Heterogeneous Computing with OpenACC

Alexei Strelchenko

CaSToRC @ The Cyprus Institute

LinkSceem User Meeting, June 27th, 2012

Outline

- Challenges of heterogeneous programming
- What is OpenACC
- First example
- OpenACC Programming Model
- Tips and hints
- Summary

Challenges of heterogeneous programming

- Levels of parallelism:
  - multiple device with separate (physical) address spaces
  - multiple threads inside a device
  - vector parallelism inside the thread
- Types of parallel programming models:
  - Message Passing (distributed memory systems)
  - Task parallel (shared memory systems, thread based)
  - Data parallel, streaming processing
- Exploiting heterogeneous environment:
  - GPGPU APIs and programming languages (OpenCL, CUDA)
  - Libraries (CUBLAS, CUSPARSE, MAGMA etc)
  - Compiler directives (PGI Accelerator Directives, OpenACC)

Programming with OpenACC directives

- Developed by NVIDIA, PGI, Cray and CASP
- Provides:
  - Compiler directives to specify parallel regions (in C/Fortran)
  - Implicit accelerator initialization
  - Implicit data transfers between host and accelerator
  - Interoperability with CUDA C/Fortran and GPU libs
- Almost as simple as OpenMP

OpenACC: how it looks like

- OpenMP:

```
int main() {
  double pi = 0.0;
#pragma omp parallel for reduction(+:pi)
  for(long i=0; i<N; i++){
    double t = (double)((i+0.5)/N);
    pi += 4.0/(1.0+t*t);
  }
}
```

- OpenACC code

```
int main() {
  double pi = 0.0;
#pragma acc kernels
  for(long i=0; i<N; i++){
    double t = (double)((i+0.5)/N);
    pi += 4.0/(1.0+t*t);
  }
}
```

OpenACC directive syntax

- C:

#pragma acc *directive [clause [,] clause] ...]*

parallel code block

- Fortran:

!$acc *directive [clause [,] clause] ...]*

parallel code block

!$acc end *directive*

OpenACC main constructs

- Parallel and kernel regions
- Parallel loops
- Data regions
- Runtime API

Kernels constructs

- Each loop compiled into a separate GPU kernel
- C:

```
#pragma acc kernels
{
  for(int i=0; i<N; i++){
  ....
  }//end of loop
  for(int j=0; j<N; j++){
  ....
  }//end of loop
}
```

- Fortran:

```
!$acc kernels
do i=0,i<N
   ....
end do
do j=0,j<N
   ....
end do
!$acc end kernels
```

Kernels construct syntax

- C:

```
#pragma acc kernels [clause ...]
{
  structured block
}
```

- Fortran:

```
!$acc kernels [clause ...]
  structured block
!$acc end kernels
```

- Clauses:
  - if (*condition*)
  - async (*expression*)
  - data management clauses

Loop construct syntax

- C:

```
#pragma acc loop [clause ...]
{
  for (...)
    ...
}
```

- Fortran:

```
!$acc loop [clause ...]
  do ...
    ...
  end do
!$acc end kernels
```

- Clauses:
  - if (*condition*)
  - async (*expression*)
  - data management clauses, execution clauses

SXPY example

- C:

```
#include <openacc.h>
void sxpy (float *x, float *y, int n){
#pragma acc parallel loop independent
   for (int i < 0; i < n; i++)
     y[i] += x[i];
}
```

- Fortran:

```
subroutine sxpy (x, y, n)
integer :: n, i
real :: x(:), y(:)
!$acc parallel loop
   do i = 1, n
     y(i) = x(i) + y(i)
   end do
!$acc end parallel
end subroutine
```

- *independent* clause used in loop directive:

limits the effect of pointer aliasing (for C)

SXPY example

- The restrict keyword

```
#include <openacc.h>
void sxpy (float *x, float *restrict y, int n){
#pragma acc kernels
   for (int i < 0; i < n; i++)
     y[i] += x[i];
}
```

- float *restrict ptr :

alternative to *independent* clause: no aliasing assumed for ptr

Code compilation with PGI

- C:

  pgcc -acc [-Minfo=accel] -ta=nvidia, host -o sxpy sxpy.c

- Fortran:

  pgf90 -acc [-Minfo=accel] -ta=nvidia, host -o sxpy sxpy.f90

- Options:
  - -acc: enables OpenACC recognition
  - -Minfo=accel: enables compiler feedback
  - -ta=nvidia, cc1x cc2x cc3x: sets target architecture
  - -ta=nvidia, cuda4.x: sets cuda toolkit version
  - -ta=nvidia, fastmath O3: device code optimizations

Compiler output

```
pgcc -acc -Minfo=accel -ta=nvidia, host -o sxpy sxpy.c

sxpy:
     13, Generating copy(y[0:len])
         Generating copyin(x[0:len])
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
     14, Loop is parallelizable
         Accelerator kernel generated
         14, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
             CC 1.0 : 4 registers; 48 shared, 4 constant, 0 local memory bytes
             CC 2.0 : 8 registers; 4 shared, 60 constant, 0 local memory bytes
```

Compiler output

```
pgcc -acc -Minfo=accel -ta=nvidia, host -o sxpy sxpy.c

sxpy:
     13, Generating copy(y[0:len])
         Generating copyin(x[0:len])
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
     14, Loop is parallelizable
         Accelerator kernel generated
         14, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
             CC 1.0 : 4 registers; 48 shared, 4 constant, 0 local memory bytes
             CC 2.0 : 8 registers; 4 shared, 60 constant, 0 local memory bytes
```

Compiler output

```
pgcc -acc -Minfo=accel -ta=nvidia, host -o sxpy sxpy.c

sxpy:
     13, Generating copy(y[0:len])
         Generating copyin(x[0:len])
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
     14, Loop is parallelizable
         Accelerator kernel generated
         14, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
             CC 1.0 : 4 registers; 48 shared, 4 constant, 0 local memory bytes
             CC 2.0 : 8 registers; 4 shared, 60 constant, 0 local memory bytes
```

Compiler output

```
pgcc -acc -Minfo=accel -ta=nvidia, host -o sxpy sxpy.c

sxpy:
     13, Generating copy(y[0:len])
         Generating copyin(x[0:len])
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
     14, Loop is parallelizable
         Accelerator kernel generated
         14, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
             CC 1.0 : 4 registers; 48 shared, 4 constant, 0 local memory bytes
             CC 2.0 : 8 registers; 4 shared, 60 constant, 0 local memory bytes
```

Toy model: 2d Ising

The model (zero external field):

$$\mathcal{Z}(\beta) = \sum_{\sigma_i} exp(-\beta\ H[\sigma_i]), H[\sigma_i] = -J \sum_{<i,j>} \sigma_i \sigma_j$$

Here: $\beta = \frac{1}{T}$ and $\sigma = \pm 1$

Some observables:

- Internal energy per site $E = \langle H \rangle / V$
- Specific heat $C_V = \beta(\langle H^2 \rangle - \langle H \rangle^2) / V$
- Magnetization $M = \langle |\mu| \rangle / V, \mu = \sum_i \sigma_i$

2D Ising: Metropolis algorithm

- Idea (importance sampling):

Draw configurations according to their Boltzmann weight:

$$P^{eq}[\sigma_i] \propto exp(-\beta\ H[\sigma_i])$$

- The algorithm:

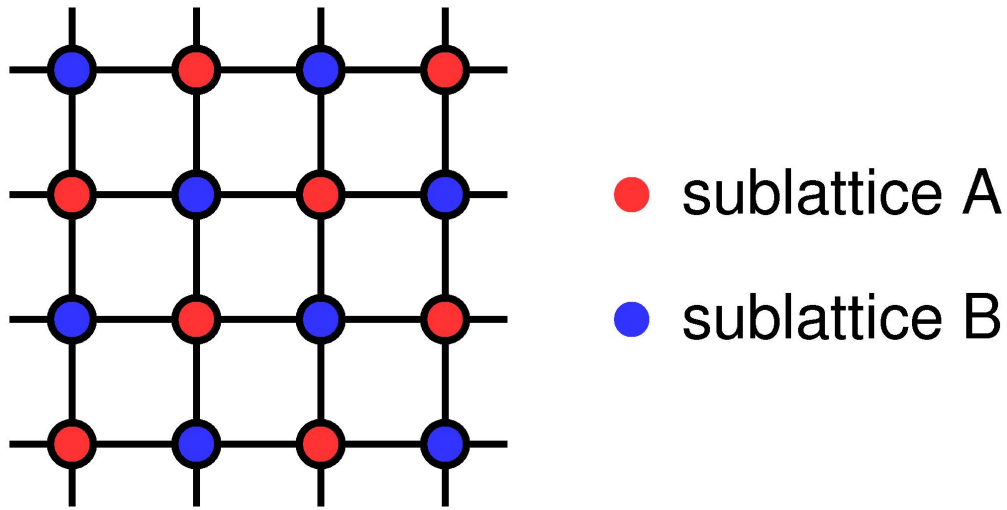
Propose locally a flip of a single spin and accept this update with probability:

$$W(\sigma_i \rightarrow \sigma_i') = min(1, exp(-\beta\ (H' - H))])$$

If the energy is lowered, the spin flip is always accepted.
If the energy is increased, accept flip with a certain probability.
Selecting spins: At random, sequentially, etc.

2D Ising: Parallel implementation

- Checker-board decomposition:



- **Note**: The practical implementation needs (good) PRNG!

2D Ising: Metropolis update

Input – A/B spin sublattices: $A[NY][NXh]$, $B[NY][NXh]$

Update 'A' spins:

For $j = 0, 1, .., NY$ **do**:
  For $i = 0, 1, .., NXh$ **do**:

    Calculate energy difference (mind BC!):
    $\Delta E = 2A[j][i](B[j][i+1] + B[j][i-1] + B[j+1][i] + B[j-1][i])$
    If $\Delta E < 0$ **Then**: flip 'A' spin $A[j][i] * = -1$
    Else:
      Compute $accept = exp(-\beta \Delta E)$
      Generate random number ω
      If $\omega < accept$ **Then**: flip 'A' spin $A[j][i] * = -1$

  End of i-loop
End of j-loop

Update 'B' spins: $A \leftrightarrow B$

2D Ising : Metropolis update with OpenMP

Start thermalization sweeps

Update 'A' spins:

```
#pragma omp num_threads(OMP_NUM_THREADS) private(tid, step)
{
 tid = omp_get_thread_num();
 step = NY/ omp_get_num_threads();
 Load PRNG seeds
 for (j = tid * step ; j < (tid + 1) * step ; j++)
   for (i = 0 ; i < NXh ; i++){
     Calculate neighbor coordinates: ip1, im1, jp1, jm1
     ΔE = 2A[j][i](B[j][ip1] + B[j][im1] + B[jp1][i] + B[jm1][i])
     Perform Metropolis update with PRNG
   } //end of nested loops
} //end of omp parallel region
```

Update 'B' spins: $A \leftrightarrow B$

End thermalization sweeps

2D Ising : naive parallelization with OpenACC

Start thermalization sweeps

#pragma acc kernels
{
Update 'A' spins:
for $(j = 0\ ;\ j < NY\ ;\ j++)$
  for $(i = 0\ ;\ i < NXh\ ;\ i++)\{$
    Calculate neighbor coordinates: $ip1, im1, jp1, jm1$
    $\Delta E = 2A[j][i](B[j][ip1] + B[j][im1] + B[jp1][i] + B[jm1][i])$
    Load PRNG seeds
    Perform Metropolis update with PRNG
  }
Update 'B' spins: $A \leftrightarrow B$
} //end of acc kernels region

End thermalization sweeps

Data construct

- C:

```
#pragma acc data [clause ...]
{
  structured block
}
```

- Fortran:

```
!$acc data [clause ...]
  structured block
!$acc end data
```

- Clauses:
  - if (*condition*)
  - async (*expression*)
  - data management clauses (*copy()*, *create()* etc.)

Data clauses

- copy(list):

Allocates memory on GPU and copies data H2D when entering region and copies data D2H when exiting the region

- copyin(list):

Allocates memory on GPU and copies data H2D when entering region

- copyout(list):

Allocates memory on GPU and copies data D2H when exiting the region

- create(list):

Allocates memory on GPU, no data transfers

- present(list):

Data is already present on GPU from another containing data region

Data clauses: array shaping

- Data clauses can be used on *data*, *kernels* or *parallel* directives:
- C:

```
#pragma acc data copy(a[n:len])
{
  structured block
}
#pragma acc kernels copyin(b[0:n])
{
  structured block
}
```

- Fortran:

```
!$acc data copy(a[n:len])
  structured block
!$acc end data

!$acc kernels copyin(b[1:n])
  structured block
!$acc end kernels
```

2D Ising : OpenACC data region

Start thermalization sweeps

```
#pragma acc data copy(A[0:NY][0:NXh], B[0:NY][0:NXh])
{
#pragma acc kernels
 {
  Update 'A' spins:
  for (j = 0 ; j < NY ; j++)
    for (i = 0 ; i < NXh ; i++){
      Calculate neighbor coordinates: ip1, im1, jp1, jm1
      ΔE = 2A[j][i](B[j][ip1] + B[j][im1] + B[jp1][i] + B[jm1][i])
      Load PRNG seeds
      Perform Metropolis update with PRNG
    }
  Update 'B' spins:   A ↔ B
 } //end of acc kernels region
}//end of acc data region
```

End thermalization sweeps

Update Construct

- C:

```
#pragma acc update [clause ...]
{
  structured block
}
```

- Clauses:
  - if (*condition*)
  - async (*expression*)
  - host(list)
  - device(list)

- Fortran:

```
!$acc update [clause ...]
  structured block
!$acc end data
```

General tips

- Nested loops are best for parallelization
- Iterations must be independent (restrict keyword/independent clause)
- Compiler must be able to figure out sizes of data region
- Use contiguous memory for multi-dimensional arrays
- Function calls must be inlineable (-Minline)
- Conditional compilation with _OPENACC macro

Reduction clause

- Reduction clause is allowed on *parallel* construct:
  - C:

```
#pragma acc parallel reduction(operation : var)
{
  structured block with reduction on var
}
```

  - Fortran:

```
!$acc kernels reduction(operation : var)
  structured block with reduction on var
!$acc end kernels
```

2D Ising : compute microstate energy per spin

```
#pragma acc data copy(A[:][:], B[:][:])
{

 Update spins

 Compute energy per spin (e.g. via subroutine call):
 double microE = 0.0;
#pragma acc parallel present_or_copyin(A[:][:], B[:][:]) reduction(+: microE)
 {
  for (j = 0 ; j < NY ; j++)
    for (i = 0 ; i < NXh ; i++){
      Calculate neighbor coordinates: ip1, im1, jp1, jm1
      microE += -A[j][i](B[j][ip1] + B[j][im1] + B[jp1][i] + B[jm1][i])
    }
 } //end of acc parallel region
}//end of acc data region
```

OpenACC execution model

- OpenACC exe model three levels: *gang*, *worker* and *vector*
- Allows to better fit a particular target architecture
- On CPU : *gang*, *worker* and *vector*
  - gang(*number*) :number of CPUs in the system
  - worker(*number*) :number of CPU cores
  - vector(*number*) : e.g. SIMD instructions (SSE, AVX)
- On GPU : *gang* and *vector*
  - gang(*number*) :number of thread blocks per grid
  - vector(*number*) :number of threads per block

Execution model examples

- 64 thread blocks, 64 threads per block:

($N = 128 \times 64$, each thread executes 2 loop instructions)

```
#pragma acc kernels loop gang(64) vector(64)
  for(int i = 0; i < N; i++) y[i] += x[i];
```

- 128 thread blocks, 128 threads per block:

```
#pragma acc parallel num_gangs(128) vector_length(64)
{
#pragma acc loop
  for(int i = 0; i < N; i++) y[i] += x[i];
}
```

- Multi-dimensional exe domain

(128×256 blocks, each block of 32×64 threads)

```
#pragma acc kernels loop gang(256) vector(64)
  for(int j = 0; j < N; j++)
#pragma acc loop gang(128) vector(32)
    for(int i = 0; i < M; i++) ...
```

2D Ising : explicit multi-dimensional execution

Start thermalization sweeps

#pragma acc data copy(A[:][:], B[:][:])

{

Update 'A' spins:

#pragma acc kernels loop gang(16) vector(32) independent

{

for $(j = 0\ ;\ j < NY\ ;\ j++)$

#pragma acc loop gang(16) vector(32) independent

for $(i = 0\ ;\ i < NXh\ ;\ i++)$ {

Calculate neighbor coordinates: $ip1, im1, jp1, jm1$

$\Delta E = 2A[j][i](B[j][ip1] + B[j][im1] + B[jp1][i] + B[jm1][i])$

PRNG seeds

Perform Metropolis update with PRNG

} //end of acc loop region

} //end of acc kernels loop region

Update 'B' spins: $A \leftrightarrow B$

}//end of acc data region

End thermalization sweeps

OpenACC RT API

- System setup routines
  - acc_init(*acc_device_nvidia*)
  - acc_set_device_type(*acc_device_nvidia*)
  - acc_set_device_num(dev_id, *acc_device_nvidia*)
- Resource allocation routines:
  - void* acc_malloc(size_t);
  - void acc_free(void*)
- Synchronization routines:
  - acc_async_wait(int);
  - acc_async_wait_all()

Interoperability with CUDA: deviceptr clause

- deviceptr(*list*):

Declare that the ptrs in *list* refer to device ptrs that need not be allocated or moved between host and device

- Example:

```
#include <openacc.h>
cudaMalloc (&x, n*sizeof(float));
cudaMalloc (&y, n*sizeof(float));
  ....
#pragma acc data deviceptr(x, y)
{
#pragma acc kernels
    for (int i < 0; i < n; i++)
      y[i] += x[i];
}
```

Interoperability with CUDA: host_data construct

Makes the address of device data available on the host

- use_device(*list*):

Use the device address for any variable in the list

- Example:

```
#include <openacc.h>
#pragma acc data copy(x[0:n])
{
#pragma acc host_data use_device(x)
    use_cuda_ptr(x);
}
```

OpenACC with OpenMP

- Multi-GPU execution with OpenMP:

```
#include <openacc.h>
#include <omp.h>

#pragma omp parallel num_threads(number_of_gpus)
{
    int id = omp_get_thread_num();
    acc_set_device_num(id, acc_device_nvidia );
#pragma acc parallel
    {
    //Do something on GPU id;
    }
}
```

OpenACC with MPI

- Multi-GPU execution with MPI:

```
#include <openacc.h>
#include <mpi.h>

  int my_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, & my_rank);
  int num_dev = acc_get_num_devices(acc_device_nvidia);
  int id = my_rank % num_dev;
  acc_set_device_num(id, acc_device_nvidia );
#pragma acc parallel
    {
    //Do something on GPU id;
    }
```

OpenACC future perspectives

- Integration into OpenMP
- More tools (GNU)
- More languages (C++)
- More targets: CPUs, AMD GPUs, Intel MIC
- Interoperability with OpenCL
- Multiple GPUs

Thank you!