Programming Accelerators using Directives

Credits: Introduction to OpenACC and toolkit – Jeff Larkin, Nvidia – Oct 2015
Heterogeneous Parallel Computers

• Composed of
  – CPU(s)
    • Low-latency processor optimized for sequential execution
    • large memory size and deep memory hierarchy
  – Accelerator(s)
    • high throughput SIMD or MIMD processors optimized for data-parallel execution
    • high performance limited memory size and small depth memory hierarchy

• Example
  – Multisocket compute server
    • Host: two-socket 20 – 40 Intel Xeon cores with 128 – 512 GB CC-NUMA shared memory
    • Accelerators: 1-8 accelerators (e.g. Nvidia Cuda cards, Intel Xeon Phi cards) connected via PCIe x16 interfaces (16GB/s)
      – host controls data to/from accelerator memory
Basic Programming Models

- **Offload model**
  - idea: offload computational kernels
    - send data to accelerator
    - call kernel(s)
    - retrieve data
  - accelerator-specific compiler support
    - Cuda compiler (nvcc) for Nvidia GPUs
    - Intel vectorizing compiler (icc -mmic) for Intel Xeon Phi KNL
      - #pragma offload target(mic:n) in(...) out(...) inout(...)
  - accelerator-neutral OpenCL → OpenACC
    - Cuda-like notation
    - OpenACC compiler can target Nvidia or Intel Xeon Phi (RIP)
Emerging Programming Models

• directive model
  – idea: identify sections of code to be compiled for accelerator(s)
    • data transfer and kernel invocation generated by compiler

  – accelerator-neutral efforts
    • OpenACC
      – #pragma acc parallel loop
        for (...) { ... }
      – gang, worker, vector (threadblock, warp, warp in SIMT lockstep)
      – gcc 5, PGI, Cray, CAPS, Nvidia compilers
    • OpenMP 5.1
      – similar directives to (but more general than) OpenACC
      – includes all previous shared memory parallelization directives
      – implemented by gcc 11.2
Matrix multiplication

- library implementations

**Figure 2.** FP32 performance comparison of cuBLAS, Intel MKL and CUDA on Tesla V100
Matrix multiplication
• using parallelization directives

Figure 3. Dependence of FP32 performance on matrix sizes and programming model on Tesla V100
Introduction to OpenACC

Jeff Larkin, NVIDIA Developer Technologies
Why OpenACC?
OpenACC
Simple | Powerful | Portable

Fueling the Next Wave of Scientific Discoveries in HPC

main()
{
    <serial code>
    #pragma acc kernels
    // automatically runs on GPU
    {
        <parallel code>
    }
}

University of Illinois
PowerGrid- MRI Reconstruction

70x Speed-Up
2 Days of Effort

RIKEN Japan
NICAM- Climate Modeling

7-8x Speed-Up
5% of Code Modified

8000+
Developers
using OpenACC

http://www.cray.com/sites/default/files/resources/OpenACC_213462.12_OpenACC_Cosmo_CS_FNL.pdf
http://www.openacc.org/content/experiences-porting-molecular-dynamics-code-gpus-cr-ay-xk7
OpenACC Directives

- Incremental
- Single source
- Interoperable
- Performance portable
- CPU, GPU, MIC

```c
#pragma acc data copyin(a,b) copyout(c)
{
    ...
    #pragma acc parallel
    {
        #pragma acc loop gang vector
        for (i = 0; i < n; ++i) {
            z[i] = x[i] + y[i];
            ...
        }
    }
    ...
}
```
Accelerated Computing Fundamentals
Accelerated Computing
10x Performance & 5x Energy Efficiency for HPC

CPU
Optimized for Serial Tasks

GPU Accelerator
Optimized for Parallel Tasks
What is Heterogeneous Programming?

- Compute-Intensive Functions: A few % of Code, A large % of Time
- Rest of Sequential CPU Code

GPU + CPU
Portability & Performance

Accelerated Libraries
- High performance with little or no code change
- Limited by what libraries are available

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

Parallel Language Extensions
- Greater flexibility and control for maximum performance
- Often less portable and more time consuming to implement
Code for Portability & Performance

- Libraries
  - Implement as much as possible using portable libraries

- Directives
  - Use directives for rapid and portable development

- Languages
  - Use lower level languages for important kernels
OpenACC Programming Cycle
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 Parallelism

```c
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]));
        }
    }
    iter++;
}
```

Data dependency between iterations.

Independent loop iterations

Independent loop iterations
OpenACC kernels Directive

The kernels directive identifies a region that may contain loops that the compiler can turn into parallel kernels.

```
#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.
Parallelize with OpenACC kernels

```c
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++;
}
```

Look for parallelism within this region.
Building the code

$ 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 */
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 seconds
- **Memory Copy**: 62 seconds
PCIe Copies

104ms/iteration
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]));
    }
  }
  ...
}

A, Anew resident on host

These copies happen every iteration of the outer while loop!

A, Anew resident on accelerator

Copy

Copy

Copy

A, Anew resident on host

A, Anew resident on accelerator
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
Optimize Loop Performance
Express Data Movement
Express Parallelism
Data regions

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.

```plaintext
#pragma acc data
{
    #pragma acc kernels
    ...
    #pragma acc kernels
    ...
}
```

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
Express Data Locality

```c
#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++;
}
```

Copy A to/from the accelerator only when needed.
Create Anew as a device temporary.
Rebuilding the code

```bash
$ 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
Speed-Up (Higher is Better)

Intel Xeon E5-2698 v3 @ 2.30GHz (Haswell)

vs.

NVIDIA Tesla K40

Single Thread: 1.00X
2 Threads: 1.90X
4 Threads: 3.20X
6 Threads: 3.74X
8 Threads: 3.83X
OpenACC: 19.89X

Socket/Socket: 5.2X
Identify Available Parallelism

Optimize Loop Performance

Express Parallelism

Express Data Movement
The loop Directive

The `loop` directive gives the compiler additional information about the next loop in the source code through several clauses.

- **independent** - all iterations of the loop are independent
- **collapse(N)** - turn the next N loops into one, flattened loop
- **tile(N[,M,...])** - break the next 1 or more loops into tiles based on the provided dimensions.

These clauses and more will be discussed in greater detail in a later class.
Optimize Loop Performance

```c
#pragma acc data copy(A) create(Anew)
while ( err > tol && iter < iter_max ) {
    err=0.0;
#pragma acc kernels
{
#pragma acc loop device_type(nvidia) tile(32,4)
    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 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++;
}
```

“Tile” the next two loops into 32x4 blocks, but only on NVIDIA GPUs.
Intel Xeon E5-2698 v3 @ 2.30GHz (Haswell) vs. NVIDIA Tesla K40

Speed-Up (Higher is Better)

Single Thread  2 Threads  4 Threads  6 Threads  8 Threads  OpenACC  OpenACC Tuned

1.00X  3.20X  3.74X  3.83X  19.89X  21.22X
The OpenACC Toolkit
Introducing the New OpenACC Toolkit
Free Toolkit Offers Simple & Powerful Path to Accelerated Computing

- PGI Compiler
  Free OpenACC compiler for academia

- NVProf Profiler
  Easily find where to add compiler directives

- GPU Wizard
  Identify which GPU libraries can jumpstart code

- Code Samples
  Learn from examples of real-world algorithms

- Documentation
  Quick start guide, Best practices, Forums

http://developer.nvidia.com/openacc