Introduction to Parallel Programming with Single and Multiple GPUs

Frank Mueller

http://moss.csc.ncsu.edu/~mueller/mpigpu

Objective

- Learn to program GPUs for general-purpose computing tasks
- Specific to NVIDIA GPUs
  - CUDA programming abstraction
- Compare performance to CPU threads
  - OpenMP – could also integrate (not shown)
- Show integration within clusters
  - Multiple nodes with GPUs
- Hands-on exercises (more C than Fortran): laptop → ARC cluster

Not covered in this short course:

- Advanced GPU performance tuning (memory, async. kernels etc.)
- OpenCL
- PGI compiler directives for accelerators

Learning Modules

- Lecture style:

1. GPU programming with CUDA
2. CUDA reduction
   - As a sample optimization
3. ARC Cluster

- Hands-on programming
  - Several exercises

Hands-on Exercises: Computing π (Pi)

- Running Example: compute Pi with increasing parallelism
- Numerical approach

1. C/Fortran
2. C + OpenMP (threads)
3. C/Fortran + CUDA
   1. Simple kernel (1 thread)
   2. 1 Block (512 threads)
   3. Shared memory
   4. Grid of blocks (32k threads)
   5. GPU reduction
4. C/Fortran + MPI
5. C + MPI + OpenMP
6. C + MPI + CUDA

Login on ARC Cluster

- On Windows systems, use:
  - Putty / any ssh client
  - Connect to arc.csc.ncsu.edu
- On Linux systems:
  ssh <myname>@arc.csc.ncsu.edu
- At/after login:
  - password: xxx
  - Hit enter twice when asked to generate RSA keys
- qsub -I -q c2050
- cd mpigpu

Activate CUDA

- Edit your ~/.bashrc file to append the 4 lines below:

```
vim ~/.bashrc
    export PATH=".:~/bin:/usr/local/bin:/usr/bin:$PATH"
    export PATH="/usr/local/cuda/bin:$PATH"
    export LD_LIBRARY_PATH="/usr/local/cuda/lib64:
       /usr/local/cuda/lib:$LD_LIBRARY_PATH"
    export MANPATH="/usr/share/man:$MANPATH"
```

(in one line)

- Log out and back in to activate the new settings.
- Prepare for exercises:

```
wget http://moss.csc.ncsu.edu/~mueller/mpigpu/hw/mpigpu.zip
unzip mpigpu.zip
cd mpigpu
```

Approximation of Pi

Integration to evaluate π

Computer approximations to π by using numerical integration.
Know

$$\tan(45^0) = 1;$$

same as

$$\tan\frac{\pi}{4} = 1;$$

So that;

$$4 * \tan^{-1} 1 = \pi$$

From the integral tables we can find

$$\tan^{-1} x = \int \frac{1}{1+x^2} dx$$

or

$$\tan^{-1} 1 = \int_0^1 \frac{1}{1+x^2} dx$$

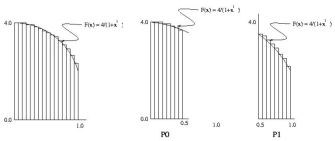
Using the mid-point rule with panels of uniform length $h = 1/n$, for various values of n. Evaluate the function at the midpoints of each subinterval (x_{i-1}, x_i) $i * h - h/2$ is the midpoint. Formula for the integral is

$$x = \sum_{i=1}^{n} f(h * (i - 1/2))$$

$$\pi = h * x$$

where

$$f(x) = \frac{4}{1+x^2}$$



PI in C (stylized)

```
#include <stdio.h>
#include <math.h>
#include "mytime.h"

double integrate(int n) {
  double sum, h, x;
  int i;
  h = 1.0 / (double) n;
  for (i = 1; i <= n; i++) {
    x = h*((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
  }
  return sum * h;
}
```

```
int main(int argc, char *argv[]) {
  int n;
  double PI25DT = 3.14159…;
  double pi;

  while (1) {
    printf("Enter # intervals: ");
    scanf("%d",&n);
    if (n == 0)
      break;
    pi = integrate(n);

   printf("pi=%.16f, error=%.16f\n",
          pi, fabs(pi - PI25DT));
}
```

```
make pi-c
pi-c
Enter the number of intervals: (0 quits) 1000000
pi is approximately 3.1415926535897643, Error is 0.0000000000000289
wall clock time = 0.038800
Enter the number of intervals: (0 quits) 0
```


PI in Fortran (stylized)

```
      program main
           integer n, i
           double precision  PI25DT
           parameter (PI25DT = 3.14159…)
           double precision  pi, h, sum, x, f, a

           f(a) = 4.d0 / (1.d0 + a*a)
      10   write(6,98)
      98   format('Enter # intervals: ')
           read(5,99) n
      99   format(i10)
           if ( n .le. 0 ) goto 30
           h = 1.0d0/n
           sum = 0.0d0
```

```
make pi-f
pi-f
Enter the number of intervals: (0 quits)
1000000
pi is approximately 3.1415926535897643, Error is 0.0000000000000289
wall clock time = 0.038800
Enter the number of intervals: (0 quits)
0
```

PI for PGI w/ OpenACC in C (stylized)

```
#include <stdio.h>
#include <math.h>
#include "mytime.h"

double integrate(int n) {
  double sum, h, x;
  int i;
  h = 1.0 / (double) n;
#pragma acc parallel
  for (i = 1; i <= n; i++) {
    x = h*((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
  }
  return sum * h;
}
```

```
int main(int argc, char *argv[]) {
  int n;
  double PI25DT = 3.14159…;
  double pi;

  while (1) {
    printf("Enter # intervals: ");
    scanf("%d",&n);
    if (n == 0)
      break;
    pi = integrate(n);

   printf("pi=%.16f, error=%.16f\n",
          pi, fabs(pi - PI25DT));
  }
```

```
make pi-pgicu-c
pi-pgicu-c
Enter the number of intervals: (0 quits) 1000000
pi is approximately 3.1415926535897643, Error is 0.0000000000000289
wall clock time = 0.038800
Enter the number of intervals: (0 quits) 0
```


PI for PGI Fortran w/ OpenACC

```
      program main
           integer n, i
           double precision  PI25DT
           parameter (PI25DT = 3.14159…)
           double precision  pi, h, sum, x, f, a

           f(a) = 4.d0 / (1.d0 + a*a)
      10   write(6,98)
      98   format('Enter # intervals: ')
           read(5,99) n
      99   format(i10)
           if ( n .le. 0 ) goto 30
           h = 1.0d0/n
           sum  = 0.0d0
```

```
make pi-pgicu-f
pi-pgicu-f
Enter the number of intervals: (0 quits)
1000000
pi is approximately 3.1415926535897643, Error is 0.0000000000000289
wall clock time = 0.020000
Enter the number of intervals: (0 quits)
0
```

```
           goto 10
      30   end
```


PI for CUDA in C

```
int main(int argc, char *argv[]) {
  int n;
  int *n_d; // device copy of n8
  …
  double *pi_d; // device copy of pi
  struct timeval startwtime, …;
  // Allocate memory on GPU
  cudaMalloc( (void **) &n_d, sizeof(int) * 1 );
  cudaMalloc( (void **) &pi_d, sizeof(double) * 1 );

  while (1) {
  …
    if (n == 0)
      break;
    // copy from CPU to GPU
    cudaMemcpy( n_d, &n, sizeof(int) * 1, cudaMemcpyHostToDevice);
    integrate<<< 1, 1 >>>(n_d, pi_d);
    // copy back from GPU to CPU
    cudaMemcpy( &pi, pi_d, sizeof(double) * 1, cudaMemcpyDeviceToHost);
    …
  }
  // free GPU memory
  cudaFree(n_d);
  cudaFree(pi_d);
}
```

PI for CUDA in C

```
#include <stdio.h>
#include <math.h>
#include "mytime.h"

// GPU kernel
__global__ void integrate(int
  *n, double *sum) {
  double h, x;
  int i;

  *sum = 0.0;
  h = 1.0 / (double) *n;
  for (i = 1; i <= *n; i++) {
    x = h * ((double)i - 0.5);
    *sum += 4.0 / (1.0 + x*x);
  }
  *sum *= h;
}
```

```
cp pi-c.c pi-cu.cu
vim pi-cu.cu
make pi-cu
pi-cu
Enter the number of intervals: (0 quits) 1000000
…
```


PI in Fortran: need GPU kernel in C

```
      h = 1.0d0/n
      sum  = 0.0d0
      do 20 i = 1, n, 1
        x=h * (dble(i) - 0.5d0)
        sum = sum + f(x)
20    continue
      pi = h * sum
```

```
//File: pi-cu-integrate-f.cu

// GPU kernel
__global__ void integrate(int
  *n, double *sum) {
  double h, x;
  int i;

  *sum = 0.0;
  h = 1.0 / (double) *n;
  for (i = 1; i <= *n; i++) {
    x = h * ((double)i - 0.5);
    *sum += 4.0 / (1.0 + x*x);
  }
  *sum *= h;
}
```

- Bottom line: some C required for CUDA
- File pi-cu-integrate-f.cu provided

PI in Fortran: need wrapper in C

```
//(cont.): pi-cu-integrate-f.cu

// notice underscore "_" after function name! → for gfortran
extern "C" void fortran_call_integrate_(int *n, double *pi) {
  int *n_d; // device copy of n
  double *pi_d; // device copy of pi

  // Allocate memory on GPU
  cudaMalloc( (void **) &n_d, sizeof(int) * 1 );
  cudaMalloc( (void **) &pi_d, sizeof(double) * 1 );

  // copy from CPU to GPU
  cudaMemcpy( n_d, n, sizeof(int) * 1, cudaMemcpyHostToDevice);
  integrate<<< 1, 1 >>>(n_d, pi_d);

  // copy back from GPU to CPU
  cudaMemcpy( pi, pi_d, sizeof(double)*1, cudaMemcpyDeviceToHost);

  // free GPU memory
  cudaFree(n_d);
  cudaFree(pi_d);
}
```

PI in Fortran (stylized)

```
      program main
            integer n, i
            double precision  PI25DT
            parameter (PI25DT = 3.14159…)
            double precision  pi, h, sum, x, f, a

            f(a) = 4.d0 / (1.d0 + a*a)
      10    write(6,98)
      98    format('Enter # intervals: ')
            read(5,99) n
      99    format(i10)
            if ( n .le. 0 ) goto 30
            call fortran_call_integrate(n, pi)
        write(6, 96) pi, abs(pi - PI25DT)
      96    format('pi=', F18.16,
           +       ', error=', F18.16)
            goto 10
      30    end
```

```
cp pi-f.f pi-cu-f.f
vim pi-cu-f.f
make pi-cu-f
pi-cu-f
Enter the number of intervals: (0 quits)
1000000
…
```


PI for CUDA with 512 Threads (1 Block)

```
…
#include "mytime.h"
#define THREADS 512
// GPU kernel
…
  sum[threadIdx.x] = 0.0;
  for (i=threadIdx.x+1; i<=*n;
       i+=THREADS) {
    x = h * ((double)i - 0.5);
    sum[threadIdx.x]+=4.0/(1.0+x*x);
  }
  sum[threadIDx.x] *= h;
}
```

```
cp pi-cu.cu pi-cu-block.cu
vim pi-cu-block.cu
make pi-cu-block
pi-cu-block
Enter the number of intervals…
```

```
int main(int argc, char *argv[]) {
  double mypi[THREADS];
  double *mypi_d; // device copy
   of pi
  …
  cudaMalloc( (void **) &mypi_d,
   sizeof(double) * THREADS );
    …
    integrate<<<1, THREADS>>>
      (n_d, mypi_d);
  …
    cudaMemcpy(&mypi, mypi_d,
     sizeof(double) * THREADS,
     cudaMemcpyDeviceToHost);
   pi = 0.0;
   for (i = 0; i < THREADS; i++)
     pi += mypi[i];
   …
}
…
cudaFree(mypi_d);
}
```


PI for CUDA with Shared (Fast) Memory

- Shared memory is
  - Small (a few KB)
  - Fast: single cycle access
- Global memory: ~1GB

```
…
#include "mytime.h"
#define THREADS 512
// GPU kernel
__global__ void integrate(int *n, double *gsum) {
  __shared__ double sum[THREADS];
  sum[threadIdx.x] = 0.0;
…
  gsum[threadIdx.x] = sum[threadIdx.x] * h;
}
```

```
cp pi-cu-block.cu pi-cu-shared.cu
vim pi-cu-shared.cu
make pi-cu-shared
pi-cu-shared
Enter the number of intervals…
```

```
wget http://redmpi.com/mpigpu.zip
```

PI for CUDA with Grid (32k Threads)

```
#define THREADS 512
#define MAX_BLOCKS 64
// GPU kernel, we know: THREADS == blockDim.x
__global__ void integrate(int *n, int *blocks,
   double *gsum) {
  …
  for (i = blockIdx.x*blockDim.x + threadIdx.x +
  1; i <= *n; i += blockDim.x * *blocks) {
    …
  gsum[blockIdx.x*blockDim.x + threadIdx.x] =
   sum[threadIdx.x] * h;
}
```

- 64 blocks x 512 threads
- Each block has own shared mem!!!
  - Block 0: sum[0..511]
  - Block 1: sum[0..511]
  - etc.

```
cp pi-cu-shared.cu pi-cu-grid.cu
vim pi-cu-grid.cu
make pi-cu-grid
pi-cu-grid
Enter the number of intervals…
```


PI for CUDA with Grid (32k Threads)

```
int main(int argc, char *argv[]) {
  int n, i, blocks;
  int *n_d, *blocks_d; // device copy
  …
  cudaMalloc( (void **) &blocks_d, sizeof(int) * 1 );
  cudaMalloc( (void **) &mypi_d, sizeof(double) * THREADS
                                          * MAX_BLOCKS );
  …
    cudaMemcpy( blocks_d, &blocks, sizeof(int) * 1,
                cudaMemcpyHostToDevice );

    integrate<<< blocks, THREADS >>>(n_d, blocks_d, mypi_d);

    // copy back from GPU to CPU
    cudaMemcpy( &mypi, mypi_d,
      sizeof(double) * THREADS * blocks,
      cudaMemcpyDeviceToHost );
    pi = 0.0;
    for (i = 0; i < THREADS * blocks; i++)
      pi += mypi[i];
  …
  cudaFree(blocks_d);
```

```
cp pi-cu-shared.cu pi-cu-grid.cu
vim pi-cu-grid.cu
make pi-cu-grid
pi-cu-grid
Enter the number of intervals…
```


PI for PGI w/ Reduction+Private in C (sty.)

```
#include <stdio.h>
#include <math.h>
#include "mytime.h"

double integrate(int n) {
  double sum, h, x;
  int i;
  h = 1.0 / (double) n;
#pragma acc parallel \
  reduction(+:sum) private(i,x)
  for (i = 1; i <= n; i++) {
    x = h*((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
  }
  return sum * h;
}
```

```
int main(int argc, char *argv[]) {
  int n;
  double PI25DT = 3.14159…;
  double pi;

  while (1) {
    printf("Enter # intervals: ");
    scanf("%d",&n);
    if (n == 0)
      break;
    pi = integrate(n);

   printf("pi=%.16f, error=%.16f\n",
          pi, fabs(pi - PI25DT));
  }
```

```
make pi-pgicu-reduce-c
pi-pgicu-reduce-c
Enter the number of intervals: (0 quits) 1000000
pi is approximately 3.1415926535897643, Error is 0.0000000000000289
wall clock time = 0.038800
Enter the number of intervals: (0 quits) 0
```

PI for PGI Fortran w/ Reduction+Private

```
      program main
            integer n, i
            double precision  PI25DT
            parameter (PI25DT = 3.14159…)
            double precision  pi, h, sum, x, f, a

            f(a) = 4.d0 / (1.d0 + a*a)
      10    write(6,98)
      98    format('Enter # intervals: ')
            read(5,99) n
      99    format(i10)
            if ( n .le. 0 ) goto 30
            h = 1.0d0/n
            sum  = 0.0d0
```

```
make pi-pgicu-reduce-f
pi-pgicu-reduce-f
Enter the number of intervals: (0 quits)
1000000
pi is approximately 3.1415926535897643, Error is 0.0000000000000289
wall clock time = 0.020000
Enter the number of intervals: (0 quits)
0
```

```
            goto 10
      30    end
```


PI for CUDA with Reduction

```
int main(int argc, char *argv[]) {
…
   integrate<<< blocks, THREADS >>>(n_d, blocks_d, mypi_d);
   if (blocks > 1)
      global_reduce<<< 1, 512 >>>(n_d, blocks_d, mypi_d);
    // copy back from GPU to CPU
    cudaMemcpy( &pi, mypi_d, sizeof(double) * 1,
  cudaMemcpyDeviceToHost );
```

```
make pi-cu-grid-reduce
pi-cu-grid-reduce
Enter the number of intervals…
```


PI for CUDA with Reduction

```
__global__ void integrate(int *n, int *blocks, double *gsum) {
  const unsigned int bid = blockDim.x * blockIdx.x + threadIdx.x;
  const unsigned int tid = threadIdx.x;
  …
  __shared__ double ssum[THREADS];
  double sum;

  sum = 0.0;
  h   = 1.0 / (double) *n;
  for (i = bid + 1; i <= *n; i += blockDim.x * *blocks) {
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
  }
  ssum[tid] = sum * h;
  // block reduction
  __syncthreads();
  for (i = blockDim.x / 2; i > 0; i >>= 1) { /* per block */
    if (tid < i)
      ssum[tid] += ssum[tid + i];
    __syncthreads();
  }
  gsum[bid] = ssum[tid];
}
```

PI for CUDA with Reduction

```
// number of threads must be a power of 2
__global__ static void global_reduce(int *n, int *blocks, double
  *gsum) {
    __shared__ double ssum[THREADS];
    const unsigned int tid = threadIdx.x;
    unsigned int i;

    if (tid < *blocks)
      ssum[tid] = gsum[tid * THREADS];
    else
      ssum[tid] = 0.0;
    __syncthreads();
    for (i = blockDim.x / 2; i > 0; i >>= 1) { /* per block */
        if (tid < i)
           ssum[tid] += ssum[tid + i];
        __syncthreads();
    }
    gsum[tid] = ssum[tid];
}
```


PI for MPI in C

```
#include <mpi.h>
…
double integrate(int n, int myid, int numprocs) {
  …
  for (i = myid + 1; i <= n; i += numprocs) {
  …
}

int main(int argc, char *argv[]) {
  int n, myid, numprocs;
  double PI25DT = 3.141592653589793238
  double mypi, pi;
  double startwtime, endwtime;
  int  namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  MPI_Get_processor_name(processor_name,&namelen);
  printf("MPI Task %2d on %20s\n", myid, processor_name);
  sleep(1); // wait for everyone to print
```

```
cp pi-c.c pi-mpi-c.c
vim pi-mpi-c.c
make pi-mpi-c
mpirin –np 4 pi-mpi-c
Enter the number of intervals…
```


PI for MPI in C

```
 while (1) {
    if (myid == 0) {
      printf("Enter the number of intervals: (0 quits)
  ");fflush(stdout);
      scanf("%d",&n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0)
      break;
    mypi = integrate(n, myid, numprocs);

    MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);

    if (myid == 0) {
      printf…
    }
  }
  MPI_Finalize();

  return 0;
}
```

```
cp pi-c.c pi-mpi-c.c
vim pi-mpi-c.c
make pi-mpi-c
pi-mpi-c
Enter the number of intervals…
```

PI for MPI in Fortran

```
program main
      use MPI
      integer n, myid, numprocs, i, ierr
      …
      double precision  mypi, pi, h, sum, x, f, a
      integer namelen
      character*(MPI_MAX_PROCESSOR_NAME) name
      …
      f(a) = 4.d0 / (1.d0 + a*a)
      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
      call MPI_GET_PROCESSOR_NAME(name, namelen, ierr)
      print "('MPI Task ', I3,' on ',A)", myid, trim(name)
      call MPI_BARRIER(MPI_COMM_WORLD, ierr)
10    if ( myid .eq. 0 ) then
         write(6,98)
98       format('Enter the number of i
         read(5,99) n
99       format(i10)
      endif
```

```
cp pi-f.f pi-mpi-f.f
vim pi-mpi-f.f
make pi-mpi-f
mpirun –np 4 pi-mpi-f
Enter the number of intervals…
```


PI for MPI in Fortran

```
   call MPI_BCAST( n, 1, MPI_INTEGER, 0,
      +                MPI_COMM_WORLD, ierr)
       …
       do 20 i = myid+1, n, numprocs
         x   = h * (dble(i) - 0.5d0)
         sum = sum + f(x)
 20    continue
       mypi = h * sum
c                    collect all the partial sums
       call MPI_REDUCE( mypi, pi, 1, MPI_DOUBLE_PRECISION,
      +                 MPI_SUM, 0, MPI_COMM_WORLD,ierr)
c                                  node 0 prints the answer
       if (myid .eq. 0) then
          write(6, 96) pi, abs(pi - PI25DT)
 96       format('pi is approximately:
      +           ', Error is: ', F18.1
       endif
       goto 10
 30    call MPI_FINALIZE(ierr)
       end
```

```
cp pi-f.f pi-mpi-f.f
vim pi-mpi-f.f
make pi-mpi-f
pi-mpi-f
Enter the number of intervals…
```


PI for MPI+OpenMP in C

```
double integrate(int n, int myid, int numprocs) {
  double h, x, sum;
  int i;

  sum = 0.0;
  h   = 1.0 / (double) n;
#pragma omp parallel for reduction(+:sum) private(i,x)
  for (i = myid + 1; i <= n; i += numprocs) {
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
  }
  return sum * h;
}
```

```
cp pi-mpi-c.c pi-mpi-omp-c.c
vim pi-mpi-omp-c.c
make pi-mpi-omp-c
pi-mpi-omp-c
Enter the number of intervals…
```

PI for PGI w/ MPI+OpenACC in C

```
double integrate(int n, int myid, int numprocs) {
  double h, x, sum;
  int i;

  sum = 0.0;
  h   = 1.0 / (double) n;
#pragma acc parallel reduction(+:sum) private(i,x)
  for (i = myid + 1; i <= n; i += numprocs) {
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
  }
  return sum * h;
}
```

```
cp pi-mpi-c.c pi-mpi-pgicu-c.c
vim pi-mpi-pgicu-c.c
make pi-mpi-pgicu-c
pi-mpi-pgicu-c
Enter the number of intervals…
```


PI for PGI Fortran w/ MPI+OpenACC

```
      program main
            integer n, i
            double precision  PI25DT
            parameter (PI25DT = 3.14159…)
            double precision  pi, h, sum, x, f, a

            f(a) = 4.d0 / (1.d0 + a*a)
      10    write(6,98)
      98    format('Enter # intervals: ')
            read(5,99) n
      99    format(i10)
            if ( n .le. 0 ) goto 30
            h = 1.0d0/n
            sum  = 0.0d0
```

```
cp pi-mpi-f.f pi-mpi-pgicu-f.f
vim pi-mpi-pgicu-f.f
make pi-mpi-pgicu-f
pi-mpi-pgicu-f
Enter the number of intervals: (0 quits)
1000000
pi is approximately 3.1415926535897643, Error is 0.0000000000000289
wall clock time = 0.020000
Enter the number of intervals: (0 quits)
0
```


PI for MPI+CUDA in C

```
__global__ void integrate(int *n, int *blocks, int *myid,
                          int*numprocs, double *gsum) {
  const unsigned int bid = blockDim.x * blockIdx.x + threadIdx.x;
  const unsigned int gid = *blocks * blockDim.x * *myid + bid;
  …
  for (i = gid + 1; i <= *n; i += blockDim.x * *blocks *
   *numprocs) {
    …
```

```
make pi-cu-mpi
mpirun –np 4 pi-cu-mpi
Enter the number of intervals…
```

PI for MPI+CUDA in C

```
int main(int argc, char *argv[]) {
  int n, blocks, myid, numprocs;
  int *n_d, *blocks_d, *myid_d, *numprocs_d; // device copy
  double PI25DT = 3.14159265358979323846264643;
  double mypi, pi;
  double *mypi_d; // device copy of pi
  int  namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  MPI_Get_processor_name(processor_name,&namelen);

  // Allocate memory on GPU
  cudaMalloc( (void **) &n_d, sizeof(int) * 1 );
  cudaMalloc( (void **) &blocks_d, sizeof(int) * 1 );
  cudaMalloc( (void **) &numprocs_d, sizeof(int) * 1 );
  cudaMalloc( (void **) &myid_d, sizeof(int) * 1 );
  cudaMalloc( (void **) &mypi_d, sizeof(double) * THREADS *
   MAX_BLOCKS );
```


PI for MPI+CUDA in C

```
…
  while (1) {
    if (myid == 0) {
      printf…
    }
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&blocks, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0 || blocks > MAX_BLOCKS)
      break;
    …
    cudaMemcpy( blocks_d, &blocks, sizeof(int) * 1,
                cudaMemcpyHostToDevice );
    …
    integrate<<< blocks, THREADS >>>(n_d, blocks_d,
                                     myid_d, numprocs_d, mypi_d);
    if (blocks > 1)
      global_reduce<<< 1, 512 >>>(n_d, blocks_d, mypi_d);
    …
    cudaMemcpy( &mypi, mypi_d, sizeof(double) * 1,
                cudaMemcpyDeviceToHost );
```


PI for MPI+CUDA in C

```
    MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);

    if (myid == 0) {
      printf…

  // free GPU memory
  cudaFree(n_d);
  cudaFree(blocks_d);
  cudaFree(mypi_d);

  MPI_Finalize();

  return 0;
}
```

Objectives Met?

- Able to program GPUs for general-purpose computing tasks
- Specific to NVIDIA GPUs
  - CUDA programming abstraction
- Compared performance to CPU threads
  - OpenMP – could also integrate (not shown)
- Showed integration within clusters
  - Multiple nodes with GPUs
- Hands-on exercises (more C than Fortran): laptop → ARC cluster

Not covered in this short course:

- Advanced GPU performance tuning (memory, async. kernels etc.)
- OpenCL
- PGI compiler: directives for accelerators (OpenACC)