Implementing CUDA-based GPU Acceleration to Hybrid WENO-Spectral Code

Ronald M. Caplan

Summer 2010

Introduction

This report documents a first attempt at implementing GPU parallelization to the HOPE Hybrid code. Profiling of the serial code is given for a one-dimensional test case, and the relevant routines are identified. After a quick introduction to Nvidia GPUs and the CUDA API, descriptions of the GPU implementation of the routines are developed and test codes are written to test performance. Speedup results are then shown. Recommendations are given followed by an appendix describing the included test code files and specifications of the GPU card used for the timings.

HOPE Hybrid Code

To use the code to simulate shocks, one has two options. The code can either be run with all WENO domains, or it can be run with the multiresolution analysis which dynamically switches domains from WENO to spectral and from spectral to WENO depending on the smoothness of the solution. The spectral method is much faster than the WENO method, but the multiresolution analysis is very expensive. The idea of the hybrid code is that the computational gains of the spectral method will outweigh the multiresolution analysis time because in any simulation the number of WENO domains should be quite small compared to the spectral domains..

The following timings use 100 domains, with 17 grid points for the spectral domains and 51 grid points for the WENO domains. The code is run to an end time of 1. (The number of time steps is variable since it is adaptive). RK is the time taken for the Runga-Kutta step (which contains the WENO or spectral integrators) and MR is the multiresolution switching time.

100 Domains	Steps Taken	Total Time	RK	MR
WENO-ONLY	2402	78.48 sec	76.76 sec	0 sec
HYBRID	2386	67.09 sec	16.33 sec	49.77 sec

For most problems the number of domains is what will scale up. The following timings use 4000 domains and an end time of 0.1:

4000 Domains	Steps Taken	Total Time	RK	MR
WENO-ONLY	243	332.19 sec	325.94 sec	0 sec
HYBRID	244	285.34 sec	68.11 sec	214.38 sec

One can see that the hybrid code is slightly faster than the WENO-only code. During the hybrid run, approximately 3 domains were WENO, and the rest spectral.

There are thus three main sections of code to look at for speedup:

1) WENO integrator
2) Spectral integrator
3) Multiresolution switcher.

By profiling the code, it is found that one routine stands out in each of these three sections. However due to the modulation of the code, there are many small routines which is not desirable for parallelization. The following is a listing of the potential parallelizable routines with their percentage of computation time for the timings of 100 domains given above:

	Maximum Compute Time Subroutine	Percentage of Section Computation	Percentage of Total Computation
WENO Integration in RK	PS_WENO_Euler_Long_Zico_INT()	30.25%	29.91%
Spectral Integration in RK	vcost()	34.37%	8.25%
Multiresolution Switcher	Lagrangian_Weight_2()	80.68%	59.67%

The WENO integrator Zico routine is only 30% because the WENO method has many other small routines that are part of the algorithm, which when added together make up the rest of the time. The percentage listed is for the WENO-only code without MR analysis. The second largest compute time routine is the WENO code is actually Lagrangian_weight_2(), which is already included in our list of routines for the MR code.

The Spectral integrator vcost() routine is only about 30% due to many small routines in the smoothing computations which add up to the rest of the compute time. Since the routine only is about 8% of the total compute time, it may not be valuable to parallelize the routine.

For the MR, it is clearly a good idea to parallelize Lagranigan_Weight_2(). This is even more desirable due to the fact that the routine is also called in the WENO-only code. It should be pointed out that the “2” in the routine is due to this routine being in a CASE statement which calls any one of 4 routines, all of which will be similar to structure and compute time to the “2” routine.

Nvidia GPUs

Nvidia GPUs offer great potential for parallelism. They are organized as several multiprocessors (MP) each with a number of compute cores (originally 8, but with new cards, each MP has more). All MPs are on a single card with a large global memory, but each MP has a small but very fast shared memory. The card structure could be described as a MPI/OpenMP combination cluster all on a single chip with most communication taken care of automatically.

The cards are utilized by a C code extension API called CUDA.

The lowest end CUDA capable cards run about $30 and only have 1 or 2 MPS, with a total of 16 cores. These cards only can handle floating point precision computations. The lowest end cards that can handle double precision computations (a compute capability of 1.3) run about $200. Nvidia has recently released a new GPU architecture called Fermi. The new Fermi cards are a large potential improvement over the current Tesla-based cards. Some of the relevant features of the Fermi cards are:

- 7x faster double precision performance
- Many more compute cores
- Larger shared memory per MP
- Larger number of threads per block
- IEEE compliant floating point
- ECC memory

The error-correcting memory is very good to have for long simulations. Also, the larger number of threads per block is vital in higher-dimensional settings when one thread needs to communicate with neighbor threads since more threads means more interior nodes in shared memory than boundary points. The double precision performance was a major drawback in the previous cards, and in many real-world applications, the double precision is a necessity, and so this performance increase is very useful. The lowest end Fermi cards run about $200, and highest around $1800.

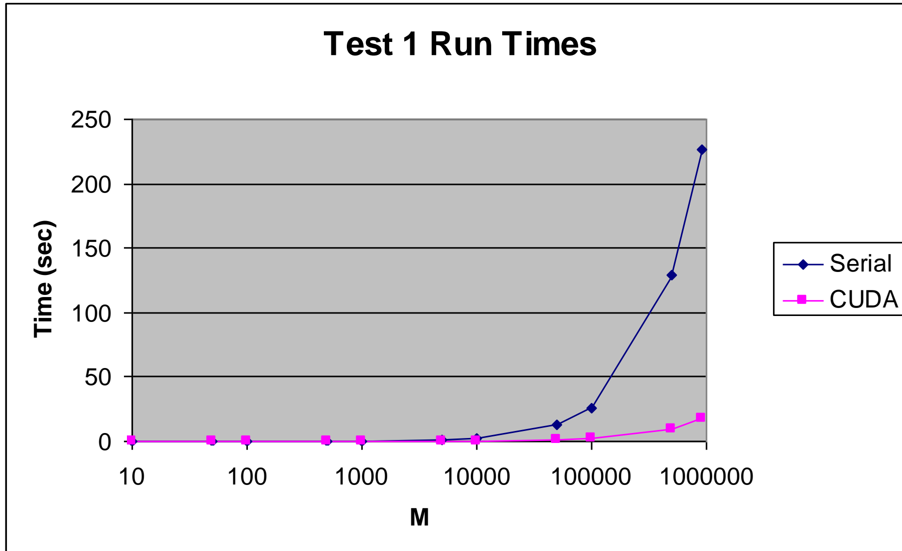
The CUDA API is native to the C programming language. It is also free. For FORTRAN implementations, wrappers are created to use CUDA. Although there are some freeware projects in the works, the most common implementation is that of PGI's compilers. However, these are costly, and so not ideal for small developers. Also, it is still unknown if the FORTRAN CUDA performance matches the native C performance or not. Due to the wrappers and automatic implementations of procedures such as memory transfer, it is possible that CUDA C is faster than CUDA FORTRAN.

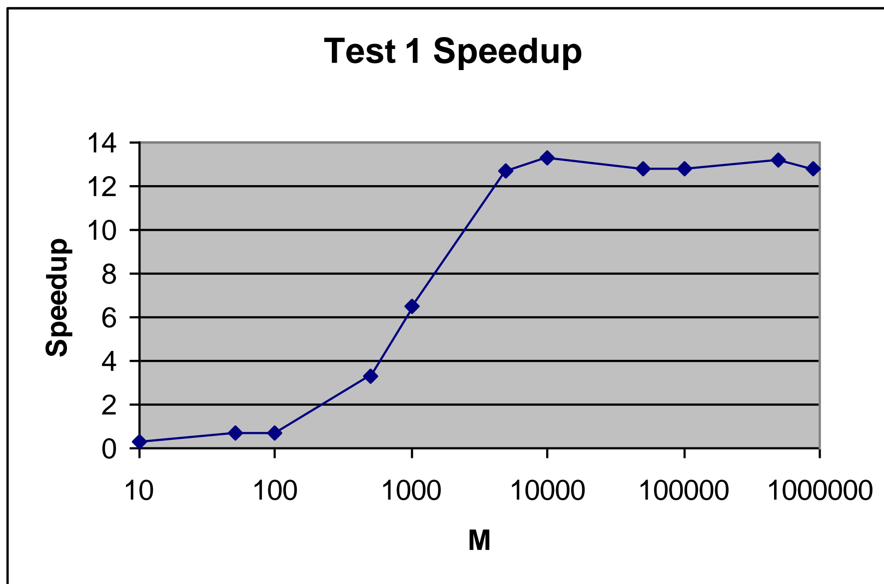
CUDA Implementations and Results

Test 1: Lagrangian_Weight_2()

The CUDA implementation works for any value of N and M (within memory restrictions). As currently written, the test program also calls a CUDA version of get_ratio_a_2() which has synchronization issues which make it only valid for M less than the CUDA blocksize. In a real implementation the serial version of get_ratio_a_2 could be used to avoid this problem, or the CUDA version could be further developed.

The timing results for N=17, and Steps=100 are:





We see that for larger values of M, a speedup of around 12 is seen, which remains constant as M increases. Thus is this routine could be run on a large array vector, it can speedup computation of the hybrid routine significantly.

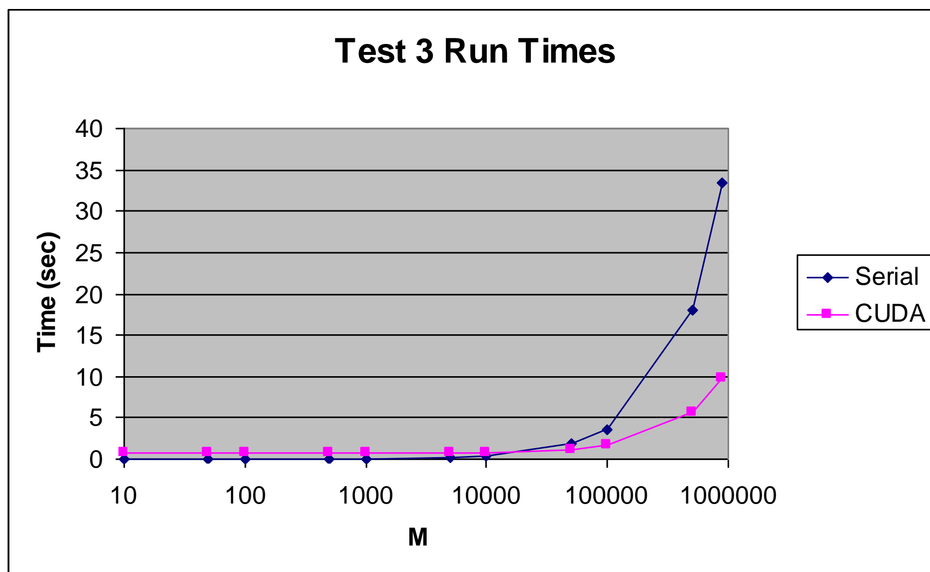
Test 2: vcost()

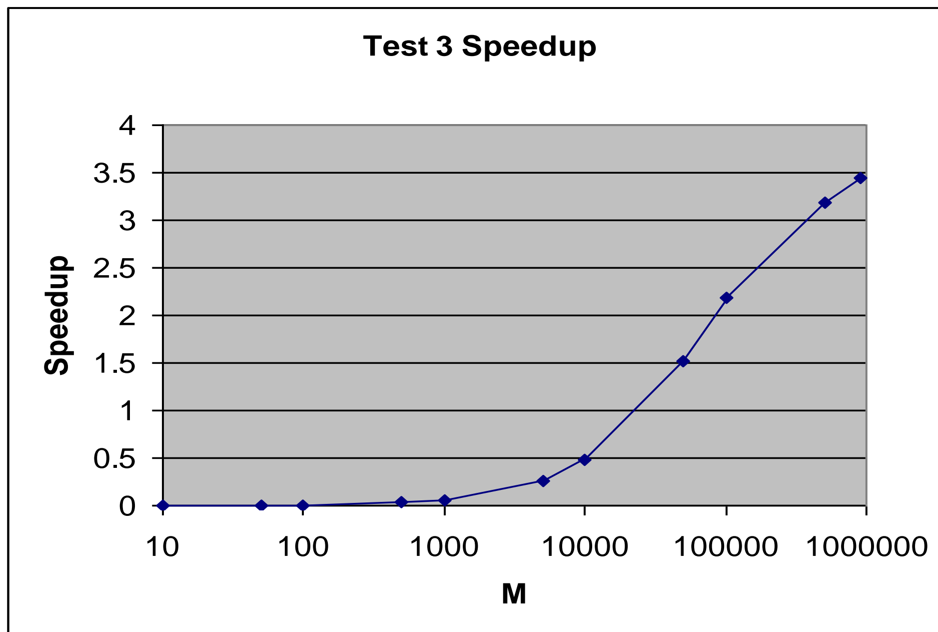
The vcost() routine calls several varied routines named vradf_ which include vradf2, vradfg, etc. Which routine is called depends on the multiplicity of N. The CUDA implementation of all vradf routines works for all N except for vradfg() which has synchronization issues, making it only valid for N less than the CUDA block size. Further development may solve this issue.

The results of the CUDA vcost yield no speedup for even very large N. This implies that unless a major flaw is found in the CUDA code, it is not worthwhile to develop a CUDA code for vcost, especially considering how little the spectral method contributes to the total compute time when compared to the MR analysis.

Test 3: PS_WENO_Euler_Long_Zico_INT()

The CUDA version of this routine works for all values of M (memory allowed) and for all orders (1-11, odd). The results are as follows:





We see that for large M, we do get some speedup, which implies that in an ALL-WENO code with 1 domain and a large M, the CUDA modifications could speed up the zico routine by a factor of 3 at least. The WENO algorithm also calls lagrangian_weight_2, which as shown above speeds up even better with CUDA.

Conclusion and Recommendations

From the results of this preliminary study, it would seem that iin its current form, the Hybrid code is not suitable for CUDA implementation. However, with a restructuring of the code, specifically in combining like-domains into large continuous memory arrays, a CUDA implementation may give good speedup results.

It would remain to be seen however, whether the results would surpass that of the more standard parallelism already implemented in the hybrid code (MPI and OpenMP). One advantage of a CUDA implementation would be that the cost of obtaining the speedup would be much less than on a standard cluster.

Further optimizations of the CUDA routines developed, and implementation on new Fermi architectures may increase performance significantly. It would also be a goo idea to compare test codes written in the native CUA C with their PGI FORTRAN counterparts to see if the FORTAN wrappers are hindering performance.

Appendix A: Outline of Test Routines

Below is a list of test code files with descriptions for all three test codes.

Test 1 is the implementation of the MR Langriangian Weight routine. The test is given an N value (size of the spectral domain) and an M value (varies on function call, but usually the size of the WENO domain). It also runs the test for a specified number of steps.

PPtest1.f90	Main program. Runs 3 versions of lag_weight, original, modified serial, and CUDA
PPtest1_mod.cuf	Module containing all subroutines.
compile1	Script to compile test code using PGI
run1	Script to run test code on Dulcinea

Test 2 is the implementation of the spectral vcost routine. The test is given an N value (size of the spectral domain) and an M value (the number of independent DCT to perform). It also runs the test for a specified number of steps.

PPtest2.f90	Main program, runs serial and CUDA vcost
Module_FFT_VFFT.F	Original vcost module routines (subset of original file)
Module_FFT_VFFT_CUDA.cuf	CUDA vcost routines
compile2	Script to compile test code using PGI
run2	Script to run test code on Dulcinea

Test 3 is the implementation of the WENO zico Routine. The test is given an "Order" value (either 1,3,5,7,9,or 11) and an M value (the size of the WENO domain). It also runs the test for a specified number of steps.

PPtest3.f90	Main program, runs serial and CUDA zico
Module_WENO_Euler_Long_Zico_CUDA.cuf	CUDA routines
Module_WENO_Euler_Long_Zico.F	Original routines (modified subset with .i files explicitly included)
Module_WENO_Coefficients_1357911.F	Module used by zico module
Module_WENO_Option.F	Module used by zico module
compile3	Script to compile test code using PGI
run3	Script to run test code on Dulcinea

Other included files:
ranlib.f – Used to generate random numbers for initial test arrays in test codes.
Pseudopack.h – Used by all test routines
PPtest2_mod.F90 – Super slow DCT routine used for test 2 verification
pgf95options.txt – PGF95 compiler options manual
Hybrid1D_Timings.xls – Serial profiling timings of hybrid code
PPtest_TIMINGS.xls – Timings of CUDA test codes.
Folder "mm" contains a CUDA matrix-multiply test code for reference

Appendix B: Dulcinea TESLA info and other CUDA card info

The specification of the CUDA card used in this report is:

Device Name:	Tesla M1060
Device Revision Number:	1.3
Global Memory Size:	4.29E+09
Number of Multiprocessors:	30
Number of Cores:	240
Concurrent Copy and Execution:	Yes
Total Constant Memory:	65536
Total Shared Memory per Block:	16384
Registers per Block:	16384
Warp Size:	32
Maximum Threads per Block:	512
Maximum Block Dimensions:	512, 512, 64
Maximum Grid Dimensions:	65535 x 65535 x 1
Maximum Memory Pitch:	2147483647B
Texture Alignment	256B
Clock Rate:	1296 MHz
Initialization time:	721569 microseconds
Current free memory	4.26E+09
Upload time (4MB)	1829 microseconds (720 ms pinned)
Download time	1205 microseconds (773 ms pinned)
Upload bandwidth	2293 MB/sec (5825 MB/sec pinned)
Download bandwidth	3480 MB/sec (5426 MB/sec pinned)

CUDA Documentation is in the folder CUDA_DOC_3.1