Cross-Platform OpenCL Code and Performance Portability for CPU and GPU Architectures Investigated with a Climate and Weather Physics Model

Han Dong, Dibyajyoti Ghosh, Fahad Zafar, Shujia Zhou
{han6, dg9, fahad3, szhou} @ umbc.edu
University of Maryland Baltimore County

***Abstract*—Current multi- and many-core computing typically involves multi-core Central Processing Units (CPU) and many-core Graphical Processing Units (GPU) whose architectures are distinctly different. To keep longevity of application codes, it is highly desirable to have a programming paradigm to support these current and future architectures. Open Computing Language (OpenCL) is created to address this problem. While the current implementations of OpenCL compiler provide the capability to compile and run on the architectures above, most of the current researches investigate the performance of GPU's as a compute device. In this paper we will investigate the portability of OpenCL across CPU and GPU architectures in terms of code and performance via a representative climate and weather physics model, NASA's GEOS-5 solar radiation model, SOLAR. An OpenCL implementation portable between CPU's and GPU's has been obtained with significant performance improvement in some CPU's and GPU's. We found that OpenCL's vector-oriented programming paradigm assists compilers with implicit vectorization and consequently significant performance gains were achieved.**

***Index Terms*—Multi-threaded environments; Parallel Applications; OpenCL; Vectorization**

I. Introduction

Current trends in computer processor development have moved from a single powerful core to multi-core such as Intel Westmere and IBM Power7 and many-core accelerators such as NVIDIA Fermi and AMD FireStream GPU's. These architectural improvements are useful only if compute-intensive applications can efficiently utilize all the resources in a computer system. However, distinct architecture differences, in particular between CPU's and GPU's, pose a very challenging task for developing an application code with longevity as well as optimal performance. OpenCL is created partially to address these issues. It offers a standard heterogeneous programming environment for applications to execute on CPU's, GPU's and various types of accelerators and mobile processors [1][2]. OpenCL provides a SPMD (Single Process Multiple Data) model for programming where the parallel portions of a program comprise a grid of work items executing the same code.

In this paper we explore the code and performance portability of OpenCL across different platforms consisting of CPU's and GPU's using a real-world, representative climate and weather physics model, solar radiation (SOLAR). Furthermore, we investigate the reasons behind performance and portability of IBM Power6, and Intel CPU's and NVIDIA GPU's in Linux and Mac OSX environments. We evaluate performance with the number of physics columns in SOLAR from 128 up to 1024. Across the various platforms we noticed significant performance gains, especially among Intel CPU processors, including over 300x speedup on certain column sizes (see Table II). These results have been adumbrated along with detailed code analysis in the following sections. We conjecture that these observed gains are facilitated by Single Instruction Multiple Data (SIMD) coding style of OpenCL kernels along with the implicit vectorization capabilities of the OpenCL SDK's provided by the vendors.

The rest of the paper is organized as follows: in Section II, related work in the area of porting code to CPU and GPU are discussed. Section III provides an analysis of the GEOS-5 climate model, specifically the solar radiation component. Section IV lists the experimental setup in terms of platforms and hardware. In Section V, we highlight our experiences in porting the serial C code to a parallel OpenCL version along with various optimizations that contributed to the overall performance gains. Section VI presents the results of our performance gains along with discussions on the code and performance portability from CPU to CPU platforms, GPU to GPU platforms, and across CPU and GPU platforms. Section VII touches upon the topic of explicit manual vectorization through the use of Intel AVX [3] intrinsic and we conclude our paper in Section VIII.

II. Related Work

Although research in OpenCL often discuss the platform independence as its main benefit [4][5][6], there are few examples of cross-platform OpenCL implementations for real world applications. Most work in parallel programming often focused on the performance of GPU devices instead of CPU devices [7][8][9] where CUDA often outperforms OpenCL in executing the same code [10]. To our knowledge, the few works that discuss performance of OpenCL in CPU's by Grosser et al [11] and Zhang et al [12] provide limited in-depth analysis on the reasons behind the performance gains even though it has been shown by Lee et al [13] that CPU's exhibits similar performance gains compared to GPU's given adequate performance tunings.

The SOLAR code was initially translated from FORTRAN to C, and ported to the IBM Cell Broadband Engine by Zhou et al [4] where detailed code analysis and performance improvements were discussed. Fahad et al [14] extended the serial, single-precision C implementation of SOLAR to the IBM Power architecture and Intel x86 architecture with OpenCL and obtained considerable performance improvement (3x to 4x) per processor core. However, their implementation only obtained an accuracy of 1.0×10^{-4} between the serial and parallel implementations and cannot execute on GPU's. In this paper, we improve their implementation with a code runnable across multiple platforms consisting of CPU's and GPU's with an accuracy of 1.0×10^{-6} and demonstrate the dramatic performance gains of executing OpenCL code on CPU processors.

III. Analysis of SOLAR Radiation Model

In a climate model, the Earth is represented with a 3 dimensional grid. Typically a latitude-longitude grid where the horizontal direction is used to solve fluid dynamics equations. A grid in the vertical direction, so-called column, is used to describe physical processes such as solar radiation, cloud, and precipitation. The NASA GEOS-5 climate model is a production-quality climate modeling code consisting of a few hundred thousand lines of code written in FORTRAN, as shown in Figure 1. In this paper, we focus on a particular portion of the code handling solar radiation effects (SOLAR), which can take around 10% of the total runtime depending upon aerosol effects. The code structure only has dependence in the vertical direction, which is representative of physical model components used in climate and weather models. Reduction in execution time of SOLAR will allow it to be utilized more frequently and consequently will help improving the predictability of climate models such as GEOS-5 and weather models such as the Weather Research and Forecasting (WRF) Model. In addition, a unified implementation that preserves performance across heterogeneous architectures will have an enormous impact on its longevity, therefore reducing the human cost and effort in maintaining the code. The program structure of SOLAR is shown in Figure 2 - bulk computations are done in SOLUV and SOLIR functions. SOLUV and SOLIR perform ultraviolet and infrared radiation computations, respectively. Both utilize many of the same methods inside the code base. The serial C version of SOLAR used in this paper utilizes single precision floats and consists of 1500 lines of code. SOLUV takes around 15% of the total runtime while SOLIR takes around 80% of the total runtime.

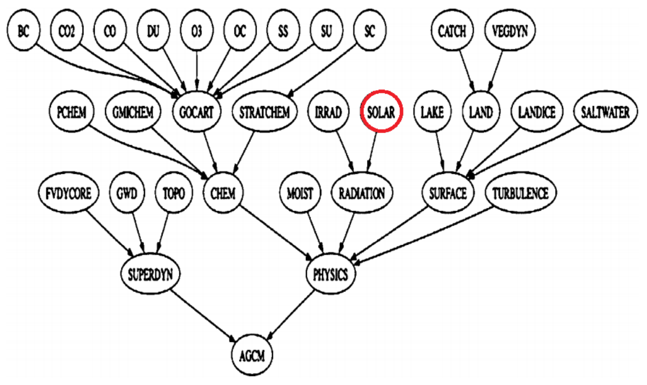
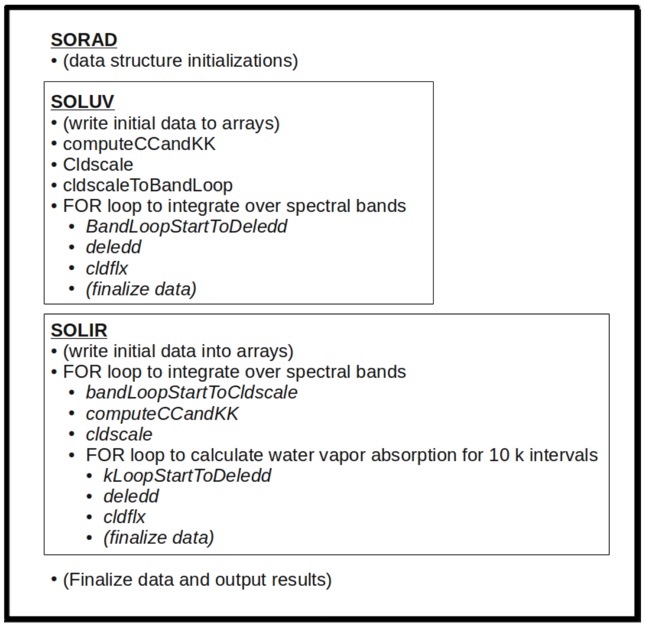


Fig. 1. NASA GEOS-5 climate model structure. The red circle highlights SOLAR code.

IV. Experimental Setup

The serial C version of SOLAR is used to confirm the accuracy of the results and for performance comparison. Table I lists the various platforms used in our work, where COMPUTE_UNITS refers to the number of computational units on each platform. This is due to OpenCL's abstract platform model [15], which maps the number of compute units to the number of threads in CPU processors and similarly to the number of Streaming Multiprocessors (SM) in GPU devices. Each compute unit consists of an array of processing elements



SORAD
- (data structure initializations)

SOLUV
- (write initial data to arrays)
- computeCCandKK
- Cldscale
- cldscaleToBandLoop
- FOR loop to integrate over spectral bands
    - *BandLoopStartToDeledd*
    - *deledd*
    - *cldflx*
    - *(finalize data)*

SOLIR
- (write initial data into arrays)
- FOR loop to integrate over spectral bands
    - *bandLoopStartToCldscale*
    - *computeCCandKK*
    - *cldscale*
    - FOR loop to calculate water vapor absorption for 10 k intervals
        - *kLoopStartToDeledd*
        - *deledd*
        - *cldflx*
        - *(finalize data)*

- (Finalize data and output results)

Fig. 2. The code structure of solar radiation model, SOLAR.

that execute the code.

The Intel Core i7-2630QM is a 4 core Sandy Bridge processor which can execute 2 simultaneous threads per core; it supports the latest Intel Streaming SIMD Extensions (SSE) 4.2 [16], Advanced Vector Extensions (AVX) [3] and it executes with maximum COMPUTE_UNITS of 8. The Intel Xeon CPU X5670 is a 6 core Nehalem processor which can execute 2 threads per core; however the node has dual Intel Xeon X5670 processors which executes with maximum COMPUTE_UNITS of 24 and it supports Intel's SSE 4.2. The Intel Core 2 Duo has 2 cores and can execute 1 thread per core; it runs with maximum COMPUTE_UNITS of 2 and supports the Intel SSE 4.1 extensions. Intel's SSE and AVX extensions are special instructions that operate on 128 bit (SSE) and 256 bit (AVX) registers, which can pack floating point numbers and execute a single instruction on each float in parallel [17].

The IBM JS22 Power6 blade has dual quad-core processors that executes with maximum COMPUTE_UNITS of 8. NVIDIA GeForce GTX 580M is a Fermi class GPU with compute capability 2.1. It has 8 SM's and 48 CUDA cores per SM (total of 384 CUDA cores). It runs with maximum COMPUTE_UNITS of 8. The GeForce GT 320M has compute capability 1.2 with 6 SMs and 12 CUDA cores per SM (total 72 CUDA cores), and runs with maximum COMPUTE_UNITS of 6.

Intel Core i7 and NVIDIA GTX 580M reside on the same workstation with Ubuntu 11.04 OS. The Intel Xeon X5670 and IBM Power6 are on different compute nodes running different versions of Red Hat Linux. The Intel Core 2 Duo and GeForce GT 320M reside on the same workstation with Mac OSX 10.6.7 OS. The serial C version of SOLAR was compiled with *-O2* optimization flag while the parallel implementations were compiled with *-O2 -cl-fast-relaxed-math -cl-mad-enable* optimization flags.

V. Porting and Optimizations

We focused our first OpenCL implementation to run on CPU due to the simplicity of implementation compared to GPU where special attention needs to be given for memory coalescing, utilization of local memory, and minimizing usage of PCI Express Bus. The CPU version does not use the PCI Express Bus and local memory is not needed as OpenCL memory objects are cached by the processor; explicit caching with local memory introduces unnecessary overhead [18]. The parallel version of SOLAR consists of 36 kernels and is executed on OpenCL compute units, either CPU cores or GPU SM's.

The functions in SOLUV and SOLIR are parallelized with insights from Intel OpenCL SDK guide [18]. Some optimizations included: using temporary data variables to decrease global memory reads and writes, removing conditional statements to decrease thread divergence within the kernels. Preprocessor macros were used for constant variables that dictated kernel loop iterations. Without preprocessor macros, the number of iterations per loop was determined at run time, usually after a kernel is issued by the command queue for execution. Our preprocessor macros enable OpenCL dynamic compilation to ensure that the variable is known at kernel compile time allowing compilers to perform implicit loop unrolling.

In SOLAR, most of the functions were easily vectorized except for the computationally expensive CLDFLX function. Listing 6 represents the serial CLDFLX which not only contains multi-dimensional arrays but also a three layer conditional statement with data dependencies. A major part of porting and optimization was spent in breaking the dependence. Listings 7, 8, and 9 represent the result of splitting CLDFLX into three kernels, *upKernel*, *downKernel*, and *reductionKernel*. CLDFLX consists of multiple arrays with eight different layer configurations for clear and/or cloudy weather conditions. Most arrays are used for initial energy flux calculations while a final array stores a summation of the previous energy fluxes. We utilized bit masks with eight bits to simulate all eight weather configurations. The *upKernel* and *downKernel* are named as such since the calculations go up and down the columns updating

Platform	Compute Units	Clock (GHz)	Environment	GCC Version	OpenCL SDK	OpenCL Specification
IBM JS22 Power6	8	4.00	Red Hat 4.1.2-48	4.1.2	IBM Power v0.3	1.1
Intel Core i7 2630QM	8	2.00	Ubuntu 11.04	4.5.2	Intel 1.5	1.1
Intel Core 2 Duo P8600	2	2.40	Mac OSX 10.6.7	4.2.1	Intel 1.1	1.0
GeForce GTX 580M	8	-	Ubuntu 11.04	4.5.2	CUDA 4.0.1	1.1
GeForce GT 320M	6	-	Mac OSX 10.6.7	4.2.1	CUDA 3.2	1.0
Dual Intel Xeon X5670	24	2.93	Red Hat 4.4.4-13	4.4.4	Intel 1.5	1.1

TABLE I
CHARACTERISTICS OF CPU'S AND GPU'S USED IN PERFORMANCE TESTING.

the initial energy fluxes due to no data dependencies across the eight different layer configurations. The *reductionKernel* has data dependencies across the layer configurations in order to compute an intermediate array to be used in the final summation. However the intermediate array was redundantly recomputed every time. This was resolved by pre-computing the intermediate array and passing it to the kernel as an input argument. The optimizations listed above benefits the GPU devices by eliminating thread divergence and data dependencies.

The parallel OpenCL implementation contains around 1800 lines of kernel code and 36 kernels. The kernels range from one-dimensional to three-dimensional. There are over 70 multidimensional arrays. Each thread in a kernel maps to each specific index in the arrays passed as arguments, thus no barriers were required. The initial porting to OpenCL performed reasonably well on both CPU and GPU platforms; performance results are elaborated more in the next section.

VI. RESULTS

Figure 3 shows the execution time for all the different platforms, excluding any initialization, data writes or reads. In Tables 1 and 2, certain information are intentionally empty for the GPU's, such as clock speeds and speedups per thread. The clocks for the GPU's are difficult to compare against CPU's as GPU's consist of multiple vector processing units with its own processing and graphics clocks. A GPU thread cannot be compared to a CPU thread as a single GPU thread is much more lightweight and can be considered as a functional unit that is part of an entire warp (32 CUDA threads) of functional units that execute a single vector instruction across the same data in lockstep.

Table II indicates reasonably good results with Core i7, Xeon, and GTX 580M. The Core i7 had the best per thread speedup (24x for each thread) at the case of 512 columns. The Xeon had the best total speedup (359x at the case of 512 columns) as it could launch 24 threads of computation compared to the 8 threads of Core i7. The GTX 580M also showed impressive speedup results even though there weren't specific GPU optimizations. Figure 3 demonstrates the scalability of the OpenCL parallel CPU implementation across the column sizes from 128 up to 1024. Mixed results were recorded with IBM JS22 Power6, Intel Core 2 Duo, and GT 320M platforms. More detailed analysis of these results are explained in the next subsections.

A. *Performance Portability*

Across different CPU platforms, the parallel code that compiled and ran on one x86 architecture ran seamlessly on the others as the majority of platforms adhered to OpenCL 1.1 specifications. One main reason behind good code portability in the parallel OpenCL implementation was due to minimal computation complexity within the kernels; the majority of the code required simple add, subtract, division and multiplication operations with the occasional if and for loops and did not utilize many of the native built in functions. We tested our GPU implementation on two platforms, Mac OSX v10.6.7 for OpenCL 1.0 specification on NVIDIA GeForce 320M with compute capability 1.2 and Ubuntu 11.04 for OpenCL 1.1 specification on NVIDIA GTX 580M with compute capability 2.1. After minute changes were made, we achieved the results with an accuracy of up to 1.0×10^{-6} difference compared to the serial C implementation. It is also important to note the parallel OpenCL implementation was not optimized for GPU's. However the code portability of OpenCL is evident even across CPU's to GPU's and vice versa. The parallel OpenCL code that compiled and ran on the various CPU platforms, compiled and executed easily on the GPU platforms.

B. *Code Portability*

1) CPU: Table II indicates substantial speedup from the Intel platforms given a limited amount of computational threads. The reason being that is the Intel OpenCL SDK provides implicit vectorization through the compiler, specifically the SSE [16] and AVX intrinsic

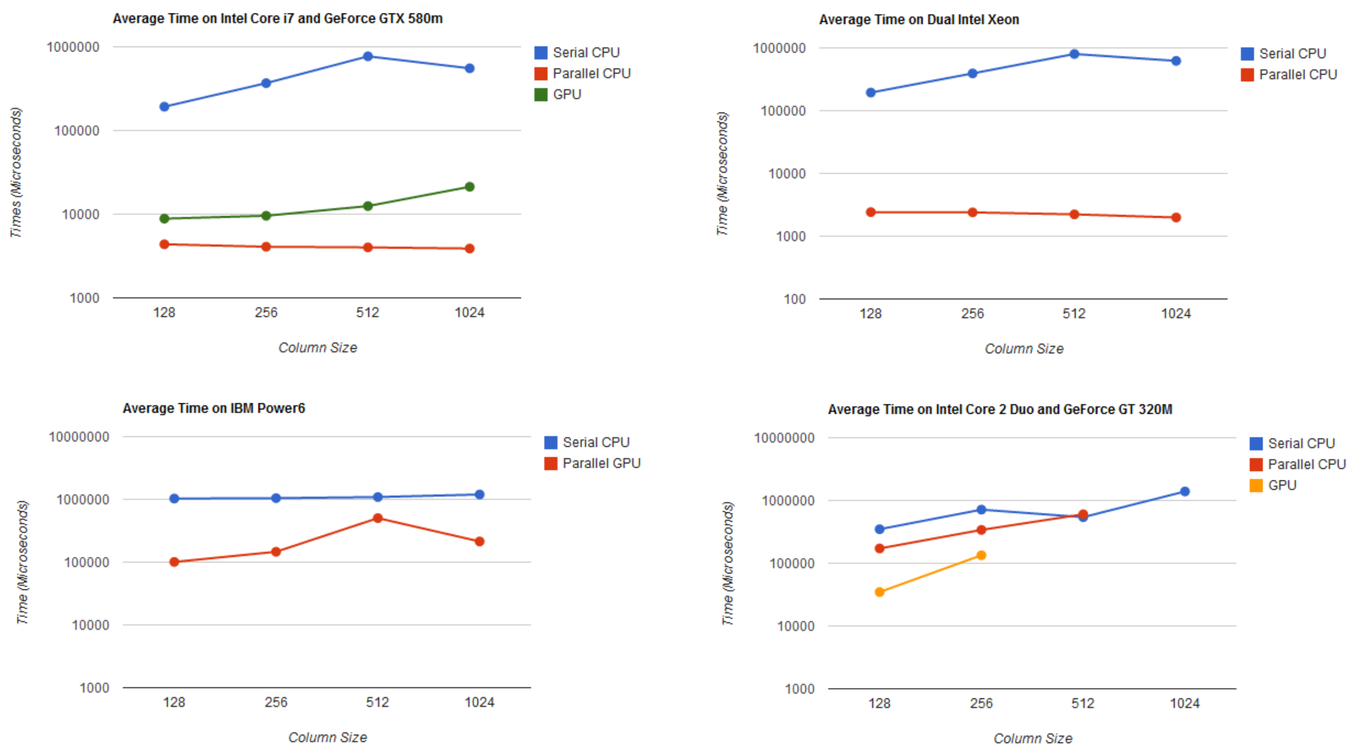


Fig. 3. Average execution time from 128 to 1024 columns

	Column Size							
	128		256		512		1024	
Platform	**Per Thread**	**Total**	**Per Thread**	**Total**	**Per Thread**	**Total**	**Per Thread**	**Total**
Intel Core i7-2630QM	5.5	55	11.25	90	24	192	17.7	142
Dual Intel Xeon X5670	3.3	80	6.8	163.7	14.9	359	13	312
GeForce GTX 580M	-	21	-	38	-	61.5	-	26
IBM JS22 Power6	1.3	10.2	0.9	7.16	0.27	2.16	0.742	5.93
Intel Core 2 Duo	1.01	2.02	1.05	2.1	0.89	0.445	-	-
GeForce GT 320M	-	10.02	-	5.329	-	-	-	-

TABLE II
SPEEDUP ACROSS ALL PLATFORMS

[3]. We utilized the Intel Offline Compiler [19] to output Intel OpenCL kernel assembly as well as ppu-objdump to output IBM OpenCL kernel assemblies. The results for the assembly of Intel platforms can be seen in Listing 1 and 2, which indicates the usage of XMM* registers which are the 128 bit Intel SSE registers [16] that are available on both the Core i7 and the Xeon X5670. In addition, we notice that the Core i7 assembly also includes YMM* register usage. The YMM* registers are the latest Intel AVX extensions [3] that supports full 8 wide floating point vectors (256 bit). Instructions such as *vmulpd* and *vpshufd* are special SIMD instructions belonging to the Intel SSE family. The main reason for the dramatic speedup in our implementation is due to the fact that the Intel OpenCL SDK helped GCC compiler to further vectorize instructions at assembly level and the coding style of OpenCL contributed greatly to this implicit vectorization. OpenCL's coding style is SIMD based as it is intended to run on GPU's too. Optimizations that are important for GPU's such as reducing thread divergence and improving stridden memory accesses greatly helps compilers for CPU's. The primary reason is due to SIMD style of kernel programming since it eliminates complex loop constructs. This helps compilers to provide more effective vectorization as it usually behaves in a conservative manner for vectorization, only proceeding when it is safe [20]; this relies upon data dependency graphs of the loops. If there are no cycles in the graph then the loop can be easily vectorized [21], and cycles are broken through the optimization of kernels originally

intended to execute on GPU's to fully exploit the SIMD feature of vector processors.

```
vmulpd YMM1, YMM4, YMM1
vpshufd XMM4, XMM5, 3
vcvtss2sd XMM4, XMM4, XMM4
vmovhlps XMM8, XMM5, XMM5
vcvtss2sd XMM8, XMM8, XMM8
```

Listing 1. OpenCL offline compiler assembly dump of a portion of kernel code on Intel i7-2630QM.

```
vmulss XMM0, XMM0, DWORD PTR [RSP +
   84]
vmovss XMM1, DWORD PTR [RIP + .
   LCPI56_0]
vaddss XMM2, XMM0, XMM1
vmovss DWORD PTR [RSP + 60], XMM2
```

Listing 2. OpenCL offline compiler assembly dumping of a portion of kernel code on Intel Xeon X5670.

In Listing 3, vectorization can also be seen on IBM Power6. Instructions starting with *va** and *vm** are the special AltiVec SIMD instruction sets [22] used for vector multiply and adds. The AltiVec instruction set enables the usage of 128 bit registers. However, the performance of the Power6 is not as good as Intel Core i7 or Xeon. The best speedup was at 128 columns where a 1.3x speedup per thread was seen. One potential reason is GCC rather than XLC compiler was utilized to compile and execute on Power6. Additionally, utilizing *-O2* and *-O3* optimization flags with GCC on Power6 produced no speedup for both serial and parallel code. In fact, the serial implementation executed slower with *-O2* and *-O3* optimization flags compared to no optimization flag usage. XLC is a commercially available compiler from IBM and is specifically designed for PowerPC architectures; it performs native implicit vectorization to utilize the AltiVec instruction sets [23]. From the work of Fahad et al [14], a 3x to 4x speedup was witnessed per core on the same JS22 IBM Power6 blade due to optimizations by XLC compiler; further evidence can be seen from [14] as the serial code on Power6 executed 2.5x faster with XLC compared to GCC. Other research has also shown XLC to perform better compared to GCC [24][25]. Why does GCC improve OpenCL performance on Intel x86 but not so much on IBM PowerPC? The reason could be due to the difference between Intel OpenCL SDK and the IBM OpenCL SDK. Intel's implementation of OpenCL 1.1 specifications is specifically optimized for Intel processors; furthermore it provides implicit vectorization by mapping the code to hardware vector units and merges OpenCL work items in order to utilize SSE and AVX intrinsic [26]. However, IBM OpenCL SDK [27] does not explicit mention implicit AltiVec vectorization is supported natively through their OpenCL compiler. Unfortunately, XLC is needed with IBM OpenCL SDK in order to implicitly generate low level AltiVec instructions which were unavailable in our test platforms. Our investigation in the IBM OpenCL kernel assembly indicated only a small fraction of AltiVec instructions were utilized while the Intel OpenCL Offline compiler indicated that both Intel Core i7 and Xeon (Listing 1 and 2) utilized more SSE and AVX intrinsic.

```
100006cc: 10 05 29 80     vaddcuw v0,
   v5,v5
100006d0: 00 00 0b b0     .long 0xbb0
100006d4: 12 00 00 00     vaddubm v16,
   v0,v0
100006d8: 00 00 04 37     .long 0x437
100006dc: 10 05 26 64     vmsumubm v0,
   v5,v4,v25
```

Listing 3. ppu-objdump of assembly on IBM Power6.

2) GPU: The GT 320M has 16 KB of local memory per SM while the GTX 580M has 49 KB of local memory per SM. Exploring local memory was severely limited due to large data size requirements in each column of SOLAR. Attempts to modify multiple kernels to use local memory blocks did not record any improvement compared to the non-local memory kernel code performance. For maximal PCI Express bandwidth utilization we experimentally used pinned memory. On PCIe Gen2 cards, pinned memory can attain greater than 5 GBps transfer rate [28]. One problem was OpenCL's limitation compared to CUDA when it comes to utilizing pinned memory. OpenCL does not have control over whether memory objects are allocated in the pinned memory. Developers can only request for pinned memory allocation by *CL_MEM_ALLOC_HOST_PTR*. The computational flow of SOLAR is not embarrassingly parallel. It included over 70 multi-dimensional arrays that needed to be allocated on global memory. This contributed to the difficulty in implementing GPU specific optimizations as we were not able to identify a specific kernel that could benefit from utilizing local memory. However, the attempts to merge kernels greatly improved GPU performance as we reduced the original 70 kernels from Fahad et al [14] to about half (36 kernels). Kernel aggregation helps to reduce kernel invocation overhead and the optimizations listed in Section V produced more performance gain. Specifically in GTX 580M, the best speedup of 61.5x was seen with the case of 512 columns. However this performance does not seem to be portable as GT 320M had a performance decrease from 10x at 128 columns to 5x at 256 columns while GTX 580M increased its speedup from 21x at 128 columns to 38x at 256 columns. The GTX 580M was using CUDA's latest OpenCL SDK that is adhering to OpenCL 1.1 specifications while the GT 320M was using an older version of the SDK that only supported the 1.0 specifications (Table I). This difference can result in compilers doing more optimizations with the GTX 580M. Both GPU's ran slower at 256 columns compared to 128 columns; the GTX 580M was 10% slower while the GT 320M was 4x slower (Figure 3). The GTX 580M not only has more cores per SM (384 CUDA Cores) compared to the GT 320M (72 CUDA Cores), but also has more memory and faster clock speeds [29]. At 256 columns, the GT 320M did not have enough physical resources to run the application efficiently. The memory limitations of the GT 320M also meant that it was not successful in running the code at column sizes 512 and 1024 as segmentation faults occurred (Table II).

VII. Discussion

The Intel Core i7 is a Sandy Bridge processor. Listing 1 indicates automatic promotion of float arrays to either float4, thereby using the 128 bit SSE registers (XMM*), or float8, which are the new 256 bit AVX registers (YMM*). Since AVX register usage was limited on Intel Core i7, we attempted to explicitly use vector data types of float8 instead of regular floats by including *__attribute__((vec_type_hint(float8)))* for each kernel header and padded floating arrays to be divisible by 8. The main benefit of utilizing manual vector data types is the ability to map the vector data to the hardware vector registers. Therefore the float8 arrays will be matched to the width of the underlying YMM* AVX registers. Although this will adversely affect the performance portability of the code given that we are targeting a specific vector width, we hope to achieve significant gains in performance for the targeted platform. We have managed to manually vectorize the SOLUV function which allows performance comparison of the manually vectorized SOLUV against the SOLUV of the original parallel OpenCL implementation. The main challenge in manual vectorization is that vector data types cannot be used in conditional statements; we utilized built-in relational functions such as *isgreater* or *isless* and called stub functions for each side of the conditional in order to resolve this. In addition, we explicitly avoided extracting vector components during computation by utilizing vector operations for all computation to eliminate forced reloading of the same vector from memory.

```
vmovaps  YMM0, YMMWORD PTR [RIP + .
   LCPI16_0]
vdivps      YMM0, YMM0, YMMWORD PTR [
   R12 + R13]
mov      RAX, QWORD PTR [RBP + 24]
vmovaps  YMMWORD PTR [RAX + R13], YMM0
mov      RAX, QWORD PTR [RBP + 32]
vmovaps  YMM0, YMMWORD PTR [RAX + R13]
vmaxps      YMM0, YMM0, YMMWORD PTR [
   RIP + .LCPI16_1]
vmovaps  YMMWORD PTR [RSP], YMM0 # 32-
   byte Spill
```

Listing 4. Intel OpenCL Offline Compiler output of assembly on Intel Core i7 with explicit AVX.

```
vmovq XMM0, RAX
vmovlhps XMM0, XMM0, XMM0
vmovaps  XMMWORD PTR [RSP + 192], XMM0
vmovaps  YMM1, YMMWORD PTR [RIP + .
   LCPI9_0]
vextractf128   XMM2, YMM1, 1
vpaddq      XMM2, XMM0, XMM2
vpshufd  XMM2, XMM2, 8
vmovaps  YMMWORD PTR [RSP + 160], YMM1
```

Listing 5. Intel OpenCL Offline Compiler output of assembly on Intel Core i7 WITHOUT explicit AVX.

Listings 4 and 5 indicate the difference in usage of registers; both listings show the same assembly. However, our results indicate a 5% to 10% speed improvement over the original parallel OpenCL code. The main reason that our speedup was not as much as expected was due to the time it took SOLUV to run compared to SOLIR; SOLUV only takes about 10% to 15% of the total runtime while SOLIR took 70% to 80% of the total runtime. We plan to further explore manual usage of AVX vector data types by converting SOLIR to utilize float8 as well.

The Intel Core 2 Duo did not show the performance improvements as seen in the Intel Core i7 and Xeon. We suspect this is due to an older version of Intel OpenCL SDK on Mac OSX (Table I). This older Intel OpenCL SDK could have a premature implementation of implicit vectorization compared to the 1.5 SDK and additionally the Core 2 Duo only supports SSE 4.1 intrinsic, which contains a limited set of SSE instructions and registers, therefore resulting in less performing assembly code. Currently, Intel OpenCL SDK 1.5 does not support Mac OSX so we are not able to fully investigate the reasons behind the results seen in the Intel Core 2 Duo. It may also be of interest to output the assembly of the executable on Mac OSX in order to confirm the existence of any vectorization.

VIII. Conclusion

We have developed a OpenCL code for a representative climate and weather physics model that is able to run across multiple different platforms with dramatic performance improvement over each core. We believe these dramatic speedups in CPU demonstrate that OpenCL provides an interface to implement light-weight multi-threaded code on CPU's as POSIX threads are often much heavier when used in parallel programming. OpenCL provides access to a multi-threaded programming and execution model while providing a lower level memory and thread management similar to that of CUDA for NVIDIA GPU's. We demonstrated autovectorization capabilities of the OpenCL compilers across platforms and have shown significant improvement gains.

References

[1] J. Y. Xu, "Opencl the open standard for parallel programming of heterogeneous systems," 2008.

[2] D. Singh, "Higher level programming abstractions for fpgas using opencl," 2011. [Online]. Available: www.eecg.toronto.edu/~jayar/fpga11/Singh_Altera_OpenCL_FPGA11.pdf

[3] INTEL.COM, "Product page," http://software.intel.com/en-us/avx, 2011.

[4] S. Zhou, D. Duffy, T. Clune, M. Suarez, S. Williams, and M. Halem, "The impact of IBM Cell technology on the programming paradigm in the context of computer systems for climate and weather models," *CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE*, pp. 2176–2186, 2009.

[5] A. Munshi, "Opencl, parallel computing on the gpu and cpu," 2008.

[6] J. Brietbart and C. Fohry, "Opencl - an effective programming model for data parallel computations at the cell broadband engine," 2010.

[7] M. J. and V. M., "GPU acceleration of numerical weather prediction," *Parallel Processing Letters Vol. 18 No. 4. World Scientific*, pp. 531–554, Dec 2008.

[8] J. M. T. H. Govett, M., "Running the NIM next-generation weather model on gpus," *The 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2010.

[9] K. R, "Gpu computing for atmospheric modeling experience with a small kernel and implications for a full model," *Computing in Science and Engineering*, 2010.

[10] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi, "Evaluating performance and portability of opencl programs," *Science And Technology*, vol. 2, pp. 781–784, 2010. [Online]. Available: http://vecpar.fe.up.pt/2010/workshops-iWAPT/Komatsu-Sato-Arai-Koyama-Takizawa-Kobayashi.pdf

[11] T. Grosser, A. Gremm, S. Veith, G. Heim, W. Rosenstiel, V. Medeiros, and M. E. de Lima, "Exploiting heterogeneous computing platforms by cataloging best solutions for resource intensive seismic applications," *INTENSIVE 2011, The Third International Conference on Resource Intensive Applications and Services*, pp. 30–36, 2011.

[12] W. Zhang, L. Zhang, S. Sun, Y. Xing, Y. Wang, and J. Zheng, "A preliminary study of opencl for accelerating ct reconstruction and image recognition," *Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE*, pp. 4059–4063, 2009.

[13] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 451–460, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1816038.1816021

[14] F. Zafar, D. Ghosh, L. Sebald, and S. Zhou, "Accelerating a climate physics model with opencl," *Symposium on Application Accelerators in High-Performance Computing 2011*, 2011.

[15] AMD, "Opencl and the amd app sdk," http://developer.amd.com/documentation/articles/pages/OpenCL-and-the-AMD-APP-SDK.aspx, April 2011.

[16] INTEL.COM, "Product page," http://www.intel.com/support/processors/sb/CS-030123.htm, Apr. 2011.

[17] S. Siewert, "Using intel streaming simd extensions and intel integrated performance primitives to accelerate algorithms." [Online]. Available: http://software.intel.com/en-us/articles/

[18] Intel, "Writing optimal opencl code with intel opencl sdk," http://software.intel.com/file/39189, 2011.

[19] INTEL.COM, "Inspect your code with the intel opencl sdk offline compiler," http://software.intel.com/en-us/articles/inspect-your-code-with-intel-opencl-sdk-offline-compiler/, Apr. 2011.

[20] M. Garzarn and S. Maleki, "Program optimization through loop vectorization," http://agora.cs.illinois.edu/download/attachments/38305904/9-Vectorization.pdf, 2010.

[21] C. M. J. Garzaran, "Loop vectorization," https://agora.cs.illinois.edu/download/attachments/28937737/10-Vectorization.pdf, 2010.

[22] IBM, "Power isa version 2.03," http://www.power.org/resources/downloads/PowerISA_203_Final_Public.pdf, 2006.

[23] IBM, "XL C/C++ for linux," http://www-01.ibm.com/software/awdtools/xlcpp/linux/.

[24] I. I. B. Buros, "Linux performance," http://www.ibm.com/developerworks/wikis/download/attachments/104533332/BillBurosSTG-LinuxonPower-PerformanceConsiderations.ppt, Feb. 2007.

[25] R. H. Team, "Survey of GCC performance," http://people.redhat.com/bkoz/benchmarks.

[26] INTEL.COM, "Intel opencl sdk," http://software.intel.com/en-us/articles/intel-opencl-sdk/, Apr. 2011.

[27] IBM, "OpenCL Development Kit for Power," https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=80367538-d04a-47cb-9463-428643140bf1.

[28] NVIDIA.COM, "Opencl best practices guide," http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/OpenCL_Best_Practices_Guide.pdf, May 2010.

[29] NVIDIA, http://www.geforce.com/Hardware/NotebookGPUs/geforce-gtx-580m.

```
for (ih=0; ih<2; ih++)
{
 for (mb=0; mb<M_BLOCK; mb++)
 {
  //calculate tda[0][ih][0][mb],tta[..],rsa
      [..]
  //calculate tda[1][ih][0][mb],tta[..],rsa
      [..]
  //calculate rra[ih][0][LM+1][mb],rxa[..]
  //calculate rra[ih][1][LM+1][mb],rxa[..]
 }
 for (L=1; L<ict; L++)
 {
  for (mb=0; mb<M_BLOCK; mb++)
  {
   //calculate tda[0][ih][L][mb],tta[..],rsa
       [..]
   //calculate tda[0][ih][L][mb],tta[..],rsa
       [..]
  }
 }
 for (im=im1-1; im<im2; im++)
 {
  for (l=ict; l<icb; l++)
  {
   for (mb=0; mb<M_BLOCK; mb++)
   {
    //update tda[m][ih][l],tta[m][ih][l],rsa[m
        ][ih][l]
   }
  }
 }
}
for (ih=0; ih<2; ih++)
{
 if(ih==0)
 {
  for (mb=0; mb<M_BLOCK; mb++)
  {
   //clear portion, update ch[mb]
  }
 }
 else
 {
  for (mb=0; mb<M_BLOCK; mb++)
  {
   //cloudy portion, update ch[mb]
  }
 }

 for (im=0; im<2; im++)
 {
  if(im==0)
  {
   for (mb=0; mb<M_BLOCK; mb++)
   {
    //clear portion, update cm[mb]
   }
  }
  else
  {
   for (mb=0; mb<M_BLOCK; mb++)
   {
    //cloudy poriton, update cm[mb]
   }
  }
  for (is=0; is<2; is++)
  {
   if(is==0)
   {
    for (mb=0; mb<M_BLOCK; mb++)
 {
  //clear portion, update ct[mb]
 }
   }
   else
   {
    for (mb=0; mb<M_BLOCK; mb++)
 {
  //cloudy portion, update ct[mb]
 }
   }
  }

  for (l=icb; l<LM+1; l++)
  {
   for (mb=0; mb<M_BLOCK; mb++)
   {
    //update tda[im][ih][l][mb], tta[....],
        rsa[....]
   }
  }
  for (l=ict; l>-1; l--)
  {
   for (mb=0; mb<M_BLOCK; mb++)
   {
    //update rra[is][im][l][mb], rxa[....]
   }
  }
  for (l=1; l<=LM+1; l++)
  {
   for (mb=0; mb<M_BLOCK; mb++)
   {
    //update fdndif[mb], flxdn[l][mb]
   }
  }
  for (l=0; l<LM+1; l++)
  {
   for (mb=0; mb<M_BLOCK; mb++)
   {
    //update fall[l][mb], fsdir[mb], fsdif[mb]
   }
  }
 }
}
```

Listing 6. Serial CLDFLX code

```
for(l = 0; l <= NLM + 1; l++)
 temp_dev_rxa = rxa[stride5D + l * NM_BLOCK +
    k];
    temp_dev_rra = rra[stride5D + l * NM_BLOCK
         + k];
    denm = 1.0 / (1.0 - dev_rsa0 *
        temp_dev_rxa);
    xx = dev_tda0 * temp_dev_rra;
    yy = dev_tta0 - dev_tda0;
    temp_fdndif = (xx * dev_rsa0 + yy) * denm;
    fupdif = (xx + yy * temp_dev_rxa) * denm;
    flxdn[l * NM_BLOCK + k] = dev_tda0 +
        temp_fdndif - fupdif;
    if(l == (NLM+1))
     fdndir[k] = dev_tda0;
     fdndif[k] = temp_fdndif;
    if(l < (NLM+1))
     temp_rr = rr[stride3D+l*NM_BLOCK+k];
  temp_rs = rs[stride3D+l*NM_BLOCK+k];
  temp_td = td[stride3D+l*NM_BLOCK+k];
  temp_ts = ts[stride3D+l*NM_BLOCK+k];
  temp_tt = tt[stride3D+l*NM_BLOCK+k];

     denm = temp_ts / (1.0 - dev_rsa0 *
         temp_rs);
         dev_tda1 = dev_tda0 * temp_td;
         dev_tta1 = dev_tda0 * temp_tt + (
             dev_tda0 * dev_rsa0 * temp_rr +
             dev_tta0 - dev_tda0) * denm;
         dev_rsa1 = temp_rs + temp_ts *
             dev_rsa0 * denm;

         dev_tda0 = dev_tda1;
         dev_tta0 = dev_tta1;
         dev_rsa0 = dev_rsa1;
```

Listing 7. upKernel: First kernel out of three produced to break data dependence in the serial CLDFLX code - loops up the columns to update tda tta rsa

```
int mask1[8] = {0,1,0,1,0,1,0,1};
int mask2[8] = {0,0,1,1,0,0,1,1};
int mask3[8] = {0,0,0,0,1,1,1,1};
i = get_global_id(0); j = get_global_id(1);
ih = mask1[j]; im = mask2[j]; is = mask3[j];
k = j * 16 + i;
stride3D = ih*(NLM+2)*NM_BLOCK;
stride5D = is*2*(NLM+2)*NM_BLOCK+im*(NLM+2)*
    NM_BLOCK;
temp = rr[is * (NLM+2) * NM_BLOCK + (NLM+1) *
    NM_BLOCK + k];
rra[is * 2 * (NLM+2) * NM_BLOCK + im * (NLM+2)
     * NM_BLOCK + (NLM+1) * NM_BLOCK + k] =
    temp;
rxa[is * 2 * (NLM+2) * NM_BLOCK + im * (NLM+2)
     * NM_BLOCK + (NLM+1) * NM_BLOCK + k] =
    temp;
dev_rra0 = temp;
dev_rxa0 = temp;
for (l = NLM; l >= 0; l --)
 temp_rr = rr[stride3D+l*NM_BLOCK+k];
 temp_rs = rs[stride3D+l*NM_BLOCK+k];
 temp_td = td[stride3D+l*NM_BLOCK+k];
 temp_ts = ts[stride3D+l*NM_BLOCK+k];
 temp_tt = tt[stride3D+l*NM_BLOCK+k];
 denm = temp_ts / (1.0 - temp_rs * dev_rxa0);
 dev_rra1 = temp_rr + (temp_td * dev_rra0 + (
     temp_tt - temp_td) * dev_rxa0) * denm;
 dev_rxa1 = temp_rs + temp_ts * dev_rxa0 *
     denm;
 rra[stride5D+l*NM_BLOCK+k] = dev_rra1;
 rxa[stride5D+l*NM_BLOCK+k] = dev_rxa1;
 dev_rra0 = dev_rra1;
 dev_rxa0 = dev_rxa1;
```

Listing 8. downKernel: Second kernel out of three produced to break data dependence in the CLDFLX code - loops down the columns to update rra rxa

```
int mask1[8] = {0,1,0,1,0,1,0,1};
int mask2[8] = {0,0,1,1,0,0,1,1};
int mask3[8] = {0,0,0,0,1,1,1,1};
mb = get_global_id(0);
l = get_global_id(1);
k = get_global_id(2);
ih = mask1[k]; im = mask2[k]; is = mask3[k];
fclr[l * NM_BLOCK + mb] = flxdn[(l+1) * (
    NM_BLOCK+1) + mb];
fall[l * NM_BLOCK + mb] = fall[l * NM_BLOCK +
    mb] + flxdn[(l+1) * (NM_BLOCK+1) + mb] *
    ctData[(ih * 4 + im * 2 + is * 1) * 128 +
    mb];
```

Listing 9. reductionKernel: Final kernel produced to break data dependence in the serial CLDFLX code - ctData is the precomputed array to eliminate dependencies across bitmasks