Parallel Computing with OpenACC

Wei Feinstein
HPC User Services

Parallel Programming Workshop 2017
Louisiana State University
AGENDA

• Fundamentals of Heterogeneous & GPU Computing
• What are Compiler Directives?
• Accelerating Applications with OpenACC
  • Identify Available Parallelism
  • Parallelize loops
  • Optimize Data Locality
  • Optimize loops
• Interoperability
Large Scale of applications

Genomics

Deep Learning

Costal Storm Prediction

Multi-Core CPU

GPU

Many-Core CPU (Intel Xeon Phi)

OpenMP

OpenACC

Phi Directives

High Level Programming (directive/pragma)
GPU Computing History

• The first Graphics Processing Unit (GPU) was designed as graphics accelerators, supporting only specific fixed-function pipelines.
• Starting in the late 1990s, the hardware became increasingly programmable, NVIDIA's first GPU in 1999.
• The General Purpose GPU (GPGPU): its excellent floating point performance.
• 2006 world's 1st solution for general computing on GPUs. CUDA was designed and launched by NVIDIA
• CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model created by NVIDIA and implemented on the GPUs that they produce.
Latency Processor

Throughput processor
Latency vs. Throughput

**Koenigsegg one**
- 270 mph
- Baton Rouge to New Orleans in 0.29 hr (18 mins)
- Seats: 2

**School bus**
- 40 mph
- BR to NO in 2 hr
- Seats: 72
Latency vs. Throughput

Koenigsegg one
- latency: 0.28 hr (18 mins)
- Throughput: $2/0.28 = 7.14$ people/hr

School bus
- Latency: 2 hr
- Throughput: $72/2 = 36$ people/hr
Comparison of Architectures

**CPU**
- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

**GPU**
- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation
- Hide latency from other threads via fast context switching
WHAT IS HETEROGENEOUS COMPUTING?

Application Execution

High Serial Performance

High Data Parallelism

CPU

GPU

6th HPC Parallel Programming Workshop
Parallel Computing with OpenACC
3 Approaches to Heterogeneous Programming

- **Libraries**: Easy to use, Most Performance
- **Compiler Directives**: Easy to use, Portable code
- **Programming Languages**: Most Performance, Most Flexibility
Examples of GPU-accelerated Libraries

- NVIDIA cuBLAS
- NVIDIA cuRAND
- NVIDIA cuSPARSE
- NVIDIA NPP
- GPU VSIPL: Vector Signal Image Processing
- CULA tools: GPU Accelerated Linear Algebra
- MAGMA: Matrix Algebra on GPU and Multicore
- NVIDIA cuFFT
- Rogue Wave Software: IMSL Library
- ArrayFire: Matrix Computations
- CUSP: Sparse Linear Algebra
- Thrust: C++ STL Features for CUDA
GPU Programming Languages

- **Fortran**
  - OpenACC, CUDA Fortran

- **C**
  - OpenACC, CUDA C

- **C++**
  - Thrust, CUDA C++

- **Python**
  - PyCUDA, Copperhead

- **C#**
  - GPU.NET

- **Numerical analytics**
  - MATLAB, Mathematica, LabVIEW
void saxpy(int N, float a, float *x, float *y){
    for (int i = 0; i < N; ++i)
        y[i] = a*x[i] + y[i];
}

x, y: vector with N elements
a: scalar
void saxpy_CPU(int n, float a, float *x, float *y) {
    for (int i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}

int main(int argc, char **argv){
    long n= 1<<20;
    float *x = (float*)malloc(n * sizeof(float));
    float *y = (float*)malloc(n * sizeof(float));
    // Initialize vector x,y
    for (int i = 0; i < n; ++i) {
        x[i] = 1.0f;
        y[i] = 0.0f;
    }
    // Perform SAXPY
    saxpy_CPU(n, a, x, y);
}
**Saxpy_cuBLAS**

```c
extern void
cublasSaxpy(int,float,float*,int,float*,int);

int main(){
    ...
    // Initialize vectors x, y
    for (int i = 0; i < n; ++i) {
        x[i] = 1.0f;
        y[i] = 0.0f;
    }
    // Perform SAXPY
    #pragma acc
    host_data use_device(x,y)
    cublasSaxpy(n, 2.0, x, 1, y, 1);
}
```

**Saxpy_OpenACC**

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

int main(){
    ...
    // Initialize vectors x, y
    for (int i = 0; i < n; ++i) {
        x[i] = 1.0f;
        y[i] = 0.0f;
    }
    // Perform SAXPY
    saxpy_ACC(n, a, x, y);
}
```

[http://docs.nvidia.com/cuda](http://docs.nvidia.com/cuda)
// define CUDA kernel function
__global__ void saxpy_kernel( float a, float* x, float* y, int n ){
    int i;
    i = blockIdx.x*blockDim.x + threadIdx.x;
    if( i <= n ) y[i] = a*x[i] + y[i];
}

void main( float a, float* x, float* y, int n ){
    float *xd, *yd;
    // manage device memory
    cudaMalloc( (void**)&xd, n*sizeof(float) );
    cudaMalloc( (void**)&yd, n*sizeof(float) );
    cudaMemcpy( xd, x, n*sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( yd, y, n*sizeof(float), cudaMemcpyHostToDevice );
    // calls the kernel function
    saxpy_kernel<<< (n+31)/32, 32 >>>( a, xd, yd, n );
    cudaMemcpy( x, xd, n*sizeof(float), cudaMemcpyDeviceToHost );
    // free device memory after use
    cudaFree( xd );
    cudaFree( yd );
}
GPU Tools

Code performance increases with the deployment of GPU tools.
3 Approaches to Heterogeneous Programming

- **Libraries**
  - Easy to use
  - Most Performance

- **Compiler Directives**
  - Easy to use
  - Portable code

- **Programming Languages**
  - Most Performance
  - Most Flexibility
AGENDA

• Fundamentals of Heterogeneous & GPU Computing
• What are Compiler Directives?
• Accelerating Applications with OpenACC
  • Identify Available Parallelism
  • Parallelize loops
  • Optimize Data Locality
  • Optimize loops
• Interoperability
What Are Compiler Directives?

```c
int main() {
    do_serial_stuff();
    #pragma acc parallel loop {
        for(int i=0; i < BIGN; i++)
        {
            ...compute intensive work
        }
    }
    do_more_serial_stuff();
}
```

- Inserts compiler hints to compute on GPU
- Data and Execution returns to the CPU
What are OpenACC Directives?

- Designed for multicore CPUs & many core GPUs
- Portable compiler hints
- Compiler parallelizes code

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

Application Code

$acc parallel

$acc end parallel

Compute-Intensive Functions

Generate Parallel Code for GPU

Rest of Sequential CPU Code

GPU

CPU

Rest of Sequential CPU Code
History of OpenACC

- OpenACC is a specification/standard for high-level, compiler directives to express parallelism on accelerators.
  - Aims to be portable to a wide range of accelerators
  - One specification for multiple vendors, multiple devices
  - Original members: CAPS, Cray, NVIDIA and PGI
- First released 1.0 Nov 2011
- 2.0 was released Jue 2013
- 2.5 was released Oct 2015

http://www.openacc.org/specification
Why OPENACC?

- **Simple**: Directives are the easy path to accelerate applications compute intensive

- **Open**: OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors

- **Portable**: GPU Directives represent parallelism at a high level, allowing portability to a wide range of architectures with the same code
Which Compilers Support OpenACC

- PGI compilers for C, C++ and Fortran
- Cray CCE compilers for Cray systems
- CAPS compilers
- NVIDIA
Using PGI compilers on Mike

Login in to SuperMike:
$ ssh userid@mike.hpc.lsu.edu

Get an interactive compute node:
$ qsub -I -l nodes=1:ppn=16 -l walltime=4:00:00
    -q shelob -A hpc_train_2017

Add the PGI compiler v15.10
$ soft add +portland-15.10

$ pgcc -V
Example: Jacobi Iteration

Iteratively converges to correct value, e.g., temperature, by computing new values at each point from the average of neighboring points.

\[ A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4} \]

- Example: Solve Laplace equation in 2D:

\[ \nabla^2 f(x, y) = 0 \]
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++ ) {
            error = fmax( error, fabs(Anew[j][i] - A[j][i]) ) ;
        }
    }
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
        }
    }
    iter++;
}  // end while loop
AGENDA

- Fundamentals of Heterogeneous & GPU Computing
- What are Compiler Directives?
- Accelerating Applications with OpenACC
  - Identify Available Parallelism
  - Parallelize loops
  - Optimize Data Locality
  - Optimize loops
  - Interoperability
Identify Available Parallelism

• Identify section of a code consuming the significant percentage of time (hot spots)
  • Routines, loops

• Profilers:
  • gpof (GNU)
  • pgprof (PGI)
  • Vampir
  • NVIDIA visual profiler
Code Profiling (pgprof)

- Compile code with profiling info
  - `$ pgcc -Mprof=ccff laplace.c`

- Generate pgprof.out
  - `$ pgcollect ./a.out`  
    --> pgprof.out

- Visualize profile info
  - `$ pgprof -exe ./a.out`
int iter = 0;

start_time = omp_get_wtime();
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 = fmax( error, fabs(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];
        }        
    }

    if(iter % 100 == 0) printf("5d, 5d, \n", iter, error);
    iter++;
}

end_time = omp_get_wtime();
printf("total time in sec: %f\n", end_time - start_time);
AGENDA

• Fundamentals of Heterogeneous & GPU Computing
• What are Compiler Directives?
• Accelerating Applications with OpenACC
  • Identify Available Parallelism
  • Parallelize loops
  • Optimize Data Locality
  • Optimize loops
  • Interoperability
General Directive Syntax and Scope

- **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
Kernels Directive

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;
  {
    y[i] = a*x[i] + y[i];
  }
}
```

A kernel is a function executed on the GPU as an array of threads in parallel.
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++ ) {
                error = fmax( error, fabs(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++;
}
Parallize with Kernels (Fortran)

```fortran
do while ( error .gt. tol .and. iter .lt. iter_max )
  error=0.0
  !$acc kernels
  do j=1,m-2
    do i=1,n-2
      Anew(i,j) = 0.25_fp_kind * ( A(i+1,j ) + A(i-1,j ) + &
                                 A(i  ,j-1) + A(i  ,j+1) )
      error = max( error, abs(Anew(i,j)-A(i,j)) )
    end do
  end do
  do j=1,m-2
    do i=1,n-2
      A(i,j) = Anew(i,j)
    end do
  end do
  !$acc end kernels
end do
```

Look for parallelism within this region.

```
pgfortran -acc -fast -ta=nvidia,time -Minfo=all laplace_kernels1.f90
```

Kernels end here
How to compile with OpenACC

Compile using PGI compiler

$ pgcc -acc -fast -ta=nvidia -Minfo=accel laplace_kernels1.c

$ pgf90 -acc -fast -ta=nvidia -Minfo=accel laplace_kernels1.f90
Compiler-Generated Info

47, Generating copyout(Anew[1:4094][1:4094])
Generating copyin(A[:4096][:4096])
Generating copyout(A[1:4094][1:4094])
48, Loop is parallelizable
49, Loop is parallelizable

Accelerator kernel generated
Generating Tesla code
48, #pragma acc loop gang /* blockIdx.y */
49, #pragma acc loop gang, vector(128) /*
blockIdx.x threadIdx.x */

52, Max reduction generated for error
56, Loop is parallelizable
57, Loop is parallelizable

Accelerator kernel generated
Generating Tesla code
56, #pragma acc loop gang /* blockIdx.y */
57, #pragma acc loop gang, vector(128) /*
blockIdx.x threadIdx.x */
Check CPU & GPU Utilization

Open a separate terminal
$ ssh -X mikexxx or shelobxxx

$ top: CPU utilization

$ nvidia-smi: GPU
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
Loop Clauses: Private & Reduction

**private**
- A copy of the variable is made for each loop iteration. It is private by default

**reduction**
- A private copy of the affected variable is generated for each loop iteration
- A reduction is then performed on the variables.
- Supports +, *, max, min, and various logical operations
while ( error > tol && iter < iter_max ) {
    error = 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++ ) {
            error = fmax( error, fabs(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++;
}
do while ( error .gt. tol .and. iter .lt. iter_max )
  error=0.0
  !$acc parallel loop reduction(max:err)
  do j=1,m-2
    do i=1,n-2
      Anew(i,j) = 0.25 *( A(i+1,j)+A(i-1,j) + A(i,j-1) + A(i,j+1) )
      error = max( error, abs(Anew(i,j)-A(i,j)) )
    end do
  end do
  !$acc end parallel

  !$acc parallel loop
  do j=1,m-2
    do i=1,n-2
      A(i,j) = Anew(i,j)
    end do
  end do
  !$acc end parallel
Kernels vs. Parallel Loops

**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.

**Parallel Loop**
- Requires analysis by programmer to ensure safe parallelism
- Will parallelize what a compiler may miss
- Straightforward path from OpenMP
OpenACC for Multicore CPUs

- Originally targeted to NVIDIA and AMD (GPUs)
- Latest PGI compiler (>=pgi15.10) generates parallel code on CPUs using OpenMP
- With the same set of OpenACC pragmas
  - GPUs: -ta=nvidia
  - CPUs: -ta=multicore using all the CPU cores
  - Or export ACC_NUM_CORES = [ 1 .. 16 ]

- $ pgcc -ta=multicore,time -Minfo=all laplace_kernels2.c
OpenACC for Multicore CPUs

48, Loop is parallelizable
Generating Multicore code
48, #pragma acc loop gang

49, Loop is parallelizable

56, Loop is parallelizable
Generating Multicore code
56, #pragma acc loop gang

57, Loop is parallelizable
Memory copy idiom, loop replaced by call to __c_mcopy8
Performance Comparison

The graph compares completion times for serial and parallel executions with increasing numbers of cores. The data suggests that there are disappointing performance outcomes when using OpenACC with kernels/parallel loops.
What Went Wrong?

$ export PGI_ACC_TIME=1
$ pgcc -ta=nvidia -Minfo=accel laplace_parallel2.c -o laplace_parallel2

$ pgcc -ta=nvidia,\texttt{time} -Minfo=accel laplace_parallel2.c -o laplace_parallel2

$ ./laplace_parallel2
Data Movement

main NVIDIA devicenum=0
time(us): **88,667,910**

47: compute region reached 1000 times
   47: data copyin transfers: 1000
       device time(us): total=8,667 max=37 min=8 avg=8

47: kernel launched 1000 times
   grid: [4094]  block: [128]
   device time(us): total=2,320,918 max=2,334 min=2,312 avg=2,320
   elapsed time(us): total=2,362,785 max=2,471 min=2,353 avg=2,362

47: reduction kernel launched 1000 times
   grid: [1]  block: [256]
   device time(us): total=14,001 max=15 min=14 avg=14
   elapsed time(us): total=32,924 max=72 min=31 avg=32

47: data copyout transfers: 1000

56: compute region reached 1000 times
   56: kernel launched 1000 times
      grid: [4094]  block: [128]
      device time(us): total=1,802,925 max=1,822 min=1,783 avg=1,802
      elapsed time(us): total=1,847,383 max=1,884 min=1,827 avg=1,847

56: data region reached 2000 times
   56: data copyin transfers: 8000
       device time(us): total=22,000,619 max=4,096 min=2,739 avg=2,750

56: data copyout transfers: 8000
   device time(us): total=20,118,762 max=2,703 min=2,498 avg=2,514

63: data region reached 1000 times
   63: data copyout transfers: 8000
       device time(us): total=20,121,450 max=2,664 min=2,498 avg=2,515

Total time: **88,667,910**
Data transfer time: **84,671,301**
Compute time: **3,996,609**
Performance Profiling by NVVP from Nvidia

- soft add +cuda-7.5.18

$ nvvp ./laplace_parallel2
Parallel Computing with OpenACC
AGENDA

- Fundamentals of Heterogeneous & GPU Computing
- What are Compiler Directives?
- Accelerating Applications with OpenACC
  - Identify Available Parallelism
  - Parallelize loops
  - Optimize Data Locality
  - Optimize loops
- Interoperability
1. Copy input data from CPU memory to GPU memory
Data Flow

1. Copy input data from CPU memory to GPU memory
2. Execute GPU Kernel
1. Copy input data from CPU memory to GPU memory
2. Execute GPU Kernel
3. Copy results from GPU memory to CPU memory
OpenACC Memory Model

Two separate memory spaces between host and accelerator

- Data transfer by DMA transfers
- Hidden from the programmer in OpenACC, so beware:
  - Latency
  - Bandwidth
  - Limited device memory size
while (error > tol && iter < iter_max) {
    error = 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++) {
            error = fmax(error, fabs(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++;
}
How to Improve Data Movement

• Use data/array on GPU as long as possible

• Move data between CUP and GPU as less-frequently as possible

• Don’t copy data back to CPU if not needed on CPU
Define Data Regions

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
    ...
}
```

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.

**present_or_copy [in|out], present_or_create, deviceptr.**
Will be made as default in the future
Array Shaping

Compiler sometimes cannot determine size of arrays, 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 ( 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++ ) {
        error = fmax( error, fabs(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++;
  }
}
Optimizing Data Locality (Fortran)

```fortran
!$acc data copy(A) create(Anew)
  do while ( error .gt. tol .and. iter .lt. iter_max )
    error=0.0_fp_kind
  !$acc kernels
  do j=1,m-2
    do i=1,n-2
      Anew(i,j) = 0.25_fp_kind * ( A(i+1,j ) + A(i-1,j ) + &
                                A(i  ,j-1) + A(i  ,j+1) )
      error = max( error, abs(Anew(i,j)-A(i,j)) )
    end do
  end do
  !$acc end kernels
  do j=1,m-2
    do i=1,n-2
      A(i,j) = Anew(i,j)
    end do
  end do
  !$acc end data
  if(mod(iter,100).eq.0 ) write(*,'(i5,f10.6)'), iter, error
  iter = iter + 1
  !$acc end data
```
main NVIDIA devicenum=0
 time(us): 2,413,950
 43: data region reached 1 time
   43: data copyin transfers: 8
   device time(us): total=22,409 max=2,812 min=2,794 avg=2,801
 48: compute region reached 1000 times
   48: data copyin transfers: 1000
   device time(us): total=21,166 max=54 min=11 avg=21
   48: kernel launched 1000 times
   grid: [4094] block: [128]
   device time(us): total=2,320,508 max=2,336 min=2,310 avg=2,320
   elapsed time(us): total=2,365,313 max=2,396 min=2,355 avg=2,365
   48: reduction kernel launched 1000 times
   grid: [1] block: [256]
   device time(us): total=14,000 max=14 min=14 avg=14
   elapsed time(us): total=33,893 max=67 min=32 avg=33
 48: data copyout transfers: 1000
   device time(us): total=15,772 max=45 min=13 avg=15
 68: data region reached 1 time
  68: data copyout transfers: 9
   device time(us): total=20,095 max=2,509 min=30 avg=2,232
main  NVIDIA  devicenum=0
time(us): 88,667,910
47: compute region reached 1000 times
  47: data copyin transfers: 1000
    device time(us): total=8,667 max=37 min=8 avg=8
  47: kernel launched 1000 times
    grid: [4094]  block: [128]
    device time(us): total=2,320,918 max=2,334 min=2,312 avg=2,320
    elapsed time(us): total=2,362,785 max=2,471 min=2,353 avg=2,362
  47: reduction kernel launched 1000 times
    grid: [1]  block: [256]
    device time(us): total=14,001 max=15 min=14 avg=14
    elapsed time(us): total=32,924 max=72 min=31 avg=32
  47: data copyout transfers: 1000
    device time(us): total=16,973 max=49 min=14 avg=16
47: data region reached 1000 times
47: data copyin transfers: 8000
  device time(us): total=22,404,937 max=2,886 min=2,781 avg=2,800
56: compute region reached 1000 times
56: kernel launched 1000 times
  grid: [4094]  block: [128]
  device time(us): total=1,802,925 max=1,822 min=1,783 avg=1,802
  elapsed time(us): total=1,847,383 max=1,884 min=1,827 avg=1,847
56: data region reached 2000 times
56: data copyin transfers: 8000
  device time(us): total=22,000,619 max=4,096 min=2,739 avg=2,750
56: data copyout transfers: 8000
  device time(us): total=20,118,762 max=2,703 min=2,498 avg=2,514
63: data region reached 1000 times
63: data copyout transfers: 8000
  device time(us): total=20,121,450 max=2,664 min=2,498 avg=2,515
6th HPC Parallel Programming Workshop
Parallel Computing with OpenACC
Performance

![Graph showing completion time in seconds for different cores and OpenACC configurations.](image)
AGENDA

• Fundamentals of Heterogeneous & GPU Computing
• What are Compiler Directives?
• Accelerating Applications with OpenACC
  • Identify Available Parallelism
  • Parallelize loops
  • Optimize Data Locality
• Optimize loops
• Interoperability
OpenACC: 3 Levels of Parallelism

- **Vector** threads work in lockstep (SIMD/SIMT parallelism)
- **Workers** compute a vector
- **Gangs** have 1 or more workers and share resources (such as cache, the streaming multiprocessor, etc.)
- Multiple gangs work independently of each other
CUDA Kernels: Parallel Threads

A kernel is a function executed on the GPU as an array of threads in parallel.

All threads execute the same code, can take different paths.

Each thread has an ID Select input/output data Control decisions.

```plaintext
float x = input[threadIdx.x];
float y = func(x);
output[threadIdx.x] = y;
```
CUDA Kernels: Subdivide into Blocks

- Threads are grouped into blocks
CUDA Kernels: Subdivide into Blocks

- Threads are grouped into blocks
- Blocks are grouped into a grid
CUDA Kernels: Subdivide into Blocks

- Threads are grouped into **blocks**
- **Blocks** are grouped into a **grid**
- A **kernel** is executed as a **grid of blocks of threads**
MAPPING OPENACC TO CUDA
OpenACC Execution Model on CUDA

- The OpenACC execution model has three levels: gang, worker, and vector

- For GPUs, the mapping is implementation dependent. Some possibilities:
  - gang==block, worker==warp, and vector==threads of a warp

- Depends on what the compiler thinks is the best mapping for a problem

- code portability is reduced
Gang, Worker, Vector Clauses

• gang, worker, and vector can be added to a loop clause
• A parallel region can only specify one of each gang, worker, vector
• Control the size using the following clauses on the parallel region
  • num_gangs(n), num_workers(n), vector_length(n)

```
#pragma acc kernels loop gang
for (int i = 0; i < n; ++i)
  #pragma acc loop
  vector(128)
  for (int j = 0; j < n; ++j)
    ...
```

```
#pragma acc parallel vector_length(128)
#pragma acc loop gang
for (int i = 0; i < n; ++i)
  #pragma acc loop vector
  for (int j = 0; j < n; ++j)
    ...
```
**Collapse Clause**

The **collapse(n)** clause transforms the following `n` tightly nested loops into one, flattened loop.

- Useful when individual loops lack sufficient parallelism or more than 3 loops are nested (gang/worker/vector)

```c
#pragma acc parallel
#pragma acc loop
collapse(2)
for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
        ...
```

```c
!$acc parallel
!$acc loop collapse(2)
do j=1,N-1
do i=1,N-1
    ...
```

---

**Loops must be tightly nested**
The “restrict” keyword in C

- Avoid pointer aliasing
  - 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”*
  - In simple, the ptr will only point to the memory space of itself

- 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.

Routine Construct

Specifies that the compiler should generate a device copy of the function/subroutine and what type of parallelism the routine contains.

Clauses:

```
gang/worker/vector/seq (sequential)
```

Specifies the level of parallelism contained in the routine.
#pragma acc routine vector
void foo(float* v, int i, int n) {
    #pragma acc loop vector
    for (int j=0; j<n; ++j) {
        v[i*n+j] = 1.0f/(i*j);
    }
}

#pragma acc parallel loop
for (int i=0; i<n; ++i) {
    foo(v,i);
    //call on the device
}
Update Construct

- Fortran
  - `#pragma acc update host/device [clause ...]`
- C
  - `!$acc update host/device [clause ...]`
- 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.
Asynchronous Execution

- Activated through `async [(int)]` clause on these directives:
  - parallel
  - kernels
  - update

Without `async`: host waits for device to finish execution
With `async`: host continues with code following directive

- Optional integer argument may be used to explicitly refer to region in a `wait` directive
- Two activities with same value are executed in the order the host process encounters them
- Two activities with different values may be executed in any order
The “wait” Directive

Executable directive

C:

```c
#pragma acc wait [(int)]
```

Fortran:

```fortran
!$acc wait [(int)]
```

Host thread waits for completion of asynchronous activities

Optional argument:

wait for asynchronous activity with argument in `async` clause
#pragma acc parallel loop async(1) // kernel A
#pragma acc parallel loop async(2) // kernel B
#pragma acc wait(1,2) async(3)
#pragma acc parallel loop async(3) // wait(1,2) // or wait directive
// kernel C
#pragma acc parallel loop async(4) wait(3) // kernel D
#pragma acc parallel loop async(5) \ wait(3) // kernel E
#pragma acc wait(1)
//kernel F on host
AGENDA

• Fundamentals of Heterogeneous & GPU Computing
• What are Compiler Directives?
• Accelerating Applications with OpenACC
  • Identify Available Parallelism
  • Parallelize loops
  • Optimize Data Locality
  • Optimize loops

• Interoperability
3 Approaches to Heterogeneous Programming

- **Libraries**: Easy to use, Most Performance
- **Compiler Directives**: Easy to use, Portable code
- **Programming Languages**: Most Performance, Most 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
“host_data” Construct

C/C++
    #pragma acc kernels host_data use_device(list)

Fortran
    !$acc kernels host_data use_device(list)

- Make the address of device data available on host
- Specified variable addresses refer to device memory
- Variables must be present on device

deviceptr data clause: inform compiler that the data already resides on the GPU
SAXPY

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

- A function in the standard Basic Linear Algebra Subroutines (BLAS) library
cublasSaxpy from cuBLAS library

```c
void cublasSaxpy( int n,
const float *alpha,
const float *x,
int incx,
float *y,
int incy)
```

• A function in the standard Basic Linear Algebra Subroutines (BLAS) library, which is a GPU-accelerated library ready to be used on GPUs.
• cuBLAS: GPU-accelerated drop-in library ready to be used on GPUs.
**Saxpy_acc**

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

int main()
{
    ...
    // Initialize vectors x, y
    #pragma acc data create(x[0:n]) copyout(y[0:n])
    #pragma acc parallel loop
    for (int i = 0; i < n; ++i)
    { x[i] = 1.0f; y[i] = 0.0f; }
    // Perform SAXPY
    saxpy_acc(n, a, x, y);
}
```

**Saxpy_cuBLAS**

```c
extern void
cublasSaxpy(int, float, float*, int, float*, int);

int main()
{...
    // Initialize vectors x, y
    #pragma acc data create(x[0:n]) copyout(y[0:n])
    #pragma acc parallel loop
    for (int i = 0; i < n; ++i)
    { x[i] = 1.0f; y[i] = 0.0f; }
    // Perform SAXPY
    #pragma acc host_data use_device(x,y)
    cublasSaxpy(n, 2.0, x, 1, y, 1);
    ...
}
```

[http://docs.nvidia.com/cuda](http://docs.nvidia.com/cuda)
void saxpy_acc(int n, float a, float *x, float *y) {
    #pragma acc parallel loop
    for (int i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}

int main(){
    ...
    // Initialize vectors x, y
    #pragma acc data create(x[0:n]) copyout(y[0:n])
    #pragma acc parallel loop
    for (int i = 0; i < n; ++i) {
        x[i] = 1.0f; y[i] = 0.0f;
    }
    // Perform SAXPY
    saxpy_acc(n, a, x, y);
}
    ...

extern void
cublasSaxpy(int,float,float*,int,float*,int);

int main(){
    ...
    // Initialize vectors x, y
    #pragma acc data create(x[0:n]) copyout(y[0:n])
    #pragma acc parallel loop
    for (int i = 0; i < n; ++i) {
        x[i] = 1.0f; y[i] = 0.0f;
    }
    // Perform SAXPY
    #pragma acc deviceptr (x,y)
cublasSaxpy(n, 2.0, x, 1, y, 1);
    ...

http://docs.nvidia.com/cuda
Optimize Data Locality

Optimize Loop Performance

Parallelize Loops with OpenACC

Code Profiling
More Than One GPUs?
Device Management

• Internal control variables (ICVs):
  • `acc-device-type-var`
    → Controls which type of accelerator is used
  • `acc-device-num-var`
    → Controls which accelerator device is used

• Setting ICVs by API calls
  • `acc_set_device_type()`
  • `acc_set_device_num()`

• Querying of ICVs
  • `acc_get_device_type()`
  • `acc_get_device_num()`
OpenACC APIs

acc_get_num_devices

- Returns the number of accelerator devices attached to host and the argument specifies type of devices to count

C:
- int acc_get_num_devices(acc_device_t)

Fortran:
- Integer function acc_get_num_devices(devicetype)
OpenACC APIs

acc_set_device_num

- Sets ICV ACC_DEVICE_NUM
- Specifies which device of given type to use for next region Can not be called in a parallel, kernels or data region

C:
- Void acc_set_device_num(int, acc_device_t)

Fortran:
- Subroutine
  acc_set_device_num(devicenum, devicetype)
OpenACC APIs

- **acc_get_device_num**
  - Return value of ICV `ACC_DEVICE_NUM`
  - Return which device of given type to use for next region
  - Can not be called in a parallel, kernels or data region

- **C:**
  - Void `acc_get_device_num(acc_device_t)`

- **Fortran:**
  - Subroutine `acc_get_device_num(devicetype)`
#pragma acc routine seq
void saxpy(int n, float a, float *x, float *restrict y) {
    #pragma acc loop //kernels
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i]/2.3/1.2;
}

int main(int argc, char **argv)
{
    int n = 1<<40;
    float *x = (float*)malloc(n*sizeof(float));
    float *y = (float*)malloc(n*sizeof(float));
    for (int i = 0; i < n; ++i) {
        x[i] = 2.0f;
        y[i] = 1.0f;
    }

    int gpu_ct=acc_get_num_devices(acc_device_nvidia);
    int tid=0;
    #pragma omp parallel private(tid) num_threads(gpu_ct)
    {
        int i=omp_get_thread_num();
        acc_set_device_num(i,acc_device_nvidia);
        #pragma acc data copyin(n) copyin(x[0:n]) copyout(y[0:n])
        {
        #pragma acc kernels
            for (int j=0; j<n*n; j++)
            {
                saxpy(n, 3.0f, x, y);
            }
        }
    }
}
Directive-based programming with multiple GPU cards

[wfeinste@shelob030 openacc17]$ pgcc -acc -mp -fast saxpy_2gpu.c
[wfeinste@shelob030 openacc17]$ nvidia-smi
Mon May 29 02:20:46 2017

+-----------------------------------------------------------------------------+
| NVIDIA-SMI 352.93     Driver Version: 352.93 |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id              |
| Fan  Temp  Perf Pwr:Usage/Cap| Memory-Usage | GPU-Util Compute M. |
|===============================+======================+======================|
|   0  Tesla K20Xm         On   | 0000:20:00.0     Off |          0 |
| N/A   18C    P0    62W / 235W |     87MiB /  5759MiB | 99%      Default |
|   1  Tesla K20Xm         On   | 0000:8B:00.0     Off |          0 |
| N/A   19C    P0    63W / 235W |     87MiB /  5759MiB | 99%      Default |
+-----------------------------------------------------------------------------+

| Processes:                                                                 |
|-----------------------------|-----------------------|-------------------|
| GPU   PID  Type Process name | GPU Memory Usage  |
|-----------------------------|-----------------------|-------------------|
|   0  18225  C  ./a.out      | 71MiB | 71MiB |
|   1  18225  C  ./a.out      | 71MiB | 71MiB |
Directive-based programming on multi-GPUs

- OpenACC only supports one GPU
- Hybrid model:
  - OpenACC + OpenMP to support multi-GPU parallel programming
  - Data management
Getting Started for Labs

• Connect to mike cluster:
  • `ssh username@mike.hpc.lsu.edu`

• Login in to the interactive node
  `qsub -l -A xxx -l walltime=2:00:00 -l nodes=1:ppn=16 -q shelob`

• Open another terminal
  `ssh -X shelobxxx /mikexxx`
General Steps for Labs

- Code profiling to identify the target for parallelization
  - `pgprof`: PGI visual profiler
  - `pgcc -Minfo=ccff mycode.c -o mycode`
  - `pgcollect mycode`
  - `pgprof -exe mycode`
- Add OpenACC pragmas/directives
  - `pgcc -acc -ta=nvidia,time -Minfo=accel app.c -o app`
  - `pgf90 -acc -ta=nvidia,time -Minfo=accel app.f90 -o app`
Exercise 1

For the matrix multiplication code
\[ A \cdot B = C \]
where:
\[ a_{i,j} = i + j \]
\[ b_{i,j} = i \cdot j \]
\[ c_{i,j} = \sum_k a_{i,k} \cdot b_{k,j} \]

1. For mm_acc_v0.c, speedup the matrix multiplication code segment using OpenACC directives/pragmas

2. For mm_acc_v1.c:
   - Change A, B and C to dynamic arrays, i.e., the size of the matrix can be specified at runtime;
   - Complete the function matmul_acc using the OpenACC directives;
   - Compare performance with serial and OpenMP results
Exercise 2

Calculate \( \pi \) value using the equation:
\[
\int_0^1 \frac{4.0}{(1.0 + x^2)} = \pi
\]
with the numerical integration:
\[
\sum_{i=1}^{n} \frac{4.0}{(1.0 + x_i \cdot x_i)} \Delta x \approx \pi
\]

Speedup the code segment using OpenACC directives/pragmas