Programming Heterogeneous Many-cores Using Directives

HMPP - OpenAcc

François Bodin, Florent Lebeau, CAPS entreprise
Introduction

• Programming many-core systems faces the following dilemma
  o The constraint of keeping a unique version of codes, preferably mono-language
    • Reduces maintenance cost
    • Preserves code assets
    • Less sensitive to fast moving hardware targets
    • Codes last several generations of hardware architecture
  o Achieve "portable" performance
    • Multiple forms of parallelism cohabiting
      – Multiple devices (e.g. GPUs) with their own address space
      – Multiple threads inside a device
      – Vector/SIMD parallelism inside a thread
    • Massive parallelism
      – Tens of thousands of threads needed
  • For legacy codes, directive-based approach may be an alternative
Profile of a Legacy Application

- Written in C/C++/Fortran
- Mix of user code and library calls
- Hotspots may or may not be parallel
- Lifetime in 10s of years
- Cannot be fully re-written
- Migration can be risky and mandatory

```
while(many){
    ...
    mylib1(A,B);
    ...
    myuserfunc1(B,A);
    ...
    mylib2(A,B);
    ...
    myuserfunc2(B,A);
    ...
}
```
Heterogeneous Many-Cores

- Many general purposes cores coupled with a massively parallel accelerator (HWA)

CPU and HWA linked with a PCIx bus

Data/stream/vector parallelism to be exploited by HWA e.g. CUDA / OpenCL
Outline of the Presentation

- Usual parallel programming techniques in HPC
- Many-core architecture space
- Demonstration code description
- Directives-based programming
- Going further with OpenHMPP
Usual Parallel Programming Techniques in HPC
Styles of Parallel Programming

• Task Parallel
  o Thread based
  o Mostly shared memory ➔ OpenMP directive based API

• Data Parallelism ➔ SIMD / SIMT
  o Same computation applied to all elements of a collection
  o Data distribution over the processors
  o Frequently exploit loop-level parallelism

• Message Passing ➔ MPI Library
  o Mostly distributed memory
  o Frequently use data distribution techniques
Task Based Example: Threads Fork and Join

- The whole the programming environment assumes some level of thread managements
  - Gang scheduling
  - Thread / data affinity / migration
- For instance OpenMP Parallel Loops
  - Express distribution of loop iterations over the threads
  - Various scheduling clauses

```c
#pragma omp parallel for schedule(static, 3)
for (i=0; i<N; i++){
    dowork(i);
}
```
Single Program Multiple Data (SPMD)

- This is the dominant form of parallelism in scientific computing
  - No thread creation overhead
  - Simple
  - Shared or distributed memory architecture

- MPI message passing based on this model
  - Explicit control of all inter-processor communication

![Diagram showing SPMD concept with threads and communication]

- Thread 0
- Thread 1
- Thread 2

Same code

Send

Receive

asynchronous

synchronous
initialize
if (my_rank == 0) {
    sum = 0.0;
    for (source = 1; source < num_procs; source++) {
        MPI_RECV(&value, 1, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &status);
        sum += value;
    }
} else {
    MPI_SEND(&value, 1, MPI_FLOAT, 0, tag, MPI_COMM_WORLD);
}
finalize

SPMD MPI code
1. **Domain decomposition**

forall cells of PE
read properties, state, neighbors;
for time = 1 to end
forall cells of PE
compute next step
update neighbors ghost cells

1. **Add ghost cells**
Many-Core Architecture Space
Heterogeneous Architecture Space

- Achieving "portable" performance across architecture space

- Heterogeneity
  - Different parallel models
  - Different ISAs
  - Different compilers
  - Different memory systems
  - Different libraries

- A code must be written for a set of hardware configurations
  - 6 CPU cores + MIC
  - 24 CPU cores + GPU
  - 12 cores + 2 GPUs
  - ...

Codes need to move in this space and new HWs to come
Where Are We Going?

Evolution of Processing Units in Future Processors

- CUDA/NVIDIA Tesla release
- Specialized Manycores (GPGPU)
- Frequency based Performance Improvement Era
- Manycores processors reaching the general purpose market
- Non migrated applications do not scale up

* Frequency based on Intel Processor (max.)
** Number of Processing Units (~cores x threads)
Intel CPU, NVIDIA GPU
Usual Parallel Programming Won't Work Per Se

- Exploiting heterogeneous many-core with MPI parallel processes
  - Extra latency compared to shared memory use
    - MPI implies some copying required by its semantics (even if efficient MPI implementations tend to reduce them)
    - Cache trashing between MPI processes
  - Excessive memory utilization
    - Partitioning for separate address spaces requires replication of parts of the data
    - When using domain decomposition, the sub-grid size may be so small that most points are replicated (i.e. ghost cells)
    - Memory replication implies more stress on the memory bandwidth which finally prevent scaling

- Exploiting heterogeneous many-core with thread based APIs
  - Data locality and affinity management non trivial
  - Reaching a tradeoff between vector parallelism (e.g. using the AVX instruction set), thread parallelism and MPI parallelism
  - Threads granularity has to be tuned depending on the core characteristics (e.g. SMT, heterogeneity)
  - Most APIs are shared memory oriented
Domain Decomposition Parallelism

32x32x32 cell domain
ghost cells 2 ↔
ghost cells / domain cells = 0.42

1 process → 8 processes

16x16x16 cell domain
ghost cells 2 ↔
ghost cells / domain cells = 0.95
Flexible Code Generation Required

- The parallel programming API must not assume too much about the HW targets
Auto-Tuning is Required to Achieve Some Performance Portability

- The more optimized a code is, the less portable it is
  - Optimized code tends to saturate some hardware resources
  - Parallelism ROI varies a lot
    - i.e. # threads and workload need to be tuned
  - Many HW resources not virtualized on HWA (e.g. registers, #threads)

Example of an optimized versus a non optimized stencil code
Demonstration Code Description
HydroC Code

- **HydroC** is a *summary* from RAMSES
  - Used to study large scale structure and galaxy formation.
  - Includes classical algorithms we can find in many applications codes for Tier-0 systems
  - Solves compressible Euler equations of hydrodynamics, based on finite volume numerical method using a second order Godunov scheme for Euler equations
  - The algorithms have not been modified
  - ~1500 LoC, two versions, Fortran and C, MPI

- **GNU Compiler 4.4.5, MPICH, NV SDK 5.1, CAPS OpenACC, compiler flag -O3**

- **More at**
  - [http://irfu.cea.fr/Phocea/Vie_des_labos/Ast/ast_sstechnique.php?id_ast=904](http://irfu.cea.fr/Phocea/Vie_des_labos/Ast/ast_sstechnique.php?id_ast=904)

*Pierre-François Lavallée, Guillaume Colin de Verdière, Philippe Wautelet, Dimitri Lecas, Jean-Michel Dupays*  

*IDRIS/CNRS, CEA,Centre DAM*
HydroC Application – CPU Version

- **To compile:**

  ```
  $ make
  ```

- **To execute:**

  ```
  $ ./HydroC -i input.base
  ```

- **Output:**

  ```
  +--------------------------+
  | nx=400                  |
  | ny=500                  |
  | nxystep=500             |
  | tend=40.000             |
  | nstepmax=81             |
  | noutput=100000          |
  | dtoutput=0.000          |
  +--------------------------+
  202 252
  --> Step=  1, 1.33631e-03, 1.33631e-03  {296.098 Mflops 39290168} (0.133s)
  --> Step=  2, 2.67261e-03, 1.33631e-03  {332.653 Mflops 39293940} (0.118s)
  . . .
  --> Step=  80, 4.56073e-01, 6.74684e-03  {329.866 Mflops 39398490} (0.119s)
  --> Step=  81, 4.62848e-01, 6.77480e-03  {318.822 Mflops 39399392} (0.124s)
  Hydro ends in 00:00:10.307s (10.307).
  ```
HydroC Migration Steps

Hotspots
- Understand your performance goal (analysis, definition and achievement)
- Know your hotspots (analysis, code reorganization, hotspot selection)
- Establish a validation process
- Set a continuous integration process with the validation

Hours to Days

Define your Parallel Project

Parallelization
- Optimize CPU code
- Exhibit application SIMT parallelism
- Parallelize for Many-core
- Validate execution

Days to Weeks

Port your Application on Many-core

Many-core operational application with known potential

Phase 1
Phase 2

Tuning
- Reduce data transfers
- Optimize kernel execution
- Provide feedback to application programmers for improving algorithm data structures/…
- Consider multiple devices

Weeks to Months

Optimize Your Many-core Application

A corporate project
- Purchasing Department
- Scientists
- IT Department

S0630-31 www.caps-entreprise.com
Migration Process, Tool View

- CPU Code Analysis
  - CPU Profiler

- GPU Programming
  - OpenHMPP/OpenACC
  - Accelerated Libraries
    - Proxy to CuFFT, CuBlas, Cula, ArrayFire, etc.

- GPU Code Analysis
  - Kernel performance
  - CPU-GPU data transfers
    - Tracing visualization tool
HydroC Call-Graph

main (100%)

hydro_godunov (97.0%)

riemann (48.2%)
slope (11.2%)
trace (10.1%)

updateConservativeVars (5.6%)

qleftright (4.9%)
constoprim (4.6%)
cmpflx (3.0%)
gatherConservativeVars (5.4%)
compute_deltat (2.9%)
equationofstate (5.0%)
### Flat profile:

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self seconds</th>
<th>time seconds</th>
<th>calls</th>
<th>self ms/call</th>
<th>total ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>48.24</td>
<td>4.77</td>
<td>4.77</td>
<td>162</td>
<td>29.45</td>
<td>29.45</td>
<td>riemann</td>
</tr>
<tr>
<td>11.23</td>
<td>5.88</td>
<td>1.11</td>
<td>162</td>
<td>6.85</td>
<td>6.85</td>
<td>slope</td>
</tr>
<tr>
<td>10.11</td>
<td>6.88</td>
<td>1.00</td>
<td>162</td>
<td>6.17</td>
<td>6.17</td>
<td>trace</td>
</tr>
<tr>
<td>5.56</td>
<td>7.43</td>
<td>0.55</td>
<td>162</td>
<td>3.40</td>
<td>3.40</td>
<td>updateConservativeVars</td>
</tr>
<tr>
<td>5.36</td>
<td>7.96</td>
<td>0.53</td>
<td>162</td>
<td>3.27</td>
<td>3.27</td>
<td>gatherConservativeVars</td>
</tr>
<tr>
<td>4.96</td>
<td>8.45</td>
<td>0.49</td>
<td>203</td>
<td>2.41</td>
<td>2.41</td>
<td>equation_of_state</td>
</tr>
<tr>
<td>4.85</td>
<td>8.93</td>
<td>0.48</td>
<td>162</td>
<td>2.96</td>
<td>2.96</td>
<td>qlefttright</td>
</tr>
<tr>
<td>4.55</td>
<td>9.38</td>
<td>0.45</td>
<td>162</td>
<td>2.78</td>
<td>2.78</td>
<td>constoprim</td>
</tr>
<tr>
<td>3.03</td>
<td>9.68</td>
<td>0.30</td>
<td>162</td>
<td>1.85</td>
<td>1.85</td>
<td>cmpflx</td>
</tr>
<tr>
<td>1.92</td>
<td>9.87</td>
<td>0.19</td>
<td>41</td>
<td>4.64</td>
<td>7.05</td>
<td>compute_deltat</td>
</tr>
<tr>
<td>0.10</td>
<td>9.88</td>
<td>0.01</td>
<td>162</td>
<td>0.06</td>
<td>0.06</td>
<td>make_boundary</td>
</tr>
<tr>
<td>0.10</td>
<td>9.89</td>
<td>0.01</td>
<td>8</td>
<td>1.25</td>
<td>1.25</td>
<td>ToBase64</td>
</tr>
<tr>
<td>0.00</td>
<td>9.89</td>
<td>0.00</td>
<td>891</td>
<td>0.00</td>
<td>0.00</td>
<td>Dmalloc</td>
</tr>
<tr>
<td>0.00</td>
<td>9.89</td>
<td>0.00</td>
<td>164</td>
<td>0.00</td>
<td>0.00</td>
<td>cclock</td>
</tr>
<tr>
<td>0.00</td>
<td>9.89</td>
<td>0.00</td>
<td>81</td>
<td>0.00</td>
<td>0.00</td>
<td>Imalloc</td>
</tr>
<tr>
<td>0.00</td>
<td>9.89</td>
<td>0.00</td>
<td>81</td>
<td>0.00</td>
<td>0.00</td>
<td>allocate_work_space</td>
</tr>
</tbody>
</table>

We start here for code migration.
Directive-based Programming
Directives-based Approaches

- Supplement an existing serial language with directives to express parallelism and data management
  - Preserves code basis (e.g. C, Fortran) and serial semantic
  - Competitive with code written in the device dialect (e.g. CUDA)
  - Incremental approach to many-core programming
  - Mainly targets legacy codes

- Many variants
  - OpenHMPP
  - PGI Accelerator
  - OpenACC
  - OpenMP Accelerator extension
  - ...

- OpenACC is a new initiative by CAPS, CRAY, PGI and NVidia
  - A first common subset presented at SC11
How Does Directive Based Approach Differ from CUDA or OpenCL?

- HMPP/OpenACC parallel programming model is parallel loop centric
- CUDA and OpenCL parallel programming models are thread centric

```c
void saxpy(int n, float alpha, float *x, float *y){
#pragma acc independent
for(int i = 0; i<n; ++i)
    y[i] = alpha*x[i] + y[i];
}
```

```c
__global__
void saxpy_cuda(int n, float alpha,
    float *x, float *y) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if(i<n) y[i] = alpha*x[i]+y[i];
}
```

```c
int nblocks = (n + 255) / 256;
saxpy_cuda<<<nblocks, 256>>>(n, 2.0, x, y);
```
Two Styles of Directives in CAPS Workbench

- Functions based, i.e. codelet (OpenHMPP)
- Code regions based (OpenACC)

```c
#pragma hmpp myfunc codelet, ...
void saxpy(int n, float alpha, float x[n], float y[n]){
    #pragma hmppcg gridify(i)
    for(int i = 0; i<n; ++i)
        y[i] = alpha*x[i] + y[i];
}
```

```c
#pragma acc kernels ...
{
    for(int i = 0; i<n; ++i)
        y[i] = alpha*x[i] + y[i];
}
```
HMPP/OpenACC Compiler flow (Src to Src Cpler)

$ hmpp gcc filetocompile.c -o mybinary.exe

HMPP Compiler Driver

C++ Frontend

C Frontend

Fortran Frontend

Instrumentation and extraction module

Instrumented Code

CPU Compiler (e.g. gcc, ifort, ...)

Executable

OpenCL/CUDA/... Code Generation

readable Cuda/OpenCL code

HWA Compiler

HWA Code DLib

HMPP/OpenACC Runtime Library
Code Generation based on "Loop Nest Gridification"

- The loop nest gridification process converts parallel loop nests in a grid of GPU/HWA threads
  - Use the parallel loop nest iteration space to produce the threads

```c
#pragma hmppcg gridify(i)
for(int i = 0; i<10; ++i){
    y[i] = alpha*x[i] + y[i];
}
```

```c
int i = blockIdx.x* blockDim.x + threadIdx.x;
if( i<10 )
    y[i]=alpha*x[i]+y[i];
```
Remote Procedure Call on an Accelerator

• Running on the accelerator is implemented using some form of RPC (Remote Procedure Call)

```c
#pragma hmpp f1 codelet
myfunc(...){
    ...
    for()
        for()
            ...
        ...
}
```

```c
main(){
    ...
    #pragma hmpp f1 callsite
    myfunc(V1[k],V2[k]);
    ...
}
```
A RPC sequence consists in 5 basic steps:

1. Allocate the GPU and the memory
2. Transfer the input data: CPU => GPU
3. Compute
4. Transfer the output data: CPU <= GPU
5. Release the GPU and the memory
RPC Example (Codelet Style)

```c
#pragma hmpp sobelfilter advancedload, data["original_image"]...
for (i=0; i<NB_RUNS; i++){
    #pragma hmpp sobelfilter callsite
    sobelFilter( original_image, edge_image, size, size);
}
#pragma hmpp sobelfilter delegatedstore, data["edge_image"]
```

Execution on the GPU

Upload GPU data

Download GPU data
OpenACC Initiative

• Express data and computations to be executed on an accelerator
  o Using marked code regions

• Main OpenACC constructs
  o Parallel and kernel regions
  o Parallel loops
  o Data regions
  o Runtime API

• OpenACC support released in April 2012 (HMPP Workbench 3.1)
  o OpenACC Test Suite provided by University of Houston

• Visit http://www.openacc-standard.com for more information
OpenACC Data Management

- Mirroring duplicates a CPU memory block into the HWA memory
  - Mirror identifier is a CPU memory block address
  - Only one mirror per CPU block
  - Users ensure consistency of copies via directives

![Diagram of CPU Memory and HWA Memory with HMPP RT Descriptor](image-url)
OpenACC Execution Model

- Host-controlled execution
- Based on three parallelism levels
  - Gangs – coarse grain
  - Workers – fine grain
  - Vectors – finest grain
Parallel Loops

- The loop directive describes iteration space partitioning to execute the loop; declares loop-private variables and arrays, and reduction operations
- Clauses
  - gang [(scalar-integer-expression)]
  - worker [(scalar-integer-expression)]
  - vector [(scalar-integer-expression)]
  - collapse(n)
  - seq
  - independent
  - private(list)
  - reduction(operator:list )

```c
#pragma acc loop gang(NB)
for (int i = 0; i < n; ++i){
    #pragma acc loop worker(NT)
    for (int j = 0; j < m; ++j){
        B[i][j] = i * j * A[i][j];
    }
}
```
Code Generation

#pragma acc loop gang(NB)
for (int i = 0; i < n; ++i){
    #pragma acc loop worker(NT)
    for (int j = 0; j < m; ++j){
        B[i][j] = i * j * A[i][j];
    }
}

Virtual machine

Gang workers

Gang workers

Gang workers

Gang workers

compiler dep.

Virtual to physical machine mapping

Iteration spaces distribution

Stream Multi-Processor

GPU thread

GPU thread

GPU thread

Physical machine
Kernel Regions

- Parallel loops inside a region are transformed into accelerator kernels (e.g. CUDA kernels)
  - Each loop nest can have different values for gang and worker numbers

- Clauses
  - if(condition)
  - async[(scalar-integer-expression)]
  - copy(list)
  - copyin(list)
  - copyout(list)
  - create(list)
  - present(list)
  - present_or_copy(list)
  - present_or_copyin(list)
  - present_or_copyout(list)
  - present_or_create(list)
  - deviceptr(list)

```c
#pragma acc kernels
{
    #pragma acc loop independent
    for (int i = 0; i < n; ++i){
        for (int j = 0; j < n; ++j){
            for (int k = 0; k < n; ++k){
                B[i][j*k%n] = A[i][j*k%n];
            }
        }
    }
}

#pragma acc loop gang(NB)
for (int i = 0; i < n; ++i){
    #pragma acc loop worker(NT)
    for (int j = 0; j < m; ++j){
        B[i][j] = i * j * A[i][j];
    }
}
```
Iterations Mapping

#pragma acc loop gang(2)
for (i=0; i<n; i++){
    #pragma acc loop worker(2)
    for (j=0;j<m;j++){
        iter(j,j);
    }
}

for (i=0; i<n/2; i++){
    for (j=0;j<m;j++){
        iter(i,j);
    }
}

for (i=n/2+1; i<n; i++){
    for (j=0;j<m;j++){
        iter(i,j);
    }
}

for (i=0; i<n/2; i++){
    for (j=0;j<m;j+=2){
        iter(i,j);
    }
}

for (i=0; i<n/2; i++){
    for (j=1;j<m;j+=2){
        iter(i,j);
    }
}

Gang 0

Gang 0, Worker 0

Gang 1

Gang 0, Worker 0

Gang 0, Worker 1

Distribution scheme is compiler dependant (here simplified scheme)
HydroC OpenACC Migration Step

Hotspots
- Understand your performance goal (analysis, definition and achievement)
- Know your hotspots (analysis, code reorganization, hotspot selection)
- Establish a validation process
- Set a continuous integration process with the validation

Parallelization
- Optimize CPU code
- Exhibit application SIMT parallelism
- Parallelize for Many-core
- Validate execution

Define your Parallel Project

Port your Application on Many-core

Optimize Your Many-core Application

Tuning
- Reduce data transfers
- Optimize kernel execution
- Provide feedback to application programmers for improving algorithm data structures/
- Consider multiple devices

Hours to Days

Days to Weeks

Weeks to Months

Phase 1
Many-core operational application with known potential

Phase 2
A corporate project
- Purchasing Department
- Scientists
- IT Department
Porting the Riemann Hotspot

```c
void riemann ()
{
    #pragma acc kernels
    copy( qleft[0:Hnvar*Hstep*Hnxyt], \
         qright[0:Hnvar*Hstep*Hnxyt], \
         qgdnv[0:Hnvar*Hstep*Hnxyt], \
         sgnm[0:Hstep*Hnxyt] )
{
    #pragma acc loop independent
    for (int s = 0; s < slices; s++){
        for (int i = 0; i < narray; i++){
            ...
        }
    }
    ...

    #pragma acc kernels
    copy( qleft[0:Hnvar*Hstep*Hnxyt], \
         qright[0:Hnvar*Hstep*Hnxyt], \
         sgnm[0:Hstep*Hnxyt], \
         qgdnv[0:Hnvar*Hstep*Hnxyt] )
{
    #pragma acc loop independent
    for (int invar = IP + 1; invar < Hnvar; invar++){
        for (int s = 0; s < slices; s++){
            ...
        }
    }
    }
}

Allocate and copy data from host to device and device to host and deallocate at the end of the block.

1D gridification

Copy data from device to host and deallocate

Potential Speedup
\( S_p = \frac{1}{1 - 0.4824} = 1.93 \)
Porting the Riemann Hotspot

- CAPS OpenACC compiler feedback

  ...  
  hmppcg: [Message DPL3000] riemann.c:86: Loop 's' was gridified (1D)
  ...
  ...
  hmppcg: [Message DPL3000] riemann.c:266: Loop 'invar' was gridified (1D)
  ...

- Indicating successful parallel code generation
Porting the Riemann Hotspot

- Code execution feedback

[ 0.275872] (0) INFO : Enter kernels (queue=none, location=riemann.c:79)
[ 0.275886] (0) INFO : Allocate qleft[0:1008000] (element_size=8, host_address=0x7f5239905010, memory_space=cudaglob, queue=none, location=riemann.c:79)
[ 0.275962] (0) INFO : Upload qleft[0:1008000] (element_size=8, host_address=0x7f5239905010, queue=none, location=riemann.c:79)
[ 0.277318] (0) INFO : Allocate qright[0:1008000] (element_size=8, host_address=0x7f523914c010, memory_space=cudaglob, queue=none, location=riemann.c:79)
[ 0.277465] (0) INFO : Upload qright[0:1008000] (element_size=8, host_address=0x7f523914c010, queue=none, location=riemann.c:79)
[ 0.278861] (0) INFO : Allocate qgdnv[0:1008000] (element_size=8, host_address=0x7f5238993010, memory_space=cudaglob, queue=none, location=riemann.c:79)
[ 0.278997] (0) INFO : Upload qgdnv[0:1008000] (element_size=8, host_address=0x7f5238993010, queue=none, location=riemann.c:79)
[ 0.280249] (0) INFO : Allocate sgnm[0:252000] (element_size=4, host_address=0x197bd90, memory_space=cudaglob, queue=none, location=riemann.c:79)
[ 0.280324] (0) INFO : Upload sgnm[0:252000] (element_size=4, host_address=0x197bd90, queue=none, location=riemann.c:79)
...[ 0.281202] (0) INFO : Call __hmpp_acc_region__54seznto (queue=none, location=riemann.c:79)
...[ 0.285177] (0) INFO : Download sgnm[0:252000] (element_size=4, host_address=0x197bd90, queue=none, location=riemann.c:79)
[ 0.285775] (0) INFO : Free sgnm[0:252000] (element_size=4, host_address=0x197bd90, queue=none, location=riemann.c:79)
[ 0.285853] (0) INFO : Download qgdnv[0:1008000] (element_size=8, host_address=0x7f5238993010, queue=none, location=riemann.c:79)
[ 0.289225] (0) INFO : Free qgdnv[0:1008000] (element_size=8, host_address=0x7f5238993010, queue=none, location=riemann.c:79)
[ 0.289390] (0) INFO : Download qright[0:1008000] (element_size=8, host_address=0x7f523914c010, queue=none, location=riemann.c:79)
[ 0.290886] (0) INFO : Free qright[0:1008000] (element_size=8, host_address=0x7f523914c010, queue=none, location=riemann.c:79)
[ 0.291045] (0) INFO : Download qleft[0:1008000] (element_size=8, host_address=0x7f5239905010, queue=none, location=riemann.c:79)
[ 0.292503] (0) INFO : Free qleft[0:1008000] (element_size=8, host_address=0x7f5239905010, queue=none, location=riemann.c:79)
[ 0.292659] (0) INFO : Leave kernels (queue=none, location=riemann.c:79)
Parallel Regions

- Start parallel activity on the accelerator device
  - Gangs of workers are created to execute the accelerator parallel region
  - Can share HWA resources across loops nests → one region = one CUDA kernel
  - SPMD style code without barrier

- Clauses
  - if(condition)
  - async[(scalar-integer-expression)]
  - num_gangs(scalar-integer-expression)
  - num_workers(scalar-integer-expression)
  - vector_length(scalar-integer-expression)
  - reduction(operator:list)
  - copy(list)
  - copyin(list)
  - copyout(list)
  - create(list)
  - present(list)
  - present_or_copy(list)
  - present_or_copyin(list)
  - present_or_copyout(list)
  - present_or_create(list)
  - deviceptr(list)
  - private(list)
  - firstprivate(list)

```c
#pragma acc parallel num_gangs(BG),
   num_workers(BW)
{
  #pragma acc loop gang
  for (int i = 0; i < n; ++i){
    #pragma acc loop worker
    for (int j = 0; j < n; ++j){
      B[i][j] = A[i][j];
    }
  }
}
for(int k=0; k < n; k++){
  #pragma acc loop gang
  for (int i = 0; i < n; ++i){
    #pragma acc loop worker
    for (int j = 0; j < n; ++j){
      C[k][i][j] = B[k-1][i+1][j] + ...;
    }
  }
}
```
#pragma acc parallel num_gangs(2)
{
  #pragma acc loop gang
  for (i=0; i<n; i++){
    for (j=0;j<m;j++){
      iter1(j,j);
    }
  }
}

#pragma acc loop gang
for (i=0; i<n/2; i++){
  for (j=0;j=m;j++){
    iter1(i,j);
  }
}

#pragma acc parallel num_gangs(2)
{
  #pragma acc loop gang
  for (i=0; i<n; i++){
    iter2(i);
  }
}

for (i=n/2+1; i<n; i++){
  for (j=0;j<m;j++){
    iter1(i,j);
  }
}

for (i=0; i<n/2; i++){
  for (j=0;j=m;j++){
    iter1(i,j);
  }
}

for (i=0; i<n; i++){
  iter2(i);
}

for (i=n/2+1; i<n; i++){
  iter2(i);
}
Data Management Directives

- Data regions define scalars, arrays and sub-arrays to be allocated in the device memory for the duration of the region
  - Explicit management of data transfers using clauses or directives
- Many clauses
  - if(condition)
  - copy(list)
  - copyin(list)
  - copyout(list)
  - create(list)
  - present(list)
  - present_or_copy(list)
  - present_or_copyin(list)
  - present_or_copyout(list)
  - present_or_create(list)
  - deviceptr(list)

```c
#pragma acc data copyin(A[1:N-2]),
    copyout(B[N])
{
    #pragma acc kernels
    {
        #pragma acc loop independant
        for (int i = 0; i < N; ++i){
            A[i][0] = ...;
            A[i][M - 1] = 0.0f;
        }
        ...
    }
    #pragma acc update host(A)
    ...
    #pragma acc kernels
    for (int i = 0; i < n; ++i){
        B[i] = ...;
    }
}
```
Runtime API

- Set of functions for managing device allocation (C version)

  - `int acc_get_num_devices(acc_device_t)`
  - `void acc_set_device_type(acc_device_t)`
  - `acc_device_t acc_get_device_type(void)`
  - `void acc_set_device_num(int, acc_device_t)`
  - `int acc_get_device_num(acc_device_t)`
  - `int acc_async_test(acc_device_t)`
  - `void acc_async_test_all()`
  - `void acc_async_wait(int)`
  - `void acc_async_wait_all()`
  - `void acc_init(acc_device_t)`
  - `void acc_shutdown(acc_device_t)`
  - `void* acc_malloc(size_t)`
  - `void acc_free(void*)`
  - `...`
HydroC Migration Step

**Define your Parallel Project**
- Understand your performance goal (analysis, definition and achievement)
- Know your hotspots (analysis, code reorganization, hotspot selection)
- Establish a validation process
- Set a continuous integration process with the validation

**Port your Application on Many-core**
- Optimize CPU code
  - Exhibit application SIMT parallelism
  - Parallelize for Many-core
  - Validate execution

**Optimize Your Many-core Application**
- Reduce data transfers
- Optimize kernel execution
- Provide feedback to application programmers for improving algorithm data structures/...
- Consider multiple devices

**Hotspots**
- Hours to Days

**Parallelization**
- Days to Weeks

**Tuning**
- Weeks to Months

**Phase 1**
- Many-core operational application with known potential

**Phase 2**

A corporate project
- Purchasing Department
- Scientists
- IT Department

www.caps-entreprise.com
void riemann ()
{
    #pragma acc kernels
    pcopyin(qleft[0:Hnvar*Hstep*Hnxyt], qright[0:Hnvar*Hstep*Hnxyt]) \ 
    pcopyout(qgdnv[0:Hnvar*Hstep*Hnxyt], sgnm[0:Hstep*Hnxyt]) \ 
    pcopyin( Hnvar,Hstep,Hnxyt,smallp_, smallpp_, gamma6_,slices, 
              narray, Hsmallr, Hsmallc, Hniter_riemann, Hgamma)
{
    #pragma acc loop independent
    for (int s = 0; s < slices; s++){
        
        for (int i = 0; i < narray; i++){
            ...
        }
    }\ 
    ...\ 
    #pragma acc kernels
    pcopyin(qleft[0:Hnvar*Hstep*Hnxyt], qright[0:Hnvar*Hstep*Hnxyt],\ 
           sgnm[0:Hstep*Hnxyt]), \ 
    pcopyout(qgdnv[0:Hnvar*Hstep*Hnxyt]) \ 
    pcopyin(Hnvar, Hstep, Hnxyt, slices, narray)
{
    #pragma acc loop independent
    for (int invar = IP + 1; invar < Hnvar; invar++){
        
        for (int s = 0; s < slices; s++){
            ...
        }
    }
    ...
    }
}
Riemann Kernel Optimizations

```c
void riemann(){
    #pragma acc kernels
    pcopyin(qleft[0:Hnvar*Hstep*Hnxyt], qright[0:Hnvar*Hstep*Hnxyt]) \n    pcopyout(qgdnv[0:Hnvar*Hstep*Hnxyt], sgnm[0:Hstep*Hnxyt]) \n    pcopyin( Hnvar,Hstep,Hnxyt,smallp_, smallpp_, gamma6_,slices, 
            narray, Hsmallr, Hsmallc, Hniter_riemann, Hgamma)
    {
        #pragma acc loop independent
        for (int s = 0; s < slices; s++){
            #pragma acc loop independent
            for (int i = 0; i < narray; i++){
                ...
            }
        }
    }
    ...
    #pragma acc kernels
    pcopyin(qleft[0:Hnvar*Hstep*Hnxyt], qright[0:Hnvar*Hstep*Hnxyt],
            sgnm[0:Hstep*Hnxyt]) \ 
    pcopyout(qgdnv[0:Hnvar*Hstep*Hnxyt]) \ 
    pcopyin(Hnvar, Hstep, Hnxyt, slices, narray)
    {
        #pragma acc loop independent
        for (int invar = IP + 1; invar < Hnvar; invar++){
            #pragma acc loop independent
            for (int s = 0; s < slices; s++){
                ...
            }
        }
    }
    ...
}
```
Riemann Kernel Optimizations

- CAPS OpenACC compiler feedback

hmppcg: [Message DPL3001] riemann.c:87: Loops 'i' and 's' were gridified \((2D)\)

hmppcg: [Message DPL3001] riemann.c:267: Loops 's' and 'invar' were gridified \((2D)\)

- Indicating effective parallelization
Porting the Riemann Hotspot

- Total execution time (CPU: i5-2400, 1 GPU: GTX 465)
  - Original CPU application: 10.3 sec. (serial)
  - Naive OpenACC porting: 11.1 sec.
  - Optimized transfers: 9.8 sec. ($S_p = 1.05$)
  - Optimized kernels: 7.9 sec. ($S_p = 1.31$ over a potential of 1.9)
HydroC Migration Step

Define your Parallel Project
- Understand your performance goal (analysis, definition and achievement)
- Know your hotspots (analysis, code reorganization, hotspot selection)
- Establish a validation process
- Set a continuous integration process with the validation

Port your Application on Many-core
- Optimize CPU code
  - Exhibit application SIMD parallelism
  - Parallelize for Many-core
  - Validate execution

Optimize Your Many-core Application
- Reduce data transfers
- Optimize kernel execution
- Provide feedback to application programmers for improving algorithm data structures...
- Consider multiple devices

A corporate project
- Purchasing Department
- Scientists
- IT Department

Phase 1
- Many-core operational application with known potential

Phase 2
- Hours to Days
- Days to Weeks
- Weeks to Months
Further Optimizations

Reuse of data on GPU across multiple compute regions with a data region

hydro_godunov (97.0%)

main (100%)

compute_deltat (2.9%)

riemann (48.2%)
slope (11.2%)
trace (10.1%)

updateConservativeVars (5.6%)

gatherConservativeVars (5.4%)
equationofstate (5.0%)

qleftright (4.9%)
constoprim (4.6%)
cmpflx (3.0%)
void hydro_godunov (...)
{
#pragma acc data \
create(qleft[0:H.nvar], qright[0:H.nvar], \
    q[0:H.nvar], qgdnv[0:H.nvar], \
    flux[0:H.nvar], u[0:H.nvar], \
    dq[0:H.nvar], e[0:Hstep], c[0:Hstep], \
    sgnm[0:Hstep], qxm[0:H.nvar], qxp[0:H.nvar]) \
copy(uold[0:H.nvar*H.nxt*H.nyt]) \
copyin(Hstep)
{
    for (j = Hmin; j < Hmax; j += Hstep){
        // compute many slices each pass
        int jend = j + Hstep;
        if (jend >= Hmax)
            jend = Hmax;
        . . .// the work here
    } // for j
}//end of data region

Data are left on the GPU during the step loop. pcopy clauses are used into called routines
Full Application

- With the same strategy, the full application have been ported with OpenACC

- The following hotspots have been accelerated
  - cmplx
  - updateConservativeVar
  - gatherConservativeVar
  - constoprim
  - equationofstate
  - qleftright
  - riemann
  - slope
  - trace

Achieved speedup = 3x and still room for improvement

1 week of development

60 directives, 4% of the LoC
A Few Rules When Using OpenACC

- Use kernels regions as a default
  - Parallel region semantic less trivial
- Add the copy-in and copy-out data clauses
  - Automatic variables scoping is compiler dependant
- Use gangs and workers before using vectors
  - Meanings is more uniform across compilers
  - Vector is more a hints, gang is not (especially in parallel section)
  - Use them in the order gang/worker/vector with one level of each
- Check different values of gangs and workers for performance
  - Similar to CUDA thread block tuning
- Avoid calls to routine in kernels/regions that are not in the same file
  - Inter-procedural inlining may not always work
  - This is ongoing discussion in the consortium

www.caps-entreprise.com
void convSM_N(typeToUse A[M][N], typeToUse B[M][N]){
    int i, j, k; int m=M, n=N;
    #pragma acc kernels pcopyin(A[0:m]) pcopy(B[0:m])
    {
        typeToUse c11, c12, c13, c21, c22, c23, c31, c32, c33;

        c11 = +2.0f;  c21 = +5.0f;  c31 = -8.0f;
        c12 = -3.0f;  c22 = +6.0f;  c32 = -9.0f;
        c13 = +4.0f;  c23 = +7.0f;  c33 = +10.0f;

        #pragma acc loop gang(16)
        for (int i = 1; i < M - 1; ++i) {
            #pragma acc loop worker(16)
            for (int j = 1; j < N - 1; ++j) {
                  +c21*A[i-1][j+0]+c22*A[i+0][j+0]+c23*A[i+1][j+0]
                  +c31*A[i-1][j+1]+c32*A[i+0][j+1]+c33*A[i+1][j+1];
            }
        }
    }
}//kernels region
DNA Distance Application with OpenACC

• Biomedical application part of Phylip package,
  o Main computation kernel takes as input a list of DNA sequences for each species
    • Code is based on an approximation using Newton-Raphson method (SP)
    • Produces a 2-dimension matrix of distances
  o Experiments performed in the context of the HMPP APAC CoC*

• Performance
  o OpenMP version, 4 & 8 threads, Intel(R) i7 CPU 920 @ 2.67GHz
  o 1 GPU Tesla C2070

*http://competencecenter.hmpp.org/category/hmpp-coc-asia/
Sobel Filter Performance Example

- Sobel Filter benchmark
- Size
  - ~ 200 lines of C code
- GPU C2070 improvement
  - x 24 over serial code on Intel i7 CPU 920 @ 2.67GHz
- Main porting operation
  - Inserting 6 OpenACC directives
Going Further with OpenHMPP Compiler
What is in OpenHMPP and not in OpenACC

- Library integration directives
  - Needed for a “single source many-core code” approach

- Tracing Open performance APIs
  - Allows to use tracing third party tools

- Tuning directives
  - Loops nest transformations

- External and native functions
  - Allows to include CUDA code into kernels

- Multiple devices management
  - Data collection / map operation

- And many more features
  - Loop transformations directives for kernel tuning
  - buffer mode, UVA support, …
Library Integration
Dealing with Libraries

- Library calls can usually only be partially replaced
  - No one-to-one mapping between libraries (e.g. BLAS, FFTW, CuFFT, CULA, ArrayFire)
  - No access to all application codes (i.e. avoid side effects)
  - **Want a unique source code**

- Deal with multiple address spaces / multi-HWA
  - Data location may not be unique (copies, mirrors)
  - Usual library calls assume shared memory
  - Library efficiency depends on updated data location (long term effect)

- Libraries can be written in many different languages
  - CUDA, OpenCL, OpenHMPP, etc.

- OpenACC uses device pointers (deviceptr clause)
  - Accelerator specific, not a portable technique
Library Mapping Example

**FFT Warcraft**

```c
fftw_plan fftwf_plan_dft_r2c_3d(
  sz, sy, sx,
  work1, work2,
  FFTW_ESTIMATE);

fftwf_execute(p);

fftwf_destroy_plan(p);
```

**NVIDIA cuFFT**

```c
cufftHandle plan;
cufftPlan3d(&plan, sz, sy, sx, CUFFT_R2C);
cufftExecR2C(plan, (cufftReal*) work1,
  (cufftComplex *) work2);
cufftDestroy(plan);
```
Proxy Directives "hmppalt" in HMPP3.0

• A proxy indicated by a directive is in charge of calling the accelerated library
• Proxies get the execution context from the HMPP runtime
• Proxies are used only to selected calls to the library

```c
CALL INIT(A,N)
CALL ZFFT1D(A,N,0,B)  ! This call is needed to initialize FFTE
CALL DUMP(A,N)

!$hmppalt ffte call , name="zfft1d", error="proxy_err"
CALL ZFFT1D(A,N,-1,B)
CALL DUMP(A,N)

C
C SAME HERE
!$hmppalt ffte call , name="zfft1d", error="proxy_err"
CALL ZFFT1D(A,N,1,B)
CALL DUMP(A,N)
```

Replaces the call to a proxy that handles GPUs and allows to mix user GPU code with library ones.
Library Interoperability in HMPP 3.0

HMPP Runtime API

Native GPU Runtime API

GPU Lib
- ...
- ...
- gpuLib(…)
- ...

CPU Lib
- ...
- ...
- cpuLib1(…)
- ...
- cpuLib3(…)
- ...

C/CUDA/…

... call libRoutine1(…)
... ...
... #pragma hmppalt
... call libRoutine2(…)
... ...
... ...
... call libRoutine3(…)
... ...
Library Integration with HMPPALT – Example

• The proxy set **blascuda** to translate BLAS calls into cuBLAS calls is composed of
  o A Fortran include file **blascuda.inc**
  o A shared library **libblascuda.so**
  o Some documentation
• A typical BLAS code

```fortran
SUBROUTINE doit(n,X,Y)
    REAL :: X(n), Y(n)
    REAL prod
    CALL sscal(n, 2.0, 1, X)
    prod = sdot(n, 1, X, 1, Y)
END SUBROUTINE

SUBROUTINE doit(n,X,Y)
    INCLUDE “blascuda.inc”
    REAL :: X(n), Y(n)
    REAL prod
    !$HMPPALT blascuda call, name=”sscal”
    CALL sscal(n, 2.0, 1, X)
    !$HMPPALT blascuda call, name=”sdot”
    prod = sdot(n, 1, X, 1, Y)
END SUBROUTINE
```

• Must be compiled with HMPP and linked with **libmyblas.so**
  o ifort -O3 -shared -o libmyblas.so myblas.f90 -lcublas
  o hmpp ifort myApplication.f90 -o myApplication.exe -lmyblas
Different Proxies Settings

• Library level CPU-GPU data transfer management
  o Data updates are performed inside the proxies
  o Can be redundant with accelerated users code

• Users level CPU-GPU data transfer management
  o No data updates are performed inside the proxies
  o Allow to optimize the data transfers between users and library kernels
Profiling with Paraver
Profiling with Paraver – Overview

- Paraver is a trace visualization tool developed by the Barcelona Supercomputing Center

- HMPP 3.x supports application profiling through additional plugins
Profiling with Paraver – Filters

- Paraver is able to apply filters on the trace files to retrieve specific information

- Dedicated filters ease GPGPU porting for HMPP applications

<table>
<thead>
<tr>
<th>Global View</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intermediate View</td>
</tr>
<tr>
<td>Codelet Execution</td>
</tr>
<tr>
<td>Detailed View</td>
</tr>
<tr>
<td>Codelet Execution</td>
</tr>
</tbody>
</table>
Tracing Interface with Vampir
OpenHMPP Tracing Interface

• API specifying callbacks to trace OpenHMPP directives events

• Examples of event query functions
  o phmpp_event_kind_t phmpp_get_event_kind(phmpp_event_t *event, ...);
  o phmpp_event_state_t phmpp_get_event_state(phmpp_event_t *event,...);
  o unsigned int phmpp_get_event_instance_id(phmpp_event_t *event, ...);
  o unsigned int phmpp_get_event_semantic_id(phmpp_event_t *event,...);
  o phmpp_code_t * phmpp_get_event_code(phmpp_event_t * event);
  o phmpp_hardware_t *phmpp_get_event_hardware(phmpp_event_t * event);
  o ...

static void phmpp_init(void){
  phmpp_register_prologue_callback(init);
  phmpp_register_event_callback(PHMPP_EK_CODELET_REQUEST, codelet_event);
  phmpp_register_event_callback(PHMPP_EK_TRANSFER_REQUEST, transfer_event);
  phmpp_register_epilogue_callback(finish);
}
Tuning Directives
Code Tuning Directives

- Directive-based HWA kernel code transformations
- Directives preserve original CPU code

```c
#include <mygroup>

#pragma hmpp <mygroup> sgemm codelet, args[tout].io=inout, &
#pragma hmpp & args[*].mirror args[*].transfer=manual

void sgemm(float alphav[1], float betav[1], const float t1[SIZE][SIZE],
const float t2[SIZE][SIZE], float tout[SIZE][SIZE]) {

    int j, i;
    const float alpha = alphav[0], beta = betav[0];

    #pragma hmppcg(CUDA) unroll i:4, j:4, split(i), noremainder(i,j), jam
    #pragma hmppcg gridify (j,i)
    for( j = 0 ; j < SIZE ; j++ ) {
        for( i = 0 ; i < SIZE ; i++ ) {
            int k;
            float prod = 0.0f;
            for( k = 0 ; k < SIZE ; k++ ){
                prod += t1[k][i] * t2[j][k];
            }
            tout[j][i] = alpha * prod + beta * tout[j][i];
        }
    }
}
```

Loop transformations

Apply only when compiling for CUDA
Tuning Example – 1*

- **HMPP-transformed PolyBench codes using CUDA and OpenCL:**
  - Given in terms of speedup over default (non-transformed) HMPP code
  - Compared with results of manually-written CUDA/OpenCL implementation
  - HMPP transformations give speedup over default in 8 of the 14 transformed codes using CUDA and 6 of the 14 codes using OpenCL

- **CUDA Results:**

![Graph showing speedup over default HMPP CUDA configuration](image)

*From "Autotuning a High-Level Language Targeted to GPU Kernels", S. Grauer-Gray, R. Searles, L. Xu, S. Ayalasomayajula, J. Cavazos Supercomputing 2011, University of Delaware*
Tuning Example – 2*

*From "Autotuning a High-Level Language Targeted to GPU Kernels", S. Grauer-Gray, R. Searles, L. Xu, S. Ayalasomayajula, J. Cavazos Supercomputing 2011, University of Delaware
Codelet External Functions
External Functions Directives

**sum.h**

```c
#ifndef SUM_H
#define SUM_H

float sum( float x, float y );

#endif /* SUM_H */
```

**sum.c**

```c
#include "sum.h"

#pragma hmp function=hmppcg_functions.dpil

function,target=CUDA
float sum( float x, float y ) {
    return x+y;
}
```

**extern.c**

```c
#include "sum.h"

int main(int argc, char **argv) {
    int i, N = 64;
    float A[N], B[N];
    ...
    #pragma hmp cdl region, args[B].io=inout,
    target=CUDA
    {
        for( int i = 0 ; i < N ; i++ )
        B[i] = sum( A[i], B[i] );
    }
    ...
}
```

Declare ‘sum’ as a function to be called in a codelet

‘sum’ is an external function

$ hmpp --function=hmppcg Functions.dpil$ gcc source_files
Integrating CUDA Code in Codelets
Native Cuda/OpenCL Functions Support

- Support of handwritten CUDA and OpenCL codes
  - Enables manual optimizations for advanced users

```c
float f1(float x, float y, float alpha, float pow)
{
    float res;
    res = alpha * cosf(x);
    res += powf(y, pow);
    return sqrtf(res);
}
```

```c
#pragma hmpp mycdlt codelet, ...
void myfunc(int size, float alpha, float pow, float *v1, float *v2, float *res)
{
    int i;
    float t[size], temp;
    #pragma hmppcg include (<native.cu>)
    #pragma hmppcg native (f1)
    for(i = 0; i < size; i++)
        res[i] = f1(v1[i],v2[i], alpha, pow);
}
```

- **Native function in CUDA**

  ```c
  __device__ float f1(float x, float y, float alpha, float pow)
  {
    float res;
    res = alpha * __cosf(x);
    res += __powf(y, pow);
    return __fsqrt_rn(res);
  }
  ```

- **Include file containing native function definition**

  - Use different GPU sqrt

- ‘f1’ is a native function
Multi-Accelerator Programming with OpenHMPP
Most HMPP directives are extended with a device clause

```
#pragma hmpp <MyLabel> MyCodelet command, ..., device="expr"
```

The device clause requires an index within the allocated devices (0,1,...)

All arguments in a multi-device group must use the Mirror Storage Policy, each mirror is allocated on a single device

```
#pragma hmpp <MyLabel> allocate, data[x] , device="..."
```

All mirrored arguments of a callsite must be allocated on the same device
- The callsite is automatically performed on that device thanks to the "owner computes rule"
Collections of Data in OpenHMPP 3.x

```
struct {
    float * data;
    int size;
} Vec;
```
OpenHMPP 3.0 Map Operation on Data Collections

```c
#pragma hmpp <mgrp> parallel
for(k=0;k<n;k++) {
    #pragma hmpp <mgrp> f1 callsite
    myparallelfunc(d[k],n);
}
```
Debugging with Allinea DDT
Debugging with Allinea DDT (1)

• Debugging tool for parallel computing developed by Allinea

• Supports
  o MPI
  o OpenMP
  o CUDA
  o OpenACC, OpenHMPP CUDA applications

• Features
  o Host debugging
  o Device debugging
  o CAPS error handler
Debugging with Allinea DDT (2)

Codelet

Breakpoint

Display array on device
Conclusion

• Directive-based approaches are currently one of the most promising track for heterogeneous many-cores
  o Preserve code assets
  o At node level help separating parallelism description from implementation

• Need to integrate libraries and user codes
  o Requires interoperability between runtimes

• Auto-tuning is one of the future keys to efficient portability
  o Has to be part of the programming API
GTC Special Offering

• CAPS OpenACC Compiler for $199
  o OpenACC 1.0 directives
  o C and Fortran to CUDA - for Linux
  o One-year maintenance and support
  o 1 perpetual license
  o Ends June 15th

• See us on Booth #28 or on Exxact booth #36