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
Motivation

• Explore OpenCL in accelerating a real world computationally intensively application. (NASA climate and weather physics model)

• Investigate both the performance and code portability of OpenCL with GPUs and CPUs.

• Extend the work of Zafar et al [1] by:
  – Producing a baseline OpenCL code that compiles and runs on both CPUs and GPUs.
  – Maintain the accuracy of serial code.
Outline

• Solar Radiation Model
• Experimental Setup
• Porting and Optimizations
• Results
• Explicit AVX Registers
• Conclusion
SOLAR RADIATION MODEL
The components of solar and infrared radiation together can take at least 20% computing time of the atmosphere model of NASA GEOS 5.
NASA GEOS-5

• Solar radiation component of NASA’s GEOS-5 takes ~10% of model computation time.

• NASA is interested in analysis of performance and cost benefit using non traditional computing systems.

• GEOS-5 - 20+ years old, written in Fortran (mostly), still evolving.

• Cannot be entirely rewritten due to production constraints.
Processes in a Climate Model

http://www.ucar.edu/communications/CCSM/overview.html
Code Structure of SOLAR

**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)
### Experimental Setup

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

**TABLE I**

*Characteristics of CPU’s and GPU’s used in performance testing.*
PORTING AND OPTIMIZATIONS
OpenCL Compilation Model

- OpenCL uses Dynamic/Runtime compilation model [2]
  1. Code is first compiled to an Intermediate Representation (IR)
     - Done once and IR is stored
  2. IR is compiled to machine code for execution
     - Application loads IR and performs compilation during run time

- Preprocessor macros were used for constant variables that dictated kernel loop iterations.

- Preprocessor macros enable OpenCL dynamic compilation to ensure that the variable is known at kernel compile time allowing compilers to perform implicit loop unrolling.
for (ih=0; ih<2; ih++)
    for (mb=0; mb<M_BLOCK; mb++)
        //calculate tda[0][ih][0][mb], tta[...], rsa[...]
        //calcualte 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[1][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]
```c
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;
rra[is * 2 * (NLM+2) * NM_BLOCK + im * (NLM+2) * NM_BLOCK + (NLM+1) * NM_BLOCK + k] = temp;
dev_rra0 = temp;
dev_rx0 = 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_rx0);
    dev_rra1 = temp_rr + (temp_td * dev_rra0 + (temp_tt - temp_td) * dev_rx0) * denm;
    dev_rx0 = temp_rs + temp_ts * dev_rx0 * denm;
    dev_rra1 = dev_rra1;
    dev_rx0 = dev_rx0;
    rra[stride5D+l*NM_BLOCK+k] = dev_rra1;
    rxa[stride5D+l*NM_BLOCK+k] = dev_rx0;
    dev_rra0 = dev_rra1;
    dev_rx0 = dev_rx0;
```
for(\(l = 0; l \leq NLM + 1; l++\))

\[
\begin{align*}
\text{temp\_dev\_rx} & = \text{rx}[\text{stride5D} + l \times \text{NM\_BLOCK} + k]; \\
\text{temp\_dev\_rr} & = \text{rr}[\text{stride5D} + l \times \text{NM\_BLOCK} + k]; \\
\text{denm} & = 1.0 / (1.0 - \text{dev\_rsa0} \times \text{temp\_dev\_rx}); \\
\text{xx} & = \text{dev\_tda0} \times \text{temp\_dev\_rr}; \\
\text{yy} & = \text{dev\_tda0} - \text{dev\_tda0}; \\
\text{temp\_fnddif} & = (\text{xx} \times \text{dev\_rsa0} + \text{yy}) \times \text{denm}; \\
\text{fudpif} & = (\text{xx} + \text{yy} \times \text{temp\_dev\_rx}) \times \text{denm}; \\
\text{flxdn}[l \times \text{NM\_BLOCK} + k] & = \text{dev\_tda0} + \text{temp\_fnddif} - \text{fudpif}; \\
\end{align*}
\]

if(\(l == (NLM+1)\))

\[
\begin{align*}
\text{fnddir}[k] & = \text{dev\_tda0}; \\
\text{fnddif}[k] & = \text{temp\_fnddif}; \\
\end{align*}
\]

if(\(l < (NLM+1)\))

\[
\begin{align*}
\text{temp\_rr} & = \text{rr}[\text{stride3D} + l \times \text{NM\_BLOCK} + k]; \\
\text{temp\_rs} & = \text{rs}[\text{stride3D} + l \times \text{NM\_BLOCK} + k]; \\
\text{temp\_td} & = \text{td}[\text{stride3D} + l \times \text{NM\_BLOCK} + k]; \\
\text{temp\_ts} & = \text{ts}[\text{stride3D} + l \times \text{NM\_BLOCK} + k]; \\
\text{temp\_tt} & = \text{tt}[\text{stride3D} + l \times \text{NM\_BLOCK} + k]; \\
\text{denm} & = \text{temp\_ts} / (1.0 - \text{dev\_rsa0} \times \text{temp\_rs}); \\
\text{dev\_tdal} & = \text{dev\_tda0} \times \text{temp\_td}; \\
\text{dev\_ttal} & = \text{dev\_tda0} \times \text{temp\_tt} + (\text{dev\_tda0} \times \text{dev\_rsa0} \times \text{temp\_rr} + \text{dev\_tda0} - \text{dev\_tda0}) \times \text{denm}; \\
\text{dev\_rsal} & = \text{temp\_rs} + \text{temp\_ts} \times \text{dev\_rsa0} \times \text{denm}; \\
\text{dev\_tda0} & = \text{dev\_tdal}; \\
\text{dev\_tda0} & = \text{dev\_ttal}; \\
\text{dev\_tda0} & = \text{dev\_rsal}; \\
\end{align*}
\]
CLDFLX Parallel

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];

c1[l * NM_BLOCK + mb] = flxdn[(l+1) * (NM_BLOCK+1) + mb];

call[1 * NM_BLOCK + mb] = call[1 * NM_BLOCK + mb] + flxdn[(l+1) * (NM_BLOCK+1) + mb] * ct_data[(ih * 4 + im * 2 + is * 1) * 128 + mb];
RESULTS
Accomplishments

- A single parallel OpenCL code runnable across multiple platforms consisting of IBM Cell Processors, multicore CPUs and GPUs.

- Achieved parallel implementation accuracy of $1.0 \times 10^{-6}$ in numerical differences when compared to serial implementation (increased from $1.0 \times 10^{-4}$ of Fahad et al [1]).

- Discovered OpenCL can enable CPU devices to achieve dramatic performance improvements.
## Performance Results

<table>
<thead>
<tr>
<th>Platform</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Per Thread</td>
<td>Total</td>
<td>Per Thread</td>
<td>Total</td>
</tr>
<tr>
<td>Intel Core i7-2630QM</td>
<td>5.5</td>
<td>55</td>
<td>11.25</td>
<td>90</td>
</tr>
<tr>
<td>Dual Intel Xeon X5670</td>
<td>3.3</td>
<td>80</td>
<td>6.8</td>
<td>163.7</td>
</tr>
<tr>
<td>GeForce GTX 580M</td>
<td>-</td>
<td>21</td>
<td>-</td>
<td>38</td>
</tr>
<tr>
<td>IBM JS22 Power6</td>
<td>1.3</td>
<td>10.2</td>
<td>0.9</td>
<td>7.16</td>
</tr>
<tr>
<td>Intel Core 2 Duo</td>
<td>1.01</td>
<td>2.02</td>
<td>1.05</td>
<td>2.1</td>
</tr>
<tr>
<td>GeForce GT 320M</td>
<td>-</td>
<td>10.02</td>
<td>-</td>
<td>5.329</td>
</tr>
</tbody>
</table>

**TABLE II**

**Speedup across all platforms**
Assembly Dump

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.
Intel Streaming SIMD Extensions

• Designed by Intel and introduced in 1999.

• Increases performance when the same operation are performed on multiple data objects.

• Registers:
  – SSE
  – SSE2
  – SSE3
  – SSE4
  – AVX
How does it work?

- Intel SSE packs multiple data into fixed size registers and applies same instructions to all data in parallel.
How does OpenCL contribute?

- OpenCL coding style is **SIMD** based as it is intended to run on GPUs.

- Optimizations that are important for GPUs such as reducing thread divergence and improving coalesced memory accesses helps CPU compilers.

- **SIMD** style of kernel programming eliminates complex loop constructs. This helps compilers by providing more effective vectorization as it usually behaves in a conservative manner for vectorization [3][4].

- Data dependence and cycles are broken through the optimization of kernels originally intended to execute on GPUs to fully exploit the SIMD feature of CPU vector processors.
GPU Results

• Reduced the original 70 kernels from Zafar et al [1] to about half (36 kernels).

• Exploring local memory was severely limited due to the simplified kernels.

• Development Time vs Performance
Explicit AVX Registers

- **Difficulties:**
  - Affect the performance portability due to targeting a specific vector width
  - Vector data types cannot be used in conditional statements
  - Utilized built-in relational functions such as *isgreater* or *isless* and called stub functions for each side of the conditional
  - Pad arrays to be divisible by 8
Execution time comparisons of serial code compiled with GCC, serial code compiled with Intel ICC (12.1.4) on Intel i7-2630QM CPU, and parallel OpenCL implementations.
Performance Results

Execution time comparison between OpenCL code and OpenCL code using explicit AVX intrinsic on Intel Core i7-2630QM CPU on 128 column size.
Conclusion

• Developed an OpenCL code for a representative climate and weather physics model that is able to run across multiple platforms.

• OpenCL’s kernel programming and execution model facilitates the compiler to vectorize the code and consequently improve performance.
References


