Stencil Code in OpenACC
OpenACC Example: Stencil Code

Copy b array in and out of device memory

```c
#pragma acc data copy(b[0:n][0:m]) create(a[0:n][0:m])
{
    for (iter = 1; iter <= p; ++iter)
    {
        #pragma acc kernels
        {
            for (i = 1; i < n-1; ++i)
            {
                for (j = 1; j < m-1; ++j)
                {
                    a[i][j] = w0*b[i][j] + w1*(b[i-1][j]+b[i+1][j]+b[i][j-1]+b[i][j+1]) + w2*(b[i-1][j-1]+b[i-1][j+1]+b[i+1][j-1]+b[i+1][j+1]);
                }
            }
            for (i = 1; i < n-1; ++i)
            {
                for (j = 1; j < m-1; ++j)
                {
                    b[i][j] = a[i][j];
                }
            }
        }
    }
}
```

Calculate new value from neighbors

Swap input/output arrays

a array only exists in device memory

OpenACC Example: Stencil Code

Calculate new value from neighbors

Swap input/output arrays

Copy b array in and out of device memory

```c
#pragma acc data copy(b[0:n][0:m]) create(a[0:n][0:m])
{
    for (iter = 1; iter <= p; ++iter)
    {
        #pragma acc kernels
        {
            for (i = 1; i < n-1; ++i)
            {
                for (j = 1; j < m-1; ++j)
                {
                    a[i][j] = w0*b[i][j] + w1*(b[i-1][j]+b[i+1][j]+b[i][j-1]+b[i][j+1]) + w2*(b[i-1][j-1]+b[i-1][j+1]+b[i+1][j-1]+b[i+1][j+1]);
                }
            }
            for (i = 1; i < n-1; ++i)
            {
                for (j = 1; j < m-1; ++j)
                {
                    b[i][j] = a[i][j];
                }
            }
        }
    }
}
```
Example: Jacobi Iteration for Poisson Equation

- Poisson equation has a simple form in 2D
  \[ u_{xx} + u_{yy} = f(x,y) \]
- Applications include
  - Electricity
  - Magnetism
  - Gravity
  - Heat distribution
  - Fluid flow
  - Torsion
- When \( f(x,y)=0 \) we call it Laplace equation
  \[
  u_{xx} = \frac{\partial^2 u}{\partial x \partial x} \approx \frac{u(x+h,y) - 2u(x,y) + u(x-h,y)}{h^2}
  \]
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++ ) {
        }
        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;
    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++;
}
OpenACC Directive Syntax

- C/C++
  
  ```
  #pragma acc directive [clause [,] clause] ...
  ```
  ... often followed by a structured code block

- Fortran
  
  ```
  !$acc directive [clause [,] clause] ...
  ```
  ... often paired with a matching end directive surrounding a structured code block:
  ```
  !$acc end directive
  ```
OpenACC Parallel Directive

Generates parallelism

```c
#pragma acc parallel
{
    When encountering the `parallel` directive, the compiler will generate 1 or more parallel gangs, which execute redundantly.
}
```
The loop directive informs the compiler which loops to parallelize.
OpenACC Parallel Loop Directive

Generates parallelism and identifies loop in one directive.

```c
#pragma acc parallel loop
for (i=0; i<N; ++i)
{
}
```

The parallel and loop directives are frequently combined into one.
Looking for Parallelism

while ( err > tol && iter < iter_max ) {
    err=0.0;
    #pragma acc parallel loop reduction(max:err)
    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]));
        }
    }
    #pragma acc parallel loop
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}

A reduction means that all n*m values for err will be reduced to just one, the max.
OpenACC Loop Directive: PRIVATE & REDUCTION

- The **private** and **reduction** clauses are not optimization clauses, they may be required for correctness.
- **Private** = a copy of the variable is made for each loop iteration
- **Reduction** = a reduction is performed on the listed variables
  - Supports: +, *, max, min, and logical operators
Building OpenACC Code with PGI Compiler

$ pgcc -fast -acc -ta=tesla -Minfo=all laplace2d.c
main:
40, Loop not fused: function call before adjacent loop
   Generated vector sse code for the loop
51, Loop not vectorized/parallelized: potential early exits
55, Accelerator kernel generated
   55, Max reduction generated for error
   56, #pragma acc loop gang /* blockIdx.x */
   58, #pragma acc loop vector(256) /* threadIdx.x */
55, Generating copyout(Anew[1:4094][1:4094])
Generating copyin(A[:][:])
Generating Tesla code
58, Loop is parallelizable
66, Accelerator kernel generated
   67, #pragma acc loop gang /* blockIdx.x */
   69, #pragma acc loop vector(256) /* threadIdx.x */
66, Generating copyin(Anew[1:4094][1:4094])
Generating copyout(A[1:4094][1:4094])
Generating Tesla code
69, Loop is parallelizable
Performance Results: Speedup (Higher is Better)

- Question:
  - Why did OpenACC had a slow down on GPUs?

- Answer:
  - Very low compute/memcpy ratio
Hardware: CPU vs. GPU

- **Main Memory RAM**: DDR3 or DDR4
  - Size: ~500 GiB
  - Bus width: 64 bits
  - Speed: ~100 GB/s

- **GPU Memory RAM**:
  - GPU Chip 0: GDDR5
    - Size: ~10 GiB
    - Bus width: 384 bits
    - Speed: ~200 GB/s
  - GPU Chip 1: HBM2
    - Size: 32 GiB
    - Bus width: 4096 bits
    - Speed: 700..900 GB/s

- **PCIexpress**
  - Speed: 15 GB/s
  - Long latency

- **GPU generations**
  - Kepler
  - Pascal/Volta

- **QPI**

- **NVLINK**
while ( err > tol && iter < iter_max ) {
    err = 0.0;
}

#pragma acc parallel loop
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]));
    }
}

These copies happen every iteration of the outer while loop!

A, Anew resident on host

A, Anew resident on host

A, Anew resident on accelerator

A, Anew resident on accelerator

A, Anew resident on accelerator

A, Anew resident on accelerator

A, Anew resident on host

A, Anew resident on host

A, Anew resident on accelerator
Looking for Parallelism

while ( err > tol && iter < iter_max ) {
    err=0.0;
#pragma acc parallel 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, abs(Anew[j][i] - A[j][i]));
        }
    }
#pragma acc parallel loop
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
The `data` directive defines a region of code in which GPU arrays remain on the GPU and are shared among all kernels in that region.

```c
#pragma acc data
{
#pragma acc parallel loop
/* . . . */
#pragma acc parallel loop
/* . . . */
}
```

Arrays used within the data region will remain on the GPU until the end of the data region.
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.

- **deviceptr( list )**
  - The variable is a device pointer (e.g. CUDA) and can be used directly on the device.
Array Shaping

- Compiler sometimes cannot determine size of arrays
  - Must specify explicitly using data clauses and array “shape”
- C/C++
  - #pragma acc data copyin(a[0:nelem]) copyout(b[s/4:3*s/4])
- Fortran
  - !$acc data copyin(a(1:end)) copyout(b(s/4:3*s/4))
- Note: data clauses can be used on data, parallel, or kernels
Data Regions Have Real Consequences

```c
int main(int argc, char** argv) {
    float A[1000];
    #pragma acc kernels
    for (int iter = 1; iter < 1000; ++iter){
        A[iter] = 1.0;
    }
    A[10] = 2.0;
}
```

Output: A[10] = 2.0
Looking for Parallelism

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

#pragma acc parallel loop reduction(max:err)
    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]));
        }
    }

#pragma acc parallel loop
    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 to/from the accelerator only when needed.
Create Anew as a device temporary.
$ pgcc -fast -acc -ta=tesla -Minfo=all laplace2d.c
main:
40, Loop not fused: function call before adjacent loop
   Generated vector sse code for the loop
51, Generating copy(A[:][:])[=]
   Generating create(Anew[:][:])[=]
   Loop not vectorized/parallelized: potential early exits
56, Accelerator kernel generated
   56, Max reduction generated for error
   57, #pragma acc loop gang /* blockIdx.x */
   59, #pragma acc loop vector(256) /* threadIdx.x */
56, Generating Tesla code
59, Loop is parallelizable
67, Accelerator kernel generated
   68, #pragma acc loop gang /* blockIdx.x */
   70, #pragma acc loop vector(256) /* threadIdx.x */
67, Generating Tesla code
70, Loop is parallelizable
The OpenACC loop Directive

• The **loop** directive gives the compiler additional information about the loop following the directive in the source code.
  
  - **independent**
    • All iterations of the loop are independent
    • This clause is implied when parent pragma is **parallel** construct
    • If unsure, use **auto** clause to have the compiler analyze the loop
    • For sequential execution use **seq** clause
  
  - **collapse(K)**
    • Turn the next K loops into one, flattened loop
  
  - **tile(N[,M,...])**
    • Break the next 1 or more loops into tiles based on the provided dimensions
    • This can be combined with **gang**, **worker**, and **vector** clauses for different mapping of tiles
Fine Tuning Loop Execution

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

#pragma acc parallel loop device_type(nvidia) tile(32,4)
  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]));
    }
  }

#pragma acc parallel loop device_type(nvidia) tile(32,4)
  for( int j = 1; j < n-1; j++ ) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

The next two loops will be tiled into 32x4 blocks on NVIDIA GPUs

Performance improvements

- K40: 14.92x → 15.46x
- P100: 34.71x → 36.78x