Introduction to OpenACC Directives

Leadership Computing Platforms, Extreme-scale Applications, and Performance Strategies

CScADS Summer Workshops 2012

Yonghong Yan

http://www.cs.uh.edu/~hpctools

University of Houston

Acknowledgements: Mark Harris (Nvidia), Duncan Pool (Nvidia)
Michael Wolfe (PGI), Sunita Chandrasekaran (UH), Barbara M. Chapman (UH)

About US

- UH HPCTools Group
  - Led by Barbara Chapman, member of OpenMP ARB
  - 4 senior and ~18 graduate students (most Ph.D)
  - http://www.cs.uh.edu/~hpctools
- Major Research and Development
  - OpenMP (NSF, DoE), OpenUH compiler
  - PGAS: OpenSHMEM (DoD), CAF (TOTAL)
  - *OpenACC test suite and compiler (NVIDIA)*
  - Heterogeneous and Embedded related (TI, SRC, Freescale)
- Myself, research assistant professor, OpenMP committee member
  - Use OpenMP, but more on implementing OpenMP and compilers

Contents

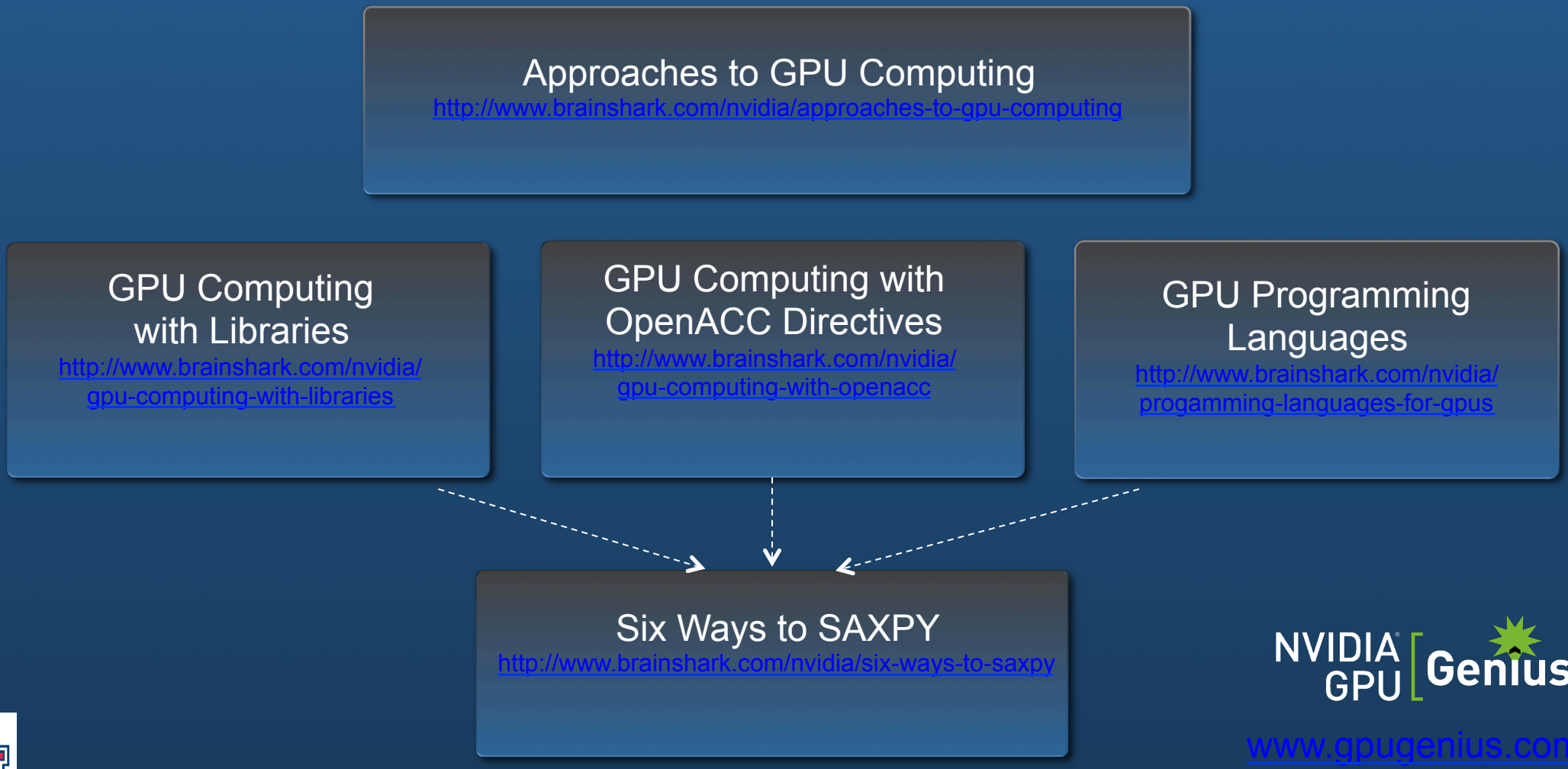
Acknowledgements and References

- The majority of the contents and slides are from the OpenACC tutorials at GTC 2012 conference given by Mark Harris (Nvidia), Duncan Pool (Nvidia), Cliff Woolley (NVIDIA), and Michael Wolfe (PGI). These contents, NVIDIA logo, and slides for this presentation are used by permission!

- Reference materials:
  - GTC 2012 tutorial http://www.gputechconf.com/gtcnew/on-demand-GTC.php?sessionTopic=&searchByKeyword=openacc&submit=&select=+&sessionEvent=2&sessionYear=&sessionFormat=#1308
  - OpenACC examples: http://www.caps-entreprise.com/fr/page/index.php?id=155&p_p=36
  - and

Introduction to GPU Software Development

Customer Presentation Series

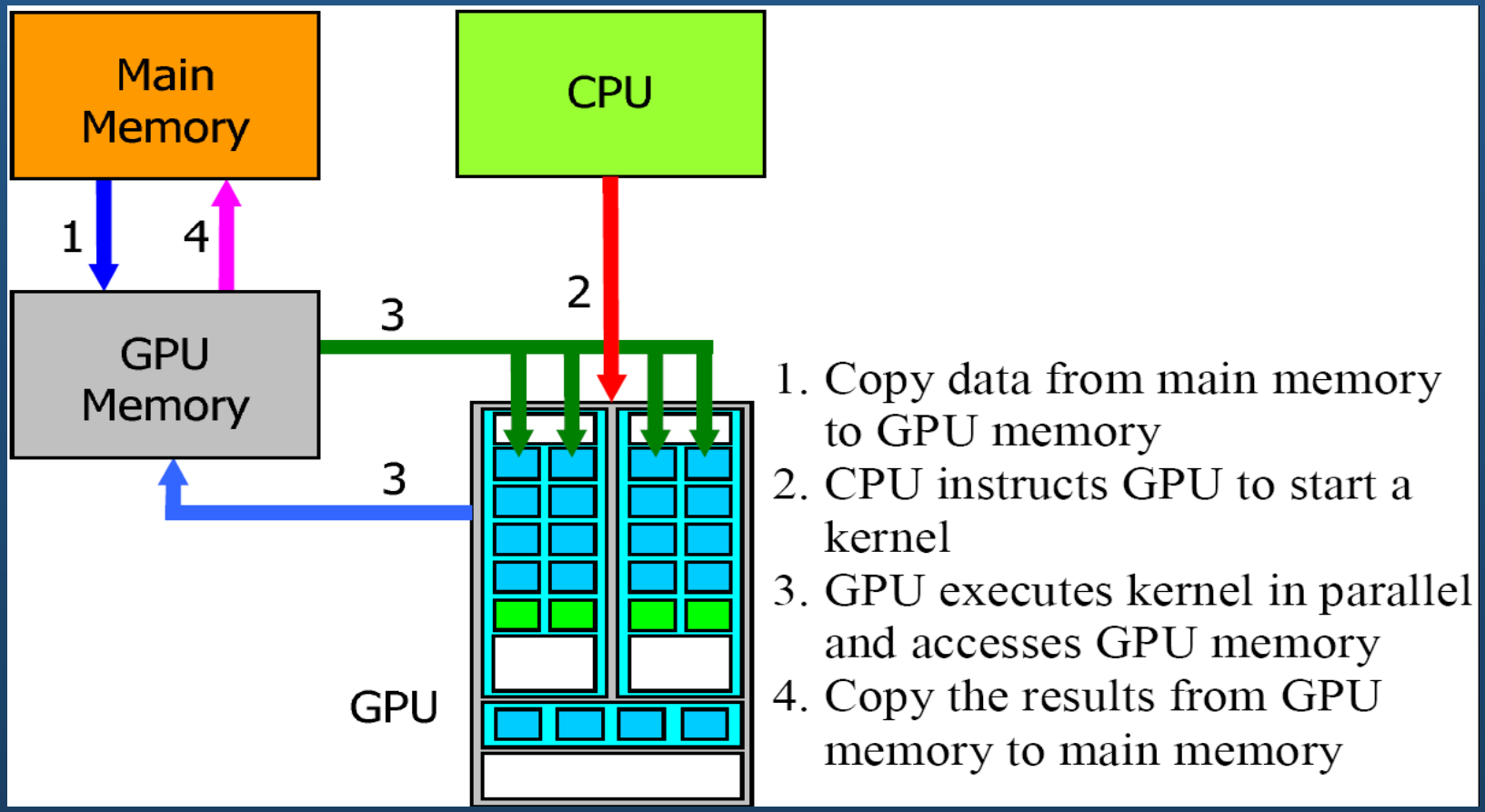


NVIDIA GPU Genius

www.gpugenius.com

Contents

Process Flow of a Kernel Call



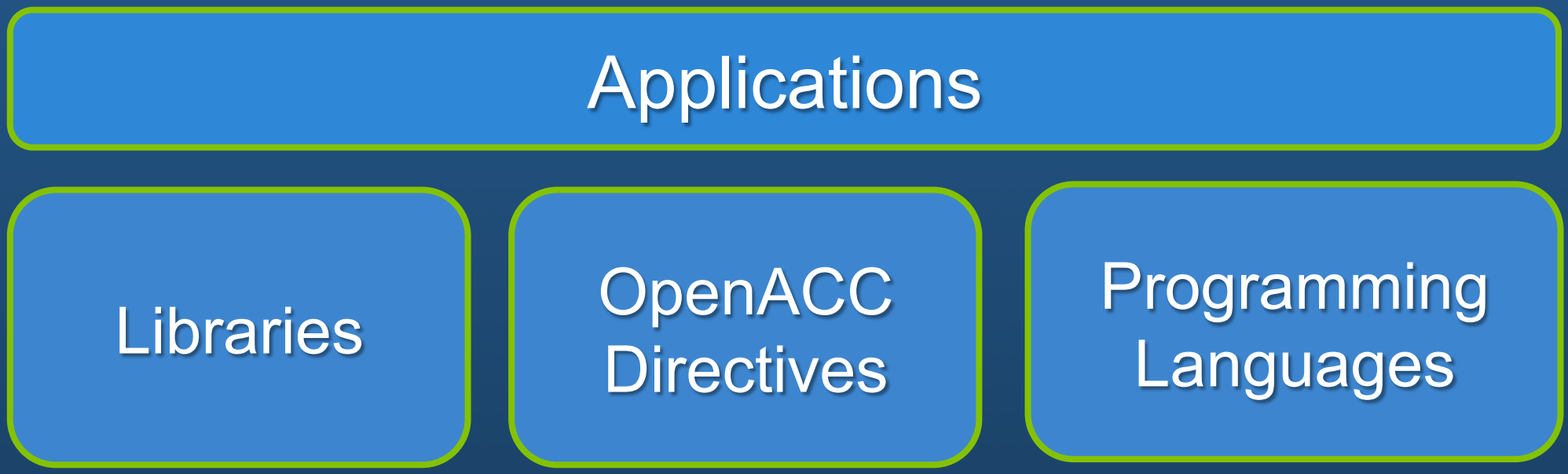
GPU Programming with CUDA

```
/* compute vector sum C = A+B */
/* Each thread performs one pair-wise addition */
__global__ void vecAdd(float* A, float* B, float* C) {
    int i = threadId.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main () {
    // cudaMalloc for A, B, and C; and cudaMemcpy for A and B
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256 >>> (d_A, d_B, d_C);
    // cudaMemcpy for C
}
```

- Exposed-communication memory hierarchy
  - Explicit memory operations: *cudaMalloc*, *cudaMemcpy*
- Compilation and execution
  - Multiple compilation stage and linking
  - Embed GPU binary as string in CPU binary

3 Ways to Accelerate Applications



3 Ways to Accelerate Applications

"Drop-in" Acceleration

Easily Accelerate Applications

Maximum Flexibility

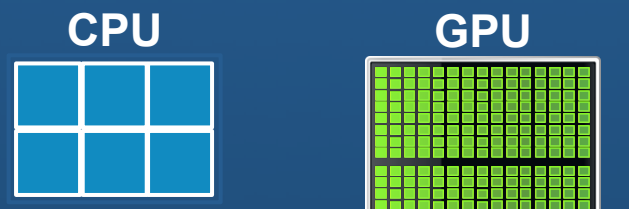
OpenACC

The Standard for GPU Directives

- **Easy:** Directives are the easy path to accelerate compute intensive applications
- **Open:** OpenACC is an open GPU directives standard
- **Powerful:** GPU Directives allow complete access to the massi parallel power of a GPU

- **Latest release:** version 1.0, Nov 2011; www.openacc.org

OpenACC Directives




```
Program myscience
  ... serial code ...
!$acc kernels
  do k = 1,n1
    do i = 1,n2
      ... parallel code ...
    enddo
  enddo
!$acc end kernels
  ...
End Program myscience
```

Your original Fortran or C code

OpenACC Compiler Hint

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs

OpenACC®
DIRECTIVES FOR ACCELERATORS

2 Basic Steps to Get Started

- **Step 1:** Annotate source code with directives:

```
!$acc data copy(util1,util2,util3) copyin(ip,scp2,scp2i)
  !$acc parallel loop
  …
  !$acc end parallel
!$acc end data
```

- **Step 2:** Compile & run:

```
pgf90 -ta=nvidia -Minfo=accel file.f
```

Contents

Single precision Alpha X Plus Y (SAXPY)

Part of Basic Linear Algebra Subroutines (BLAS) Library

$$z = \alpha x + y$$

x, y, z : vector

α : scalar

GPU SAXPY in multiple languages and libraries

OpenACC Compiler Directives

Parallel C Code

```
void saxpy(int n,
           float a,
           float *x,
           float *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);
...
```

Parallel Fortran Code

```
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)
...
```

Directive Syntax

Fortran

```
!$acc directive [clause [,] clause] …]
```

…often paired with a matching end directive surrounding a structured code block:

```
!$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* on the GPU.

```
!$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
```

kernel 1

kernel 2

Kernel:
A parallel function that runs on the GPU

Kernels Construct

Fortran

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

C

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

Clauses

```
if( condition )
async( expression )
```

Also, any data clause (and more)

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:

```
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
```

Jacobi Iteration: C Code

```
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++;
```

Iterate until converged

Iterate across matrix elements

Calculate new value from neighbors

Compute max error for convergence

Swap input/output arrays

Jacobi Iteration: OpenMP C Code

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

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

Jacobi Iteration: OpenACC C Code

```
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++;
```

Execute GPU kernel for loop nest

Execute GPU kernel for loop nest

Jacobi Iteration: OpenMP Fortran Code

```
do while ( err > tol .and. iter < iter_max )
  err=0._fp_kind

!$omp parallel do shared(m,n,Anew,A) reduction(max:err)
  do j=1,m
    do i=1,n

      Anew(i,j) = .25_fp_kind * (A(i+1, j) + A(i-1, j) + &
                                 A(i  , j-1) + A(i , j+1))

      err = max(err, Anew(i,j) - A(i,j))
    end do
  end do

!$omp end parallel do
!$omp parallel do shared(m,n,Anew,A)
  do j=1,m-2
    do i=1,n-2
      A(i,j) = Anew(i,j)
    end do
  end do
!$omp end parallel do
iter = iter +1
end do
```

Parallelize loop across CPU threads

Parallelize loop across CPU threads

Jacobi Iteration: OpenACC Fortran Code

```
do while ( err > tol .and. iter < iter_max )
  err=0._fp_kind

!$acc kernels reduction(max:err)
  do j=1,m
    do i=1,n

      Anew(i,j) = .25_fp_kind * (A(i+1, j) + A(i-1, j) + &
                                 A(i  , j-1) + A(i , j+1))

      err = max(err, Anew(i,j) - A(i,j))
    end do
  end do
!$acc end kernels
!$acc kernels
  do j=1,m-2
    do i=1,n-2
      A(i,j) = Anew(i,j)
    end do
  end do
!$acc end kernels
  iter = iter +1
end do
```

Execute GPU kernel for loop nest

Execute GPU kernel for loop nest

Contents

Data Construct

Fortran

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

C

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

General Clauses

```
if( condition )
async( expression )
```

Manage data movement. Data regions may be nested.

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`

Jacobi Iteration: OpenACC C Code

```
#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++;
```

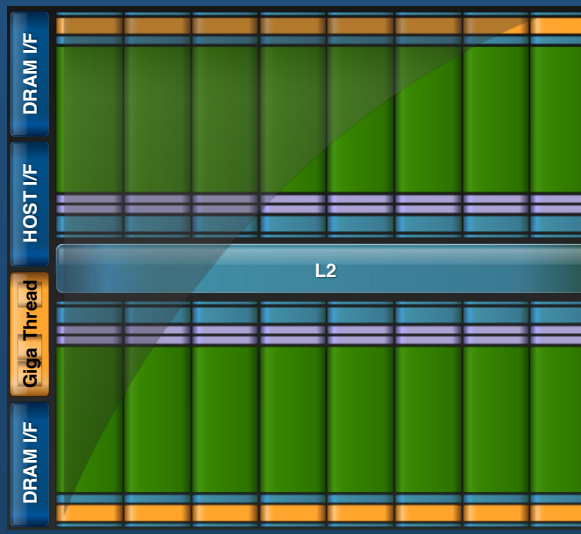
Copy A in at the beginning of loop, out at the end. Allocate Anew on acc

Contents

GPU Architecture: Two Main Components

Global memory

- Analogous to RAM in a CPU server
- Accessible by both GPU and CPU
- Currently up to 6 GB
- Bandwidth currently up to almost 180 GB/s for Tesla products
- ECC on/off option for Quadro and Tesla products

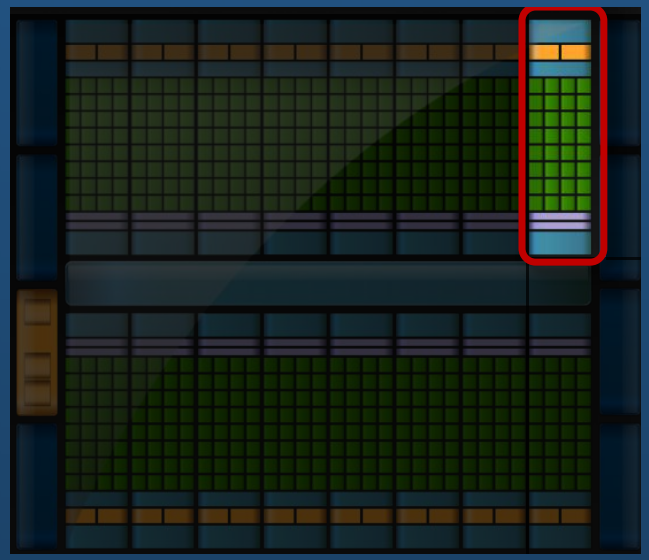


Streaming Multiprocessors (SMs)

- Perform the actual computations
- Each SM has its own:
  - Control units, registers, execution pipelines, caches

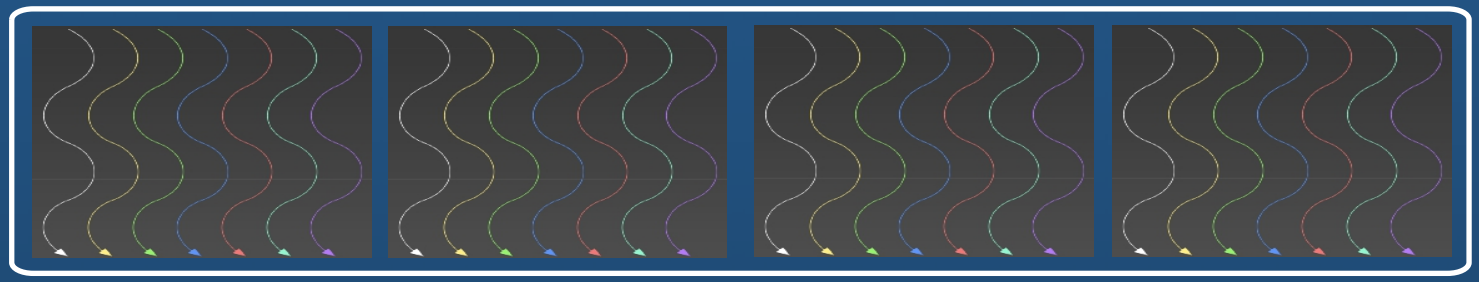
GPU Architecture - Fermi: Streaming Multiprocessor (SM)

- 32 CUDA Cores per SM
  - 32 fp32 ops/clock
  - 16 fp64 ops/clock
  - 32 int32 ops/clock
- 2 warp schedulers
  - Up to 1536 threads concurrently
- 4 special-function units
- 64KB shared mem + L1 cache
- 32K 32-bit registers



CUDA Kernels Dimensionality

- `vecAdd<<< N/256, 256 >>> (d_A, d_B, d_C);`



- A kernel is executed as a grid of blocks of threads
  - Threads are grouped into blocks
  - Blocks are grouped into a grid
- The structure could be viewed as nested `loop`
  - The inner loop → threads; outer loop → blocks

Map CUDA Execution Model to OpenACC

- Nested `loop`s inside a `parallel` region or `kernels`
- The OpenACC execution model has three levels: gang, worker, and vector

Fortran

```
!$acc loop [clause …]
    structured block
!$acc end loop
```

C

```
#pragma acc loop [clause …]
    { structured block }
```

Clauses:

collapse(*n*)

gang *[*(*scalar-integer-expression*)*]*

worker *[*(*scalar-integer-expression*)*]*

vector *[*(*scalar-integer-expression*)*]*

…

- Typical mapping for GPU:
  - gang==block, worker==warp, and vector==threads of a warp
  - omit "worker" and just have gang==block, vector==threads of a block

Mapping OpenACC to CUDA threads and blocks

n = 128000

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

Uses whatever mapping to threads and blocks the compile chooses. Perhaps 16 blocks, 25 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 12 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 12 threads, each thread executes one iteration of the loop, using parallel

Mapping OpenACC to CUDA threads and blocks

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

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

100 thread blocks, each with
apparently 1 thread, each threa
redundantly executes the loop

Mapping OpenACC to CUDA threads and blocks

```
n = 12800;

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

#pragma acc kernels loop gang(50), vector(128)
   for( int i = 0; i < n; ++i ) y[i] += a*x[i];
```

100 thread blocks, each with 12 threads, each thread executes one iteration of the loop

50 thread blocks, each with 128 threads. Each thread does two elements worth of work

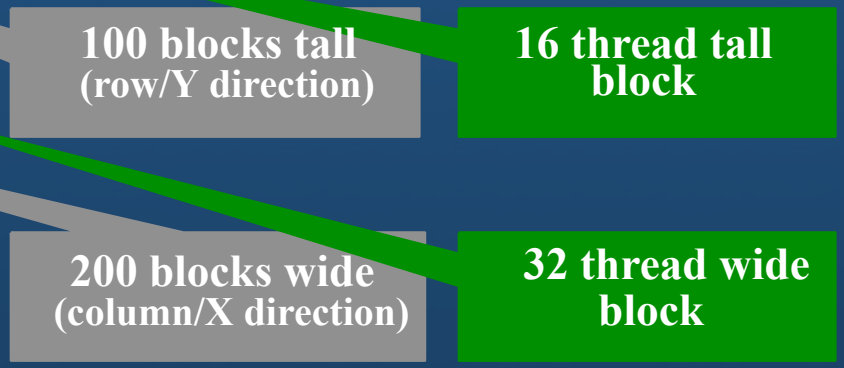
Doing multiple iterations per thread can improve performance by amortizing the cost of setup

Mapping OpenACC to CUDA threads and blocks

Nested loops generate multi-dimensional blocks and grids:

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

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



Jacobi Iteration: OpenACC C v1

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

#pragma acc kernels loop 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, fabs(Anew[j][i] - A[j][i]);
    }
  }

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

  iter++;
```

Jacobi Iteration: OpenACC C v2

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

#pragma acc kernels loop reduction(max:err)
  for( int j = 1; j < n-1; j++) {
#pragma acc loop gang(16) vector(32)
    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, fabs(Anew[j][i] - A[j][i]);
    }
  }
#pragma acc kernels loop
  for( int j = 1; j < n-1; j++) {
#pragma acc loop gang(16) vector(32)
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
```

Leave compiler to choose Y dimension for grids and blocks

Grids are 16 blocks wide, blocks are 32 threads wide

Leave compiler to choose Y dimension for grids and blocks

Grids are 16 blocks wide, blocks are 32 threads wide

Jacobi Iteration: OpenACC C v3

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

#pragma acc kernels loop
  for( int j = 1; j < n-1; j++) {
#pragma acc loop gang(16) vector(32)
    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]);
    }
  }
#pragma acc kernels loop reduction(max:err)
  for( int j = 1; j < n-1; j++) {
#pragma acc loop gang(16) vector(32)
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = 0.25 * (Anew[j][i+1] + Anew[j][i-1] +
                        Anew[j-1][i] + Anew[j+1][i]);
      err = max(err, fabs(A[j][i] - Anew[j][i]);
    }
  }
  iter+=2;
```

Need to switch back to copying Anew in to accelerator so that halo cells will be correct

Can calculate the max reduction on 'error' once per pair, so removed it from this loop

Replace memcpy kernel with second instance of the stenc kernel

Only need half as many times through the loop now

Contents

Further study

- Topics not covered
  - OpenACC `parallel`
  - Using CUDA Libraries with OpenACC, mixing CUDA with OpenACC
  - Import clauses: `async`, `reduction`, and `if`
  - `update`, `declare` and `cache` directive, `host_data`, `deviceptr`
  - `wait` directive
  - OpenACC runtime APIs
  - OpenACC Environment Variables
- The references mentioned before
- Performance tuning and profiling

The end!