Enhancing Productivity in Programming GPUs using OpenACC

Saber Feki

KAUST Supercomputing Laboratory

GPU Tutorial Day – September 18th, 2013
Agenda

• GPU programming models
• OpenACC directives and clauses
• OpenACC runtime library
• OpenACC and CUDA libraries
• The future: OpenACC 2.0 and beyond
GPU programming models

Applications

Libraries
“Drop-in” Acceleration

OpenACC Directives
Easily Accelerate Applications

Programming Languages
Maximum Flexibility
GPU programming models

• Programming languages (expert users):
  – CUDA: **Compute Unified Device Architecture** developed by NVIDIA
  – OpenCL: **Open Computing Language**: open standard for parallel programming of heterogeneous systems

• Directive based programming (productivity and portability)
  – HMPP (CAPS), ACC (PGI)
  – OpenACC, **The Standard for GPU Directives** (NVIDIA’s PGI, Cray, and CAPS)

• GPU accelerated libraries: cuFFT, cuBLAS, MAGMA, cuSPARSE, etc …
OpenACC, the standard

- By NVIDIA, CRAY, PGI and CAPS
- The standard was announced in November 2011 at Supercomputing conference
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
High-level, with low-level access

• Compiler directives to specify parallel regions in C & Fortran
  – OpenACC compilers offload parallel regions from host to accelerator
  – Portable across OSes, host CPUs, accelerators, and compilers

• Create high-level heterogeneous programs
  – Without explicit accelerator initialization,
  – Without explicit data or program transfers between host and accelerator

• Programming model allows programmers to start simple
  – Enhance with additional guidance for compiler on loop mappings, data location, and other performance details

• Compatible with other GPU languages and libraries
  – Interoperate between CUDA C/Fortran and GPU libraries
  – e.g. CUFFT, CUBLAS, CUSPARSE, etc
OpenACC directive syntax

• Fortran
  !$acc directive [clause [,] clause ] …]
  … often paired with a matching end directive
  !$acc end directive
• 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)*

```acc
!$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 (example with PGI)

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

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

- **Compiler output:**

  ```bash
  [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

**SAXPY in C**

```c
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];
}

... // Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

**SAXPY in Fortran**

```fortran
subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
$!acc kernels
    do i=1,n
        y(i) = a*x(i)+y(i)
    enddo
$!acc end kernels
end subroutine saxpy

...
$ Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...
```
C tip: the restrict keyword Declaration

- Declaration of intent given by the programmer to the compiler
  - Applied to a pointer, e.g. `float *restrict ptr`
  - Meaning: “for the lifetime of ptr, only it or a value directly derived from it (such as ptr + 1) will be used to access the object to which it points”*
- Limits the effects of pointer aliasing
- OpenACC compilers often require restrict to determine independence
  - Otherwise the compiler can’t parallelize loops that access ptr
  - Note: if programmer violates the declaration, behavior is undefined
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;
}

*restrict: “I promise y does not alias x”
```
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++;
}
PGI Accelerator Compiler output (C)

```c
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

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

**GPU:** NVIDIA Tesla M2070

<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>
What went wrong?

Set `PGI_ACC_TIME` environment variable to ‘1’

Accelerator Kernel Timing data

```
./openacc-workshop/solutions/001-laplace2D-kernels/laplace2d.c
main
  69: region entered 1000 times
      time(us): total=77524918 init=240 region=77524678
      kernels=4422961 data=66464916
    w/o init: total=77524678 max=83398 min=72025 avg=77524
  72: kernel launched 1000 times
      grid: [256x256] block: [16x16]
      time(us): total=4422961 max=4543 min=4345 avg=4422
  ./openacc-workshop/solutions/001-laplace2D-kernels/laplace2d.c
main
  57: region entered 1000 times
      time(us): total=82135902 init=216 region=82135686
      kernels=8346306 data=86775717
    w/o init: total=82135686 max=159083 min=76575 avg=82135
  60: kernel launched 1000 times
      grid: [256x256] block: [16x16]
      time(us): total=8201000 max=8297 min=8187 avg=8201
  64: kernel launched 1000 times
      grid: [1] block: [256]
      time(us): total=145306 max=242 min=143 avg=145
acc_init.c
acc_init
  29: region entered 1 time
      time(us): init=158248
```

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
    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]));
        }
    }
    ...  
}
```

These copies happen every iteration of the outer while loop!*

And note that there are two #pragma acc kernels, so there are 4 copies per while loop iteration!

*Saber Feki*
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**.
Jacobi Iteration: OpenACC C Code, Revisited

```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++;
}```
<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>

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

**GPU:**
- NVIDIA Tesla M2070

Saber Feki
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
Update construct

- Used to update existing data after it has changed in its corresponding copy (e.g. update device copy after host copy changes)
- Move data from GPU to host, or host to GPU.
- Data movement can be conditional, and asynchronous.
- **Fortran**
  ```fortran
  !$acc update [clause ...]
  ```
- **C**
  ```c
  #pragma acc update [clause ...]
  ```
- **Clauses**
  - `host( list )`
  - `device( list )`
  - `if( expression )`
  - `async( expression )`
Further speedups

• OpenACC gives us more detailed control over parallelization
  – Via gang, worker, and vector clauses
• By understanding more about OpenACC execution model and GPU hardware organization, we can get higher speedups on this code
• By understanding bottlenecks in the code via profiling, we can reorganize the code for higher performance (e.g. nvprof, nvvp )
Finding Parallelism in your code

- (Nested) for loops are best for parallelization
- Large loop counts needed to offset GPU/memcpy overhead
- Iterations of loops must be independent of each other
  - To help compiler: restrict keyword (C), independent clause
- Compiler must be able to figure out sizes of data regions
  - Can use directives to explicitly control sizes
- Pointer arithmetic should be avoided if possible
  - Use subscripted arrays, rather than pointer-indexed arrays.
- Function calls within accelerated region must be inlineable.
Tips and Tricks

- (PGI) Use time option to learn where time is being spent
  - PGI_ACC_TIME = 1 (environment variable)
- Eliminate pointer arithmetic
- Inline function calls in directives regions
  - (PGI): -Minline or –Minline=levels:<N>
- Use contiguous memory for multi-dimensional arrays
- Use data regions to avoid excessive memory transfers
- Conditional compilation with _OPENACC macro
• 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

- 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
Mapping OpenACC to CUDA

• 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

```c
#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.

```c
#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

```c
#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

Saber Feki
Mapping OpenACC to CUDA threads and blocks

• Nested loops generate multi-dimensional blocks and grids:

  #pragma acc kernels loop gang(100), vector(16)
  for( ... )

  100 blocks tall (row/Y direction)

  16 thread tall block

  #pragma acc loop gang(200), vector(32)
  for( ... )

  200 blocks wide (column/X direction)

  and 32 thread wide
Selecting block size (e.g., vectors per gang)

• Total number of threads in a block between 256 and 512 is usually a good number
  – Overly small blocks will limit the # of concurrent threads due to limitation on maximum # of concurrent blocks/SM
  – Overly large blocks can hinder performance, e.g., by increasing cost of any synchronizations/barrier among all the threads in a block

• All CUDA-capable GPUs to date prefer # threads per block to be a multiple of 32 if possible
  – …Since 32 threads is the warp size of current CUDA-capable GPUs
  – Non-multiples of 32 waste some resources and cycles
  – Furthermore, a multiple of 32 threads wide (x-dimension) is best (facilitates coalesced memory access to adjacent memory addresses)
Other clauses for loop directive

#pragma acc loop [cluases]

- **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
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`
OpenACC Runtime Library Routines I

- **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 II

- `acc_async_test(int)`: test for completion of associated asynchronous activities
- `acc_async_test_all()`: test for completion of all asynchronous activities
- `acc_async_wait(int)`: wait for completion of associated asynchronous activities
- `acc_async_waitall()`: wait for completion of all asynchronous activities
OpenACC Runtime Library Routines III

- **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 and CUDA libraries

Applications

Libraries

OpenACC Directives

Programming Languages

CUDA Libraries are interoperable with OpenACC

“Drop-in” Acceleration

Easily Accelerate Applications

Maximum Flexibility

Saber Feki
CUDA Libraries

NVIDIA cuBLAS
NVIDIA cuRAND
NVIDIA cuSPARSE
NVIDIA NPP
GPU VSIPL
CULA tools
MAGMA
CUDA cuFFT
ROGUE WAVE SOFTWARE
libjacket
CU SP
Thrust

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

GPU accelerated Libraries
Sharing data with libraries

- CUDA libraries and OpenACC both operate on device arrays
- OpenACC provides mechanisms to interoperate 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
Enhancing Productivity in Programming GPUs using OpenACC

Saber Feki

KAUST Supercomputing Laboratory

GPU Tutorial Day – September 18th, 2013