Accelerating a climate physics model with OpenCL

Fahad Zafar, Dibyajyoti Ghosh, Lawrence Sebald, Shujia Zhou
{fahad3, dg9, lsebald1, szhou} @ umbc.edu
University of Maryland Baltimore County

***Abstract*—Open Computing Language (OpenCL) is fast becoming the standard for heterogeneous parallel computing. It is designed to run on CPUs, GPUs, and other accelerator architectures. By implementing a real-world application, a solar radiation model component widely used in climate and weather models, we show that OpenCL multi-threaded programming and execution model can dramatically increase performance even on CPU architectures. Our preliminary investigation indicates that low-level vector instructions and code representations in OpenCL contribute to dramatic performance improvement over the serial version when compared with the execution of the serial code compiled across various compilers on multiple platforms with auto vectorization flags. However, the portability of OpenCL implementations needs to improve, even for CPU architectures.**

***Index Terms*—Multi-threaded programming; Parallel computing; Heterogeneous architectures; Climate Model; IBM Cell B.E.; OpenCL; Vectorization; Compilers; GCC; ICC; IBM XLC**

I. Introduction

The demand to increase forecast predictability has been pushing climate and weather models to increase model grid resolution and include more physical processes. Accelerating climate simulations with the help of emerging multicore computing paradigms provide us with tools to address these demands. Current trends in the computing industry have moved from optimizing performance gains on single-core processors to increasing the overall performance through parallel computing with many-core processors. This trend has been all too common in GPUs in the past. Now it is being widely adopted by CPU architectures as well.

OpenCL provides a standard heterogeneous parallel programming environment for applications to execute on CPUs, GPUs, and other accelerators such as DSPs and mobile processors [1], [2]. To support such a wide array of processor architectures, OpenCL puts forward a thread-extensive model for programming. OpenCL code typically comprises of multiple kernels that are executed over a multi-threaded grid. There is tremendous interest to develop an OpenCL application and run it on CPUs as well as GPUs since OpenCL claims to be a standard cross-platform programming framework. Many authors have discussed architectural independence as benefits of OpenCL [3], [1], [4]. Though there are many examples and code fragments available, cross-platform OpenCL implementation examples are hard to find for real-world applications. In addition, to our knowledge there is no report available on the portability of OpenCL in terms of code and performance with CPU as the compute device across platforms.

The OpenCL execution model is based on kernel execution where the kernel is a function declared in a program and is executed on a compute device. In this paper, we use the term OpenCL compute device and OpenCL device interchangeably. On submission of a kernel instance called a *work-item* the host device allocates a global ID for the work item. Each work item executes the same code but on different datasets and can take a different path of execution. Work items are organized into *work-groups*. Work-groups are given unique work-group IDs with the same dimensionality as that of the index space used to define the work-items. Following this logical assignment of data and tasks, work-items execute concurrently on the processing elements of a compute device. The host maintains a queuing data structure called a *command-queue* to coordinate *in-order* or *out-of-order* execution of kernels on the device.

In this paper we investigate how OpenCL will perform in CPUs and its portability among CPUs through parallelization of a real-world climate physics application, the Goddard solar radiation model component [5] with the OpenCL implementations on IBM PowerPC and POWER6 blades as well as on the Intel x86 architecture with Mac OS X. The solar radiation model component was originally written in Fortran. Over the years the structure of this Fortran code has been stable. It represents a typical climate physics model component whose calculations are carried out along the vertical (altitudinal) direction (so-called column physics). This particular code was ported to the IBM Cell Broadband Engine by Zhou et al [6] where detailed code structure analysis and performance gains were reported. We implemented the serial version of the C code in OpenCL *version 1.0*, manually optimized all sections

of the code to run in a multi-threaded fashion using OpenCL kernels and benchmarked the code on IBM JS21 and JS22 blades, on a POWER6 AIX system and on Mac OS X *versions 10.6.4 and 10.6.7*. We used the approach of extracting compute-intensive kernels without changing the overall code structure. This allowed us to have multiple manual optimizations for specific architectures while maintaining the code structure. We found a 3X$\sim$4X performance improvement per core over the original serial code compiled with GCC. We believe these dramatic performance gains on the CPU show that OpenCL provides interface to implement light-weight multi-threading code on the CPU [7]. POSIX Threads is known to be heavy-weight threads when used in multi-threaded programming, which increases program memory requirements and adds to context switching costs. OpenCL provides access to a multi-threaded programming and execution model as well as a low level API for memory and thread management. Its relaxed memory consistency model is similar to that of CUDA [8] developed for NVIDIA GPUs. We report our preliminary investigation results on OpenCL performance on CPUs and on serial code performance across platforms when compiled with GCC, Intel C++ and IBM XLC compilers. We also touch upon vectorization results from these three compilers with respect to two reference code sections in the serial code.

Our findings show that the GCC auto vectorization[9], [10] implementation fails in certain cases where the OpenCL compiler succeeds in generating fast executable code. This is particularly true for certain nested loop constructs, though the GCC vectorization unit performs as expected in many other cases in this model. Similar results were obtained from Intel ICC compiler and IBM XLC compiler for these nested loop constructs. We compiled the serial version of the solar radiation model component code with the GCC for the SSE{2,3,4}[11] and the Altivec[12] target instruction sets respectively. IBM XLC serial code compilation used Altivec[13] as target architecture. OpenCL kernels for sections of code comprising of these loop constructs have been compared with the GCC, ICC and XLC vectorization report for these code sections. The details of these findings will be elaborated in the Performance Analysis subsection under Section III. The increased performance of our OpenCL implementation can be attributed to implicit instruction and data parallelism by the OpenCL compiler on Intel[14], [15] and IBM platforms as well as to the efficient execution pipelining and better memory management.

Based on our experience, current implementations of OpenCL from IBM and Apple are not portable in terms of code. The same OpenCL code runnable in IBM JS21 and JS22 blades did not compile or run seamlessly in Mac OS X as one would expect. Some code modifications were required to get correct results. Thread scheduling differences on the platforms caused the program to output correct answer in one case (IBM blades) while the incorrect answer in the other case (Mac OS X). After successfully porting the first two sections of the code with OpenCL, the speedups on Mac OS X with Intel CPU are similar to those on the IBM platform, which indicates the OpenCL performance is portable across platforms to some extent.

The rest of the paper is organized as follows. In Section II, we discuss the solar radiation code structure and its OpenCL implementation. In Section III we present the results on IBM blades, POWER6 AIX and Mac OS X and we discuss potential reasons behind dramatic performance gain on CPUs with OpenCL. In Section IV we discuss related works and further analysis of performance gain and conclude in Section V.

II. Code Structure and OpenCL Implementation

A production-quality climate or weather model code can span up to a few hundred thousand lines. Most of these codes are written in Fortran. We used single-precision, C version of a representative, compute-intensive physics model component code with about 1500 lines as a serial baseline. This baseline is used to confirm the correctness of the results obtained with the multi-threaded parallel version as well as to calculate the performance speedup achieved against it. The selected code is the solar radiation model component, SOLAR, which has been widely used in climate and weather models. This particular version of the code is from the production version of the NASA GEOS-5 climate physics model. This model component along with the IRRAD (the infrared radiation component) typically takes 20 percent of the execution time[6]. The exact fraction of SOLAR execution time compared to the total model varies among the models. For example, in NASA GEOS5 atmospheric model, the solar radiation can take 5%$\sim$10%. However that fraction could increase significantly if the aerosol effect is included. Reducing the execution time of SOLAR can benefit the climate and weather simulations in two ways: allowing a reduced overall simulation time and an increase in the complexity of models within the same simulation time. The climate physics model component executes calculations only along the vertical (altitudinal) direction. In a climate or weather physics model, most of the physical processes in a whole or partial globe are modeled with such columns

at each point of a horizontal grid such as the latitudinal-longitudinal grid. In the parallel version, SOLAR is

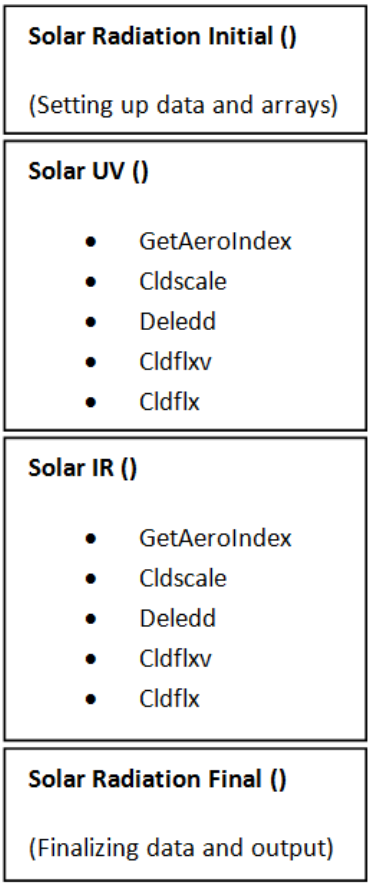

Fig. 1. The code structure of the solar radiation model component. The same routine structure was mapped to OpenCL kernels in the parallel version. One routine was mapped to multiple kernels in some cases.

multi-threaded and runnable in multiple processor cores within a shared-memory system. The parallel code is composed of over about 70 kernels. These kernels execute on an OpenCL compute device, which can be the host CPU cores or GPU streaming multiprocessors depending on user defined initialization parameters. Thus, the OpenCL code can be tested for various simulation system sizes (i.e., numbers of physics columns), which is bounded only by the memory limitations. All performance results presented in this paper use 128 columns that are independent of each other. The kernels are multi-dimensional, depending on the data arrays used. The kernels do not have a one-to-one mapping with the serial solar radiation code functions. We have written multiple kernels for each subroutine in some particular cases. In this way we ensure maximal utilization of the hardware computing resources. Most functions with conditional flow of codes were split into separate kernels working in synchronization to break the linear program flow and executed concurrently.

All data arrays are stored in OpenCL global memory. There is no data transfer between the host and the OpenCL device (CPU) other than the initialization variables and the final results. All the computations have been done inside a blade or desktop computer. Next, we spread the kernel execution over an n by m thread grid. n and m are valued with respect to the arrays present in the specific kernels so that each thread deals with one element of the array in the typical OpenCL threading fashion. The code contains about 70 arrays ranging from 1 to 3 dimensions. Kernel executions are synchronized with barriers to ensure the data integrity. Two code segments which we analyze later has been labeled as *section 1* (Appendix A) and *section 2* (Appendix C) in the paper. The first code segment belongs to the *Solar Radiation Initial()* and the second code segment belongs to SolarIR() section of the climate physics model.

III. Results

A. Performance Measurement

We tested the complete parallel implementation of the serial code in OpenCL on the IBM PowerPC and POWER processor architectures. The parallel code output results within a difference of less than 1 percent (maximum difference of 0.0001) compared to the (original) serial version. The solar radiation code is divided into four sections (Figure 1). Each section contains anywhere between $5-25$ kernels. The code uses *integer* and *floating point* data types. Dramatic performance improvement was noticed for parallel execution on CPU. Our analysis shows that the speedup can be attributed to implicit vectorization support by the OpenCL compiler infrastructure along with the kernel execution model and its threading architecture[16]. Further, splitting the functional routines into small and distributed kernels has allowed maximum parallelization of the code and resulted in a faster completion of the task. These fine tunings for such a complex implementation were possible due to the simpler programming interface provided by OpenCL as compared to manual SIMD vectorization. Explicit vector programming is time consuming, error prone and also machine dependent. Moreover, vectorization is not trivial. It has been observed that difficulties in optimizing code for SIMD architecture stems from hardware constraints too [17]. Certain kernels were executed *out of order* to maximize throughput while all of them were executed according to the size and dimensions of the arrays being manipulated. Thus, no work cycles were wasted due to inconsistent size and dimensions for memory objects.

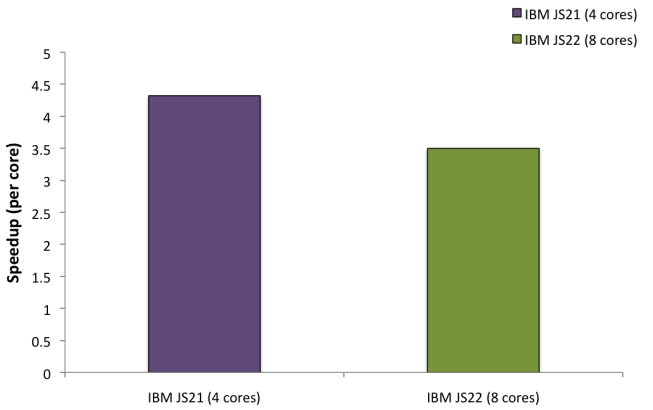

Fig. 2. Total speedup achieved per processor core using the OpenCL parallel implementation compared to the serial C version on CPU architectures.

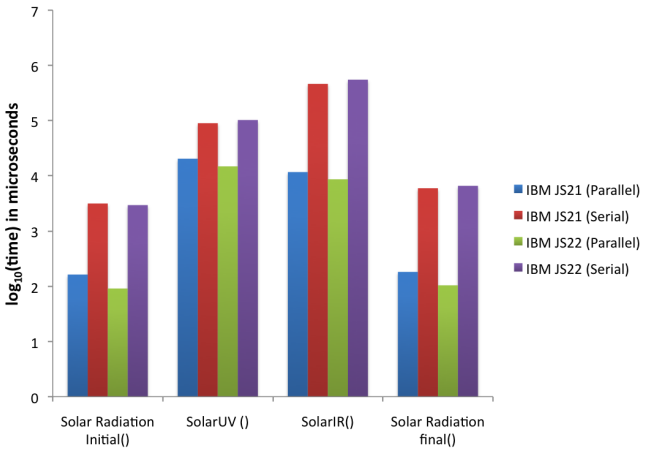

Fig. 3. Performance gain comparison per section of the code for IBM JS21 and IBM JS22

Figure 2 presents the performance results on the IBM JS21 and JS22 blades. The JS21 blade contains 2 dual-core PowerPC970MP 2.3GHz processors while the JS22 blade has 2 quad-core POWER6 4.0 GHz processors. All the tests were run with number of parallel compute cores on the OpenCL device set to CL_DEVICE_MAX_COMPUTE_UNITS. GCC has been used only for serial performance measurement on a single core. Figure 3 shows comparative performance gains per section of the code. The highest speedup is observed in the compute-intensive part (*SolarIR()*). The results produced by parallel and serial versions of the code were first run in the test mode where the serial version and parallel versions were run side by side. After each subroutine execution, all the arrays were cross-checked element by element to verify results. At the end of execution, when all sections of the code had passed the checks, the parallel code was run in benchmark mode where the timing data were collected. In both figure 3 and figure 4 we have plotted logarithmic value of the timing data attained on various platforms for the four code sections.

1) Porting to Mac OS X: OpenCL is designed to support broad processor architectures with the goal of writing once and running everywhere. A user simply compiles an application written in OpenCL with compile flags to locally optimize the code and the code is supposed to be portable to all supported platforms. Our experience with porting OpenCL did not go this smoothly. We ran the OpenCL implementations for IBM PowerPC and POWER processor on Mac OS X with limited success. The initial OpenCL code did not compile due to some library dependency issues and the first results obtained were incorrect, as the Apple OpenCL drivers did not execute the code with expected results. It appears that this is caused by premature OpenCL implementations from different vendors. We modified parts of the code to compile it correctly and reordered function calls to get correct results. Only then we were able to test the first two parts of the SOLAR code (Figure 4). One of the modifications deals with loading arrays of dimension higher than two on Mac OS X OpenCL compute device. The *clCreateBuffer*[18] OpenCL API requires host pointer as one of the input parameters. The IBM blade compatible code handles this data reference without any error while on Mac OpenCL we had to modify array representation from higher dimensionality to one dimension to pass it as host pointer reference to the OpenCL compute device. Additionally, the *clCreateContextFromType*[19] API worked on the Mac OS X OpenCL implementation while we used a custom procedure on IBM systems to get the same result as *clCreateContextFromType* didnt work on the IBM OpenCL implementation. OpenCL *version 1.1* might solve some of the porting issues we encountered.

At the time of writing this paper, we have results for the first two sections of the code and they look very promising on an alternate platform compared to the complete IBM OpenCL implementation. The speedup registered in the parallel version of *SolarUV()* section for Mac OS X (Intel 2.66 GHz Core 2 Duo) is about 2.67 (Figure 4) compared to 4.4 and 6.9 (Figure 3) for IBM JS21 and JS22 in the same section. One conclusion to be drawn at this point is that OpenCL vectorization speeds up applications across multiple CPU platforms. We are still in process of optimizing the complete SOLAR code for Mac OS X but the speedup achieved is consistent with the results obtained on IBM platform.

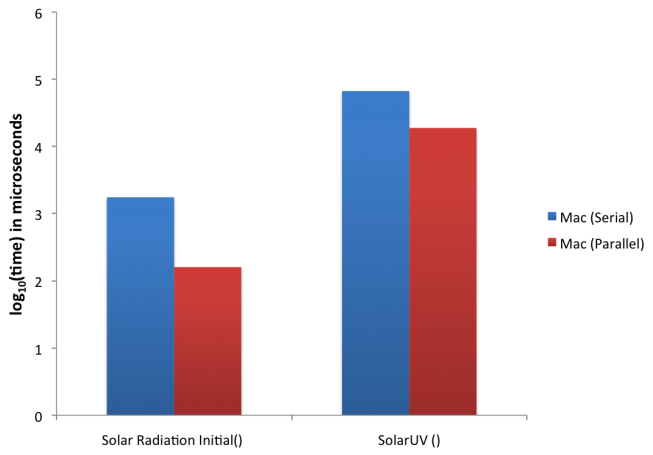

Fig. 4. Performance gain comparison achieved for the first two sections of the code on the Mac OS X. CPU Processor : Intel 2.66 GHz Core 2 Duo

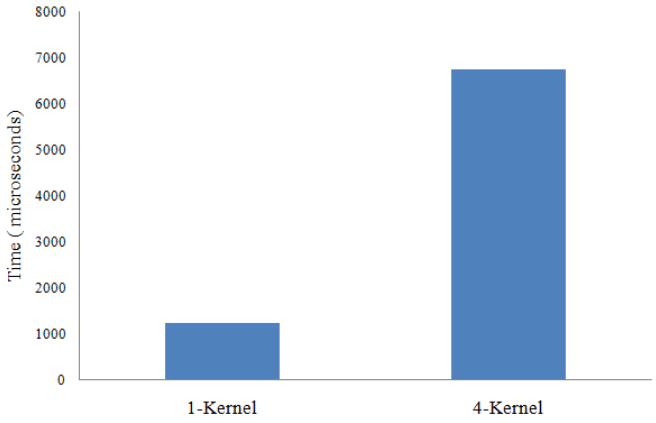

Fig. 5. Timing results for 30 executions of cldflx() subroutine with a single kernel and 4-kernel split implementation on the IBM JS22.

2) Subroutine multi-kernel implementation: Porting C code to OpenCL provides a design decision challenge based on the nested dependency structure of the code. Dividing one subroutine with multiple levels of iteration loops and a mix of decision statements can be tricky at times. We noticed that splitting some subroutines into multiple kernels at times sped up the processing, while in some cases it reduced performance. Figure 5 presents the results obtained by splitting the *cldflx()* function into 4 kernels based on the 4 major iteration loops inside the *cldflx()* code structure. *cldflx()* computes the transmittances and reflectance for a composite of layers. Layers are added one at a time, going down from the top. There are 4 major iteration loops within the function that execute with nested code elements contained within each loop.

It is evident that dividing serial code to the smallest parallelizable element is not the best decision every time when converting from serial to parallel OpenCL code. The slowdown occurs due to the additional cost of loading and setting up the kernels, which cannot be overridden by the speedup (if any) offered by using additional kernel executions in this case. Contrary to this example, not splitting the *SolarIR()* function, which contains a huge part of the physics implementation into multiple kernels, results in a loss of performance. The reason behind this is that many parts of the code that can benefit from a parallel execution are serial and can only benefit from the limited compiler specific vectorization. Thus it is important to know when to split the code and whether splitting the code will produce a performance benefit with the additional cost of executing a separate kernel.

B. Performance Analysis

In this subsection we explore the reasons behind the performance gains achieved in our work. We selected two code segments from the serial version of climate model to analyze vectorization behavior of complex nested loop structures across compilers. For comparing results we used disassembled output from GCC *version 4.2.1* for the serial code against the Intel OpenCL Offline Compiler[14] generated OpenCL kernel assembly output along with the vectorization reports from GCC *versions 4.1.2 and 4.2.1*, Intel C++ Compiler (ICC) *version 12.0.4* and IBM XLC compiler *version 10.1* on multiple platforms. The solar radiation code was compiled on the Mac OS X *version 10.6.7* platform using GCC *version 4.2.1 (Apple Inc. build 5664)*, Intel C++ Compiler *version 12.0.4* and on IBM blades using GCC *version 4.1.2 20080704 (Red Hat 4.1.2-48)* and on POWER6 AIX using IBM XLC *version 10.1* and GCC *version 4.1.2*. We refer to the first segment as *section 1* (check Appendix A for a complete listing of the code segment) and the second segment as *section 2* (check Appendix C for Section 2 nested loop structure listing) in this paper. We have used the first segment to represent sections of the serial code where GCC, Intel C++ compiler and IBM XLC compiler all fail to vectorize instructions leading to overall performance degradation of the solar radiation code. GCC′s tree-ssa based loop vectorization module complains under a number of circumstances[20], two of which we noticed occurring frequently in solar radiation model component compilation phase:

1) Vectorization failure due to un-handled data references.
2) Vectorization failure due to too many basic blocks in the loop.

ICC and XLC vectorization modules reported similar results. The compiler behaves in a conservative manner when it comes to vectorization - it will vectorize only if it absolutely safe to do it[21]. Compiler looks at the data dependence graphs of the innermost loops for vectorization. If there are no cycles in the graph, then the loop can be vectorized although sometimes statement reordering becomes necessary[22].

A basic block refers to a sequence of instructions having a single entry point and a single exit point[23] meaning no jump instruction can be a part of a basic block and no basic block can be a target of a jump instruction. During compilation GCC decomposes a program into multiple basic blocks each of which forms a vertex of the *control flow graph (CFG)* used by the tree-ssa based vectorization unit of GCC.

In *section 1* code segment our zone of interest is the nested for-loop construct shown in Listing 1 (line number **2-4** in Appendix A). GCC, ICC and IBM XLC all report vectorization failure for the nested for-loop with unhandled data reference errors. Similarly, for *section 2* all of these compilers reported vectorization failure. In our OpenCL kernel implementation we changed the two-dimensional array representation (line number **6-10,13,15,21** in Appendix A) to a one-dimensional array (line number **27-30,32,34,38,39** in Appendix B) and only the innermost for-loop was retained (line number **37-40** in Appendix B). We noticed that changing code representations in the serial code from multidimensional arrays to a single dimension row major representation to mimic OpenCL kernel code structure didnt facilitate vectorization of *section 1* and *section 2*. We conjecture that elimination of complex loop constructs in the OpenCL kernel contributes to effective vectorization by OpenCL compiler through bypassing unhandled data references and the too many basic blocks issue we encountered.

```
for (k=0; k<LM; k++){
   for (i=0; i<M_BLOCK; i++){
      dp[k][i] = pl[k+1][i]-pl[k][i];
      pa[k][i] = 0.5*(pl[k][i]+pl[k
         +1][i]);
      scal[k+1][i] = dp[k][i]*pow(pa[k
         ][i]/300.,.8);
      wh[k][i] = 1.02*wa[k][i]*scal[k
         +1][i]* (1.+0.00135*(ta[k][i
         ]-240.)) + 1.e-9;
```

Listing 1. A part of serial code for section 1

The corresponding GCC disassembler output is given below in Listing 2.

```
for (k=0; k<LM; k++){
   for (i=0; i<M_BLOCK; i++)
add     ebx,0x1
mov     edx,DWORD PTR [ebp-0x6d09c]
cmp     ebx,0x80
jne     5ae0 <sorad+0x280>
```

Listing 2. A part of GCC assembly dump for section 1

A part of the vectorized assembly code generated by Intel OpenCL Offline Compiler for section 1 is given below.

```
pshufd   XMM0, XMM0, 0
paddd XMM0, XMMWORD PTR [LCPI4_0]
movaps   XMMWORD PTR [ESP + 32], XMM0
   # 16-byte Spill
mov      EAX, EBX # Reload Reuse
imul         EAX, DWORD PTR [ESP + 28]
    # 4-byte Folded Reload
movd  XMM0, EAX
pshufd   XMM0, XMM0, 0
paddd XMM0, XMMWORD PTR [ESP + 32] #
   16-byte Folded Reload
movd  EBX, XMM0
mov          EAX, DWORD PTR [EBP + 12]
movups   XMM0, XMMWORD PTR [EAX + 4*
   EBX]
```

Listing 3. A part of OpenCL assembly dump of section 1 kernel

pshufd, paddd, movaps, movups are special SIMD instructions belonging to Intel Advanced Vector Extensions.

We noticed a speedup of about ∼40x for *section 1* code segment on the Mac OS X. The second code segment we selected from solar radiation code (refer to the Appendix C for complete listing of the second segment) contains five levels of nested loop construct (refer to line number **30-60** in Appendix C). Intel OpenCL Offline Compiler successfully vectorized this code segment as well.

Figure 6 shows the execution time data for *section 1* and *section 2* on IBM JS21, JS22 blades, IBM POWER6 AIX and Mac OS X 10.6.7. The running time for these code sections and for the overall serial code is much better when compiled on ICC and IBM XLC compared to GCC. This should come as no surprise as IBM XLC and ICCs overall optimization modules are known to perform better when compared with GCC[24], [25]. Though GCC compiled code execution was much faster on Mac OS X *version 10.6.7* as compared to other platforms.

The OpenCL compiler on IBM architectures uses the Altivec instruction set, while the OpenCL compiler on Intel architectures uses Streaming SIMD Extensions 4.1

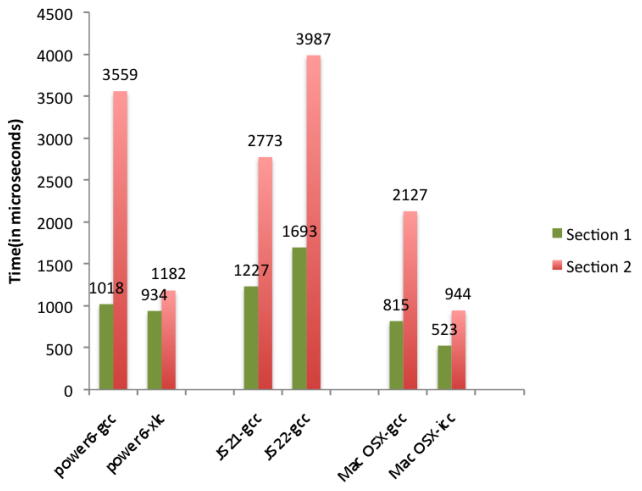

Fig. 6. Timing results for serial code compiled on JS21-GCC, JS22-GCC, Mac OS X-GCC,ICC and POWER6 AIX-GCC,IBM XLC

[26]. We implemented a minor subset of the SOLAR code with the Altivec instruction set and found that the execution time was cut in half compared to the serial version. There is an extensively nested program flow in the SOLAR code along with multi-dimensional arrays and different operations depending on conditional statements. The code becomes very complex, and it becomes very time consuming to implement a subroutines like *SolarIR()* with the Altivec instruction set. The complex code structure is also the reason why a typical C compiler such as GCC cannot utilize the vector instruction sets since the compiler cannot find the appropriate known patterns for this code. This is the problem that OpenCL seems to solve, bringing about orders of magnitude in speedup for applications targeted to run on CPUs, compared to GPUs that are already built on the foundations of multi-core paradigms.

IV. Discussion

These kinds of performance gains through vectorization on the CPU are not new. VAST ([27]) uses the native SIMD vector instructions and increases performance dramatically. [28] reports a relative speedup of 20.5X to 27.3X depending on the vector compiler (vcc and vlc) they used for certain implementations on VMX hardware. A simple dot product example gains up to 4.9X speedup with automatic vectorization. They also show that some hand tuning techniques can further improve the performance. Thus, if a programmer is provided with an easy interface to the vector libraries for complex programming tasks, better performance is often achievable.

OpenCL is itself implemented as a library with a built-in compiler to compile kernels at runtime. The current IBM implementation is based around a modified version of their XLC compiler. XLC is designed specifically for the POWER architecture. The use of XLC by IBM in their implementation of OpenCL should come as no surprise and it explains why XLC is capable of sophisticated Altivec code generation. The OpenCL implementation in Mac OS X is based on Low Level Virtual Machine (LLVM) with the Clang front-end. LLVM was designed as an infrastructure for building compilers, with a large focus on optimized code generation. LLVM supports the Intel architecture quite well, explaining why it creates such well-optimized code from the OpenCL kernel functions that we have implemented on Mac OS X thus far.

It should be noted that the OpenCL compiler might make certain assumptions that GCC cannot afford to make for naive C code. For example, it can assume that the kernel functions are not trying to do non-computational tasks such as file I/O. The OpenCL compiler can assume that the given computation is meant to be run as a many-threaded piece of code. OpenCL kernel functions are generally small, self-contained. They are usually compute-intensive and are executed over an abstract index space called N-Dimensional computation domain. For instance, 16 parallel units of work could be associated with an index space from 0 to 15. Alternatively, using 2-tuples, those 16 units of work could be associated with (0,0) to (3,3). OpenCL kernels can be grouped to form workgroups. Several of these workgroups are concurrently executed on available OpenCL compute devices[15]. The OpenCL compiler can potentially take these assumptions into account in its code generation. For both *section 1* and *section 2* GCC, ICC and IBM XLC reported vectorization failure. It will be interesting to further study how these compilers handle vectorization if the climate model′s complex nested for-loop constructs are broken down to simpler loop constructs as was done with the OpenCL kernel functions, but it would require a lot more effort and time.

We plan to modify the OpenCL code appropriately to run on GPU. We are interested in finding out how different OpenCL implementation techniques that perform well on CPUs will behave on GPUs and whether the performance can be portable, in addition to the code portability. But the potential problem of GPU memory constraints will limit the use of large global memory datasets. At the time of writing this paper we are still working on porting the IBM specific parallel code in its entirety to the Mac OS X implementation of OpenCL as well as to Nvidia′s Fermi architecture.

V. Conclusion

We have successfully demonstrated that the multi-threaded programming and execution models of OpenCL can significantly increase the performance of a real-world climate and weather application, solar radiation model component, in IBM POWER and PowerPC and POWER6 CPU architectures. Similar performance improvement has also been obtained in Intel CPUs. The immature implementation of OpenCL from various vendors prevents us from running the same OpenCL code on Intel CPU architectures, which is runnable on IBM CPU architectures. Our preliminary investigations with Altivec instructions as well as OpenCL generated assembly code reveals that the dramatic performance improvement in CPUs arises from a much better implicit vectorization support provided by the OpenCL compiler infrastructure as compared to auto-vectorization support provided by popular compilers like GCC, ICC and IBM XLC.

Acknowledgment

We would like to thank the insightful comments and suggestions from the reviewers. This work is partially supported by IBM through the Center for Hybrid Multicore Productivity Research, UMBC.

References

[1] J. Y. Xu, "OpenCL the open standard for parallel programming of heterogeneous systems," 2008, institute of Information and Mathematical Sciences Massey University at Albany, Auckland, New Zealand.

[2] A. C. a. T. T. C. Desh Singh, "Higher level programming abstractions for fpgas using opencl," www.eecg.toronto.edu/~jayar/fpga11/Singh_Altera_OpenCL_FPGA11.pdf, 2011.

[3] A. Munshi, "OpenCL, parallel computing on the GPU and CPU," *SIGGRAPH*, 2008.

[4] J. Breitbart and C. Fohry, "OpenCL - an effective programming model for data parallel computations at the cell broadband engine," *Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW), IEEE International Symposium*, 2010.

[5] M. D. Chou and M. J. Suarez, "A solar radiation parameterization (clir-ad-sw) for atmospheric studies," 1999.

[6] 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," pp. 2176–2186, 2009.

[7] L. Howes, "OpenCL parallel computing for CPUs and GPUs," *Advanced Micro Devices (AMD) presentation*, http://developer.amd.com/gpu_assets/OpenCL_Parallel_Computing_for_CPUs_and_GPUs_201003.pdf.

[8] NVIDIA, "NVIDIA CUDA compute unified device architecture," 2007, http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf .

[9] i. GCC Team, Ira Rosen, "Auto-vectorization in gcc," http://gcc.gnu.org/projects/tree-ssa/vectorization.html#using, Apr. 2011.

[10] D. Naishlos, "Autovectorization in gcc," 2004, pp. 105–117.

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

[12] I. Inc., "Tuning options to consider with gcc," http://www.ibm.com/developerworks/wikis/display/hpccentral/Tuning+options+to+consider+with+gcc, Apr. 2009.

[13] B. V. Atyam and I. I. Calvin Sze, "Application-performance tuning and optimization on power6," http://www2.fz-juelich.de/jsc/datapool/jump/JUMP-AIX-POWER6-AppsPerformanceTuning-wp032008.pdf, Feb. 2008.

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

[15] I. Ofer Rosenberg, "Optimizing opencltm on cpus," http://www.khronos.org/developers/library/2010_siggraph_bof_opencl/OpenCL-BOF-Intel-SIGGRAPH-Jul10.pdf, July 2010.

[16] K. Group, "Opencl 1.1 specification (revision 36, september 30, 2010)," http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf, Apr. 2011.

[17] M. S. at Computational Research Laboratories (CRL) India, "Vectorization - writing c/c++ code in vector format," http://software.intel.com/en-us/articles/vectorization-writing-cc-code-in-vector-format/, Jan. 2011.

[18] K. Group, "clcreatebuffer , opencl 1.0 manual," http://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/clCreateBuffer.html.

[19] ——, "clcreatedevicefromtype , opencl 1.0 manual," http://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/clCreateContextFromType.html.

[20] U. o. I. a. U.-C. Saeed Maleki, "Evaluation of vectorizing compilers," https://agora.cs.illinois.edu/download/attachments/26546085/Vectorization-Seminar.pdf, Apr. 2011.

[21] M. Garzarn and U. o. I. a. U.-C. Saeed Maleki, "Program optimization through loop vectorization," https://agora.cs.illinois.edu/download/attachments/38305904/9-Vectorization.pdf, 2010.

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

[23] R. M. Stallman and the GCC Developer Community, "Gnu compiler collection internals," http://gcc.gnu.org/onlinedocs/gcc-4.0.4/gccint.pdf, 2010-11.

[24] I. I. Bill 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 release notes," http://software.intel.com/file/33857, 2010-11.

[27] CresentBaySoftware, "VAST . Advanced compiler optimization for high-performance systems," 2005, "http://www.crescentbaysoftware.com".

[28] B. Gibbs, R. Arenburg, D. Bonaventure, B. Elkin, R. Grima, and A. Wang, "IBM E server bladecenter js20 powerpc 970 programming environment," 2005, "IBM.com/redbooks".

[29] J. Michalakes and M. Vachharajani, "GPU acceleration of numerical weather prediction. proceedings of the workshop on large scale parallel processing," *IPDPS, Miami*, 2008.

VI. Appendix A - Section 1 serial code

```

   for (k=0; k<LM; k++)
   {
      for (i=0; i<M_BLOCK; i++)
      {
         dp[k][i] = pl[k+1][i]-pl[k][i];

         pa[k][i]    = 0.5*(pl[k][i]+pl[k+1][i]);
         scal[k+1][i] = dp[k][i]*pow(pa[k][i]/300.,.8);
         wh[k][i]    = 1.02*wa[k][i]*scal[k+1][i]
         * (1.+0.00135*(ta[k][i]-240.)) + 1.e-9;

         oh[k][i]    = 1.02*oa[k][i]*dp[k][i]*466.7 + 1.e-8;

         x=1.02*10000.*dp[k][i];



         for (l=0; l<NA_NUM; l++)
         {
            qaero[l][k][i] = x*raero[l][k][i];
         }
      }

   }
```

VII. Appendix B - Section 1 OpenCL kernel source

```

__kernel void
sectionB(    __global float * dp,
       __global float * pl,
       __global float * pa,
       __global float * scal,
       __global float * wh,
       __global float * wa,
       __global float * ta,
       __global float * swh,
        __global float * oh,
         __global float * oa,
           __global float *qaero,
           __global float *raero,
           __global const int *hM_BLOCK,
           __global const  int *hLM,
           __global const  int *hNA_NUM
){
         // Vector element index
         int i = get_global_id(0);
         int k = get_global_id(1);

         int M_BLOCK = hM_BLOCK[0];
         int LM = hLM[0];
         int NA_NUM = hNA_NUM[0];

         dp[k * M_BLOCK + i] = pl[(k+1) * M_BLOCK + i]-pl[k * M_BLOCK + i];
         pa[k * M_BLOCK + i]    = 0.5f*(pl[k * M_BLOCK + i]+pl[(k+1) * M_BLOCK + i]);
         scal[(k+1) * M_BLOCK + i] = dp[k * M_BLOCK + i] * pow(pa[k * M_BLOCK + i]/300.0f, .8f);
         wh[k * M_BLOCK + i]    = 1.02f*wa[k * M_BLOCK + i]*scal[(k+1) * M_BLOCK + i]
         * (1.0f+0.00135f*(ta[k* M_BLOCK + i]-240.0f)) + 1.e-9f;
         oh[k * M_BLOCK + i]    = 1.02f*oa[k * M_BLOCK + i]*dp[k * M_BLOCK + i]*466.7f + 1.e-8f;

         float x=1.02f*10000.0f*dp[k * M_BLOCK + i];

         int l = 0;
         for (l=0; l<NA_NUM; l++){
            qaero[l * LM * M_BLOCK  + k* M_BLOCK + i] = x*raero[l * LM * M_BLOCK + k* M_BLOCK + i];
         }
}
```

VIII. Appendix C - Section 2 serial code

```
for (ih=ih1-1; ih<ih2; ih++) {
      /*level 1 nesting*/
      if(ih==0) {
         /*level 2 nesting*/
```

```
        for (mb=0; mb<M_BLOCK; mb++)
            {
            ch[mb]=1.0-cc[0][mb];
        }
    }
    else {
        for (mb=0; mb<M_BLOCK; mb++)
            {
            ch[mb]=cc[0][mb];
        }
    }
    /*level 1 nesting*/
    for (im=0; im<2; im++) {
        /*level 2 nesting*/
        if(im==0) {
            /*level 3 nesting*/
            for (mb=0; mb<M_BLOCK; mb
                ++) {
                cm[mb]=ch[mb]*(1.0-cc
                    [1][mb]);
            }
        }
        /*level 2 nesting*/
        else {
            /*level 3 nesting*/
            for (mb=0; mb<M_BLOCK; mb
                ++) {
                cm[mb]=ch[mb]*cc[1][mb
                    ];
            }
        }
        /*level 2 nesting*/
        for (is=is1-1; is<is2; is++)
            {
            /*level 3 nesting*/
            if(is==0) {
                /*level 4 nesting*/
                for (mb=0; mb<M_BLOCK;
                    mb++) {
                    ct[mb]=cm[mb]*(1.0-
                        cc[2][mb]);
                }
            }
            /*level 3 nesting*/
            else {
                /*level 4 nesting*/
                for (mb=0; mb<M_BLOCK;
                    mb++) {
                    ct[mb]=cm[mb]*cc[2][
                        mb];
                }
            }
            /*.....*/
            /*level 3 nesting*/
            for (l=0; l<LM+1; l++) {
                /*level 4 nesting*/
                for (mb=0; mb<M_BLOCK;
                    mb++) {
                    /*level 5 nesting*/
                    if(ih==0 && im==0 &&
                        is==0) {
                        fclr[l][mb]=flxdn
                            [l+1][mb];
                    }
                    fall[l][mb]=fall[l][
                        mb]+flxdn[l+1][mb
                        ]*ct[mb];
                }
            }
            /*.....*/
        }
    }
}
```