

CSC 447: Parallel Programming for Multi-Core and Cluster Systems

OPENACC

Instructor: Haidar M. Harmanani

Spring 2018

Programming Approaches

Libraries

“Drop-in” Acceleration

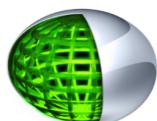
OpenACC Directives

Easily Accelerate Apps

Programming Languages

Maximum Flexibility

Development Environment



Nsight IDE
Linux, Mac and Windows
GPU Debugging and Profiling

CUDA-GDB debugger
NVIDIA Visual Profiler

Open Compiler Tool Chain



Enables compiling new languages to CUDA platform, and CUDA languages to other architectures

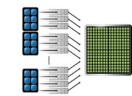
Hardware Capabilities



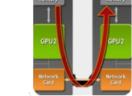
Dynamic Parallelism



HyperQ



GPUDirect



3 Ways to Accelerate Applications

Applications

Libraries

“Drop-in”
Acceleration

OpenACC Directives

Easily Accelerate
Applications

Programming Languages

Maximum
Flexibility

3 Ways to Accelerate Applications

Applications

Libraries

“Drop-in”
Acceleration

OpenACC Directives

Easily Accelerate
Applications

Programming Languages

Maximum
Flexibility

Libraries: Easy, High-Quality Acceleration

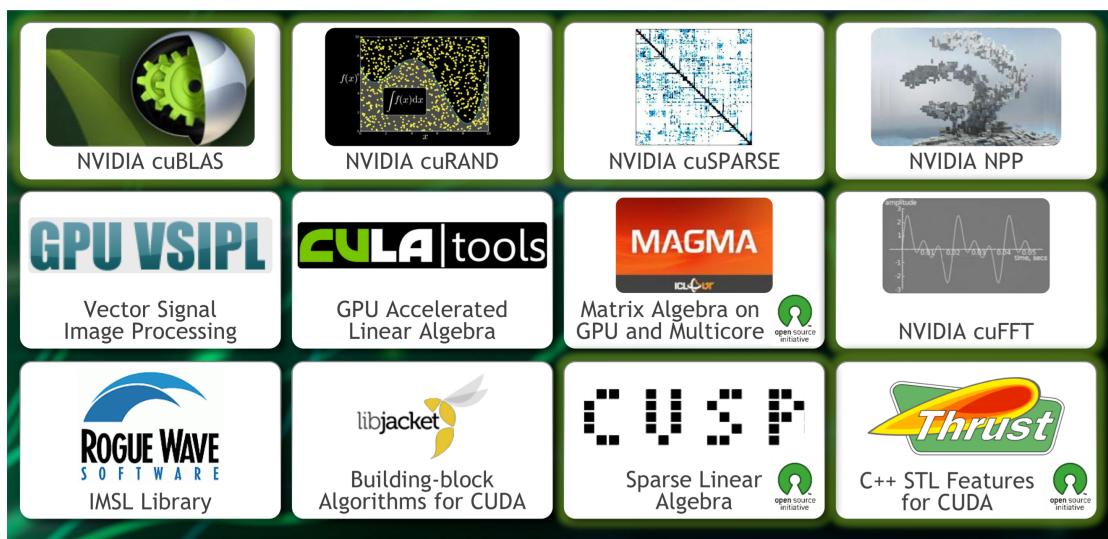
Ease of use: Using libraries enables GPU acceleration without in-depth knowledge of GPU programming

“Drop-in”: Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes

Quality: Libraries offer high-quality implementations of functions encountered in a broad range of applications

Performance: NVIDIA libraries are tuned by experts

Libraries: Easy, High-Quality Acceleration



CUDA Math Libraries

High performance math routines for your applications:

- cuFFT - Fast Fourier Transforms Library
- cuBLAS - Complete BLAS Library
- cuSPARSE - Sparse Matrix Library
- cuRAND - Random Number Generation (RNG) Library
- NPP - Performance Primitives for Image & Video Processing
- Thrust - Templatized C++ Parallel Algorithms & Data Structures
- math.h - C99 floating-point Library

Included in the CUDA Toolkit [Free download @ www.nvidia.com/getcuda](http://www.nvidia.com/getcuda)

More information on CUDA libraries:

<http://www.nvidia.com/object/gtc2010-presentation-archive.html#session2216>

Deep Learning: Latest Addition

- NVIDIA cuDNN is a GPU-accelerated library of primitives for deep neural networks
- Designed to be integrated into higher-level machine learning frameworks



Libraries: Easy, High-Quality Acceleration

```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
```

$$z = \alpha x + y$$

x, y, z: vector
\alpha: scalar

```
saxpy(1<<20, 2.0, x, y);
```

Standard C Code

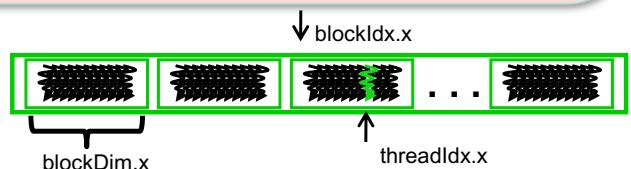
$$1 << 20 = 1,048,576$$

Libraries: Easy, High-Quality Acceleration

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

```
saxpy<<<4096,256>>>(N, 2.0, x, y);
```

CUDA C Code



Libraries: Easy, High-Quality Acceleration

Copy n elements from a vector x in host memory space to a vector d_x in GPU memory space

```
cublasInit();  
cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1);  
cublasSetVector(N, sizeof(y[0]), y, 1, d_y, 1);  
  
// Perform SAXPY on 1M elements  
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);  
  
cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1);  
cublasShutdown();
```

CUBLAS C Code

Libraries: Easy, High-Quality Acceleration

- Step 1: Substitute library calls with equivalent CUDA library calls

saxpy (...) ➔ cublasSaxpy (...)

- Step 2: Manage data locality

with CUDA: cudaMalloc(), cudaMemcpy(), etc.

with CUBLAS: cublasSetVector(), cublasGetVector(), etc.

- Step 3: Rebuild and link the CUDA-accelerated library

nvcc myobj.o -l cublas

3 Ways to Accelerate Applications

Applications

Libraries

“Drop-in”
Acceleration

OpenACC
Directives

Easily Accelerate
Applications

Programming
Languages

Maximum
Flexibility

OpenACC Directive Syntax

- `#pragma acc directive [clause [,] clause] ...]`
Often followed by a structured code block

C tip: the restrict keyword

- 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

SAXPY Revisited: OpenMP

SAXPY

- Single-Precision A·X Plus Y

$$z = \alpha x + y$$

x, y, z : vector
 α : scalar

```
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
    #pragma omp parallel for
    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 Revisited: OpenACC

SAXPY

- Single-Precision A·X Plus Y

$$z = \alpha x + y$$

x, y, z : vector
 α : scalar

```
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
    #pragma acc parallel loop
    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);
...
```

**OpenACC is not
GPU Programming.**



**OpenACC is
Exposing Parallelism
in your code.**

Complete SAXPY example code

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

```
#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];
}
```

*restrict:
"I promise y does not
alias x"

```
int main(int argc, char **argv)
{
    int N = 1<<20; // 1 million floats

    if (argc > 1)
        N = atoi(argv[1]);

    oat *x = (float*)malloc(N * sizeof(float));
    oat *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;
}
```

Compile and run

- C:

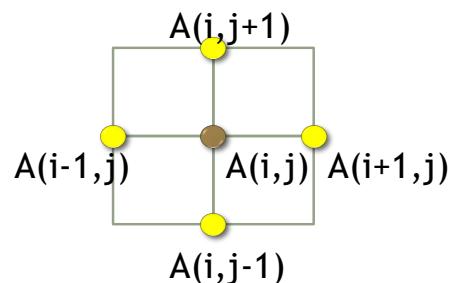
```
pgcc -acc -ta=nvidia -Minfo=accel -o saxpy_acc
saxpy.c
```

- Compiler output:

```
pgcc -acc -Minfo=accel -ta=nvidia -o saxpy_acc saxpy.c
saxpy:
  8, Generating copyin(x[:n-1])
  Generating copy(y[:n-1])
  Generating compute capability 1.0 binary
  Generating compute capability 2.0 binary
  9, Loop is parallelizable
  Accelerator kernel generated
  9, #pragma acc loop worker, vector(256) /* blockIdx.x threadIdx.x */
      CC 1.0 : 4 registers; 52 shared, 4 constant, 0 local memory bytes; 100% occupancy
      CC 2.0 : 8 registers; 4 shared, 64 constant, 0 local memory bytes; 100% occupancy
```

Example 2: Jacobi Iteration

- Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.
 - Common, useful algorithm
 - Example: Solve Laplace equation in 2D:



Jacobi Iteration Code

```
while ( error > tol && iter < iter_max )  
{  
    error=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]);  
  
            error = max(error, 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++;  
}
```

Iterate until converged

Iterate across matrix elements

Calculate new value from neighbors

Compute max error for convergence

Swap input/output arrays

OpenMP C Code

```
while ( error > tol && iter < iter_max ) {
    error=0.0;

#pragma omp parallel for shared(m, n, Anew, A)
    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]);

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

#pragma omp parallel for shared(m, n, Anew, A)
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
```

Parallelize loop across
CPU threads

Parallelize loop across
CPU threads

First Attempt: OpenACC C

```
while ( error > tol && iter < iter_max ) {
    error=0.0;

#pragma acc kernels
    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]);

            error = max(error, 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++;
}
```

Execute GPU kernel for
loop nest

Execute GPU kernel for
loop nest

First Attempt: Compiler output

```
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 error
  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
```

OpenACC parallel vs. kernels

PARALLEL

Requires analysis by programmer to ensure safe parallelism

Straightforward path from OpenMP

KERNELS

Compiler performs parallel analysis and parallelizes what it believes safe

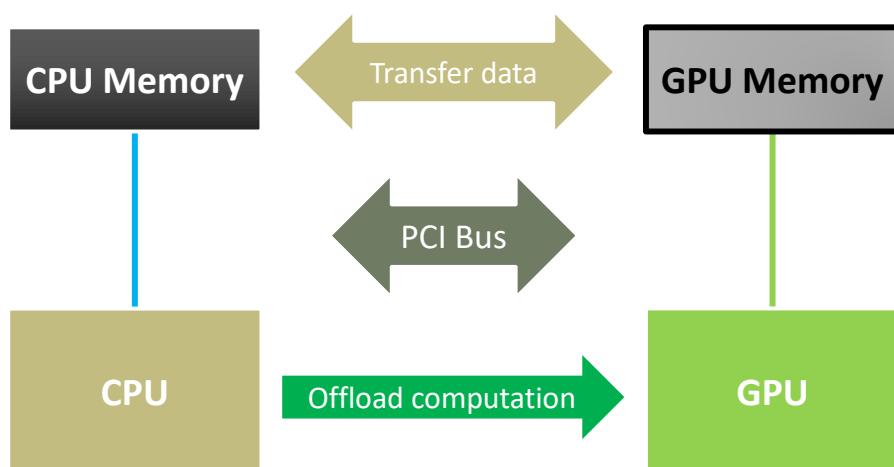
Can cover larger area of code with single directive

Both approaches are equally valid and can perform equally well.

First Attempt: Performance

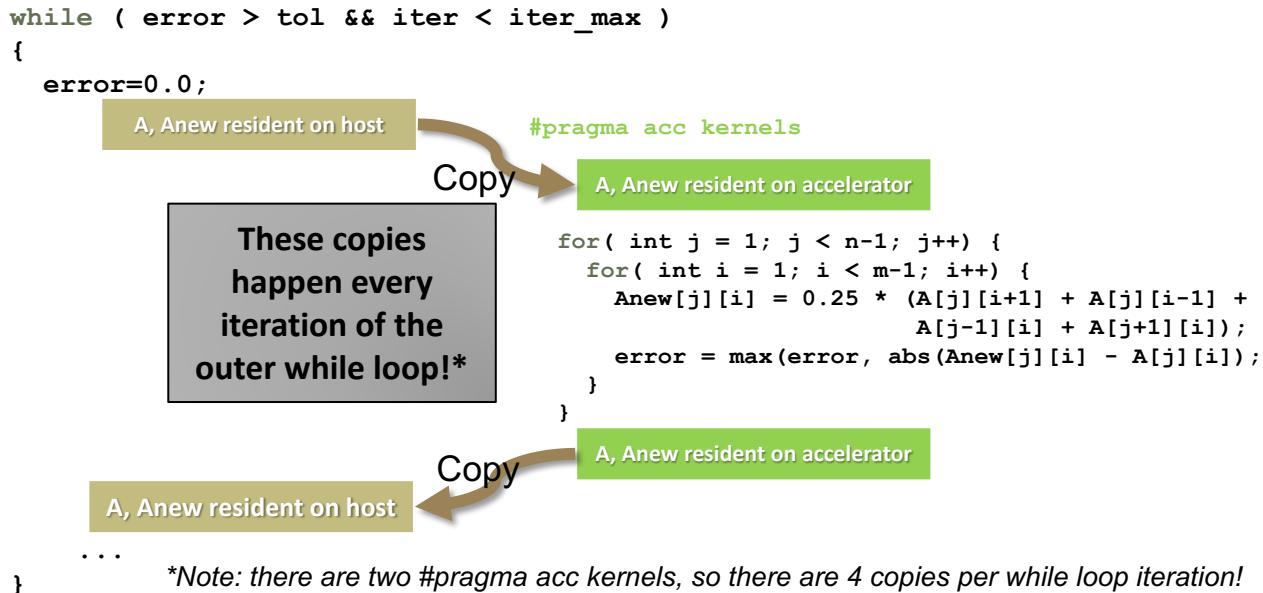
Execution	Time (s)	Speedup	
CPU 1 OpenMP thread	69.80	--	GPU: NVIDIA Tesla M2070
CPU 2 OpenMP threads	44.76	1.56x	CPU: Intel Xeon X5680 6 Cores @ 3.33GHz
CPU 4 OpenMP threads	39.59	1.76x	
CPU 6 OpenMP threads	39.71	1.76x	
OpenACC GPU	162.16	0.24x FAIL	Speedup vs. 1 CPU core Speedup vs. 6 CPU cores

Basic Concepts



For efficiency, decouple data movement and compute off-load

Excessive Data Transfers



DATA MANAGEMENT

Data Construct

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

Manage data movement. Data regions may be nested.

Data Clauses

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

Array Shaping

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

```
#pragma acc data copyin(a[0:size-1]), copyout(b[s/4:3*s/4])
```

- Note: data clauses can be used on data, kernels or parallel

Update Construct

```
#pragma acc update [clause ...]  
  
if( expression )  
async( expression )
```

- 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.

Second Attempt: OpenACC C

```
#pragma acc data copy(A), create(Anew)
while ( error > tol && iter < iter_max ) {
    error=0.0;

#pragma acc kernels
    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]);

            error = max(error, 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++;
}
```

Copy A in at beginning of loop, out at end. Allocate Anew on accelerator

Second Attempt: Performance

Execution	Time (s)	Speedup
CPU 1 OpenMP thread	69.80	--
CPU 2 OpenMP threads	44.76	1.56x
CPU 4 OpenMP threads	39.59	1.76x
CPU 6 OpenMP threads	39.71	1.76x
OpenACC GPU	13.65	2.9x

GPU: NVIDIA Tesla M2070

CPU: Intel Xeon X5680
6 Cores @ 3.33GHz

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

Speedup vs. 1 CPU core

Speedup vs. 6 CPU cores

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
- Will tackle these later on

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
-ta=nvidia, time
- Eliminate pointer arithmetic
- Inline function calls in directives regions
(PGI): -inline or -inline, levels (<N>)
- Use contiguous memory for multi-dimensional arrays
- Use data regions to avoid excessive memory transfers
- Conditional compilation with _OPENACC macro

COMPLETE OPENACC API

Kernels Construct

C

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

Parallel Construct

C

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

```
private( list )
firstprivate( list )
reduction( operator:list )
Also any data clause
```

Parallel Clauses

<code>num_gangs (expression)</code>	Controls how many parallel gangs are created (CUDA <code>gridDim</code>).
<code>num_workers (expression)</code>	Controls how many workers are created in each gang (CUDA <code>blockDim</code>).
<code>vector_length (list)</code>	Controls vector length of each worker (SIMD execution).
<code>private(list)</code>	A copy of each variable in list is allocated to each gang.
<code>firstprivate (list)</code>	private variables initialized from host.
<code>reduction(operator:list)</code>	private variables combined across gangs.

Loop Construct

C

```
#pragma acc loop [clause ...]
{ loop }
```

Detailed control of the parallel execution of the following loop.

Loop Clauses

<code>collapse(n)</code>	Applies directive to the following n nested loops.
<code>seq</code>	Executes the loop sequentially on the GPU.
<code>private(list)</code>	A copy of each variable in list is created for each iteration of the loop.
<code>reduction(operator:list)</code>	private variables combined across iterations.

Loop Clauses Inside parallel Region

Gang	Shares iterations across the gangs of the parallel region.
worker	Shares iterations across the workers of the gang.
vector	Execute the iterations in SIMD mode.

Loop Clauses Inside kernels Region

gang [(*num_gangs*)]

Shares iterations across across at most *num_gangs* gangs.

worker [(*num_workers*)]

Shares iterations across at most *num_workers* of a single gang.

vector [(*vector_length*)]

Execute the iterations in SIMD mode with maximum *vector_length*.

independent

Specify that the loop iterations are independent.

OTHER SYNTAX

Other Directives

<code>cache</code> construct	Cache data in software managed data cache (CUDA shared memory).
<code>host_data</code> construct	Makes the address of device data available on the host.
<code>wait</code> directive	Waits for asynchronous GPU activity to complete.
<code>declare</code> directive	Specify that data is to be allocated in device memory for the duration of an implicit data region created during the execution of a subprogram.

Runtime Library Routines

C

```
#include "openacc.h"

acc_async_wait
acc_async_wait_all
acc_shutdown
acc_on_device
acc_malloc
acc_free
```

Environment and Conditional Compilation

`ACC_DEVICE device`

Specifies which device type to connect to.

`ACC_DEVICE_NUM num`

Specifies which device number to connect to.

`_OPENACC`

Preprocessor directive for conditional compilation. Set to OpenACC version

3 Ways to Accelerate Applications

Applications

Libraries

OpenACC
Directives

Programming
Languages

“Drop-in”
Acceleration

Easily Accelerate
Applications

Maximum
Flexibility