Programming GPUs with OpenACC

Saber Feki
Computational Scientist Lead
Supercomputing Core Laboratory, KAUST
saber.feki@kaust.edu.sa
GPU architecture
CPU-GPU memory model

PCle Interconnect 16X - 8GB/s (gen 2) and 15.75GB/s (gen 3), very thin pipe! Kepler K40 2,880 cuda cores 1.48 Tflops/s
GPU programming

Applications

Libraries
OpenACC Directives
Programming Languages

“Drop-in” Acceleration
Easily Accelerate Applications
Maximum Flexibility
OpenACC, the standard

- By NVIDIA, CRAY, PGI and CAPS
- The standard was announced in Nov 2011 at SC11 conference
- OpenACC 2.0 released in summer 2013
- Now, 20+ partners from academia and industry
OpenACC advantages

• **Easy:** Directives are the easy path to accelerate compute intensive applications

• **Open:** OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors

• **Powerful:** GPU Directives allow complete access to the massive parallel power of a GPU
Directive syntax

- Fortran

```fortran
!$acc directive [clause [,] clause ] ...
```
... often paired with a matching end directive

```fortran
!$acc end directive
```

- C

```c
#pragma acc directive [clause [,] clause ] ...
```
Often followed by a structured code block
kernels: Your first OpenACC Directive

- Each loop executed as a separate *kernel* *(a parallel function that runs on the GPU)*

```cpp
!$acc kernels
do i=1,n
   a(i) = 0.0 b(i) = 1.0
   c(i) = 2.0
end do
do i=1,n
   a(i) = b(i) + c(i)
end do
!$acc end kernels
```
Compile and run

• **C:**
  
  `pgcc -acc [-Minfo=accel] -o saxpy_acc saxpy.c`

• **Fortran:**
  
  `pgf90 -acc [-Minfo=accel] -o saxpy_acc saxpy.f90`

• **Compiler output:**

  [sfeki@c4hdn saxpy]$ pgcc -acc -ta=nvidia -Minfo=accel -o saxpy saxpy.c

  saxpy:
  
  5, Generating present_or_copyin(x[0:n])
  Generating present_or_copy(y[0:n])
  Generating compute capability 1.0 binary
  Generating compute capability 2.0 binary

  6, Loop is parallelizable
  Accelerator kernel generated

  6, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
  CC 1.0 : 8 registers; 48 shared, 0 constant, 0 local memory bytes
  CC 2.0 : 12 registers; 0 shared, 64 constant, 0 local memory bytes
SAXPY example, revisited

**Trivial first example**
- Apply a loop directive
- Learn compiler commands

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

int main(int argc, char **argv)
{
    int N = 1<<20; // 1 million floats
    if (argc > 1)
        N = atoi(argv[1]);

    float *x = (float*)malloc(N * sizeof(float));
    float *y = (float*)malloc(N * sizeof(float));

    for (int i = 0; i < N; ++i)
        x[i] = 2.0f;
    y[i] = 1.0f;
}

saxpy(N, 3.0f, x, y);
return 0;
```
```c
while ( err > tol && iter < iter_max ) {
    err=0.0;

    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++) {

            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                 A[j-1][i] + A[j+1][i]);

            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
```
while ( err > tol && iter < iter_max ) {
    err=0.0;

    #pragma acc kernels reduction(max:err)
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                A[j-1][i] + A[j+1][i]);

            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

    #pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
pgcc -acc -ta=nvidia -Minfo=accel -o laplace2d_acc laplace2d.c

main:
57, Generating copyin(A[:4095][:4095])
   Generating copyout(Anew[1:4094][1:4094])
   Generating compute capability 1.3 binary
   Generating compute capability 2.0 binary
58, Loop is parallelizable
60, Loop is parallelizable
   Accelerator kernel generated
   58, #pragma acc loop worker, vector(16) /* blockIdx.y threadIdx.y */
   60, #pragma acc loop worker, vector(16) /* blockIdx.x threadIdx.x */
      Cached references to size [18x18] block of 'A'
      CC 1.3 : 17 registers; 2656 shared, 40 constant, 0 local memory bytes; 75% occupancy
      CC 2.0 : 18 registers; 2600 shared, 80 constant, 0 local memory bytes; 100% occupancy
64, Max reduction generated for err
69, Generating copyout(A[1:4094][1:4094])
   Generating copyin(Anew[1:4094][1:4094])
   Generating compute capability 1.3 binary
   Generating compute capability 2.0 binary
70, Loop is parallelizable
72, Loop is parallelizable
   Accelerator kernel generated
   70, #pragma acc loop worker, vector(16) /* blockIdx.y threadIdx.y */
   72, #pragma acc loop worker, vector(16) /* blockIdx.x threadIdx.x */
      CC 1.3 : 8 registers; 48 shared, 8 constant, 0 local memory bytes; 100% occupancy
      CC 2.0 : 10 registers; 8 shared, 56 constant, 0 local memory bytes; 100% occupancy
## Performance

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 1 OpenMP thread</td>
<td>69.80</td>
<td>--</td>
</tr>
<tr>
<td>CPU 2 OpenMP threads</td>
<td>44.76</td>
<td>1.56x</td>
</tr>
<tr>
<td>CPU 4 OpenMP threads</td>
<td>39.59</td>
<td>1.76x</td>
</tr>
<tr>
<td>CPU 6 OpenMP threads</td>
<td>39.71</td>
<td>1.76x</td>
</tr>
<tr>
<td>OpenACC GPU</td>
<td>162.16</td>
<td>0.24x FAIL</td>
</tr>
</tbody>
</table>

**CPU:** Intel Xeon X5680 6 Cores @ 3.33GHz

**GPU:** NVIDIA Tesla M2070
What went wrong?

Set `PGI_ACC_TIME` environment variable to ‘1’

Accelerator Kernel Timing data
./openacc-workshop/solutions/001-laplace2D-kernels/laplace2d.c

**4.4 seconds**
- `w/o init`: total=77524678 max=83398 min=72025 avg=77524
- `kernels=4422961 data=66464916`
- `grid: [256x256] block: [16x16]`
- `time(us): total=4422961 max=4543 min=4345 avg=4422`

**66.5 seconds**
- `w/o init`: total=77524678 max=83398 min=72025 avg=77524
- `kernels=4422961 data=66464916`
- `grid: [256x256] block: [16x16]`
- `time(us): total=4422961 max=4543 min=4345 avg=4422`

**8.3 seconds**
- `w/o init`: total=82135686 max=159083 min=76575 avg=82135
- `kernels=8346306 data=6677517`
- `grid: [256x256] block: [16x16]`
- `time(us): total=82135686 max=159083 min=76575 avg=82135`

**66.8 seconds**
- `w/o init`: total=82135686 max=159083 min=76575 avg=82135
- `kernels=8346306 data=6677517`
- `grid: [256x256] block: [16x16]`
- `time(us): total=82135686 max=159083 min=76575 avg=82135`

**Huge Data Transfer Bottleneck!**
- Computation: 12.7 seconds
- Data movement: 133.3 seconds
Excessive data transfer

```c
while ( err > tol && iter < iter_max ) {
    err=0.0;
    #pragma acc kernels reduction(max:err)
    A, Anew resident on host
    Copy
    A, Anew resident on accelerator
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++) {
            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }  
...  
}
And note that there are two #pragma acc kernels, so there are 4 copies per while loop iteration!
```
Another way of detecting it: NVIDIA Profiler

- Use nvprof for profiling the GPU application:

<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>44.73%</td>
<td>13.0367s</td>
<td>9000</td>
<td>1.4485ms</td>
<td>832ns</td>
<td>1.6954ms</td>
<td>[CUDA memcpy HtoD]</td>
</tr>
<tr>
<td>44.39%</td>
<td>12.9395s</td>
<td>9000</td>
<td>1.4377ms</td>
<td>2.2720us</td>
<td>1.7537ms</td>
<td>[CUDA memcpyDtoH]</td>
</tr>
<tr>
<td>6.76%</td>
<td>1.97137s</td>
<td>1000</td>
<td>1.9714ms</td>
<td>1.9698ms</td>
<td>1.9758ms</td>
<td>main_74_gpu</td>
</tr>
<tr>
<td>3.34%</td>
<td>972.58ms</td>
<td>1000</td>
<td>972.58us</td>
<td>971.59us</td>
<td>973.67us</td>
<td>main_85_gpu</td>
</tr>
<tr>
<td>0.78%</td>
<td>227.16ms</td>
<td>1000</td>
<td>227.16us</td>
<td>226.69us</td>
<td>227.97us</td>
<td>main_78_gpu_red</td>
</tr>
</tbody>
</table>

- Use NVVP GUI: NVIDIA Visual Profiler:
Data construct

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

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

- Manage data movement. Data regions may be nested

- **General Clauses**
  
  `if( condition )`

  `async( expression )`
Data clauses

- **copy** (list) Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.
- **copyin** (list) Allocates memory on GPU and copies data from host to GPU when entering region.
- **copyout** (list) Allocates memory on GPU and copies data to the host when exiting region.
- **create** (list) Allocates memory on GPU but does not copy.
- **present** (list) Data is already present on GPU from another containing data region.
- and **present_or_copy**[in|out], **present_or_create**, **deviceptr**.
Array shaping

• Compiler sometimes cannot determine size of arrays
• Must specify explicitly using data clauses and array “shape”
• C

#pragma acc data copyin(a[0:size]), copyout(b[s/4:3*s/4])

• Fortran

!$acc data copyin(a(1:size)), copyout(b(s/4:3*s/4))

• Note: data clauses can be used on data, kernels or parallel
```c
#pragma acc data copy(A), create(Anew)
while ( err > tol && iter < iter_max ) {
    err=0.0;

#pragma acc kernels reduction(max:err)
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {

            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
            A[j-1][i] + A[j+1][i]);

            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

#pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;}
```
## Performance numbers

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 1 OpenMP thread</td>
<td>69.80</td>
<td>--</td>
</tr>
<tr>
<td>CPU 2 OpenMP threads</td>
<td>44.76</td>
<td>1.56x</td>
</tr>
<tr>
<td>CPU 4 OpenMP threads</td>
<td>39.59</td>
<td>1.76x</td>
</tr>
<tr>
<td>CPU 6 OpenMP threads</td>
<td>39.71</td>
<td>1.76x</td>
</tr>
<tr>
<td>OpenACC GPU</td>
<td>13.65</td>
<td>2.9x</td>
</tr>
</tbody>
</table>
New NVIDIA profiles

The NVIDIA Visual Profiler is used to analyze the performance of a program. The figure shows the timeline for a process labeled 'Process 11006' with a thread ID of 1046591744. The profiler measures various aspects of the program's performance, including profiling overhead, memory copy, and compute operations.

The output shows the performance metrics in milliseconds, such as:

- Total time: 6.613979 s
- Breakdown of time:
  - Process: 0.250090
  - Memory: 0.001204
  - Compute: 0.000994
  - Other: 0.000503
  - Total: 6.613979 s
CUDA Kernels

- Threads are grouped into **blocks**
- **Blocks** are grouped into a **grid**
- A **kernel** is executed as a **grid of blocks of threads**
Thread blocks

• Thread blocks allow cooperation
  – Cooperatively load/store blocks of memory that they all use
  – Share results with each other or cooperate to produce a single result
  – Synchronize with each other

• Thread blocks allow scalability
  – Blocks can execute in any order, concurrently or sequentially
  – This independence between blocks gives scalability:
    • A kernel scales across any number of SMs
Mapping OpenACC to CUDA I

- The OpenACC execution model has three levels: gang, worker, and vector
- Allows mapping to an architecture that is a collection of Processing Elements (PEs)
- One or more PEs per node
- Each PE is multi-threaded
- Each thread can execute vector instructions

- **Tile** pragma in OpenACC 2.0
Mapping OpenACC to CUDA II

• For GPUs, the mapping is implementation-dependent. Some possibilities:
  – gang==block, worker==warp, and vector==threads of a warp
  – omit “worker” and just have gang==block, vector==threads of a block

• Depends on what the compiler thinks is the best mapping for the problem

• ...But explicitly specifying that a given loop should map to gangs, workers, and/or vectors is optional anyway
  – Further specifying the number of gangs/workers/vectors is also optional
  – So why do it? To tune the code to fit a particular target architecture in a straightforward and easily re-tuned way.
OpenACC loop directive and clauses

```
#pragma acc kernels loop
for( int i = 0; i < n; ++i ) y[i] += a*x[i];
```

Uses whatever mapping to threads and blocks the compiler chooses. Perhaps 16 blocks, 256 threads each

```
#pragma acc kernels loop gang(100), vector(128)
for( int i = 0; i < n; ++i ) y[i] += a*x[i];
```

100 thread blocks, each with 128 threads, each thread executes one iteration of the loop, using kernels

```
#pragma acc parallel num_gangs(100), vector_length(128)
{
    #pragma acc loop gang, vector
    for( int i = 0; i < n; ++i ) y[i] += a*x[i];
}
```

100 thread blocks, each with 128 threads, each thread executes one iteration of the loop, using parallel
Mapping OpenACC to CUDA threads and blocks

- Nested loops generate multi-dimensional blocks and grids:
  
  ```c
  #pragma acc kernels loop gang(100), vector(16)
  for( ... )
  
  #pragma acc loop gang(200), vector(32)
  for( ... )
  ```
Other clauses for loop directive

#pragma acc loop [clauses]

• independent: for independent loops
• seq: for sequential execution of the loop
• Reduction : for reduction operation such as min, max, etc...
### Jacobi example … again

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 1 OpenMP thread</td>
<td>69.80</td>
<td>--</td>
</tr>
<tr>
<td>CPU 2 OpenMP threads</td>
<td>44.76</td>
<td>1.56x</td>
</tr>
<tr>
<td>CPU 4 OpenMP threads</td>
<td>39.59</td>
<td>1.76x</td>
</tr>
<tr>
<td>CPU 6 OpenMP threads</td>
<td>39.71</td>
<td>1.76x</td>
</tr>
<tr>
<td>OpenACC GPU</td>
<td>13.65</td>
<td>2.9x</td>
</tr>
</tbody>
</table>

Note: same code runs in 9.78s on NVIDIA Tesla M2090 GPU

With Kernels and data directives
Jacobi example … again

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 1 OpenMP thread</td>
<td>69.80</td>
<td>--</td>
</tr>
<tr>
<td>CPU 2 OpenMP threads</td>
<td>44.76</td>
<td>1.56x</td>
</tr>
<tr>
<td>CPU 4 OpenMP threads</td>
<td>39.59</td>
<td>1.76x</td>
</tr>
<tr>
<td>CPU 6 OpenMP threads</td>
<td>39.71</td>
<td>1.76x</td>
</tr>
<tr>
<td>OpenACC GPU</td>
<td>10.98</td>
<td>3.62x</td>
</tr>
</tbody>
</table>

Note: same code runs in 7.58s on NVIDIA Tesla M2090 GPU

After adding loop directive with gang and vector clauses
An opportunity for Auto-tuning

- Gang and vector values can be auto-tuned for the application, targeting the available accelerator device.

An opportunity for Auto-tuning

Input code annotated with OpenACC

Accelerator Specification

Automatic code generator

Runtime evaluation and selection

Database

Input code annotated with OpenACC

```c
#pragma acc kernels
#pragma acc loop independent
da += b;
for (x = 4 ; x < nx-4; x++)
    for (y = 4; y < ny-4; y++)
        for (z = 4; k < nz-4; z++)
            U[x][y][z] = c1*V[x][y][z] + ....
```

Accelerator Specification

Automatic code generator

Runtime evaluation and selection

Database
Jacobi example … again

- Which other optimization we can further do?
  - Restructuring the code will enhance both CPU and GPU version
  - Hint: reduce memory operations
OpenACC Runtime Library

• In C:
  #include “openacc.h”

• In Fortran:
  #include ‘openacc_lib.h’ or
  use openacc

• Contains:
  – Prototypes of all routines
  – Definition of datatypes used in these routines including
    enumeration type describing types of accelerators
OpenACC Runtime Library Definitions

- **openacc_version** with a value yyyymm (year and month of the openacc version)
- **acc_device_t** : type of accelerator device
  - acc_device_none
  - acc_device_default
  - acc_device_host
  - acc_device_not_host
• **acc_get_num_devices**: returns the number of devices of the given type attached to the host

• **acc_set_device_type**: tells which type of device to use when executing an accelerator parallel or kernel region.

• **acc_get_device_type**: tells which type of device to be used for the next accelerated region

• **acc_set_device_num**: specify which device to use

• **acc_get_device_num**: returns the device number of the specified device type that will be used to run the next accelerator parallel or kernels region
OpenACC Runtime Library Routines

- **acc_init**: initialize the runtime, can be used to isolate the initialization cost from the computation cost
- **acc_shutdown**: shut down the connection to the device and free any allocated resources
- **acc_malloc**: allocate memory on the accelerator device
- **acc_free**: frees memory on the accelerator device
OpenACC Runtime Library Routines: use case

- Porting an MPI code to multiple GPUs.
- Example in running on 8 nodes, with 4 GPUs each, i.e. 32 MPI processes

- acc_init()
- acc_set_device_num( rank%4)

- Each node runs 4 MPI processes, each of them is offloading compute kernels to a separate GPU

S. Feki, A. Al-Jarro, H. Bağcı. Multiple GPUs Electromagnetics Simulations using MPI and OpenACC, Poster in GPU Technology Conference, San Jose, California, USA, March 24-27, 2014
OpenACC and CUDA libraries

Applications

Libraries

OpenACC Directives

CUDA Libraries are interoperable with OpenACC

“Drop-in” Acceleration

Easily Accelerate Applications

Programming Languages

Maximum Flexibility
GPU accelerated libraries

- NVIDIA cuBLAS
- NVIDIA cuRAND
- NVIDIA cuSPARSE
- NVIDIA NPP
- GPU VSIPPL
- CULA tools
- MAGMA
- NVIDIA cuFFT
- Rogue Wave Software
- libjacket
- CUSP
- Thrust

Vector Signal Image Processing
GPU Accelerated Linear Algebra
Matrix Algebra on GPU and Multicore
IMSL Library
Building-block Algorithms for CUDA
Sparse Linear Algebra
C++ STL Features for CUDA
Sharing data with libraries

• CUDA libraries and OpenACC both operate on device arrays
• OpenACC provides mechanisms for interoperability with library calls
  – deviceptr data clause
  – host_data construct

• Note: same mechanisms useful for interoperability with custom CUDA C/C++/Fortran code
deviceptr Data Clause

deviceptr( list ) Declares that the pointers in list refer to device pointers that need not be allocated or moved between the host and device for this pointer.

Example:

• C
  #pragma acc data deviceptr(d_input)

• Fortran
  !$acc data deviceptr(d_input)
host_data Construct

- Makes the address of device data available on the host.
- deviceptr( list ) Tells the compiler to use the device address for any variable in list. Variables in the list must be present in device memory due to data regions that contain this construct.

- Example
  - C
  
  #pragma acc host_data use_device(d_input)

- Fortran
  
  !$acc host_data use_device(d_input)
Summary on device pointers

• Use deviceptr data clause to pass pre-allocated device data to OpenACC regions and loops
• Use host_data to get device address for pointers inside acc data regions
• The same techniques shown here can be used to share device data between OpenACC loops and
  – Your custom CUDA C/C++/Fortran/etc. device code
  – Any CUDA Library that uses CUDA device pointers
Thanks!