPU is much smaller than a dGPU. The A10-7850 Kaveri APU comes with 8 GPU cores that are capable of running 512 total threads at a time. This is significantly less powerful than the K40, however it comes with a memory architecture that could make up for the decreased computational power, and a cost of $129.99 at the time of writing.

Kaveri APUs, like the A10, come equipped with what is referred to as heterogeneous unified memory architecture (hUMA). This memory model allows the CPU and GPU to share the same memory space, which means that the APU has the potential to eliminate the data copy overhead that can reduce speedups

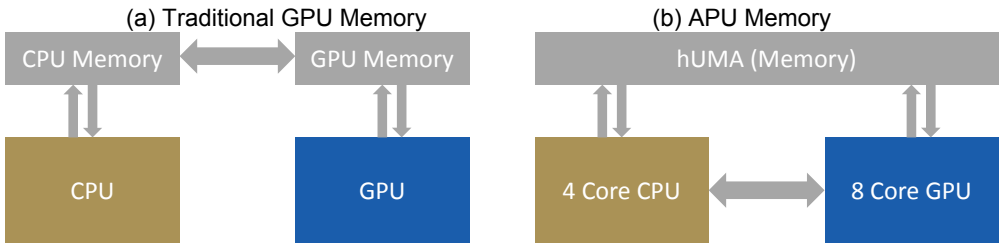


Figure 1.—Simplified memory models.

gained by running on the GPU. Many applications may not require the full potential of the K40, but do have to deal with the data transfer overhead of its memory architecture. The APU was designed to eliminate that data transfer overhead, and has the potential to perform better than the K40 for some applications. A simplified diagram of the memory model is shown in Figure 1(b).

V. Memory Management

As mentioned before, traditional GPU memory architecture requires data to be transferred from the CPU to the GPU and back during computation. This problem is especially prevalent in codes that were written before the use of the GPU for general purpose computing. Legacy codes like APNASA have been optimized and validated for years on the CPU, which means that there is likely going to be data transfer bottlenecks when porting the code to the GPU. These data transfers are fully capable of negating the benefit of the GPU, however there are a few ways to try to limit the amount of data transfers within a program. Three of these methods are described below:

A. Device-Resident Variables

One way to limit transfers is to create variables that are copied over to the GPU and left there for the duration of the program. Every action that modifies these variables within the code should then be done on the GPU, and not the CPU. Then, when a CPU process needs the data from the GPU, only the information needed will be copied to the RAM, reducing the amount of data transfers greatly.

The limitation of this method is that the pieces of data that are put onto the GPU need to have a large scope, meaning that the information stored needs to be declared early on in the program, and should be used throughout the duration of the program. Within APNASA, many of the large arrays used are generated at a subroutine level, and then deleted from memory after the subroutine has completed. If the subroutine is called only one time, then this is a good method to use, as the data could be copied over at the beginning of the subroutine and then copied out at the end. As it is currently implemented, however, the most time intensive subroutines are called many times within the duration of the program. While copying the data at the beginning and end of the subroutine may reduce the amount of copies, the data will still be copied many times, adding to the run time significantly. Generating program-level arrays could make this a viable process, however that would mean restructuring much of the code, which was out of the scope of the project.

B. Concurrent Data Transfers

Large GPUs like the K40 have the ability to run several processes at once. This means that the card is able to copy data to and from the device at the same time. They are also able to run two kernels at once, and do data transfers for one kernel while the other is running. OpenACC allows the programmer to use these features, and through some fine tuning, there are ways to help reduce the amount time that the computer spends doing only data transfers. This process does not eliminate the data transfer overhead, but can reduce it. This method was used during the current effort when running on the Tesla K40.

C. Shared Memory

In a system like the Kaveri APUs, where the memory is shared, there should be no data transfers between the devices, which would eliminate the data transfer costs. The zero-copy features of the APU are not yet implemented in PGI's OpenACC package though, so in practice there is still overhead due to data transfer. This is a simple copy within the RAM, however, and data does not have to be transferred across the PCIe bus.

VI. Tests

There were three separate codes that were ported to the GPU in the current effort. First, to make sure that the system was working correctly, a dummy script was accelerated. Then, once the hardware was confirmed to be working, an APNASA Pre-Processing Script was run on the GPU. Once that was complete, an APNASA subroutine was examined and ported to the GPU. Each of the tests were performed on both systems and are described in more detail below.

A. Dummy Script

The dummy script was approximately thirty lines of Fortran 90 code that created an array of 100,000 elements. The code then multiplied each element by 1.00001, 1,000,000 times. Because the code was so simple and parallelizable, it was easy to determine if the results from the GPU were accurate. The code also provided a simple way to determine if each hardware system was functioning without error.

B. APNASA Pre-processing Script

Prior to jumping into the complexities associated with a 3D CFD code, a fairly simple pre-processing script was used as a way to continue validating the process of porting code to the GPU, as well as explore the possibilities of accelerating applications other than APNASA. The pre-processing script was a short script that took very little time in the first place, and is used to prepare mesh files to be run through APNASA.

C. APNASA Subroutine

Once finished with the pre-processing script, the more time intensive APNASA subroutines were examined for accelerating. The “filter” subroutine was selected due its significance within the code. Within 50 iterations of the APNASA demo case, it is called 50 times and accounts for 9.24 percent of the total runtime of the code. Additionally, it has the highest per-call runtime of the top five subroutines. Within this subroutine, two loops were identified as having a large potential for acceleration due to the large amount of data being changed and the simplicity of the loops.

VII. Results

Part of the GNU profiler output described in Section III is listed in Table 1. This data was generated by running 50 iterations of APNASA’s demo case. This demo case consists of a rotor and stator gridded with 51 radial points, 51 tangential points, and 271 axial points. The case was run on a workstation with a quad core Intel Xeon processor and 4 GB of DDR3 RAM. The total runtime of the 50 iterations was 169.65 s. One thing to notice from the profiling output is that even the most time consuming parts of the code do not take much time to run. The runge and filter subroutines take the most individual time per call, but that time is still well under a second. The remaining subroutines run sufficiently quickly in their current configurations so they are not candidates for speedups. This modular nature of the APNASA code is something to note, as it makes it less of a candidate for GPU acceleration.

TABLE 1.—TOP FIVE SUBROUTINES’ PROFILING DATA

Subroutine name	Percent of total runtime, percent	Calls	Seconds per call
runge	25.3	200	0.21 s
gstres	13.96	2550	0.01 s
hstres	13.72	2601	0.01 s
fstres	13.10	2550	0.01 s
filter	9.24	50	0.31 s

TABLE 2.—TEST RESULTS

Test	NVIDIA Tesla K40			AMD A10-7850K		
	CPU	GPU	Multiplier	CPU	GPU	Multiplier
Dummy script	67.6 s	1.02 s	66.27x	68.64 s	5.04 s	13.62x
Pre-processing script	1.53 s	1.67 s	0.92x	0.87 s	0.81 s	1.07x
50 CFD iterations	76.2 s	79.6 s	0.96x	100.44 s	107.52 s	0.93x

The results of the tests are shown in Table 2. The dummy script was sped up drastically on both the APU and dGPU system. The dGPU system was able to achieve better speedups due to the larger computational power. The pre-processing script was sped up on the APU system, but not the K40 system. This may have to do with the reduced copy time of the APU as well as the reduced initialization time of the APU. Attempting to speed up APNASA as a whole did not work on either system, with slightly better performance coming from the dGPU. This has to do with the additional computational power and that the dGPU is able to run multiple kernels concurrently, as well as copy data to and from the device while running kernels. The GPU code run on the K40 was well-optimized. Many of the optimization techniques were not available on the APU however, so there was a relative reduction in performance when using the APU.

VIII. GPU Feasibility

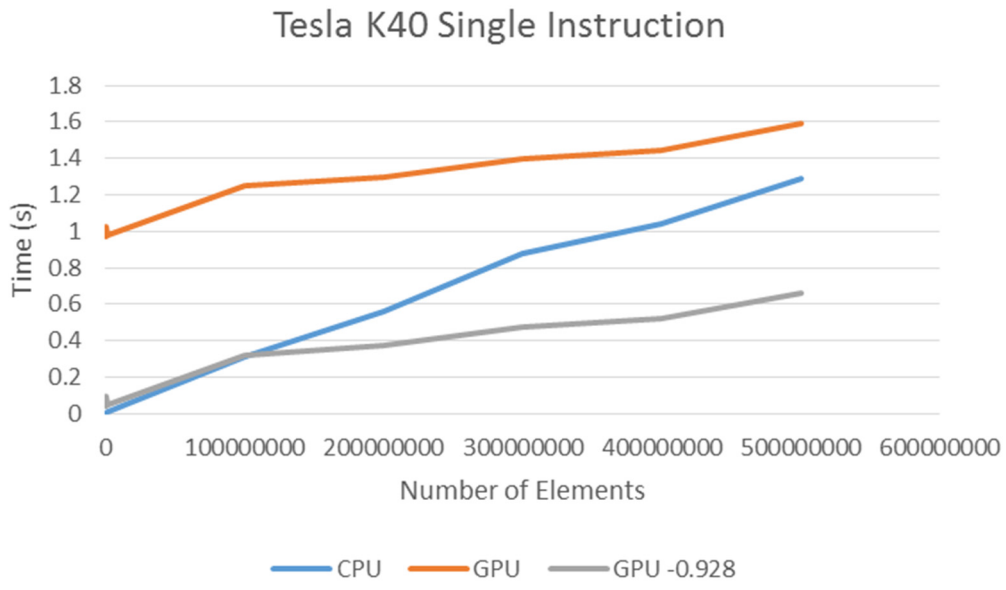
In order to determine whether APNASA and other codes would be good candidates for GPU acceleration, the threshold at which a kernel is doing enough work to achieve speedups on the GPU was found in two ways, and on both systems. Each method is explained in more detail below.

The information below contains two sets of GPU data; the total run time of the code when run on the GPU, and the total run time minus the minimum GPU time, which is used to represent the kernel execution time without the initialization time of the GPU. The initialization time was calculated by a separate test that ran a dummy kernel and calculated the total run time. In practice, the threshold amount for a given application will be somewhere on the CPU line of the graph, between the two GPU data lines. This threshold will depend on the number of kernels that can distribute the initialization costs between themselves. As more kernels are being run on the GPU, the lower the threshold amount will be.

A. Single Instruction per Element

First, the number of elements in an array was varied while the number of instructions being performed on each element was kept at a constant value, which was one in this case. This test was used to determine the number of elements an array would need in order to be accelerated on the GPU. The runtime was noted after code completion on both the CPU and the GPU. The results are shown in Figure 2.

Interestingly, when the initialization time is ignored, the dGPU system and the APU system seem to perform similarly. In order to speed up the runtime of a kernel on either of the GPUs, assuming that the startup time is distributed among many kernels, the number of elements in the array needs to be upwards of 100,000,000. We can also see from this data that the dGPU system has a minimum run time of about 0.9 s, and the APU has one of about 0.3 s. This helps to illustrate the existence of the initialization costs as well as points out that any program being ported over to the GPU on either system needs to have a runtime in excess of each of these values in order to be viable to run on the GPU, which may help to explain why the pre-processing script saw speedups on the APU, but not on the K40 system.



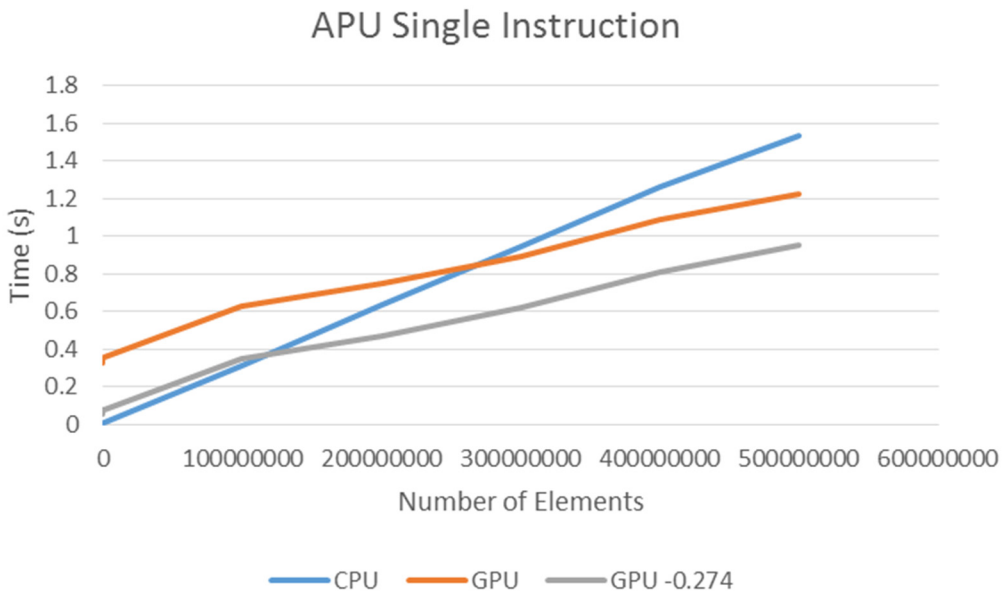


Figure 2.—Single instruction test results.

B. Fixed Array Size

Because APNASA primarily uses a fixed mesh size for each blade row during a simulation, it is desired to know how many instructions per element would be necessary to justify porting the data to the GPU. This was found by fixing the array size to one representative of an APNASA mesh, and varying the number of instructions per element within the array. The number of instructions ranged from 1-600 instructions per element. The results are shown in Figure 3. As before, the APU and dGPU behave similarly when the startup costs are ignored. We see that in order for a kernel to be run on the GPU rather than the CPU, the number of instructions per element must exceed 100. The portions of code being run on the GPU from APNASA did not meet this requirement, which seems to indicate that there was little chance of speedups.

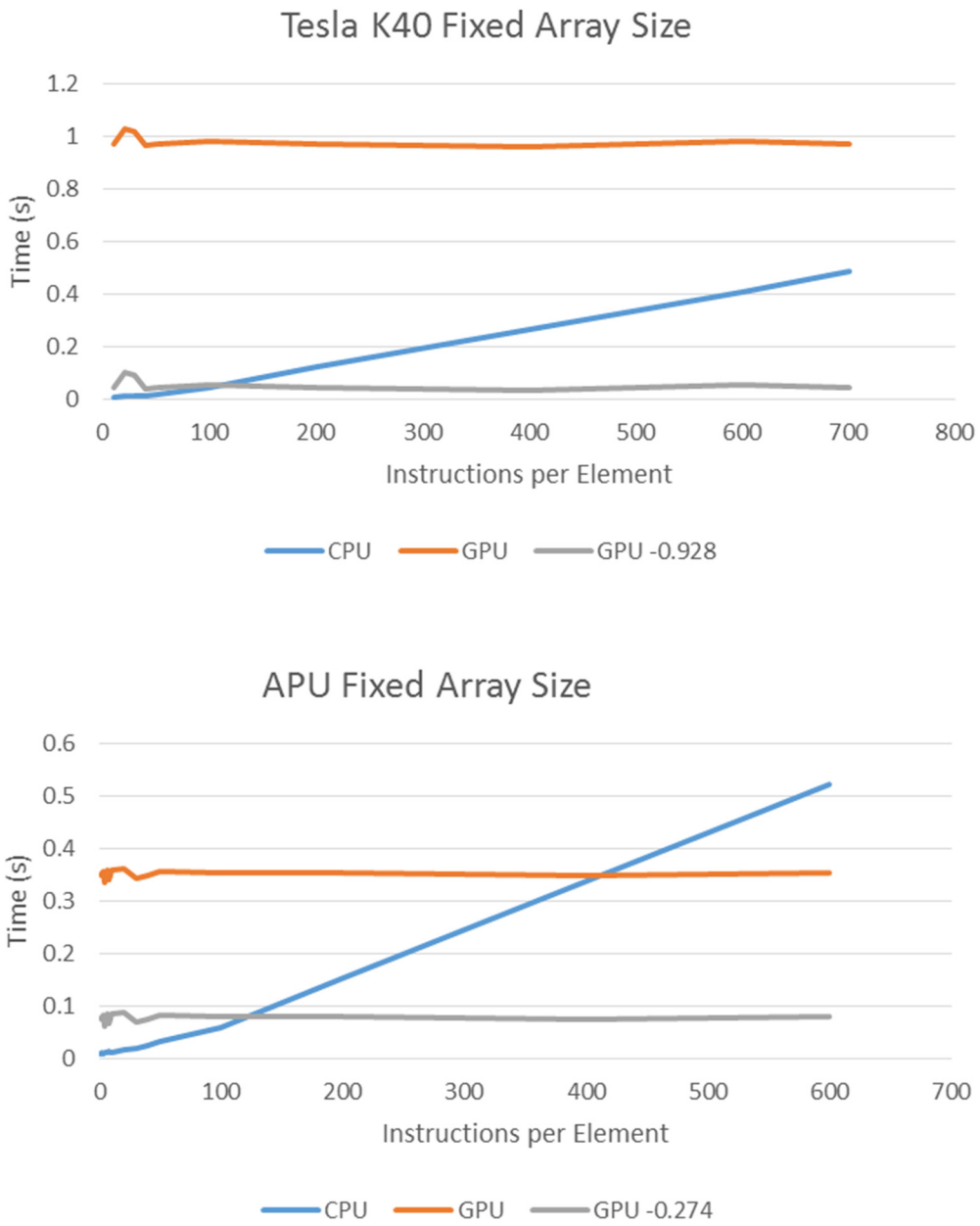


Figure 3.—Fixed array size test results.

IX. OpenACC Data Management

The accelerated processing unit's hardware allows for complete sharing of data between the CPU and the GPU. This ability should greatly improve the runtime of codes that are able to utilize it as compared to the dGPU runtime. However this project was not able to utilize the full potential of the shared memory due to limitations of the PGI OpenACC implementation at the time of writing, which does not yet support the zero-copy feature of the architecture. As it is, the data is being unnecessarily copied in the RAM. While the data transfer time is reduced, it is not eliminated as expected. Once this feature is implemented, it may be worth revisiting attempts to accelerate APNASA on the APU.

An interface was developed in an attempt to force OpenACC to recognize the shared memory of the architecture. Because we did not have access to the PGI compiler source code, we were only able to generate an external routine to add data to this table. It was successful in getting OpenACC to see the data, however the OpenCL code generated by the compiler was not able to modify the data. Creating an external routine to force the compilers to recognize the shared memory does not seem to be possible without having access to the source code of the compilers.

X. Conclusions

While the GPU is a modern powerhouse of computation, it is not able to speed up certain parallel applications. This is due partially to the significant overhead costs associated with running code on the GPU. There are often bottlenecks created when a large amount of data is transferred from the CPU to the GPU and back, as well as large initialization costs just to get the GPU running. APNASA is written in a very modular way, meaning that it uses many, quickly running subroutines that are called many times to do computation. It does not seem to do enough parallel work on large arrays to overcome the overhead costs of running code on the GPU. With new technologies, it may be possible, but at the moment, without significant code restructuring, it will not be possible to accelerate APNASA using the GPU.

In order for an application to be accelerated from GPU computing, it needs to be structured in a specific way. Any code that has potential to be sped up on the GPU needs to exceed the threshold values involving the amount of data, and the amount of work being done to the data. Without significant parallel potential, and the ability to store data on the GPU for long periods of time, it is unlikely that GPU computing will be advantageous for many applications without code restructuring.

Appendix A.—GPROF Profiling Output

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
25.30	42.92	42.92	200	0.21	0.27	runge_
13.96	66.61	23.69	2550	0.01	0.01	gstres_
13.72	89.89	23.28	2601	0.01	0.01	hstres
13.10	112.11	22.22	2550	0.01	0.01	fstres_
9.24	127.79	15.68	50	0.31	1.67	filter_
6.98	139.64	11.85	100	0.12	0.12	resid_
6.20	150.16	10.52	11	0.96	1.10	eddycm_
5.01	158.66	8.50	50	0.17	0.17	step_
1.17	160.65	1.99				powf.J
0.99	162.33	1.68	50	0.03	0.03	error_
0.92	163.89	1.56	11	0.14	0.14	damp1_
0.67	165.03	1.14	1	1.14	166.72	MAIN__
0.42	165.75	0.72	11	0.07	0.07	baxi2d_
0.37	166.37	0.62	1	0.62	0.62	resdke_
0.30	166.88	0.51				expf.J
0.22	167.26	0.38	200	0.00	0.00	baxist_
0.22	167.63	0.37	200	0.00	0.00	bcjv_
0.19	167.95	0.32	1	0.32	0.32	baxi_
0.13	168.17	0.22	200	0.00	0.00	bcext_
0.11	168.35	0.18	200	0.00	0.00	bcwall_
0.08	168.49	0.14	1	0.14	0.14	bforc1_
0.07	168.61	0.12	200	0.00	0.00	runge2_
0.06	168.71	0.10				logf.J
0.05	168.80	0.09	1	0.09	0.09	metric_
0.05	168.88	0.08	1	0.08	1.44	bforc3_
0.05	168.96	0.08	1	0.08	0.08	initb_
0.05	169.04	0.08				matherr
0.04	169.11	0.07				cosf.J
0.04	169.18	0.07				for_read_seq_xmit
0.04	169.24	0.06	50	0.00	0.00	filte2_
0.04	169.30	0.06	1	0.06	0.06	sarea_
0.03	169.35	0.05	51	0.00	0.00	fstr2_
0.02	169.39	0.04	51	0.00	0.00	hstr2_
0.02	169.43	0.04				expf
0.02	169.46	0.03	2315548	0.00	0.00	wf_
0.02	169.49	0.03	50	0.00	0.00	step2_
0.01	169.51	0.02	10	0.00	0.00	edycm2_
0.01	169.53	0.02	1	0.02	0.02	output_
0.01	169.55	0.02				for__desc_ret_item.
0.01	169.57	0.02				powf
0.01	169.58	0.01	200	0.00	0.00	bckv_
0.01	169.59	0.01	10	0.00	0.00	baxiske_
0.01	169.60	0.01	1	0.01	0.18	input_
0.01	169.61	0.01	1	0.01	0.01	resdke2_

```
0.01    169.62    0.01       1     0.01     0.01  sarea2_
0.01    169.63    0.01                          __libm_error_support
0.01    169.64    0.01                          for__desc_ret_item
0.01    169.65    0.01                          for__desc_zero_length_item
0.00    169.65    0.00     201     0.00     0.00  bcj2v_
0.00    169.65    0.00     200     0.00     0.00  bcinl2_
0.00    169.65    0.00     100     0.00     0.00  resid2_
0.00    169.65    0.00      51     0.00     0.00  gstr2_
0.00    169.65    0.00      50     0.00     0.00  error2_
0.00    169.65    0.00      11     0.00     0.00  propty_
0.00    169.65    0.00      10     0.00     0.00  damp2_
0.00    169.65    0.00       1     0.00     0.00  bcext2_
0.00    169.65    0.00       1     0.00     0.00  bforc2_
0.00    169.65    0.00       1     0.00     0.00  bforc4_
0.00    169.65    0.00       1     0.00     0.00  bfread_
0.00    169.65    0.00       1     0.00     0.00  gridg_
0.00    169.65    0.00       1     0.00     0.00  metri2_
0.00    169.65    0.00       1     0.00     0.00  ptcalc_
```

% time	the percentage of the total running time of the program used by this function.
cumulative seconds	a running sum of the number of seconds accounted for by this function and those listed above it.
self seconds	the number of seconds accounted for by this function alone. This is the major sort for this listing.
calls	the number of times this function was invoked, if this function is profiled, else blank.
self ms/call	the average number of milliseconds spent in this function per call, if this function is profiled, else blank.
total ms/call	the average number of milliseconds spent in this function and its descendents per call, if this function is profiled, else blank.
name	the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.

Appendix B.—Source Code for Dummy Script

This code was used to ensure that the GPU systems were working correctly as well as ensure that the OpenACC directives were properly porting code to the GPU.

```
      PROGRAM MAIN
C
      REAL, ALLOCATABLE, DIMENSION (:) :: X      ! (IL)

C
      NL = 1000000
      IL = 100000

C
C  ALLOCATE ARRAYS
C
      ALLOCATE(X(IL))
C
C  INITIALIZE ARRAY X
C

      DO I=1,IL
      X(I) = FLOAT(I)
      END DO

C
C  PERFORM A LOT OF WORK IN PARALLEL ON ARRAY X
C


c$acc kernels
      DO N=1,NL
c$acc loop
      DO I=1,IL
      X(I) = 1.00001 * X(I)
      END DO
      END DO
c$acc end kernels



C
      PRINT *, X(1), X(500), X(5000), X(10000)
      STOP
C
      END
```

Appendix C.—Source Code for GPU Feasibility Test

This code was used to determine the amount of computation needed to achieve speedups on the GPU. The dimensions of the array X were changed, as well as the number of instructions in the second loop.

```
      PROGRAM MAIN

C DECLARE VARIABLES
      REAL, ALLOCATABLE, DIMENSION (:,:,:) :: X
      INTEGER ::
     . IL = 61,
     . JL = 61,
     . KL = 200,
     . C = 0

C ALLOCATE X
      ALLOCATE(X(IL,JL,KL))

C POPULATE X
      DO K=1,KL
      DO J=1,JL
      DO I=1,IL
      X(I,J,K) = 1.0000
      END DO
      END DO
      END DO

C PERFORM TASKS ON X

c$acc kernels
      DO K=1,KL
      DO J=1,JL
      DO I=1,IL
      X(I,J,K) = X(I,J,K) * 2
      END DO
      END DO
      END DO
c$acc end kernels

      STOP
      END
```

Appendix D.—Source Code for C/Fortran Interface

This code was used in an attempt to force PGI's implementation of OpenACC to recognize the shared memory of the accelerated processing unit. It is discussed in Section IX.

Fig D1. main.f

```
      PROGRAM MAIN

      USE ISO_C_BINDING
C     INITIALIZE VARIABLES
      REAL, TARGET, ALLOCATABLE, DIMENSION (:) :: X
      INTEGER :: IL = 100

C     ALLOCATE VARIABLES
      ALLOCATE(X(IL))

C     INITIALIZE X
      DO I=1,IL
      X(I) = FLOAT(I)
      END DO

C     CALL C FUNCTION
      CALL MAP(X,SIZE(X))


C     CHANGE X TO 15
c$acc kernels loop present(X)
      DO I=1,IL
      X(I) = 15
      END DO

      CALL UNMAP(X)
C     PRINT PART OF ALL POINTERS AND X
      PRINT *, X

      END
```

Fig D2. map_.c

```
/*
Function: map
maps inputted fortran data to OpenACC present data table
*/

#include <stdio.h>
#include <stdlib.h>
#include <openacc.h>

void map_(f1, size)
float f1[];
int size;
```

```
{
acc_map_data(f1,f1,size*sizeof(float));

printf("\t%li\n", f1);
printf("\t%f\t%f\t%f\n", f1[0], f1[49], f1[99]);
}
```

Fig D3. unmap_.c

```
/*
Function: unmap
unmaps data from OpenACC present data table
*/
#include <openacc.h>

void unmap_(f1)
float f1[];
{
acc_unmap_data(f1);
}
```

References

1. Thibault, Julien C., Senocak, Inanc; "CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows," forty-seventh Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition, Orlando, FL, 2009.
2. Adamczyk, J.J., 1985, "Model Equation for Simulating Flows in Multistage Turbomachinery," ASME 85-GT-226.
3. Adamczyk, J.J., Mulac, R.A., Celestina, M.L., 1986, "A Model for Closing the Inviscid Form of the Average-Passage Equation System," ASME 86-GT-227.
4. Adamczyk, J.J., Celestina, M.L., Beach, T.A., and Barnett, M., 1990, "Simulation of Three-Dimensional Viscous Flows Within a Multistage Turbine," ASME J. of Turbomachinery, Vol. 112, pp. 370–376.
5. Jesperson, Dennis C., "Acceleration of a CFD code with a GPU," Scientific Programming, Vol. 18, 2010, pp. 193–201
6. Noaje, Gabriel.; Jaillet, Christophe., Krajecki, Michael., "Source-to-source code translator: OpenMP C to CUDA," High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on , vol., no., pp. 512,519, 2–4 Sept. 2011.
7. GPROF, GNU Binutils for Ubuntu, Ver. 2.24, 1983 [cited 4 August 2015].
8. PGPROF, PGI Accelerator workstation for C and Fortran, Ver. 15.5-0, 2015, URL: https://www.pgroup.com/products/pgprof.htm [cited 4 August 2015].
9. NVVP, NVIDIA CUDA Toolkit, Ver. 6.5, 2013 [cited 4 August 2015].