Cray OpenACC Case Studies

Eric Dolven
Cray, Inc
Outline

● vdmintv
  ● Excerpted from NIM (Non-hydrostatic Icosahedral Model) dynamics, developed at NOAA ESRL
  ● F2C-ACC is the standard build environment
  ● It is desired to develop a performance comparable port to OpenACC
  ● Case study walking through performance gains with OpenACC
  ● Discuss possible reasons why F2C remains the faster option

● TPKERNEL
  ● Excerpted from the cubed sphere dynamical core developed at NOAA GFDL
  ● It is desired to run this as fast as possible on whatever platform is available
vdmintv: Background

REAL*4 :: rhsu(nz,nob)
REAL*4 :: rhsv(nz,nob)
REAL*4 :: Tgtu(nz,npp)
REAL*4 :: Tgtv(nz,npp)

!$acc parallel num_gangs(10242)
!$acc loop gang private(Tgtu,Tgtv,rhsu,rhsv)
do ipn=ips,ipe
!$acc loop vector
do k=1,nz-1
    ...
endo
CALL solveiThLS2

!$acc loop seq
do isn = 1,nprox(ipn)
!$acc loop vector
do k=1,nz-1
    ...
endo !k-loop
END SUBROUTINE solveiThLS2

subroutine solveiThLS2
!$acc loop vector
do k=1,nz
    ...
endo !k-loop
END SUBROUTINE solveiThLS2

}
**vdmintv: Data**

- **Configuration: Cray XK Kepler K20x with Interlagos**
  1. `ftn -O2 -h acc,noopm,acc_model=fast_addr`
  2. `VL_96`
  3. `VL_96 + 96 registers (-Wx,"--maxrregcount=96")`
  4. `VL_96 + 96 registers + Tgt[u|v] shared memory`
  5. `VL_96 + 96 registers + rhs[u|v] shared memory`
  6. `F2C_ACC`

<table>
<thead>
<tr>
<th></th>
<th>gputime</th>
<th>th.blk</th>
<th>reg/thrd</th>
<th>occ</th>
<th>g_ld</th>
<th>l_Id</th>
<th>sh_ld</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23855.87</td>
<td>128</td>
<td>128</td>
<td>0.25</td>
<td>1349410</td>
<td>85432</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>19833.54</td>
<td>96</td>
<td>127</td>
<td>0.234</td>
<td>1350447</td>
<td>65250</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>17739.39</td>
<td>96</td>
<td>96</td>
<td>0.281</td>
<td>1352424</td>
<td>172062</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>15621.73</td>
<td>96</td>
<td>96</td>
<td>0.281</td>
<td>1185684</td>
<td>112320</td>
<td>155448</td>
</tr>
<tr>
<td>5</td>
<td>12492.26</td>
<td>96</td>
<td>96</td>
<td>0.281</td>
<td>469824</td>
<td>17568</td>
<td>696132</td>
</tr>
<tr>
<td>6</td>
<td>8923.936</td>
<td>96</td>
<td>81</td>
<td>0.281</td>
<td>493704</td>
<td>0</td>
<td>121380</td>
</tr>
</tbody>
</table>
vdmintv: Analysis and comparison

- So why is F2C-ACC still 30% faster than the optimized OpenACC?
- Global memory access is about the same
- Shared memory access is about the same
- Local memory access is zero
  - F2C is using registers more efficiently
- **F2C demotes 2-D arrays rhsu and rhsv**
  - This currently has to be done by hand with OpenACC
  - Potentially, one could use the CACHE directive creatively

```
!ACC$DATA(<Tgtu,Tgtv:none,shared>)
!ACC$REGION (<nz>,<(ipe-ips+1)>,
          <nz,nob,nbf,amtx1:in>,
          <rhsu,rhsv:none,local,demote(1)>,
          <Tgtu,Tgtv:none,shared>) BEGIN
```
**TPKERNEL: Background**

- **Three strategies:**
  - OpenMP
  - OpenACC coarse grain parallelism (device routines)
  - OpenACC fine grain parallelism

```fortran
REAL*8 arrays(:,,:,192)
!$OMP PARALLEL
  do nt=1,niters
!$OMP DO
    do nl=1,nz
      call fv_tp_2d(arrays(1:192,1:192))
    enddo
  enddo
!$OMP END PARALLEL
```
TPKERNEL: Background

● Three strategies:
  ● OpenMP
  ● **OpenACC** coarse grain parallelism (device routines)
  ● OpenACC find grain parallelism

```fortran
REAL*8 arrays(:,:,:,:)

!$ACC ROUTINE(fv_tp_2d_acc) WORKER
do nt=1,niters
  !$ACC PARALLEL ASYNC VECTOR_LENGTH(NTPB)
  !$ACC LOOP GANG
    do nl=1,nz
      call fv_tp_2d_acc(arrays(1:192,1:192))
    enddo
  !$ACC END PARALLEL
endo
```
TPKERNEL: Background

- Three strategies:
  - OpenMP
  - OpenACC coarse grain parallelism (device routines)
  - OpenACC fine grain parallelism

```c
REAL*8 arrays(:,:,:)
do nt=1,niters
    call fv_tp_2d_acc(arrays(1:192,1:192,1:64))
enddo

subroutine fv_tp_2d_acc(arrays(:,:,:))
! Local
    real, dimension(isd:ied,jsd:jed,nz) :: qi, qj, fx2, fy2, fxx, fyy, ra
!$ACC DATA PRESENT(arrays), CREATE(<local vars>)
!$ACC PARALLEL LOOP collapse(3) gang vector
    do k=1,nz
        do j=jsd,jed
            do i=is,ie+1
                fxx(i,j,k) = xfx(i,j,k) * fx2(i,j,k)
            enddo
        enddo
    enddo
enddo
```
TPKERNEL: Data
Run on Cray XK Kepler K20x with Interlagos

<table>
<thead>
<tr>
<th>OMP THREADS</th>
<th>OMP (sec)</th>
<th>Coarse ACC (sec)</th>
<th>Fine ACC (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17.309</td>
<td>3.158</td>
<td>1.573</td>
</tr>
<tr>
<td>2</td>
<td>9.225</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4.975</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4.616</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>3.924</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- OMP scales reasonably well up to a point
- Coarse grain OpenACC is not faster than OMP
- Why is this?
TPKERNEL: Coarse grain redux

```fortran
!$ACC ROUTINE(fv_tp_2d_acc) WORKER
do nt=1,niters
  !$ACC PARALLEL ASYNC VECTOR_LENGTH(NTPB)
  !$ACC LOOP GANG
do nl=1,nz
    call fv_tp_2d_acc(arrays(:,:))
  enddo
$ACC END PARALLEL
enddo

subroutine fv_tp_2d_acc(arrays(:,:))
  ! Local
  real, dimension(isd:ied,jsd:jed) :: qi, qj, fx2, fy2, fxx, fyy, ra
!$ACC ROUTINE WORKER
!$ACC ROUTINE(xtp) WORKER
!$ACC ROUTINE(ytp) WORKER
!$ACC ROUTINE(copy_corners) WORKER

!$ACC DATA PRESENT(arrays(:,:)), CREATE(<local vars>)
```

9/17/2014
TPKERNEL: Coarse grain redux

● Local variables for all the device routines must be malloc’d on each threadblock.
● Because this totals over 4MB, they are placed on the heap
● Thread 0 for each threadblock must perform the malloc, and to be threadsafe, they must be done serially.
● This is much more expensive than OMP where private variables are allocated on the private stack of each thread.
● What I omitted to reveal is that the coarse grain OpenACC parallel runs require

```
CRAY_ACC_MALLOC_HEAPSIZE=512mb
```

Or the run fails with:
```
aprun -n1 ./executable
block 10, thread 0: malloc(2195424) failed
```
TPKERNEL: Performance portability

- **PGI**
  - 1.561 seconds on Accelerator; 20.406 seconds on Host

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.32%</td>
<td>126.42ms</td>
<td>200</td>
<td>632.12us</td>
<td>628.45us</td>
<td>638.66us</td>
<td>xtp_171_gpu</td>
</tr>
<tr>
<td>8.99%</td>
<td>121.99ms</td>
<td>200</td>
<td>609.95us</td>
<td>607.07us</td>
<td>614.78us</td>
<td>xtp_239_gpu</td>
</tr>
<tr>
<td>8.79%</td>
<td>119.37ms</td>
<td>200</td>
<td>596.82us</td>
<td>593.76us</td>
<td>601.28us</td>
<td>ytp_408_gpu</td>
</tr>
<tr>
<td>8.69%</td>
<td>117.90ms</td>
<td>200</td>
<td>589.50us</td>
<td>586.75us</td>
<td>593.76us</td>
<td>ytp_323_gpu</td>
</tr>
<tr>
<td>6.66%</td>
<td>90.434ms</td>
<td>200</td>
<td>452.17us</td>
<td>450.69us</td>
<td>453.60us</td>
<td>xtp_192_gpu</td>
</tr>
<tr>
<td>6.60%</td>
<td>89.525ms</td>
<td>200</td>
<td>447.62us</td>
<td>446.46us</td>
<td>448.96us</td>
<td>xtp_215_gpu</td>
</tr>
</tbody>
</table>

- **CCE**
  - 1.573 seconds on Accelerator; 17.601 seconds on Host

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.91%</td>
<td>124.96ms</td>
<td>200</td>
<td>624.79us</td>
<td>619.07us</td>
<td>631.97us</td>
<td>xtp$ck_L271_30</td>
</tr>
<tr>
<td>8.51%</td>
<td>119.33ms</td>
<td>200</td>
<td>596.66us</td>
<td>592.03us</td>
<td>601.83us</td>
<td>ytp$ck_L440_44</td>
</tr>
<tr>
<td>8.00%</td>
<td>112.19ms</td>
<td>200</td>
<td>560.95us</td>
<td>558.85us</td>
<td>563.43us</td>
<td>xtp$ck_L201_27</td>
</tr>
<tr>
<td>7.97%</td>
<td>111.74ms</td>
<td>200</td>
<td>558.72us</td>
<td>555.17us</td>
<td>569.57us</td>
<td>xtp$ck_L222_28</td>
</tr>
<tr>
<td>7.78%</td>
<td>109.14ms</td>
<td>200</td>
<td>545.69us</td>
<td>543.30us</td>
<td>548.64us</td>
<td>ytp$ck_L357_39</td>
</tr>
<tr>
<td>6.88%</td>
<td>96.470ms</td>
<td>200</td>
<td>482.35us</td>
<td>480.64us</td>
<td>483.97us</td>
<td>xtp$ck_L246_29</td>
</tr>
</tbody>
</table>
!$ACC END DATA