Accelerating a climate physics model with OpenCL

Fahad Zafar, Dibyajyoti Ghosh, Lawrence Sebald, Shujia Zhou
Introduction

• The demand to increase forecast predictability has been pushing climate and weather models
  ◦ increase model grid resolution
  ◦ include more physical processes.

• 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.
OpenCL

- Open Computing Language (OpenCL) is fast becoming the standard for heterogeneous parallel computing
- Run on CPUs, GPUs, and other accelerator architectures (Cell, Fusion)
- OpenCL puts forward a thread-extensive model for programming
Contributions and Focus

- A complete cross-platform real-world example in OpenCL
  - Evaluate cross-platform performance and portability
- Compare C compilers and execution environments (IBM, Mac OS X)
- Highlight the performance gain achieved by OpenCL CPU implementation over traditional C code on CPUs.
GEOS-5 Climate Model

- The NASA Goddard Earth Observing System Model, Version 5 (GEOS-5), is a currently operational climate model.

- SOLAR is a solar radiation model component [Chou et al. 99] used in GEOS-5 and other climate and weather models.

A production-quality climate or weather model code can span up to a few hundred thousand lines.

- Originally written in FORTRAN

This particular code was converted to C and ported to the IBM Cell Broadband Engine by [Zhou et al] where detailed code structure analysis and performance gains were reported.

- ~20% for SOLAR AND IRRAD, the remaining computing time breaks down as
  - ~25% for dynamics
  - ~25% for input and output data
  - ~30% for other column-physics components.

We implemented the serial version of the C code in OpenCL version 1.0

- Platforms: IBM JS21 (PowerPC) and JS22 (Power6) blades, a POWER6 AIX system, and Mac OS X versions 10.6.4 and 10.6.7 with x86 Intel processors.

Code Overview

- **Solar Radiation Initial ()**
  - (Setting up data and arrays)

- **Solar UV ()**
  - GetAeroIndex
  - Cldscale
  - Deledd
  - Cldfixv
  - Cldfix

- **Solar IR ()**
  - GetAeroIndex
  - Cldscale
  - Deledd
  - Cldfixv
  - Cldfix

- **Solar Radiation Final ()**
  - (Finalizing data and output)
Approach

- Extract compute-intensive kernels without changing the overall code structure
- Manually optimized sections of the code to run in a multi-threaded fashion using OpenCL kernels and benchmark them.
- Run in 2 Modes
  - Cross-checking with serial execution
  - Benchmark
Approach

- Compare OpenCL vs. serial GCC
  - IBM (complete)
  - Mac OS X (incomplete)
- Compare auto-vectorization with other compilers for select sections of the code
## CPU Experiment Setup

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM JS 21 blade</td>
<td>• GCC v4.1.2</td>
</tr>
<tr>
<td>IBM JS 22 blade</td>
<td>• GCC v4.1.2</td>
</tr>
<tr>
<td>IBM POWER6 AIX</td>
<td>• GCC v4.1.2</td>
</tr>
<tr>
<td></td>
<td>• IBM XLC v10.1</td>
</tr>
<tr>
<td>Intel 2.66 GHz Core 2 Duo</td>
<td>• GCC v4.2.1</td>
</tr>
<tr>
<td>Mac OS X 10.6.7</td>
<td>• Intel C++ Compiler v12.0.4</td>
</tr>
</tbody>
</table>
Code Execution and Checking

- Parallel code runs side by side
  - Cross-check Compute Device values after every kernel

Code Sample:

```c
execute_section1(df, cnt, so2);

//
// SECTION - J
//
for (k=0; k<LM; k++)
{
    for (i=0; i<N_BLOCK; i++)
    {
        so2[i][k+1] = so2[i][k] + scal[i][k+1]*cnt[i];
        so2[k+1][i] = so2[k][i] + scal[k+1][i] *cnt[i];
        df[i][k+2] = 0.053*1.0 - exp(-0.00016*sqrt(so2[i][k+1]));
        df[k+2][i] = 0.065*1.0 - exp(-0.00155*sqrt(so2[k][1][1]));
    }

execute_section1(so2, df);

//
// for solar heating due to co2 scaling follows Eq(3.5) with f=1.

// SECTION - K
//
for (i=0; i<N_BLOCK; i++)
{
    so2[i][0] = (789 * exp2)*scal[i][0];
    so2[0][i] = (789 * exp2)*scal[0][i];
}

execute_sectionK(so2, scal);
```
Manual Optimization

- 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 speed up the processing, while in some cases it reduced performance.
Findings

- Auto vectorization support in OpenCL compiler
  - better than GCC, IBM XLC, Intel ICC

- We found a 3 ~ 4X performance improvement per core over the original serial code compiled with GCC

- OpenCL provides access to a multi-threaded programming and execution model as well as a low-level API for memory and thread management
Findings

- Similar results were obtained from Intel ICC compiler and IBM XLC compiler for these nested loop constructs.

- Efficient vectorization and global optimizations contribute to drastic speedup in OpenCL.
OpenCL Parallel vs. IBM Serial

- Code implementation complete
- Tested and cross checked
- About 70 compute kernels
- The kernels do not have a one-to-one mapping with the serial solar radiation code functions.
- The code uses integer and floating point data types
Results: Speedup per core

- IBM JS21 (4 cores)
- IBM JS22 (8 cores)
Results: Performance gain per section

![Bar chart showing performance gain per section.](chart)

- **Solar Radiation Initial()**
- **SolarUV ()**
- **SolarIR()**
- **Solar Radiation final()**

Legend:
- **IBM JS21 (Parallel)**
- **IBM JS21 (Serial)**
- **IBM JS22 (Parallel)**
- **IBM JS22 (Serial)**
OpenCL Parallel vs. Mac OS X Serial

- Code Implementation Incomplete
  - First two sections tested and running
- Performance is portable to some extent
Portability Issues

- The code implemented with OpenCL 1.0 compiled correctly and executed accurately with cross-checked values in IBM JS21 and JS22 blades did not run on Mac OS X “as is”.
- Thread scheduling identified as an issue.
- Platform detection functions would crash for one platform while ran correctly for the other.
Vectorization test

- **IBM**
  - Implement a subset of the code using Altivec Instruction set.
  - Speedup = 2x

- **Intel**
  - Use Intel OpenCL viewer to look at the assembly code
A part of the serial code with GCC vectorization error output

```c
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+0.00135*(ta[k][i]-240.)) + 1.e-9;
        wh[k+1][i] = wh[k][i]+wh[k][i];
    }
}
```
Vectorization Analysis - II

A part of the OpenCL code with vectorized instruction set for the loop-construct in the last slide.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>imul EBX, EAX</td>
<td>4-byte Folded Reload</td>
</tr>
<tr>
<td>add EBX, DWORD PTR [ESP + 28]</td>
<td>4-byte Folded Reload</td>
</tr>
<tr>
<td>add EDI, DWORD PTR [ESP + 28]</td>
<td>4-byte Folded Reload</td>
</tr>
<tr>
<td>mov DWORD PTR [ESP + 16], EDI</td>
<td>4-byte Spill</td>
</tr>
<tr>
<td>mov ecx, DWORD PTR [ESP + 56]</td>
<td></td>
</tr>
<tr>
<td>movs XMM0, DWORD PTR [EBX + 4*EDI]</td>
<td></td>
</tr>
<tr>
<td>subb XMM0, DWORD PTR [EBX + 4*EDI]</td>
<td></td>
</tr>
<tr>
<td>mov EDX, DWORD PTR [EBX + 116]</td>
<td>4-byte Folded Reload</td>
</tr>
<tr>
<td>mov EDX, DWORD PTR [ESP + 112]</td>
<td></td>
</tr>
<tr>
<td>mov EDX, DWORD PTR [EDX]</td>
<td></td>
</tr>
<tr>
<td>mov EAX, DWORD PTR [ESP + 32]</td>
<td></td>
</tr>
<tr>
<td>movs DWORD PTR [EDI + 4*EBX], XMM0</td>
<td></td>
</tr>
<tr>
<td>mov EAX, DWORD PTR [ESP + 16]</td>
<td>4-byte Reload</td>
</tr>
<tr>
<td>addb XMM0, DWORD PTR [ECX + 4*EDI]</td>
<td></td>
</tr>
<tr>
<td>mov ecx, DWORD PTR [ESP + 60]</td>
<td></td>
</tr>
<tr>
<td>movs DWORD PTR [ECX + 4*EDI], XMM0</td>
<td></td>
</tr>
<tr>
<td>divs XMM0, DWORD PTR [LDTT3 + LDTT3 + 1]</td>
<td></td>
</tr>
<tr>
<td>movs XMM0, DWORD PTR [LDTT3 + LDTT3 + LDTT3 + 1]</td>
<td></td>
</tr>
<tr>
<td>call _kernel_eval_u0`povf2</td>
<td></td>
</tr>
<tr>
<td>movs XMM0, DWORD PTR [XMM]</td>
<td>4-byte Spill</td>
</tr>
</tbody>
</table>

pshufd, paddd, movaps, movups are special SIMD instructions belonging to Intel Advanced Vector Extensions.
Why is OpenCL Faster on the CPU?

- 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.
Why is OpenCL Faster on the CPU?

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

- Light weight OpenCL threads
  - Better memory management?

- It should be noted that the OpenCL compiler might make certain assumptions that GCC cannot afford to make for naive C code.
  - The OpenCL compiler can assume that the given computation is meant to be run as a many-threaded piece of code
Benchmarking Other Compilers

Select two sections of SOLAR with complex nested loop constructs

Compile and run serial sections on GCC, IBM XLC, Intel C++ compiler

Compared timing data
Timing compilers on different platforms

![Bar chart showing timing comparisons between different platforms for Section 1 and Section 2. The x-axis represents different platforms such as power6-gcc, power6-ix, SP2-gcc, SP22-gcc, Mac OS-gcc, and Mac OSX-gcc. The y-axis represents time in microseconds. Each platform has two bars indicating the time for Section 1 and Section 2.](chart.png)
Conclusion

- Multithreaded programming and execution models of OpenCL can significantly increase the performance
  - IBM POWER and PowerPC and POWER6 CPU architectures
  - Similar performance improvement has also been obtained in Intel CPUs

- 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.

- Across compiler infrastructure Intel ICC on Mac OS X 10.6.7 fares best followed by IBM XLC on POWER6 AIX
Future Work

- We plan to modify the OpenCL code appropriately to run on GPU.

- Apply OpenMP optimization to serial code and compare to OpenCL version.

- Identify programming practices that work best/worst on GPU and CPU platforms.
Questions ?

Thank You