OpenACC performance tuning and profiling

Alistair Hart
Cray Exascale Research Initiative Europe

Contents

● First we talk about a few tuning tips for OpenACC
  ● How long is a piece of string?
    ● optimisation tends to be a case-by-case process
    ● OpenACC makes it easier; e.g. changing loop schedules, caching...
    ● the same general rules apply as for CPUs
      ● there is still some trial and error

● Then we talk about how you can get the information you need to make these tuning decisions
  ● Discuss in the context of Cray tools: CrayPAT and Reveal
  ● But the same principles apply for other profilers
Tuning OpenACC performance

- **Tuning needs input:**
  - There are three main sources of information; make sure you use them:
    - Compiler feedback (static analysis)
      - Loopmark files for CCE; `-minfo=accel` for PGI
    - Runtime commentary (CCE only)
    - Code profiling
      - CrayPAT
      - Nvidia compute profiler
      - `pgprof` for PGI

- **Remember the Golden Rules of performance tuning:**
  - Always profile the code yourself
  - Always verify claims like "this is always the slow routine"
  - Codes/computers change
  - Optimize the real problem running on the production system
    - A small testcase running on a laptop will have a very different profile
  - Optimize the parts of the code that take the most time
    - Even if these are not the exciting bits of the code
    - E.g., it might not be GPU compute; it might be MPI comms, I/O...
  - Remember that the balance of CPU/GPU/network/I/O will change as you go
    - So keep profiling

Tuning OpenACC codes

- **The main optimisation you can do is minimising data movements**
  - How can I tell if these are important?
    - CrayPAT will show the proportion of time spent in data transfers
    - Loopmark comments will tell you which arrays might be transferred
    - Runtime commentary will tell you which arrays actually were moved and how often
  - What can I do?
    - Use data regions to keep data resident on the accelerator where possible
    - Only transfer the data you need
      - If you only need to transfer some of an array (e.g., halo data, debugging values), rather than use `copy` clause, use `create` clause and `update` directive.
    - Another gotcha: beware of GPU memory allocation
      - If a routine using big temporary arrays is called many times, even `create` clause can have a big overhead
      - Consider keeping the array(s) allocated between calls
        - Add it to a higher data region as `create` and use `present` clause in subprogram
        - (Not good for a memory-bound code, of course)
Kernel optimisation

- The next optimisation is to make sure all the kernels vectorise
  - How can I tell if this is a problem?
    - if a kernel is surprisingly slow on accelerator
    - in a wildly different place in the the profile compared to running on CPU
    - examine the loopmark compiler commentary files
  - loop iterations should be divided over both the threads in a threadblock (vector) and over the threadblocks (gang)
    - CCE: you should see either:
      - If a single loop is divided over both levels of parallelism, look for: \texttt{Gg}
      - If two different loops divided, look for \texttt{G} and \texttt{2g-s} (maybe with numbers between)
  - generally want to vectorise the innermost loop
    - usually fastest-moving array index, for coalescing
  - if not, can the inner loop be vectorised?
    - i.e. can loop iterations be computed in any order?
    - if not, rewrite code
      - avoid loop-carried dependencies
      - e.g. buffer packing: calculate rather than increment
      - these rewrites will probably perform better on CPU also

Forcing compiler to vectorise

- If the loop is vectorisable, guide the compiler
  - a gentle hint:
    - put "acc loop independent" directive above this loop
    - could also use CCE directive "ldir\$ concurrent"
      - see "man intro\_directives" for details
  - a direct order:
    - put "acc loop vector" directive above this loop
    - check the code is still correct and running faster, though:
      - the compiler might not be vectorising for a good reason

- If the inner loops is vectorising but performance is bad
  - is the inner loop really the one to vectorise?
    - here we should vectorise the i-loop (know \texttt{mmax} small)
  - put "acc loop seq" directive above loop
    - then executed redundantly by every thread
It's all vectorising, but still performing badly

- Profile the code and start whacking moles
  - optimise the thing that is taking the time
  - if it really is a GPU compute kernels...

- GPUs need lots of parallel tasks to work well

- OpenACC loop schedules are limited by the loop bounds
  - "tall, skinny" loopnests \((j=1:\text{big}; i=1:\text{small})\) won't schedule well
    - less than 32 iterations won't even fill a warp, so wasted SIMT
  - "short, fat" loopnests \((j=1:\text{small}; i=1:\text{big})\) also not good
    - want lots of threadblocks to swap amongst SMs
  - collapse clause is way of increasing flexibility
    - the compiler may use this automatically (\(C\) in loopmark)
    - no guarantee that it is faster
      - e.g. index rediscovery requires expensive integer divisions

Avoiding temporary arrays

- Perfect loop nests often perform better than imperfect
  - Imperfect loop nests often use temporary arrays
    - e.g. in a stencil like MultiGrid, to avoid additional duplicated computation
  - With OpenACC, these arrays are privatised; too big for shared memory
    - Imperfect loop nest also means scheduling decisions are restricted

- Try two approaches; which (if any) faster depends on code
  - Remove temporary arrays by manually inlining (eliminate array \(b\))
    - one perfect loop nest; cache clause can use shared mem/regs where needed
  - Manually privatise arrays and fission the loopnest \((b(i)\rightarrow b(i,j))\)

\[
\begin{align*}
\text{DO} & \ j = 1,N \\
\text{DO} & \ i = 0,M+1 \\
& \ b(i) = a(i,j+1) + a(i,j-1) \\
\text{ENDDO}\end{align*}
\]

\[
\begin{align*}
\text{DO} & \ j = 1,N \\
\text{DO} & \ i = 1,M \\
& \ c(i,j) = a(i+1,j+1) + a(i+1,j-1) + a(i-1,j+1) + a(i-1,j-1) \\
\text{ENDDO}\end{align*}
\]
Using the collapse clause

- Consider a three-level loopnest (i inside j inside k)
  - needs to be perfectly nested to use collapse
  - Collapse all three loops and schedule across GPU
    - "acc parallel loop collapse(3) gang worker vector" above k-loop
      - probably don't need "gang worker vector" here
    - Schedule inner two loops over threads in threadblock
      - "acc parallel loop gang" above k-loop
      - "acc loop collapse(2) vector" above j-loop
        - don't need "gang"; enough warps are used to cover all the iterations
    - Schedule outer two loops over the threadblocks
      - "acc parallel loop collapse(2) gang" above k-loop
      - "acc loop vector" above i-loop
    - Schedule outer two loops together over entire GPU
      - "acc loop collapse(2) gang worker vector" above k-loop
      - "acc loop seq" above i-loop
    - Schedule k-loop and i-loop together over entire GPU
      - collapsed loops must be perfectly nested; you'll need to reorder the code

More drastic performance optimisations

- Would reordering your data structures help?
  - Example:
    - \( N_{\text{max}} \) particles each have \( S_{\text{max}} \) internal properties
      - code combines the internal properties together for each particle separately
    - CPU code usually stores data as \( f(S_{\text{max}}, N_{\text{max}}) \) or \( f[N_{\text{max}}][S_{\text{max}}] \)
      - good cache reuse when we access all the properties of a particle
    - GPU code would normally parallelise over the particles
      - each thread processes the internal properties of a single particle
        - first warp would attempt vector load of \( s^{th} \) prop. of 32 particles: \( f(s, 1:32) \)
          - no coalescing (where vector load is contiguous block of memory)
            - very poor performance (even if \( S_{\text{max}} \) is small)
    - Better to reorder data so site index fastest: \( f_{\text{gpu}}(N_{\text{max}}, S_{\text{max}}) \)
      - vector load of \( f_{\text{gpu}}(1:32, s) \) now stride-1 in memory
        - if code memory-bandwidth bound, a big improvement
  - Quite an effort to reorder data structures in the code
    - but... may also see benefits on CPU
      - especially with AVX (and longer vectors in future processors)
How do I get the information I need?

- Using Cray performance tools on a Cray XK6
- There are other profilers
  - this shows the sort of breakdown that is useful

Programming Models Supported for the GPU

- Goal is to provide whole program analysis for programs written for x86 or x86 + GPUs
- Development focus is on support of CCE with OpenACC directives
- Cray XK programming models supported
  - OpenACC, CUDA, PGI accelerator directives
Introduction to Performance Statistics

- Default statistics collected when accelerated directives are encountered with **tracing**
  - Host time for kernel launches, data copies and synchronization with the accelerator
  - Accelerator time for kernel execution and data copies
  - Data copy size to and from the accelerator

- Collection enabled by default for programs built with CCE

- Collection enabled with runtime environment variable for CUDA

Collecting Accelerator Statistics

- **Tracing** produces GPU performance data:
  - `$ pat_build -u my_program`
  - `$ pat_build -w my_program`

- **Sampling** does not collect GPU performance data (although samples may be seen within CUDA libraries)
  - `$ pat_build my_program`
  - `$ pat_build -O apa my_program`

- The following disables compilation for OpenACC directives and therefore collection of accelerator statistics:
  - `$ cce -h noacc`
  - `$ cce -h profile_generate my_program.f`
Accelerator Table Column Definitions

- **Host Time%**
  - percentage of wallclock time for events
- **Host Time**
  - wallclock time, in seconds, for the event
- **Acc Time**
  - amount of time the event executed on the accelerator
- **Acc Copy In**
  - amount of data copied to the accelerator
- **Acc Copy Out**
  - amount of data copied from the accelerator
- **Calls**
  - the number of time the event occurred

_All of the above are summed for regions and functions_

Accelerator Table Layout

- **Notes** section at the beginning of the tables contains helpful information describing how the table was generated and suggestions on how to produce additional related tables.

- **Data** presented in default text report is _organized_ as a calltree with functions/accelerated regions sorted in decreasing order _by Host Time_

- Called functions, regions and events are indented to the right
- Left-most column represents indentation in table
- By default, cells in accelerator tables that have no data are marked with ‘—’
Example Accelerator Statistics

<table>
<thead>
<tr>
<th>Table 1: Time and Bytes Transferred for Accelerator Regions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
</tr>
<tr>
<td>Time%</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>100.0%</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>100.0%</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>63.5%</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>22.1%</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>20.6%</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>18.8%</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>1.6%</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>1.1%</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>1.1%</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

GPU or Kernel Statistics

- **Current metrics supported**
  - Kernel grid size
  - Block size
  - Amount of shared memory dynamically allocated for kernel

- **Statistics are averaged across all kernel launches on all PEs**

- **Additional metrics will be available in future releases**
Example Kernel Statistics

<table>
<thead>
<tr>
<th>Function</th>
<th>Dim</th>
<th>Dim</th>
<th>Dim</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>X Dim</th>
<th>Y Dim</th>
<th>Z Dim</th>
</tr>
</thead>
</table>
| streaming ACC_KERNEL | 62163 | 1 | 1 | 1024 | 1 | 1 | 1 | 909 
| grad_exchange ACC_KERNEL | 402 | 1 | 1 | 128 | 1 | 1 | 1 | 443 
| grad_exchange ACC_KERNEL | 402 | 1 | 1 | 128 | 1 | 1 | 1 | 467 
| grad_exchange ACC_KERNEL | 402 | 1 | 1 | 128 | 1 | 1 | 1 | 500 
| cal_velocity ACC KERNEL | 400 | 1 | 1 | 512 | 1 | 1 | 1 | 1126 
| collision ACC KERNEL | 400 | 1 | 1 | 512 | 1 | 1 | 1 | 474 
| collision ACC KERNEL | 400 | 1 | 1 | 128 | 1 | 1 | 1 | 597 
| wall_boundary ACC KERNEL | 400 | 1 | 1 | 128 | 1 | 1 | 1 | 973 
| collision ACC KERNEL | 400 | 1 | 1 | 128 | 1 | 1 | 1 | 629 
| recolor ACC KERNEL | 400 | 1 | 1 | 128 | 1 | 1 | 1 | 823 
| injection ACC KERNEL | 128 | 1 | 1 | 64 | 1 | 1 | 1 | 1281 
| streaming_exchange ACC KERNEL | 128 | 1 | 1 | 128 | 1 | 1 | 1 | 829 
| streaming_exchange ACC KERNEL | 128 | 1 | 1 | 128 | 1 | 1 | 1 | 729 
| streaming_exchange ACC KERNEL | 128 | 1 | 1 | 128 | 1 | 1 | 1 | 641 
| streaming_exchange ACC KERNEL | 128 | 1 | 1 | 128 | 1 | 1 | 1 | 538 
| collision ACC KERNEL | 101 | 1 | 1 | 128 | 1 | 1 | 1 | 612 
| set_boundary micro press ACC KERNEL | 101 | 1 | 1 | 128 | 1 | 1 | 1 | 299 
| set_boundary macro press2 ACC KERNEL | 101 | 1 | 1 | 128 | 1 | 1 | 1 | 259 
| streaming ACC KERNEL | 14 | 1 | 1 | 256 | 1 | 1 | 1 | 919 

GPU Hardware Performance Counters

- Enable collection similarly to CPU counter collection:
  - CPU: PAT_RT_HWPC=group or events
  - GPU: PAT_RT_ACCPC=group or events
  - Can't mix collecting CPU and GPU counters in same run

- Enables causes change in behavior of application:
  - Host needs to synchronize with the accelerator at each event (since accelerator executes asynchronously with the host)

- Can be seen through accelerator table
  - **No counters**: time spent waiting for kernel to complete is shown with ACC_SYNC_WAIT (a synchronization created by the compiler)
  - **Counters**: perftools syncs with accelerator with each event so Host Time is exclusive time for the containing region (since waiting occurs within the event's trace point instead of in the compiler sync)
GPU HW Counter Groups

- A predefined set of groups has been created for ease of use (combines events that can be counted together)

- ACCPC groups start at 1000

- Specify group by number or name
  - PAT_RT_ACCPC=1000  OR
  - PAT_RT_ACCPC=inst_exec

- See accpc(5) man page for list of groups and their descriptions

Examples of GPU Counter Groups

- **1000, inst_exec**
  - Inst_executed: “Number of instructions executed, not including replays.”
  - Warps_launched: “Number of warps launched.”
  - Divergent_branch: “Number of divergent branches within a warp. This counter will be incremented by 1 if at least 1 thread in a warp diverges via a data dependent conditional branch.”

- **1013, active_warps**
  - active_warps: “Accumulated number of active warps per cycle. For every cycle it increments by the number of active warps in the cycle which can be in the range of 0 to 48.

- **1014, sm_cta_launched**
  - sm_cta_launched: “Number of thread blocks launched.”
Accelerator Statistics with GPU Counters

Table 2: Time and Bytes Transferred for Accelerator Regions

<table>
<thead>
<tr>
<th>Host</th>
<th>Host</th>
<th>Acc</th>
<th>Acc Copy</th>
<th>Acc Copy</th>
<th>Calls</th>
<th>Calltree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>(MBytes)</td>
<td>(MBytes)</td>
<td></td>
<td>PE=HIDE</td>
</tr>
<tr>
<td>100.0%</td>
<td>42.787</td>
<td>35.429</td>
<td>2554.726</td>
<td>2559.820</td>
<td>38164</td>
<td>Total</td>
</tr>
</tbody>
</table>

| 100.0% | 42.787 | 35.429 | 2554.726 | 2559.820 | 38164 | himenobmtxp_.ACC_DATA_REGION@li.65 |

| 39 99.6% | 42.628 | 35.273 | 2554.726 | 2559.820 | 38152 | jacobi_.ACC_DATA_REGION@li.227 |

| 5 67.4% | 28.836 | 28.168 | 0.004 | 0.004 | 5015 | jacobi_.ACC_REGION@li.309 |

| 6 66.2% | 28.324 | 28.168 | 0.004 | 0.004 | 1003 | jacobi_.ACC_REGION@li.334 (exclusive) |

| 5 9.5% | 1.998 | 1.998 | 0.004 | 0.004 | 2 | jacobi_.ACC_DATA_REGION@li.227 (exclusive) |

| 5 10.2% | 4.384 | 3.786 | 0.004 | 0.004 | 4012 | jacobi_.ACC_REGION@li.334 |

| 6 66.2% | 1.113 | 0.513 | 0.004 | 0.004 | 1003 | jacobi_.ACC_REGION@li.334 (exclusive) |

| 5 4.7% | 1.998 | 1.998 | 0.004 | 0.004 | 4012 | jacobi_.ACC_REGION@li.309 |

| 6 1.8% | 0.778 | 0.778 | 0.004 | 0.004 | 1003 | jacobi_.ACC_REGION@li.309 (exclusive) |

| 5 2.6% | 1.113 | 0.513 | 0.004 | 0.004 | 4012 | jacobi_.ACC_REGION@li.274 |

| 6 1.8% | 0.778 | 0.778 | 0.004 | 0.004 | 1003 | jacobi_.ACC_REGION@li.274 (exclusive) |

GPU Counter Statistics

Table 3: ACC Performance Counter Data

<table>
<thead>
<tr>
<th>divergent_branch</th>
<th>warps_launched</th>
<th>inst_executed</th>
<th>Calltree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>PE=HIDE</td>
</tr>
</tbody>
</table>

| 8260749 | 517114 | 9063662138 | Total    |

| 8260749 | 517114 | 9063662138 | himenobmtxp_.ACC_DATA_REGION@li.65 |

| 3 | 8256619 | 516978 | 9056964810 | jacobi_.ACC_DATA_REGION@li.227 |

| 5 | 4092240 | 64192 | 1594128080 | jacobi_.ACC_REGION@li.334 |

| 6 | 4067186 | 64016 | 7407047916 | jacobi_.ACC_REGION@li.309 |

| 5 | 16358 | 65430 | 8195170 | jacobi_.ACC_REGION@li.274 |

| 6 | 16179 | 64717 | 10548871 | jacobi_.ACC_REGION@li.286 |

27-30 Aug.12  PRACE training course, Edinburgh
Alternate Views of Accelerator Data

- Flat table of acc events (flat table of events sorted in decreasing order by Host Time)
  - `$ pat_report -O acc_fu`

- Calltree by acc time (calltree with functions/regions sorted in decreasing order by Acc Time)
  - `$ pat_report -O acc_time`

- Flat table of events sorted in decreasing order by acc time
  - `$ pat_report -O acc_time_fu`

- See ‘–O keywords’ under pat_report(1) man page

Performance Statistics for Programs with CUDA

- Cray PE supports CUDA that is contained within a function

- Accelerator performance and kernel level statistics available for code written using CUDA runtime API and CUDA driver API

- Implemented using Nvidia’s CUPTI API which provides callbacks for CUDA functions
  - Perftools uses these callbacks to collect statistics when kernels are launched and data is copied to and from the GPU
CUDA (cont’d)

- **Build program that contains CUDA:**
  - Place CUDA code in separate function (compilation unit)
  - Compile functions containing CUDA with `nvcc`
    - Add `-g` option to `nvcc` command if you plan to use performance tools
  - Link object files using a PrgEnv-XXX programming environment

- **Instrument program for data collection**
  - `pat_build -u` (trace user functions, associates results with function that contains the CUDA code)
  - `pat_build -w` (trace MAIN, associates results with MAIN)

- **Enable performance data collection:**
  - Set runtime environment variable, `PAT_RT_ACC_STATS`, to ‘all’

- **Performance statistics available in default text report**

CUDA example

- **Access software**
  - `$ module load PrgEnv-cray craype-accel-nvidia20`
  - `$ module load perftools`

- **Build program with –g to get symbol information**
  - `$ nvcc -g -arch=sm_20 -c reduce0.cu reduce6.cu`
  - `$ ftn -o gpu_cuda gpu_reduce_int_cuda.F90 reduce0.o reduce6.o`

- **Instrument program for tracing**
  - `$ pat_build -u gpu_cuda`

- **Enable collection of statistics for CUDA, and run instrumented program**
  - `$ export PAT_RT_ACC_STATS=all`

- **Create report**
  - `$ pat_report gpu_cuda+pat+26300-0t.xf > cuda_report`
CUDA Example (2)

Table 1: Profile by Function Group and Function

<table>
<thead>
<tr>
<th>Time%</th>
<th>Time</th>
<th>Imb.</th>
<th>Imb.</th>
<th>Calls</th>
<th>Group</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0%</td>
<td>0.568974</td>
<td>--</td>
<td>--</td>
<td>42.0</td>
<td>Total</td>
<td></td>
</tr>
<tr>
<td>99.9%</td>
<td>0.568510</td>
<td>--</td>
<td>--</td>
<td>40.0</td>
<td>USER</td>
<td></td>
</tr>
<tr>
<td>97.6%</td>
<td>0.555595</td>
<td>--</td>
<td>--</td>
<td>3.0</td>
<td>reduce0_cuda_</td>
<td></td>
</tr>
<tr>
<td>1.8%</td>
<td>0.009986</td>
<td>--</td>
<td>--</td>
<td>7.0</td>
<td>wake_up_gpu_</td>
<td></td>
</tr>
<tr>
<td>0.1%</td>
<td>0.000334</td>
<td>--</td>
<td>--</td>
<td>1.0</td>
<td>DL</td>
<td></td>
</tr>
<tr>
<td>0.0%</td>
<td>0.000131</td>
<td>--</td>
<td>--</td>
<td>1.0</td>
<td>ETC</td>
<td></td>
</tr>
</tbody>
</table>

CUDA Example (3)

Table 2: Time and Bytes Transferred for Accelerator Regions

<table>
<thead>
<tr>
<th>Host</th>
<th>Host</th>
<th>Acc</th>
<th>Acc Copy</th>
<th>Acc Copy</th>
<th>Calls</th>
<th>Calltree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time%</td>
<td>Time</td>
<td>Time</td>
<td>In</td>
<td>Out</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(MBytes)</td>
<td>(MBytes)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100.0%</td>
<td>0.009</td>
<td>0.004</td>
<td>8.000</td>
<td>2.000</td>
<td>15</td>
<td>Total</td>
</tr>
<tr>
<td>100.0%</td>
<td>0.009</td>
<td>0.004</td>
<td>8.000</td>
<td>2.000</td>
<td>15</td>
<td>reductions_</td>
</tr>
<tr>
<td>88.3%</td>
<td>0.008</td>
<td>0.004</td>
<td>6.000</td>
<td>2.000</td>
<td>6</td>
<td>wake_up_gpu_</td>
</tr>
<tr>
<td>6.3%</td>
<td>0.001</td>
<td>0.000</td>
<td>1.000</td>
<td>0.000</td>
<td>5</td>
<td>reduce0_cuda_</td>
</tr>
<tr>
<td>5.9%</td>
<td>0.001</td>
<td>0.000</td>
<td>1.000</td>
<td>0.000</td>
<td>2</td>
<td>reduce0_cuda_(exclusive)</td>
</tr>
<tr>
<td>5.4%</td>
<td>0.001</td>
<td>0.000</td>
<td>1.000</td>
<td>0.000</td>
<td>4</td>
<td>reduce6_cuda_</td>
</tr>
<tr>
<td>5.2%</td>
<td>0.000</td>
<td>0.000</td>
<td>1.000</td>
<td>0.000</td>
<td>2</td>
<td>reduce6_cuda_(exclusive)</td>
</tr>
</tbody>
</table>
CUDA Performance Statistics Tips

- **Limitations**
  - Labels for events don’t include line numbers or type of event (data copy, data region or kernel, etc.) like are available with CCE
  - Function association

- **Multiple kernels within a single function wrapper will have statistics reporting back to wrapper**
  - Create multiple functions for CUDA to separate results

- **Statistics also available with pat_region API**
  - Use of pat_region API associates statistics with a region (can separate statistics for different kernels)

PGI Accelerator Directive Support

- **Use same method as with CUDA code**
  - Instrument for tracing
  - Set PAT_RT_ACC_STATS to ‘all’

  OR

  - Use pat_region API around accelerated loop or region

- **Performance statistics will show up under the first containing traced function, and not the directive (as with CCE)**
Reveal

- New code restructuring and analysis assistant…
  - Uses both the performance toolset and CCE’s program library functionality to provide static and runtime analysis information
  - Assists user with the code optimization phase by correlating source code with analysis to help identify which areas are key candidates for optimization

- Key Features
  - Annotated source code with compiler optimization information
    - Feedback on critical dependencies that prevent optimization
  - Scoping analysis
    - Identify, shared, private and ambiguous arrays
      - Allow user to privatize ambiguous arrays
      - Allow user to override dependency analysis
  - Source code navigation based on performance data collected through CrayPat

Reveal 1.0 Functionality Available 2H2012

- Visual loopmark and source navigation
- Integrated loop timing statistics from CrayPat
- Scoping and parallelization assistance
- Initial focus is on Fortran programs (C and C++ will have more restricted support)
- CCE must be available on same platform as Reveal
How to use Reveal 0.1 (early alpha version)

- Collect loop statistics to identify loops to parallelize
  
  $ ftn -hprofile_generate$

- Select loops to analyze
  
  !dir$ omp_analyze_loop

- Start with clean build and create program library
  
  $ ftn -hpl=/full_path/program.pl ...$

- Launch reveal:
  
  $ reveal program.pl$

- Review loopmark information (no cce loopmark option needed during build)

How to use Reveal 0.1 (2)

- Select serial loops to scope

- Expand files and functions to look for loops with scoping information (highlighted green if scoping info present, highlighted red if unknowns present)

- Scope any unknowns

- Dump scoping information to stderr (where you launched reveal) to copy and paste into a directive in your source
Visualize CCE’s Loopmark with Performance Profile

Visualize CCE’s Loopmark with Performance Profile (2)
Visualize CCE's Loopmark (2)

Negative messages are flagged red.

View Pseudo Code for Inlined Functions

Expand to see pseudo code.

Inlined call sites marked.
Scoping Assistance

27-30 Aug 12  PRACE training course, Edinburgh

Scoping Assistance (2)

27-30 Aug 12  PRACE training course, Edinburgh
Scoping Assistance (3)

Assist user with OpenMP hints

private (a, ai b, bi, c, ...) reduction (MAX: svel) firstprivate (amid, ar, ctdx, clft, ...)

Reveal Next Steps

- OpenMP directive generation and insertion into source
- Display pseudo code to show compiler restructuring
- Add loop timing information to navigation panel
- Focus on loops with unknowns
  - Improve efficiency of tool by only forcing user to review loops with unknowns or parallelization hints
  - Ability to automatically insert directives for loops
- Highlight “interesting” compiler feedback
- Create OpenAcc directives
Release Overview

- **December 2011 – perftools 5.3.0**
  - OpenACC support, app2 for Windows

- **March 2012 – perftools/5.3.1**
  - CUDA program support

- **April 2012 – perftools/5.3.2**
  - CUDA 4.1 support

- **September 2012 – perftools/6.0.0**
  - kepler, CUDA 5.0, SNB, Reveal