INTRODUCTION OF OPENACC FOR DIRECTIVES-BASED GPU ACCELERATION

Jeff Larkin, NVIDIA Developer Technologies

AGENDA

- Accelerated Computing Basics
- What are Compiler Directives?
- Accelerating Applications with OpenACC
  - Identifying Available Parallelism
  - Exposing Parallelism
  - Optimizing Data Locality
- Next Steps
ACCELERATED COMPUTING BASICS
WHAT IS ACCELERATED COMPUTING?

Application Execution

High Serial Performance

High Data Parallelism

CPU

+ GPU
SIMPLICITY & PERFORMANCE

- **Accelerated Libraries**
  - Little or no code change for standard libraries; high performance
  - Limited by what libraries are available

- **Compiler Directives**
  - High Level: Based on existing languages; simple and familiar
  - High Level: Performance may not be optimal

- **Parallel Language Extensions**
  - Expose low-level details for maximum performance
  - Often more difficult to learn and more time consuming to implement
CODE FOR PORTABILITY & PERFORMANCE

 Libraries
• Implement as much as possible using portable libraries.

 Directives
• Use directives to implement portable code.

 Languages
• Use lower level languages for important kernels.
WHAT ARE COMPILER DIRECTIVES?
WHAT ARE COMPILER DIRECTIVES?

int main() {
    do_serial_stuff()
    for(int i=0; i < BIGN; i++)
    {
        ...compute intensive work
    }
    do_more_serial_stuff();
}
OPENACC: THE STANDARD FOR GPU DIRECTIVES

- **Simple:** Easy path to accelerate compute intensive applications
- **Open:** Open standard that can be implemented anywhere
- **Portable:** Represents parallelism at a high level making it portable to any architecture
OPENACC MEMBERS AND PARTNERS
ACCELERATING APPLICATIONS WITH OPENACC
Identify Available Parallelism

Parallelize Loops with OpenACC

Optimize Data Locality

Optimize Loop Performance
EXAMPLE: 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: \( \nabla^2 f(x,y) = 0 \)

\[
A^{k+1}(i,j) = \frac{A^k(i-1,j) + A^k(i+1,j) + A^k(i,j-1) + A^k(i,j+1)}{4}
\]
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++;
}
Identify Available Parallelism

Parallelize Loops with OpenACC

Optimize Data Locality

Optimize Loop Performance
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++;
}
Identify Available Parallelism

Parallelize Loops with OpenACC

Optimize Data Locality

Optimize Loop Performance
OPENACC DIRECTIVE SYNTAX

- **C/C++**

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

- **Fortran**

```fortran
!$acc directive [clause [,] clause] ...
```
...often paired with a matching end directive surrounding a structured code block:
```fortran
!$acc end directive
```

Don’t forget `acc`
OPENACC PARALLEL LOOP DIRECTIVE

**parallel** - Programmer identifies a block of code containing parallelism. Compiler generates a *kernel*.

**loop** - Programmer identifies a loop that can be parallelized within the kernel.

NOTE: parallel & loop are often placed together

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

**Kernel:**
A function that runs in parallel on the GPU
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++;
}
$ 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
The **kernels** construct expresses that a region *may contain parallelism* and *the compiler determines* what can safely be parallelized.

```c
#pragma acc kernels
{
    for(int i=0; i<N; i++)
    {
        x[i] = 1.0;
        y[i] = 2.0;
    }
    for(int i=0; i<N; i++)
    {
        y[i] = a*x[i] + y[i];
    }
}
```

The compiler identifies 2 parallel loops and generates 2 kernels.
while ( err > tol && iter < iter_max ) {
    err=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]);

            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++;
}
$ pgcc -fast -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, Generating copyout(Anew[1:4094][1:4094])
   Generating copyin(A[:][:])
   Generating copyout(A[1:4094][1:4094])
   Generating Tesla code

57, Loop is parallelizable

59, Loop is parallelizable
   Accelerator kernel generated
   57, #pragma acc loop gang /* blockIdx.y */
   59, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
   63, Max reduction generated for error

67, Loop is parallelizable

69, Loop is parallelizable
   Accelerator kernel generated
   67, #pragma acc loop gang /* blockIdx.y */
   69, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
OPENACC PARALLEL LOOP VS. KERNELS

PARALLEL LOOP

• Requires analysis by programmer to ensure safe parallelism
• Will parallelize what a compiler may miss
• Straightforward path from OpenMP

KERNELS

• Compiler performs parallel analysis and parallelizes what it believes safe
• Can cover larger area of code with single directive
• Gives compiler additional leeway to optimize.

Both approaches are equally valid and can perform equally well.
Why did OpenACC slow down here?

Intel Xeon E5-2698 v3 @ 2.30GHz (Haswell) vs. NVIDIA Tesla K40
Very low Compute/Memcopy ratio

Compute: 5.0s
Memory Copy: 62.2s
EXCESSIVE DATA TRANSFERS

while ( err > tol && iter < iter_max )
{
    err=0.0;
    #pragma acc kernels
    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]));
        }
    }...
}...
while ( err > tol && iter < iter_max ) {
    err=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]);
                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++;
}
Identify Available Parallelism

Parallelize Loops with OpenACC

Optimize Loop Performance

Optimize Data Locality
The **data** construct 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

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy (list)</td>
<td>Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.</td>
</tr>
<tr>
<td>copyin (list)</td>
<td>Allocates memory on GPU and copies data from host to GPU when entering region.</td>
</tr>
<tr>
<td>copyout (list)</td>
<td>Allocates memory on GPU and copies data to the host when exiting region.</td>
</tr>
<tr>
<td>create (list)</td>
<td>Allocates memory on GPU but does not copy.</td>
</tr>
<tr>
<td>present (list)</td>
<td>Data is already present on GPU from another containing data region.</td>
</tr>
</tbody>
</table>

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/C++**

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

**Fortran**

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

        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++;
  }
}
$ 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
VISUAL PROFILER: DATA REGION

Iteration 1

Was 104ms

Iteration 2
Intel Xeon E5-2698 v3 @ 2.30GHz (Haswell) vs. NVIDIA Tesla K40

Single Thread: 1.00X
2 Threads: 1.82X
4 Threads: 3.13X
6 Threads: 3.90X
8 Threads: 4.38X
OpenACC: 27.30X

Socket/Socket: 6.24X

Speed-Up (Higher is Better)
OPENACC PRESENT CLAUSE

It’s sometimes necessary for a data region to be in a different scope than the compute region.

When this occurs, the `present` clause can be used to tell the compiler data is already on the device.

Since the declaration of A is now in a higher scope, it’s necessary to shape A in the present clause.

High-level data regions and the present clause are often critical to good performance.

```c
function main(int argc, char **argv)
{
    #pragma acc data copy(A)
    {
        laplace2D(A,n,m);
    }
}

function laplace2D(double[N][M] A,n,m)
{
    #pragma acc data present(A[n][m]) create(Anew)
    while ( err > tol && iter < iter_max ) {
        err=0.0;
        ... 
    }
}
```
Identify Available Parallelism

Optimize Loops with OpenACC

Parallelize Loops with OpenACC

Optimize Loop Performance

Optimize Data Locality

Watch S5195 - Advanced OpenACC Programming on gputechconf.com
NEXT STEPS
1. **Identify Available Parallelism**
   - What important parts of the code have available parallelism?

2. **Parallelize Loops**
   - Express as much parallelism as possible and ensure you still get correct results.
   - Because the compiler *must* be cautious about data movement, the code will generally slow down.

3. **Optimize Data Locality**
   - The programmer will *always* know better than the compiler what data movement is unnecessary.

4. **Optimize Loop Performance**
   - Don’t try to optimize a kernel that runs in a few *us* or *ms* until you’ve eliminated the excess data motion that is taking *many seconds*. 
TYPICAL PORTING EXPERIENCE WITH OPENACC DIRECTIVES

Step 1: Identify Available Parallelism
Step 2: Parallelize Loops with OpenACC
Step 3: Optimize Data Locality
Step 4: Optimize Loops

Application Speed-up
Development Time
FOR MORE INFORMATION

- Check out [http://openacc.org/](http://openacc.org/)
- Share your successes at WACCPD at SC15.

- Email me: [jlarkin@nvidia.com](mailto:jlarkin@nvidia.com)
GPU strategies for the point_solve_5 kernel
POINT_SOLVE_5
PERFORMANCE COMPARISON

- CPU: One socket E5-2690 @ 3Ghz, 10 cores
- GPU: K40c, boost clocks, ECC off
- Dataset: DPW-Wing, 1M cells
- One call of point_solve5 over all colors
- No transfers
- 1 CPU core: 300ms
- 10 CPU cores: 44ms
!$acc parallel loop private(f1, f2, f3, f4, f5)
rhs_solve : do n = start, end
    [...]
    istart = iam(n)
    iend = iam(n+1)
    do j = istart, iend
        icol = jam(j)
        f1 = f1 - a_off(1,1,j)*dq(1,icol)
        f2 = f2 - a_off(2,1,j)*dq(1,icol)
        [...22 lines]
        f5 = f5 - a_off(5,5,j)*dq(5,icol)
    end do
    [...]
end do
OPENACC1 - A_OFF ACCESS PATTERN
PERFORMANCE COMPARISON

- CPU: 44 milliseconds
- OpenACC1 - L2: 78 milliseconds
- OpenACC1 - L1: 141 milliseconds
- OpenACC1 - tex: 22 milliseconds
!$acc parallel loop collapse(2) private(fk)

rhs_solve : do n = start, end
  do k = 1,5
    ...]
    istart = iam(n)
    iend = iam(n+1)
    do j = istart, iend
      icol = jam(j)
      fk = fk - a_off(k,1,j)*dq(1,icol)
    [... 3 lines]
      fk = fk - a_off(k,5,j)*dq(1,icol)
    end do
  end do
end do

dq(k,n) = fk

end do

[Split off fw/bw substitution in extra loop]
OPENACC2 - A_OFF ACCESS PATTERN
CUDA FORTRAN - ADVANTAGES

- Shared Memory: as explicitly managed cache and for cooperative reuse
- Inter thread communication in a thread block with shared memory
- Inter thread communication in a warp with __shfl() intrinsic
! Calculate n, l, k based on threadIdx

! Loop over a_off entries
istart = iam(n)
iend = iam(n+1)
do j = istart, iend
    ftemp = ftemp - a_off(k,l,j)*dq(l,jam(j))
end do

! Reduction along the rows
fk = ftemp - __shfl( ftemp, k+1*5)
fk = fk - __shfl( ftemp, k+2*5)
fk = fk - __shfl( ftemp, k+3*5)
fk = fk - __shfl( ftemp, k+4*5)
CUDA FORTRAN - 25 WIDE

warp

- active
- done
- cached
- uncached
PERFORMANCE COMPARISON

milliseconds

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>OpenACC1</th>
<th>OpenACC2</th>
<th>CUDA Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>44</td>
<td>22</td>
<td>18</td>
<td>11.5</td>
</tr>
</tbody>
</table>
FUN3D CONCLUSIONS

- Unchanged code with OpenACC: 2.0x
- Modified code with OpenACC: 2.4x, modified code runs 50% slower on CPUs
- Highly optimized CUDA version: 3.7x
- Compiler options (e.g. how memory is accessed) have huge influence on OpenACC results
- Possible compromise: CUDA for few hotspots, OpenACC for the rest
- Very good OpenACC/CUDA interoperability: CUDA can use buffers managed with OpenACC data clauses
- Unsolved problem: data transfer in a partial port cause overhead