PERFORMANCE TUNING TECHNIQUES
FOR GPU AND MIC

ASAP/TDD/CISL/NCAR
Youngsung Kim
CONTENTS

- Introduction
  - Kernel-based approach
  - Micro-architectures
- MIC performance evolutions
- CUDA-C performance evolutions
- CPU performance evolutions along with MIC evolutions
- GPU programming: Open ACC, CUDA Fortran, and F2C-ACC
- One source consideration
- Summary
What is a kernel?
- A small computation-intensive part of existing large code
- Represent characteristics of computations

Benefit of kernel-based approach
- Easy to manipulate and understand
  - CESM: >1.5M LOC
- Easy to convert to many different technologies of GPU
  - CUDA-C, CUDA-Fortran, OpenACC, and F2C-ACC
- Easy to isolate issues for analysis
  - Simplify hardware counter analysis
DG KERNEL

- **Origin**
  - a kernel derived from the computational part of the gradient calculation in the Discontinuous Galerkin formulation of the shallow water equations from HOMME.

- **Implementation from HOMME**
  - Similar to “dg3d_gradient_mass” function in “dg3d_core_mod.F90”
  - Calculate gradient of flux vectors and update the flux vectors using the calculated gradient
Floating point operations
- No dependancy between elements
- # of elements
  - Can be calculated from source code analytically
  - Ex.) When nit=1000, nelem=1024, nx=4(npts=nx*nx) ≈ 2 GFLOP

OpenMP
- Two OpenMP Parallel regions for Do loops on element index(ie)

Source code
```fortran
DO ie=1,nelem
  DO ii=1,npts
    k=MODULO(ii-1,nx)+1
    l=(ii-1)/nx+1
    s2 = 0.0_8
    DO j = 1, nx
      s1 = 0.0_8
      DO i = 1, nx
        s1 = s1 + (delta(l,j)*flx(i+(j-1)*nx,ie) * &
                    der(i,k) + delta(i,k) * &
                    fly(i+(j-1)*nx,ie) * der(j,l)) * gw(i)
      END DO  ! i loop
      s2 = s2 + s1*gw(j)
    END DO ! j loop
    grad(ii,ie) = s2
  END DO ! ii loop
  END DO ! ie loop

DO ie=1,nelem
  DO ii=1,npts
    flx(ii,ie) = flx(ii,ie)+ dt*grad(ii,ie)
    fly(ii,ie) = fly(ii,ie)+ dt*grad(ii,ie)
  END DO ! ii loop
  END DO ! ie loop
```
CPU
- Conventional multi-core: 1 ~ 16+ cores/~256-bit vector registers
- Many programming language: Fortran, C/C++, etc.
- Ex) Intel SandyBridge E5-2670
  - Peak performance(2 Sockets): 332.8 DP GFLOPS (Estimated by presenter)

GPU
- Many light-weight threads: ~2680+ threads (threading & vectorization)
- Limited programming language(extensions): CUDA-C, CUDA-Fortran, OpenCL, OpenACC, F2C-ACC, etc.
- Ex.) Nvidia K20x
  - Peak performance: 1.308 DP TFLOPS

MIC
- Many core and wider vector: 60+ cores/512-bit vector registers
- Limited programming language(extensions): C/C++, Fortran
- Ex.) Intel KNC (a.k.a. MIC)
  - Peak Performance(7120): 1.208 DP TFLOPS
UNDERSTANDING PLOT

- X-axis: # of element
  - $nx \times nx \times 8$ bytes
  - 128 bytes when $nx=4$

- Y-axis: Analytic DP FLOPS
  - # of DP FLOP / Second
    - Fixed value
    - Changeable
  - Enable comparisons across heterogeneous architectures and various technologies
THE BEST PERFORMANCE RESULTS FROM CPU, GPU, AND MIC

DG_KERNEL - Performance Results

- CUDA-C (K20x)
- MIC Fortran (7120, Native)
- CUDA-C (M2070Q)
- SandyBridge Fortran (2S E5-2670)
- SandyBridge Fortran (1S E5-2670)

Performance (Analytic DP FLOPS) vs. # of ELEMENTS

6.6x
5.4x
Compiler options
- -mmic

Environmental variables
- OMP_NUM_THREADS=240
- KMP_AFFINITY = 'granularity=fine,compact'

Native mode only
- No cost of memory copy between CPU and MIC

Supports from Intel
- R. Sasanka
MIC VER. 1

- **Source modification**
  - NONE

- **Compiler options**
  - -mmic -openmp -O3
Source code

```
i = 1
s1 = (delta(l,j)*flx(i+(j-1)*nx,ie)*der(i,k) + delta(i,k)*fly(i+(j-1)*nx,ie)*der(j,l))*gw(i)
i = i + 1
s1 = s1 + (delta(l,j)*flx(i+(j-1)*nx,ie)*der(i,k) + delta(i,k)*fly(i+(j-1)*nx,ie)*der(j,l))*gw(i)
i = i + 1
s1 = s1 + (delta(l,j)*flx(i+(j-1)*nx,ie)*der(i,k) + delta(i,k)*fly(i+(j-1)*nx,ie)*der(j,l))*gw(i)
i = i + 1
s1 = s1 + (delta(l,j)*flx(i+(j-1)*nx,ie)*der(i,k) + delta(i,k)*fly(i+(j-1)*nx,ie)*der(j,l))*gw(i)
```

Compiler options

- -mmic -openmp -O3

Performance Considerations

- Complete unroll of three nested loops
## Compiler report

... 

dg_kernel_v2_unroll.F90(38): (col. 13)  
remark: LOOP WAS VECTORIZED.  
...  
dg_kernel_v2_unroll.F90(40): (col. 22)  
remark: vectorization support: gather was generated for the variable (unknown): strided by 4.  
...  

### Instructions counts

<table>
<thead>
<tr>
<th></th>
<th>Ver. 1</th>
<th>Ver. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>vgather</td>
<td>20</td>
<td>44</td>
</tr>
<tr>
<td>vmov</td>
<td>43</td>
<td>35</td>
</tr>
<tr>
<td>vmovapd</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>vadd</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>vaddpd</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>vfmadd</td>
<td>10</td>
<td>20</td>
</tr>
</tbody>
</table>

## Hardware counters

![Comparisons of hardware counter values](image)

**INSTRUCTIONS_EXECUTED**
- dg_kernel_v1_nochange
- dg_kernel_v2_unroll

**CPU_CLK_UNHALTED**
- dg_kernel_v1_nochange
- dg_kernel_v2_unroll

**MEMORY_ACCESSSES_IN_BOTH_PIPES**
- dg_kernel_v1_nochange
- dg_kernel_v2_unroll

**VPUSTALL_REG**
- dg_kernel_v1_nochange
- dg_kernel_v2_unroll
Source code

```c
!$OMP PARALLEL DO...
DO ie=1,nelem
  DO ii=1,npts
    ...
  END DO !ii
END DO
DO ii=1,npts
  ...
END DO
END DO
```

Compiler options

- -mmic -openmp -O3 -**ALIGN ARRAY64BYTE -OPT-PREFETCH=0**

Performance Considerations

- Merged two openMP regions into one with “no wait”
- Helps to remove openMP sync. overheads
Compiler report

... 
dg_kernel_v3_vecmerge.F90(37): (col. 13) remark: LOOP WAS VECTORIZED.

... 
dg_kernel_v3_vecmerge.F90(46): (col. 32) remark: vectorization support: gather was generated for the variable (unknown): strided by 4.

... 

Instructions counts

<table>
<thead>
<tr>
<th></th>
<th>Ver. 2</th>
<th>Ver. 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>vgather</td>
<td>44</td>
<td>44</td>
</tr>
<tr>
<td>vmov</td>
<td>35</td>
<td>47</td>
</tr>
<tr>
<td>vmovapd</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>vadd</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>vaddpd</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>vfmadd</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

Hardware counters

Comparisons of hardware counter values

<table>
<thead>
<tr>
<th></th>
<th>dg_kernel_v2_unroll</th>
<th>dg_kernel_v3_vecmerge</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTRUCTIONS_EXECUTED</td>
<td>[Bars]</td>
<td>[Bars]</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED</td>
<td>[Bars]</td>
<td>[Bars]</td>
</tr>
<tr>
<td>MEMORY_ACCESSSES_IN_BOTH_PIPES</td>
<td>[Bars]</td>
<td>[Bars]</td>
</tr>
<tr>
<td>VPUSTALL_REG</td>
<td>[Bars]</td>
<td>[Bars]</td>
</tr>
</tbody>
</table>
Source code

```fortran
DO J=1,NX
  DO I=1, NX
    ji = (j-1)*NX
    !DEC$ vector always aligned
    DO II=1, NX*NX
      grad2(ii,j) = grad2(ii,j) + ( delta2(ii,j)* &
         flx(ji+i,ie)*der2(ii,i) + delta3(ii,i) * &
         fly(ji+i,ie)*der3(ii,j) ) * gw2(i)
    END DO ! ii-loop
  END DO ! i-loop
  !DEC$ vector always aligned
  DO ii=1, NX*NX
    grad(ii) = grad(ii) + grad2(ii,j) * gw2(j)
  ENDDO
  END DO ! J
```

Compiler options

- -mmic -openmp -O2 -align array64byte - opt-prefetch=0  **OPT-ASSUME-SAFE-PADDING**

Performance Considerations

- All of important loops are vectorized
- Arrays are aligned at creation as well as at being referenced

![DG_KERNEL - Performance Results (MIC 7120)](image)
ANALYSIS VER. 3 AND VER. 4

Compiler report

... 
dg_kernel_v4_nogatherscatter.F90(82): (col. 17) remark: LOOP WAS VECTORIZED.

... 
dg_kernel_v4_nogatherscatter.F90(83): (col. 21) remark: vectorization support: reference grad2 has aligned access.

...

Instructions counts

<table>
<thead>
<tr>
<th></th>
<th>Ver. 3</th>
<th>Ver. 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>vgather</td>
<td>44</td>
<td>16</td>
</tr>
<tr>
<td>vmov</td>
<td>47</td>
<td>91</td>
</tr>
<tr>
<td>vmovapd</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>vadd</td>
<td>37</td>
<td>3</td>
</tr>
<tr>
<td>vaddpd</td>
<td>33</td>
<td>3</td>
</tr>
<tr>
<td>vfmadd</td>
<td>20</td>
<td>26</td>
</tr>
</tbody>
</table>

Hardware counters

Comparisons of hardware counter values

- dg_kernel_v3_vecmerge
- dg_kernel_v4_nogatherscatter
Source code modification

DO j=1, SET_NX
DO i=1, SET_NX
  j = (j-1)*SET_NX + i
  !DEC$ vector always aligned
  DO ii=1, SET_NX*SET_NX
    grad2(ii,j) = grad2(ii,j) + 
    ( DELTA_DER2(ii,ji)*flx(ji,ie) + 
    DELTA_DER3(ii,ji)*fly(ji,ie) ) * gw(i)
  END DO
END DO
END DO
...
END DO

Compiler options
- mmic -openmp -O3 -align array64byte -opt-prefetch=0 -opt-assume-safe-padding -MP2OPT_HLO_FUSION=F

Performance Considerations
- Effectively reduce two FLOPs to one FLOP by merging two arrays into one and pre-calculates it
## Compiler report

... 
dg_kernel_v5_arrmerge.F90(70): (col. 19) 
remark: LOOP WAS VECTORIZED.
...

dg_kernel_v5_arrmerge.F90(72): (col. 8) 
remark: vectorization support: reference grad2 has aligned access.
...

### Instructions counts

<table>
<thead>
<tr>
<th></th>
<th>Ver. 4</th>
<th>Ver. 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>vgather</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>vmov</td>
<td>91</td>
<td>56</td>
</tr>
<tr>
<td>vmovapd</td>
<td>64</td>
<td>44</td>
</tr>
<tr>
<td>vadd</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>vaddpd</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>vfmadd</td>
<td>26</td>
<td>26</td>
</tr>
</tbody>
</table>

## Hardware counters

![Comparisons of hardware counter values](chart.png)
CUDA-C EVOLUTIONS

- Compiler options
  - `-O3 -arch=sm_35`
  - same to all versions
- “Offload mode” only
  - However, the time cost for data copy between CPU and GPU is not included for comparison to MIC native mode

![Performance Results Graph](chart.png)

1. First Conversion
2. Increase Occupancy
3. Big arrays on Shared Mem.
4. Constant arrays on Shared Mem.
Source code

<Host>
...
dim3 gridSize(NELEM), blockSize(NX*NX);
kernel<<<gridSize, blockSize>>>(D_flx, D_fly, D_der, ...);
...

<Kernel>
...
i_e = blockIdx.x*ColsPerBlock + threadIdx.x/(nX*nX);
i_i = threadIdx.x%(nX*nX); k = i_i%nX; l = i_i/nX;
...
for (j=0;j<nX;j++) {
    s1 = 0.0;
    for (i=0;i<nX;i++) {
        s1 = s1 + (D_delta[j*nX+l] * D_flx[i*nX*nX+i+j*nX] * 
                   D_der[k*nX+i] + D_delta[k*nX+i]*D_fly[i*e*nX*nX+i+j*nX] * 
                   D_der[l*nX+j]) * D_gw[i];
    }
    s2 = s2 + s1 * D_gw[j];
}
D_flx[i*e*nX*nX+i+j] = D_flx[i*e*nX*nX+i+j] + dt * s2;
D_fly[i*e*nX*nX+i+j] = D_fly[i*e*nX*nX+i+j] + dt * s2;
...

Performance considerations

- Converting from Fortran to CUDA-C requires considerable amount of effort to make it work correctly
- Conversion to CUDA-C force programmer to think about thread parallelization and vectorization from the beginning
Source code

<Host>
    dim3 gridSize(NELEM/8);
dim3 blockSize(NX*NX*8)

<Kernel>
    No change

Performance considerations

- Enhance theoretic occupancy by using CUDA GPU Occupancy Calculator
- 25% when blockSize=NX*NX
- 100% when blockSize=NX*NX*8
Source code

<Host>
No change

<Kernel>

double D_flx_s[nX*nX*nSizeSubBlock];

double D_fly_s[nX*nX*nSizeSubBlock];

D_flx_s[ik*nX*nX+ii] = D_flx[ie*nX*nX+ii];
D_fly_s[ik*nX*nX+ii] = D_fly[ie*nX*nX+ii];

// calculations using data pre-fetched to shared mem.

D_flx[ie*nX*nX+ii] = D_flx_s[ik*nX*nX+ii] + dt * s2;
D_fly[ie*nX*nX+ii] = D_fly_s[ik*nX*nX+ii] + dt * s2;

Performance considerations

- Pre-fetch data from DRAM to Shared Memory
- Re-use the pre-fetched data nX*nX times
**Source code**

<Host>
No change

<Kernel>
```c
__shared__ double D_der_s[nX*nX], D_delta_s[nX*nX];
__shared__ double D_gw_s[nX];
...
for (i=0;i<nX;i++) {
    s1 = s1 + (D_delta_s[j*nX+l] * D_flx_s[i*k*nX*nX+i+j*nX] * D_der_s[k*nX+i] + D_delta_s[k*nX+i] * D_fly_s[i*k*nX*nX+i+j*nX] * D_der_s[l*nX+j]) * D_gw_s[i];
}
```

**Performance considerations**

- Re-use data of static arrays that are loaded to Shared memory
Generally, performance tuning on a micro-architecture also helps to improve performance on another micro-architecture. However, it is not always true.
Source Code

\[
ie = (\text{blockidx} \times \text{x}) \times \text{NDIV} + (\text{threadidx} \times \text{x})/(\text{SET}_N \times \text{SET}_N) + 1 \\
\text{ii} = \text{MODULO}(\text{threadidx} \times \text{x} - 1, \text{SET}_N \times \text{SET}_N) + 1 \\
\text{IF (ie} > \text{SET}_N \text{ELEM) RETURN} \\
\text{k} = \text{MODULO}(\text{ii}, \text{SET}_N) + 1 \\
\text{l} = (\text{ii} - 1)/\text{SET}_N + 1 \\
\text{s2} = 0.0_8 \\
\text{DO j=1, SET}_N \\
\text{s1} = 0.0_8 \\
\text{DO i = 1, nx} \\
\text{s1} = s1 + (\text{delta}(l,j) \times \text{flx}(i+(j-1) \times \text{nx}, \text{ie}) \times \text{der}(i,k) + \& \\
\text{delta}(i,k) \times \text{fly}(i+(j-1) \times \text{nx}, \text{ie}) \times \text{der}(j,l)) \times \text{gw}(i) \\
\text{END DO ! i loop} \\
\text{s2} = s2 + s1 \times \text{gw}(j) \\
\text{END DO ! j} \\
\text{grad}(ii, \text{ie}) = s2 \\
\text{flx}(ii, \text{ie}) = \text{flx}(ii, \text{ie}) + \text{dt} \times \text{grad}(ii, \text{ie}) \\
\text{fly}(ii, \text{ie}) = \text{fly}(ii, \text{ie}) + \text{dt} \times \text{grad}(ii, \text{ie})
\]

Performance considerations

- Maintains source structure of original Fortran
- Needs understanding on CUDA threading model, especially for debugging and performance tuning
- Supports implicit memory copy, which is convenient but could negatively impact to performance if over-used.
- **Source Code**
  ```fortran
  DATA PRESENT_OR_COPY(flx,fly)
  PRESENT_OR_CREATE(grad)
  PRESENT_OR_COPYIN(gw,der,delta)
  KERNELES
  LOOP GANG(ngangs) VECTOR(neblk)
  DO ie=1, SET_NELEM
  ...
  END DO ! Ie
  END KERNELES
  END DATA
  ```

- **Performance considerations**
  - Trial and error with ngangs and neblk
  - Compiler report feature helps to understand how it affects to GPU resource allocations
  - Was not successful to use cache directive
Source Code

```fortran
!ACC$REGION
<npts:block=16>,<nelem>,<der,delta,gw:in>,<flx,fly:inout>,<grad:none>) BEGIN
!ACC$DO PARALLEL(1)
  DO i=1, nelem
!ACC$DO VECTOR(1)
  DO ii=1, nx*nx
  ...
  END DO !ii
  END DO ! i
END !ACC$REGION END
```

Performance considerations

- Similar to Open ACC in terms of programming
- Generates CUDA C source file that is readable
- Performance is between OpenACC and CUDA Fortran
- Lack of support for some language features including derived types
One source is highly desirable
- Hard to manage versions from multiple micro-architectures and multiple programming technologies
- Performance enhancement can be applied to multiple versions simultaneously

Conditional Compilation
- Macro to insert & delete code for a particular technology
- User control compilation by using compiler macro

Hard to get one source for CUDA-C
- Many scientific codes are written in Fortran
- CUDA-C has quite different code structure and should be written in C

Performance impact
- Highest performance tuning techniques hardly allow one source
Faster hardware provides us with potential to the performance. However, we can exploit the potential only through better software.

Better software on accelerators generally means that it utilizes many cores and wide vectors simultaneously and efficiently.

In practice, those massive parallelisms can be achieved by, among others, 1) re-using data that are loaded onto faster memory and 2) accessing successive array elements with aligned unit-stride manner.
THANK YOU

- Contact: youngsung@ucar.edu
- ASAP: http://www2.cisl.ucar.edu/org/cisl/tdd/asap