



11th Advanced School on **PARALLEL COMPUTING**

Computing with OpenACC Directives

Francesco Salvatore - [f.salvadore@cineca.it](mailto:f.salvadore@ Cineca.it)
SuperComputing Applications and Innovation Department





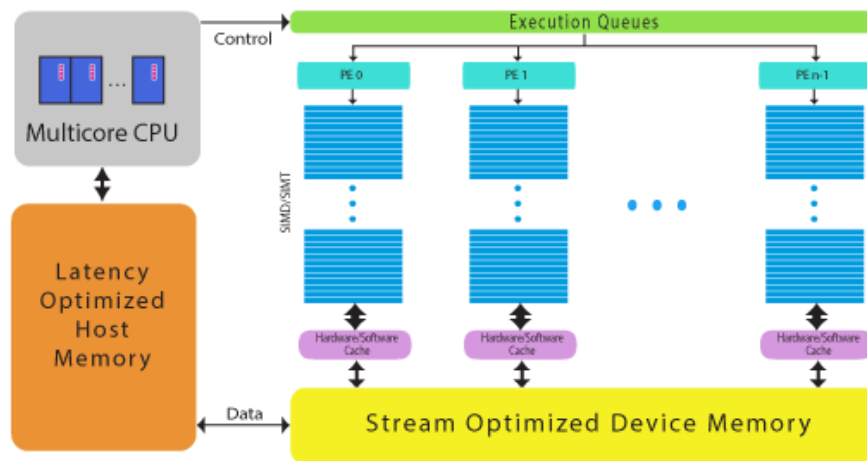
BASICS



Common Accelerator Architecture Themes

- **NVIDIA GPU**
- **ATI GPU**
- **Intel MIC**

- Separate device memory
- Many-cores
- Multithreading
- Vectors
- In-order instruction issue
- Memory strides matter
- Smaller caches



©2013 NVIDIA Corporation



3 Ways to Accelerate Applications

Applications

Libraries

“Drop-in”
Acceleration

OpenACC
Directives

Easily Accelerate
Applications

Programming
Languages

Maximum
Flexibility



OpenACC friendly Disclaimer

OpenACC Directives

**Easily Accelerate
Applications**

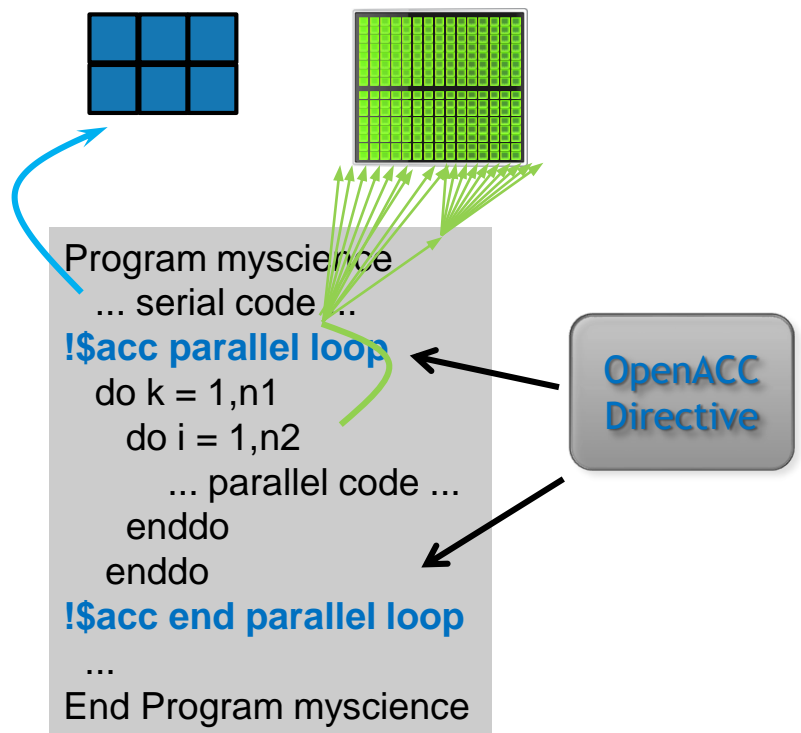
OpenACC does not make GPU programming easy.
(...)

GPU programming and parallel programming is not easy. It cannot be made easy. However, GPU programming need not be difficult, and certainly can be made straightforward, once you know how to program and know enough about the GPU architecture to optimize your algorithms and data structures to make effective use of the GPU for computing. OpenACC is designed to fill that role.

(Michael Wolfe, The Portland Group)



OpenACC Directives



Simple Compiler directives

Compiler Parallelizes code

Works on many-core GPUs &
multicore CPUs



OpenACC – Directive Based Approach

- Directives are added to serial source code
 - Manage loop parallelization
 - Manage data transfer between CPU and GPU memory
- Works with C, C++, or Fortran
 - Can be combined with explicit CUDA C/Fortran usage
- Directives are formatted as comments
 - They don't interfere with serial execution
- Maintains portability of original code



Familiar to OpenMP Programmers

OpenMP

CPU



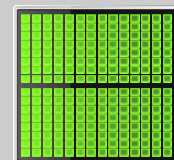
```
main() {  
  double pi = 0.0; long i;  
  
  #pragma omp parallel for reduction(+:pi)  
  for (i=0; i<N; i++)  
  {  
    double t = (double)((i+0.05)/N);  
    pi += 4.0/(1.0+t*t);  
  }  
  
  printf("pi = %f\n", pi/N);  
}
```

OpenACC

CPU



GPU



```
main() {  
  double pi = 0.0; long i;  
  
  #pragma acc parallel loop reduction(+:pi)  
  for (i=0; i<N; i++)  
  {  
    double t = (double)((i+0.05)/N);  
    pi += 4.0/(1.0+t*t);  
  }  
  
  printf("pi = %f\n", pi/N);  
}
```




OpenACC

The Standard for GPU Directives

- **Easy:** Directives are the easy path to accelerate compute intensive applications
- **Open:** OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors
- **Powerful:** GPU Directives allow complete access to the massive parallel power of a GPU





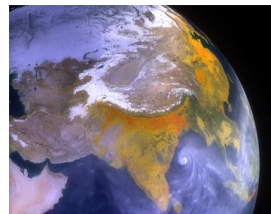
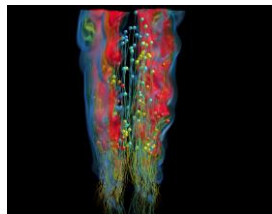
Focus on Exposing Parallelism

With Directives, tuning work focuses on *exposing parallelism*, which makes codes inherently better

Example: Application tuning work using directives for new Titan system at ORNL

S3D

Research more efficient combustion with next-generation fuels



- Tuning top 3 kernels (90% of runtime)
- 3 to 6x faster on CPU+GPU vs. CPU+CPU
- But also improved all-CPU version by 50%

- Tuning top key kernel (50% of runtime)
- 6.5x faster on CPU+GPU vs. CPU+CPU
- Improved performance of CPU version by 100%



OpenACC Specification and Website

- Full OpenACC 1.0 Specification available online
- OpenACC 2.0a revised on August 2013
<http://www.openacc-standard.org>
- Novelty in OpenACC 2.0 are significant
 - OpenACC 1.0 maybe not very mature...
- Some changes are inspired by the development of CUDA programming model
 - but the standard is not limited to NVIDIA GPUs: one of its pros is the **interoperability** between platforms

The OpenACC™ API QUICK REFERENCE GUIDE

The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators.

Most OpenACC directives apply to the immediately following structured block or loop; a structured block is a single statement or a compound statement (C or C++) or a sequence of statements (Fortran) with a single entry point at the top and a single exit at the bottom.



PGI

Version 1.0, November 2011

© 2011 OpenACC-standard.org all rights reserved.



OpenACC: Implementations and Vendors

- Standard implementation
 - CRAY provides full OpenACC 2.0 support in CCE 8.2
 - PGI does not support OpenACC 2.0, yet
 - **GNU implementation is forthcoming (expected in 5.0 release)**
- We will focus on PGI compiler
 - 30 days trial license useful for testing
- PGI:
 - all-in-one compiler, easy usage
 - sometimes the compiler tries to help you...
 - but also a constraint on the compiler to use

The OpenACC™ API QUICK REFERENCE GUIDE

The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators.

Most OpenACC directives apply to the immediately following structured block or loop; a structured block is a single statement or a compound statement (C or C++) or a sequence of statements (Fortran) with a single entry point at the top and a single exit at the bottom.

 CAPS

 CRAY
THE SUPERCOMPUTER COMPANY

 NVIDIA

PGI

Version 1.0, November 2011

© 2011 OpenACC-standard.org all rights reserved.



PGI compilers

- Check the feature you need!

| Feature | Version | Feature | Version |
|----------------------|---------|----------------------|---------|
| !\$acc kernels | 12.3 | !\$acc declare | 12.3 |
| clauses: | | clauses: | |
| if() | 12.3 | copy()/copyin() | 12.3 |
| async() | 12.3 | copyin()/copyout() | 12.3 |
| copy() | 12.3 | create() | 12.3 |
| copyin() | 12.3 | present() | 12.3 |
| copyout() | 12.3 | present_or_copy() | 12.3 |
| create() | 12.3 | present_or_copyin() | 12.3 |
| present() | 12.3 | present_or_copyout() | 12.3 |
| present_or_copy() | 12.3 | present_or_create() | 12.3 |
| present_or_copyin() | 12.3 | device_resident() | 12.6 |
| present_or_copyout() | 12.3 | deviceptr() | 12.6 |
| present_or_create() | 12.3 | | |
| deviceptr() | 12.3 | !\$acc update | 12.3 |
| | | clauses: | |
| !\$acc parallel | 12.5 | if() | 12.3 |
| clauses: | | async() | 12.3 |
| if() | 12.5 | | |
| async() | 12.5 | !\$acc cache | 12.6 |
| num_gangs() | 12.5 | | |
| num_workers() | 12.6 | !\$acc host_data | 14.1 |
| vector_length() | 12.5 | | |



A Very Simple Exercise: SAXPY

```
void saxpy(int n,  
          float a,  
          float *x,  
          float *restrict y)  
{  
  #pragma acc parallel loop  
  for (int i = 0; i < n; ++i)  
    y[i] = a*x[i] + y[i];  
}  
  
...  
// Perform SAXPY on 1M elements  
saxpy(1<<20, 2.0, x, y);  
...
```

```
subroutine saxpy(n, a, x, y)  
  real :: x(:), y(:), a  
  integer :: n, i  
  $!acc parallel loop  
  do i=1,n  
    y(i) = a*x(i)+y(i)  
  enddo  
  $!acc end parallel loop  
end subroutine saxpy  
  
...  
! Perform SAXPY on 1M elements  
call saxpy(2**20, 2.0, x_d, y_d)  
...
```



Directive Syntax

- 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 construct

- Programmer **identifies** a block of code suitable for parallelization
 - and **guarantees** that no dependency occurs across iterations
- Compiler generates parallel instructions for that loop
 - e.g., a parallel CUDA kernel for a GPU

```
#pragma acc parallel loop
for (int j=1;j<n-1;j++) {
    for (int i=1;i<n-1;i++) {
        A[j][i] = B[j][i] + C[j][i]
    }
}
```

First let us focus on the simplest usage, combining **parallel** and **loop** directives



Another approach: kernels construct

- The **kernels** construct expresses that a region **may** contain parallelism and the compiler determines what can be safely parallelized

```
!$acc kernels
```

```
do i=1,n  
  a(i) = 0.0  
  b(i) = 1.0  
  c(i) = 2.0  
end do
```

} kernel 1

```
do i=1,n  
  a(i) = b(i) + c(i)  
end do
```

} kernel 2

```
!$acc end kernels
```

C

```
#pragma acc kernels [clause ...]  
{ structured block }
```

The compiler
identifies 2 parallel
loops and generates
2 kernels



OpenACC parallel vs. kernels

parallel

- Requires analysis by programmer to ensure safe parallelism
- Straightforward path from OpenMP
- Mandatory to fully control the different levels of parallelism
- Implicit barrier at the end of the parallel region

kernels

- Compiler performs parallel analysis and parallelizes what it believes safe
- Can cover larger area of code with a single directive
- Please, write clean codes and add directives to help the compiler
- Implicit barrier at the end and between each kernel (e.g. loop)

Which is the best?



C tip: the `restrict` keyword

- Declaration of intent given by the programmer to the compiler

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”*

- Limits the effects of pointer aliasing
- OpenACC compilers often require `restrict` to determine independence between the iterations of a loop
 - Crucial when adopting `kernel`s directive, but also for other optimizations
 - Note: if the programmer violates the declaration, the behavior is undefined



Complete SAXPY example code

- Use restrict to help the compiler when adopting **kernels**
 - Apply a **loop** directive
- Be careful: **restrict** is C99 but not C++ standard

```
#include <stdlib.h>

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

*restrict:
"I promise y does not alias x"

```
int main(int argc, char **argv)
{
    int N = 1<<20; // 1 million floats

    if (argc > 1)
        N = atoi(argv[1]);

    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;
    }

    saxpy(N, 3.0f, x, y);

    return 0;
}
```



Loop Construct

- Applies to a loop which must immediately follow this directive
- Describes:
 - type of parallelism
 - loop-private variables, arrays, and reduction operations
- We already encountered it combined with the **parallel** directive
 - combining kernels and loop is also possible but limits the capability of **kernels** construct (i.e. extending to wide regions of code)

C

```
#pragma acc loop [clause ...]  
{ for block }
```

Fortran

```
!$acc loop [clause ...]  
{ do block }
```



Independent clause

- In a **kernels** construct, the **independent loop** clause helps the compiler in guaranteeing that the iterations of the loop are independent wrt each other
- E.g., consider $m > n$

```
#pragma acc kernels
#pragma acc loop independent
    for(int i;i<n;i++)
        c[i] = 2.*c[m+i];
```
- In **parallel** construct the independent clause is implied on all **loop** directives without a **seq** clause



Seq and collapse

- The `seq` clause specifies that the associated `loops` have to be executed sequentially on the accelerator
- Beware: the `loop` directive applies to the immediately following loop

```
#pragma acc parallel
#pragma acc loop collapse(2)    // independent is automatically enforced
for(int i;i<n;i++)
for(int k;k<n;k++)
#pragma acc loop seq
for(int j;j<n;j++)
    c[i][j][k] = 2.*c[i][j+1][k];
```

- `collapse(<n_loops>)` clause allows for extending `loop` to tightly nested loops
 - but the compiler may decide to collapse loops anyway, check the report!



Loop reductions

- The **reduction** clause on a **loop** specifies a reduction operator on one or more scalar variables
 - For each variable, a private copy is created for each thread executing the associated loops
 - At the end of the loop, the values for each thread are combined using the reduction clause
- Reductions may be defined even at **parallel** level (advanced topic)
- Common operators are supported:

+ * max min && ||



Finding Parallelism in your code

- (Nested) **for** loops are best for parallelization
- Large loop counts needed to offset GPU/memcpy overhead
- Iterations of loops must be independent of each other
 - To help compiler: **restrict** keyword (C), **independent** clause
- Compiler must be able to figure out sizes of data regions
 - You can use directives to explicitly control sizes (see next)
- Pointer arithmetic should be avoided if possible
 - Use subscripted arrays, rather than pointer-indexed arrays.
- Function calls within accelerated regions must be handled with care



Environment and Conditional Compilation

`ACC_DEVICE` *device*

Specifies which device type to connect to.

`ACC_DEVICE_NUM` *num*

Specifies which device number to connect to.

`_OPENACC`

Preprocessor directive for conditional compilation.
Set to OpenACC version



Runtime Library Routines

Fortran

```
use openacc  
#include "openacc_lib.h"
```

```
acc_get_num_devices  
acc_set_device_type  
acc_get_device_type  
acc_set_device_num  
acc_get_device_num  
acc_async_test  
acc_async_test_all
```

C

```
#include "openacc.h"
```

```
acc_async_wait  
acc_async_wait_all  
acc_shutdown  
acc_on_device  
acc_malloc  
acc_free
```



Selecting the device

- Device selection can be achieved by OpenACC runtime library routines
 - device type: `acc_device_cuda/acc_device_nvidia` for PGI
 - GPUs are numbered starting from 0 (PGI)

```
#ifdef _OPENACC
    int mygpu, myrealgpu, num_devices;    acc_device_t my_device_type;
#ifdef CAPS
    my_device_type = acc_device_cuda;
#elif PGI
    my_device_type = acc_device_nvidia;
#endif
    if(argc == 1) mygpu = 0; else mygpu = atoi(argv[1]);
    acc_set_device_type(my_device_type) ;
    num_devices = acc_get_num_devices(my_device_type) ;
    fprintf(stderr,"Number of devices available: %d \n ",num_devices);
    acc_set_device_num(mygpu,my_device_type);
    fprintf(stderr,"Trying to use GPU: %d \n",mygpu);
    myrealgpu = acc_get_device_num(my_device_type);
    fprintf(stderr,"Actually I am using GPU: %d \n",myrealgpu);
    if(mygpu != myrealgpu) {fprintf(stderr,"I cannot use the requested GPU: %d\n",mygpu);exit(1); }
#endif
```



Compiling and running (PGI)

- Example of compilation:

```
pgcc -acc=noautopar -ta=nvidia -Minfo=accel -o saxpy_acc saxpy.c -DPGI
```

noautopar is needed to avoid automatic parallelization of loops

- Compiler output (-Minfo=accel):

```
pgcc -acc -Minfo=accel -ta=nvidia -o saxpy_acc saxpy.c
saxpy:
  8, Generating copyin(x[:n-1])
    Generating copy(y[:n-1])
    Generating compute capability 1.0 binary
    Generating compute capability 2.0 binary
  9, Loop is parallelizable
    Accelerator kernel generated
      9, #pragma acc loop worker, vector(256) /* blockIdx.x threadIdx.x */
        CC 1.0 : 4 registers; 52 shared, 4 constant, 0 local memory bytes; 100% occupancy
        CC 2.0 : 8 registers; 4 shared, 64 constant, 0 local memory bytes; 100% occupancy
```

- For the hands-on, compile using the makefile and run by typing
 - **make pgi**
 - **./laplace2d_acc_pgi N** (N is the GPU number to use, 0 1 2 ...)



Submission scripts

Submission scripts

[submit_acc.sh](#) – [submit_omp.sh](#) – [submit_mpiacc.sh](#)

```
#!/bin/bash
#PBS -N laplace_acc
#PBS -o job.out
##PBS -e job.err
#PBS -j oe
#PBS -l walltime=00:10:00
#PBS -l select=1:ncpus=1:ngpus=1:mpiprocs=1
#PBS -q debug
#PBS -A train_scA2014
##PBS -q R426809
```

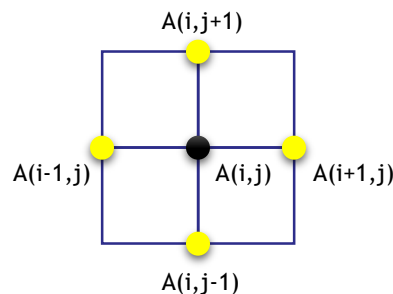
```
module load pgi/14.1
cd $PBS_O_WORKDIR
DIR=001-laplace2D-accparallel
#DIR=002-laplace2D-kernels
#DIR=003-laplace2D-collapse
#DIR=004-laplace2D-data
#DIR=005-laplace2D-declare
#DIR=006-laplace2D-function
#DIR=007-laplace2D-withcuda
#DIR=008-laplace2D-dynamic
#DIR=009-laplace2D-c++
cd $DIR
./laplace2d_acc_pgi
```



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:

$$\nabla^2 f(x, y) = 0$$



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



Jacobi Iteration C Code

```
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 = max(error, 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++;  
}
```

Note! This is an didactic implementation: the second loop could be avoided implementing a 2-step iteration
 $A \rightarrow Anew$; $Anew \rightarrow A$

Iterate until converged

Iterate across matrix elements



Calculate new value from neighbors

Compute max error for convergence

Swap input/output arrays



Base Exercise 0: OpenMP C Code

```
while ( error > tol && iter < iter_max ) {  
    error=0.0;  
  
    #pragma omp parallel for shared(m, n, Anew, A) reduction(max:error)   
    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 = max(error, abs(Anew[j][i] - A[j][i]));  
        }  
    }  
  
    #pragma omp parallel for shared(m, n, Anew, A)   
    for( int j = 1; j < n-1; j++) {  
        for( int i = 1; i < m-1; i++ ) {  
            A[j][i] = Anew[j][i];  
        }  
    }  
  
    iter++;  
}
```

Parallelize loop across
CPU threads

Parallelize loop across
CPU threads



Exercises: More Compiling Instructions

- To compile, first load environment variables for the compiler
`module load pgi/14.10`
- Then, use the provided makefile, e.g.
 - > `cd 000-laplace2D-openmp`
 - C:
 - `make pgi`

Take the elapsed times using 1 or more threads

- `setenv OMP_NUM_THREADS 6` if using tcsh
- `export OMP_NUM_THREADS=6` if using bash



GPU startup overhead

- If no other GPU process running, GPU driver may be swapped out
 - Linux specific
 - Starting it up can take 1-2 seconds
- Two options
 - Run `nvidia-smi` in persistence mode (requires root privileges)
 - Run “`nvidia-smi -q -l 30`” in the background
- Nvidia-smi should be running in persistent mode for these exercises



Exercise 1: Jacobi Acc parallel

- Task: use **acc parallel** to parallelize the Jacobi nested loops
- Edit `laplace2d.c`
 - In the `001-laplace2D-accparallel` directory
 - Add directives where needed
 - Modify the Makefile to activate the acceleration
 - **PGI compiler**
 - **Figure out the proper compilation commands (similar to SAXPY example)**
 - Compile and run the OpenACC version and compare the performances with that of OpenMP version, in the `000-laplace2D-openmp`
 - **compare the performances using `OMP_NUM_THREADS=1` and `OMP_NUM_THREADS=4` or more**



Exercise 1 Solution: OpenACC C

```
while ( error > tol && iter < iter_max ) {  
    error=0.0;  
  
    #pragma acc parallel loop reduction(max:error)  
    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 = max(error, 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++;  
}
```



Execute GPU kernel for
loop nest



Execute GPU kernel for
loop nest



Exercise 1 Solution: OpenACC Fortran

```
do while ( err > tol .and. iter < iter_max )  
  err=0._fp_kind
```

```
!$acc parallel loop reduction(max:err)
```

```
  do j=1,m
```

```
    do i=1,n
```

```
      Anew(i,j) = .25_fp_kind * (A(i+1, j ) + A(i-1, j ) + &  
                               A(i , j-1) + A(i , j+1))
```

```
      err = max(err, Anew(i,j) - A(i,j))
```

```
    end do
```

```
  end do
```

```
!$acc end parallel loop
```

```
!$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 loop
```

```
  iter = iter +1
```

```
end do
```



Generate GPU kernel for
loop nest



Generate GPU kernel for
loop nest



Exercise 1 Solution: C Makefile

```
PGCC      = pgcc -acc=noautopar -ta=nvidia,time,cuda5.0,cc35 -Minfo=accel -O3  
GCC       = gcc -O3 #-Wall -Wextra
```

```
BIN = laplace2d_acc_pgi laplace2d_acc_caps
```

```
help:
```

```
    @echo "Please specify the make target according to the compiler"  
    @echo "make pgi"
```

```
pgi: laplace2d.c
```

```
    $(PGCC) -o laplace2d_acc_$$@ $$<
```

```
clean:
```

```
    $(RM) $(BIN) __hmpp*
```



Exercise 1: Compiler output (C - PGI)

```
pgcc -acc -ta=nvidia -Minfo=accel -o laplace2d_acc laplace2d.c
main:
 57, Generating copyin(A[:4095][:4095])
    Generating copyout(Anew[1:4094][1:4094])
    Generating compute capability 1.3 binary
    Generating compute capability 2.0 binary
 58, Loop is parallelizable
 60, Loop is parallelizable
    Accelerator kernel generated
 58, #pragma acc loop worker, vector(16) /* blockIdx.y threadIdx.y */
 60, #pragma acc loop worker, vector(16) /* blockIdx.x threadIdx.x */
    Cached references to size [18x18] block of 'A'
    CC 1.3 : 17 registers; 2656 shared, 40 constant, 0 local memory bytes; 75% occupancy
    CC 2.0 : 18 registers; 2600 shared, 80 constant, 0 local memory bytes; 100% occupancy
 64, Max reduction generated for error
 69, Generating copyout(A[1:4094][1:4094])
    Generating copyin(Anew[1:4094][1:4094])
    Generating compute capability 1.3 binary
    Generating compute capability 2.0 binary
 70, Loop is parallelizable
 72, Loop is parallelizable
    Accelerator kernel generated
 70, #pragma acc loop worker, vector(16) /* blockIdx.y threadIdx.y */
 72, #pragma acc loop worker, vector(16) /* blockIdx.x threadIdx.x */
    CC 1.3 : 8 registers; 48 shared, 8 constant, 0 local memory bytes; 100% occupancy
    CC 2.0 : 10 registers; 8 shared, 56 constant, 0 local memory bytes; 100% occupancy
```




Exercise 1: Performance

2 eight-core Intel(R) Xeon(R) CPU E5-2687W @ 3.10GHz

GPU: Nvidia Tesla K20s

| Execution | Time (s) - PGI | Time (s) - CAPS |
|-----------------------|----------------|-----------------|
| CPU 1 OpenMP thread | 21.9 | 17.1 |
| CPU 2 OpenMP threads | 11.3 | 9.3 |
| CPU 4 OpenMP threads | 6.0 | 5.6 |
| CPU 8 OpenMP threads | 3.7 | 4.0 |
| CPU 16 OpenMP threads | 3.3 | 3.5 |
| OpenACC GPU | 30 | 33 |

accelerated?
NO!



Exercise 2: Jacobi Acc kernels

- Task: use **acc kernels** to parallelize the Jacobi loops
- For this simple case, no significant difference wrt **acc parallel**
 - but, try to understand the compiler report to be sure about what the compiler is doing
- Edit `laplace2d.c` in the `002-laplace2d-kernels` directory
 - any change in performances?
 - actually, no significant change...



Ex. 2 Solution: OpenACC C

```
while ( error > tol && iter < iter_max ) {
    error=0.0;

    #pragma acc kernels loop reduction(max:error)
    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 = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }

    #pragma acc kernels
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
```



Execute GPU kernel for
loop nest



Execute GPU kernel for
loop nest



Exercise 3: Collapsing loops

- Look at the compiler report:
 - GPUs work well when there is a large number of iterations to be parallelized
 - exploiting multiple loop nesting is crucial
- Use loop construct and collapse clause to optimize loops
 - which loop is actually affected? why?
- Edit `laplace2d.c` in the `003-laplace2D-collapse` directory
 - any change in performances?



Exercise 3: Performance

2 eight-core Intel(R) Xeon(R) CPU E5-2687W @ 3.10GHz

GPU: Nvidia Tesla K20s

| Execution | Time (s) - PGI | Time (s) - CAPS |
|----------------------|----------------|-----------------|
| CPU 1 OpenMP thread | 21.9 | 17.1 |
| CPU 8 OpenMP threads | 3.7 | 4.0 |
| OpenACC GPU-parallel | 30 | 33 |
| OpenACC GPU-collapse | 27 | 30 |

Small performance gain. For PGI *noautopar* is crucial, otherwise inner loop is always parallelized



What is going wrong?

- Add `-ta=nvidia,time` to compiler command line

```
Accelerator Kernel Timing data  
001-laplace2D-kernels/laplace2d.c  
main
```

```
69: region entered 1000 times  
time(us): total=77524918 init=240 region=77524678  
kernels=4422961 data=66464916
```

4.4 seconds

66.5 seconds

```
w/o init: total=77524678 max=83398 min=72025 avg=77524  
72: kernel launched 1000 times  
grid: [256x256] block: [16x16]  
time(us): total=4422961 max=4543 min=4345 avg=4422
```

```
001-laplace2D-kernels/laplace2d.c  
main
```

```
57: region entered 1000 times  
time(us): total=82135902 init=216 region=82135686  
kernels=8346306 data=66775717
```

8.3 seconds

66.8 seconds

```
w/o init: total=82135686 max=159083 min=76575 avg=82135  
60: kernel launched 1000 times  
grid: [256x256] block: [16x16]  
time(us): total=8201000 max=8297 min=8187 avg=8201  
64: kernel launched 1000 times  
grid: [1] block: [256]  
time(us): total=145306 max=242 min=143 avg=145
```

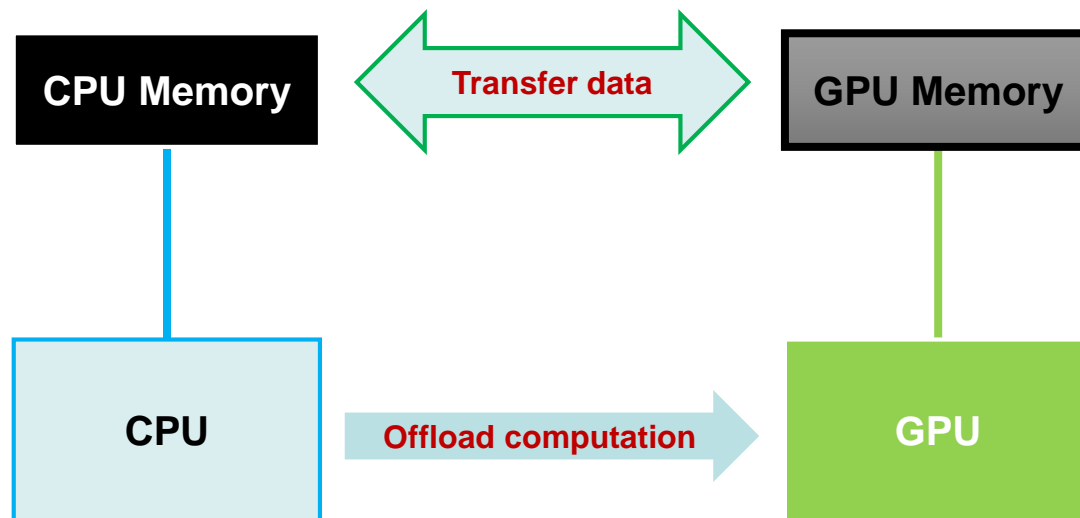
```
acc_init.c  
acc_init
```

```
29: region entered 1 time  
time(us): init=158248
```

**Huge Data Transfer
Bottleneck!**
Computation: 12.7 seconds
Data movement: 133.3 seconds



Basic Concepts



For efficiency, decouple data movement and compute off-load



Excessive Data Transfers

```
while ( error > tol && iter < iter_max ) {  
    error=0.0;
```

A, Anew resident on host

Copy

```
#pragma acc parallel loop reduction(max:error)
```

A, Anew resident on accelerator

These copies
happen every
iteration of the outer
while loop!*

```
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 = max(error, abs(Anew[j][i] - A[j][i]));  
    }  
}
```

A, Anew resident on accelerator

Copy

A, Anew resident on host

```
}  
...  
}
```

*Note: there are two #pragma acc kernels, so there are 4 copies per while loop iteration!



Golden rule: check the compiler report!

```
pgcc -acc -ta=nvidia -Minfo=accel  
laplace2d.c  
main:  
  57, Generating copyin(A[:4095][:4095])  
      Generating  
copyout(Anew[1:4094][1:4094])  
.....  
  69, Generating copyout(A[1:4094][1:4094])  
      Generating copyin(Anew[1:4094][1:4094])  
.....
```

- The compiler tries to minimize the CPU-GPU data movements
- In the previous case
 - in the first loop :
 - A is copied in, from CPU to GPU
 - Anew is copied out, from GPU to CPU
 - in the second loop:
 - Anew is copied in
 - A is copied out



DATA MANAGEMENT



Explicit data control: the naive way

- It is possible to explicitly control the data movements at the opening of the **acc parallel** regions using **data clauses**

```
#pragma acc parallel loop reduction(max:error) copyin(A) copyout(Anew)
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 = max(error, abs(Anew[j][i] - A[j][i]));
  }
}
#pragma acc parallel loop copyin(Anew) copyout(A)
for( int j = 1; j < n-1; j++) {
  for( int i = 1; i < m-1; i++ ) {
    A[j][i] = Anew[j][i];
  }
}
```

- But we cannot lower down the amount of copies because the scope of the GPU variables is limited to the accelerated regions. What now?



Data Construct

C

```
#pragma acc data [clause ...]  
{ structured block }
```

- Manages explicitly data movements
- Crucial to handle GPU data persistence
- Allows for decoupling the scope of GPU variables from that of the accelerated regions
- May be nested
- Data clauses define different possible behaviours
 - the usage is similar to that of data clauses in parallel regions

Fortran

```
!$acc data [clause ...]  
    structured block  
!$acc end data
```



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.
- and `present_or_copy[in|out]`, `present_or_create`, `deviceptr`.



Array Shaping

- The compiler sometimes cannot determine the sizes of arrays
 - you must specify them by using data clauses and array “shape”
 - you may need just a section of an array
 - sub-array syntax is allowed, in Fortran it is language-native
- C
 - `#pragma acc data copyin(a[1:size]), copyout(b[s/4:3*s/4+1])`**
- Fortran
 - `!$pragma acc data copyin(a(1:size)), copyout(b(s/4:s))`**
- Data clauses can be used on **data**, **kernels** or **parallel**



Update Executable Directive

Fortran

```
!$acc update [clause ...]
```

C

```
#pragma acc update [clause ...]
```

Clauses:

```
host( list ) Or self(list)  
device( list )
```

- Used to synchronize data among existing data when they change in the corresponding copy (e.g. update device copy after host copy changes)
- Note: subarray may be updated but updated memory must be contiguous
- Moves data from GPU to host, or host to GPU
- Data movement can be conditional and asynchronous



Exercise 4: Jacobi Data Directives

- Task: use **acc data** to minimize transfers in the Jacobi example
- Start from given `laplace2d.c` In the `002-laplace2d-data` directory
 - Add directives where needed
- Q: What speedup can you get with data + kernels directives?
 - Versus 1 CPU core? Versus 8 CPU cores?



Exercise 4 Solution: OpenACC C

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

    #pragma acc parallel loop reduction(max:error)
    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 = max(error, 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 in at the beginning of loop, out at the end. Allocate Anew on accelerator



Exercise 4: Performance

2 eight-core Intel(R) Xeon(R) CPU E5-2687W @ 3.10GHz

GPU: Nvidia Tesla K20s

| Execution | Time (s) - PGI | Time (s) - CAPS |
|----------------------|----------------|-----------------|
| CPU 1 OpenMP thread | 21.9 | 17.1 |
| CPU 8 OpenMP threads | 3.7 | 4.0 |
| OpenACC GPU-parallel | 30 | 33 |
| OpenACC GPU-collapse | 30 | 30 |
| OpenACC GPU-data | 0.9 | 1.3 |

14x compared to 1 core!
4x compared to 8 cores!



Declare data

- It is possible to further extend the scope of data on the device
 - it significantly enhances code readability and maintainability
- The **declare** directive specifies that a variable or array has to be allocated in the device memory for the duration of the implicit data region of a function
 - used in the declaration section of a function
 - may specify whether the data have to be transferred and how
 - for global scope variables, the implicit region is the whole program

C

```
float a[100];  
#pragma acc declare create(A)
```

Fortran

```
real A(100)  
!$acc declare create(A)
```



Declare data - 2

- Standard data clauses may be specified
 - copy, copyin, copyout, create, present, ... (restrictions apply for global variables)
 - specific data clauses may be employed, too
- **device_resident**
 - the memory has to be allocated on the accelerator memory and not on the host memory
- **link**
 - only a global link for the named variable should be statically created in the accelerated memory
 - to be used for large global host static data referenced within an accelerated routine
- The compiler implementations may still lag behind



Exercise 5

- Employ the **declare** directive to manage the data persistence and transfer
- Ensure that the data are correctly synchronized before and after the accelerated regions
 - beware: the default behavior for arrays is **present_or_copy** and only one number is interpreted as the size
 - hence, you need to **update** data on device or host

check the support
of your compiler
for `declare`!



Exercise 5 Solution: OpenACC C

```
double A[NN][NM];
#pragma acc declare create(A)
double Anew[NN][NM];
#pragma acc declare create(Anew)
.....
int main() {
.....
#pragma acc update device(A)
while ( error > tol && iter < iter_max ) {
    error=0.0;

#pragma acc parallel loop collapse(2) reduction(max:error)
    for( int j = 1; j < n-1; j++)
        .....
#pragma acc parallel loop collapse(2)
    for( int j = 1; j < n-1; j++)
        .....

    iter++;
}
#pragma acc update host(A)
.....
}
```



Declare A and A new to
create data on the GPU



Update A and parallelize
loops



Data and functions

- What happens when calling a function from a data region?
 - e.g., consider that the updating of A, including the loops, is performed by a function
 - the **data region** opened by the calling function applies
 - default behaviours when considering each parallel region
 - in C you just have to take care of the reduction variable using a temporary variable

```
void update_A(int n, int m, double *error)
{
    double error_loc = 0.0;
    #pragma acc parallel loop \
        reduction(max: error_loc)
    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_loc = fmax(error_loc,
                abs(Anew[j][i] - A[j][i]));
        }
    }
    *error = error_loc ;
}
```



Default behaviours and Privatization

- For arrays `present_or_copy`: no GPU allocation nor CPU-GPU copies are performed if the variable exists in a surrounding data region
 - i.e., the default behavior is shared, but private clause may be enforced to `loop`
- For scalars the rules are not trivial
 - for `acc parallel` regions, the default is `firstprivate` (private but initialized with the global value)
 - for `kernels`, default is `copy` (in and out), and private cannot be specified
 - if needed, it is usually best to specify `private` at the `loop` construct level

```
void copy_A(int n, int m) {  
  #pragma acc parallel loop  
  for( int j = 1; j < n-1; j++) {  
    #pragma acc loop private(Asca)  
    for( int i = 1; i < m-1; i++ ) {  
      Asca = Anew[j][i];  
      A[j][i] = Asca;  
    }  
  }  
}
```

check the compiler
report!
PGI compiler performs a
live-in/live-out
analysis to help you



Exercise 6

- Add directives for the case with called functions
 - Look at the compiler messages: what about intermediate copies of A and Anew? Why?
- Add also the value of A(2,2) printed together to iter,error
- Update subarray rules:
 - In fortran `v(start_x:end_x,start_y:end_y)`
 - if start or end are not specified, the array bounds are used
 - and only one number is interpreted as the end
 - In C `v[start_x:size_x][start_y:size_y]`
 - if start or size are not specified, the array bounds are used, if known
 - and only one number is interpreted as the size

multidimensional
section updates not
supported by CAPS
3.4.1, update the
whole matrix



Exercise 6: solution excerpt

C

```
if(iter % 100 == 0) {  
    #pragma acc update host(A[2:1][2:1])  
    printf("%5d, %0.6f, %0.6f", iter, error, A[2][2]);  
}
```

Fortran

```
if(mod(iter,100).eq.0 ) then  
    !$acc update host(A(2:2,2:2))  
    write(*,'(i5,f10.6,f10.6)'), iter, error, A(2,2)  
endif
```



Calling functions

- What happens if an accelerated region calls a function?

- e.g., performing the updating for each grid point

```
#pragma acc parallel loop
```

```
for( int j = 1; j < n-1; j++) {  
    for( int i = 1; i < m-1; i++ ) {  
        update_grid_point(m,n,i,j);  
        error = fmax( error, fabs(Anew[j][i] - A[j][i]));  
    }  
}
```

- According to OpenACC1.0, the only solution is to inline the function
 - it can be done manually
 - or by relying on smart usage of the preprocessor
 - or the compiler may inline the function, using adequate options



Inlining tricks

- Inlining of functions depends on the compiler skills
- But you can help the compiler
- Two important points when using PGI
 - `–Minline=name:<function name>`
 - automatic arrays defined inside the function must be avoided: pass them even if you do not need their values in the calling program
 - in Fortran, reshaping arrays (different shapes of array from caller to dummy arguments) must be explicitly requested specifying
 - `–Minline=reshape`



Linking accelerated functions

- From OpenACC 2.0, the **acc routine** allows for effective separate compilation and correct linking
 - it tells the compiler that there will be a device copy of the routine

```
#pragma acc routine
```

```
extern void update_grid_point(n,m,i,j);
```

```
#pragma acc parallel loop
```

```
for( int j = 1; j < n-1; j++) {
```

```
    for( int i = 1; i < m-1; i++ )      {
```

```
        update_grid_point(m,n,i,j);
```

```
        error = fmax( error, fabs(Anew[j][i] - A[j][i]));
```

```
    }
```

```
}
```

```
#pragma acc routine
```

```
extern void update_grid_point(n,m,i,j) {
```

```
    ...
```

```
}
```

Linking accelerated functions / 2

- Crucial to keep the existing code modularity
- Check if available with your compiler!
- It is possible to rename the device function using the **bind** clause
 - it tells the compiler that there will be a device version of the routine with a different name
 - and when defining the function **nohost** avoid the host compilation

```
#pragma acc routine bind(update_grid_point_dev)
extern void update_grid_point(n,m,i,j);
```

```
#pragma acc parallel loop
.....
```

```
#pragma acc routine nohost
extern void update_grid_point_dev(n,m,i,j) {
...
}
```



Further speedups

- OpenACC gives us more detailed control over parallelization
 - Via gang, worker, and vector clauses
- By understanding more about OpenACC execution model and GPU hardware organization, we can get higher speedups on this code
- By understanding bottlenecks in the code via profiling, we can reorganize the code for higher performance
- Will tackle these in later exercises



Tips and Tricks

- (PGI) Use time option to learn where time is being spent
 - `-ta=nvidia,time`
- Eliminate pointer arithmetic
- Inline function calls in directives regions
 - (PGI): `-inline` or `-inline,levels(<N>)`
- Use contiguous memory for multi-dimensional arrays
- Use data regions to avoid excessive memory transfers
- Conditional compilation with `_OPENACC` macro



OpenACC and CUDA interoperability



3 Ways to Accelerate Applications

Applications

Libraries

OpenACC
Directives

Programming
Languages

CUDA Libraries are
interoperable with OpenACC

“Drop-in”
Acceleration

Easily Accelerate
Applications

Maximum
Flexibility



3 Ways to Accelerate Applications

Applications

Libraries

“Drop-in”
Acceleration

OpenACC Directives

Easily Accelerate
Applications

Programming Languages

Maximum
Flexibility

CUDA Languages are
interoperable with OpenACC,
too!



Sharing data with libraries

- CUDA libraries and OpenACC both operate on device arrays
- OpenACC provides mechanisms for interop with library calls
 - deviceptr data clause
 - host_data construct
- Note: same mechanisms useful for interop with custom CUDA C/C++/Fortran code



deviceptr Data Clause

`deviceptr(list)` Declares that the pointers in *list* refer to device pointers that need not be allocated or moved between the host and device for this pointer.

Example:

C

```
#pragma acc data deviceptr(d_input)
```

Fortran

```
$(acc data deviceptr(d_input))
```



host_data Construct

Makes the address of device data available on the host.

`host_data(list)` Tells the compiler to use the device address for any variable in *list*. Variables in the list must be present in device memory due to data regions that contain this construct

Example

C

```
#pragma acc host_data use_device(d_input)
```

Fortran

```
!acc host_data use_device(d_input)
```



Example: 1D convolution using CUFFT

- Perform convolution in frequency space
 1. Use CUFFT to transform input signal and filter kernel into the frequency domain
 2. Perform point-wise complex multiply and scale on transformed signal
 3. Use CUFFT to transform result back into the time domain
- We will perform step 2 using OpenACC



Source Excerpt

```
// Transform signal and kernel
error = cufftExecC2C(plan, (cufftComplex *)d_signal,
                      (cufftComplex *)d_signal, CUFFT_FORWARD);
error = cufftExecC2C(plan, (cufftComplex *)d_filter_kernel,
                      (cufftComplex *)d_filter_kernel, CUFFT_FORWARD);

// Multiply the coefficients together and normalize the result
printf("Performing point-wise complex multiply and scale.\n");
complexPointwiseMulAndScale(new_size,
                            (float *restrict)d_signal,
                            (float *restrict)d_filter_kernel);

// Transform signal back
error = cufftExecC2C(plan, (cufftComplex *)d_signal,
                      (cufftComplex *)d_signal, CUFFT_INVERSE);
```

This function
must execute on
device data



OpenACC convolution code

```
void complexPointwiseMulAndScale(int n, float *restrict signal,  
                                float *restrict filter_kernel)  
{  
    // Multiply the coefficients together and normalize the result  
    #pragma acc data deviceptr(signal, filter_kernel)  
    {  
        #pragma acc parallel loop  
        for (int i = 0; i < n; i++) {  
            float ax = signal[2*i];  
            float ay = signal[2*i+1];  
            float bx = filter_kernel[2*i];  
            float by = filter_kernel[2*i+1];  
            float s = 1.0f / n;  
            float cx = s * (ax * bx - ay * by);  
            float cy = s * (ax * by + ay * bx);  
            signal[2*i] = cx;  
            signal[2*i+1] = cy;  
        }  
    }  
}
```

If the OpenACC compiler does not support structs in OpenACC loops, then we cast the `Complex*` pointers to `float*` pointers and use interleaved indexing



Linking CUFFT

- `#include "cufft.h"`
- Compiler command line options:

```
CUDA_PATH = /usr/local/pgi/linux86-64/2012/cuda/4.0  
CCFLAGS = -I$(CUDA_PATH)/include -L$(CUDA_PATH)/lib64  
          -lcudart -lcufft
```

Must use
PGI-provided
CUDA toolkit
paths

Must link `libcudart`
and `libcufft`



Results

```
[harrism@kollman0 cufft-acc]$ ./cufft_acc  
Transforming signal cufftExecC2C  
Performing point-wise complex multiply and scale.  
Transforming signal back cufftExecC2C  
Performing Convolution on the host and checking correctness
```

```
Signal size: 500000, filter size: 33  
Total Device Convolution Time: 11.461152 ms (0.242624 for point-wise  
convolution)  
Test PASSED
```

CUFFT + cudaMemcpy

OpenACC



Summary

- Use `deviceptr` data clause to pass pre-allocated device data to OpenACC regions and loops
- Use `host_data` to get device address for pointers inside acc data regions
- The same techniques shown here can be used to share device data between OpenACC loops and
 - Your custom CUDA C/C++/Fortran/etc. device code
 - Any CUDA Library that uses CUDA device pointers



Exercise 7

- Try to mix OpenACC and CUDA paradigms for the Laplace example
- The basic idea is to focus on some sections of code and optimize them employing CUDA APIs
- As an example, try to implement the second loop calling a `cudaMemcpy`, a DeviceToDevice copy
 - use the `host_data` directive to make the data visible to the host
 - you need to define an extern C function which calls the `cudaMemcpy`
 - passing the arrays and the extents
- Compilation, modify the Makefile consistently
 - compile the CUDA function using `nvcc` and the openacc code using PGI
 - then, link the objects together



Exercise 7 - Solution

- Replace the second accelerated loop with

```
#pragma acc host_data use_device(A,Anew)
{
    cudaFun(A,Anew,m,n);
}
```

- Create a file named `cudaFun.cu` with the CUDA function

```
extern "C" void cudaFun(double **A, double **Anew, int n, int m) {
    cudaMemcpy((double*)A,(double*)Anew,m*n*sizeof(double),
               cudaMemcpyDeviceToDevice);
}
```

- Modify the data clause for `Anew` since the `Memcpy` extends to the boundaries now...

```
#pragma acc data copy(A), copyin(Anew)
```



Exercise 7 - Makefile

```
CUDA_PATH = /cineca/prod/compilers/pgi/13.10/none/linux86-64/2013/cuda/5.0/
```

```
PGCC      = pgcc -g -acc -ta=nvidia,time,cuda5.0,cc35 -Minfo=accel -O3 -DPGI
```

```
GCC       = gcc -O3 -DCAPS #-Wall -Wextra
```

```
CAPSMC    = capsmc --codelet-required # --debug -g -G
```

```
CUDA_FLAGS = -I$(CUDA_PATH)/include -L$(CUDA_PATH)/lib64 -lcudart
```

```
pgi: laplace2d_acc_pgi.o cudaFun.o
```

```
$(PGCC) laplace2d_acc_pgi.o cudaFun.o $(CUDA_FLAGS) -o laplace2d_acc_pgi_withcuda
```

```
laplace2d_acc_pgi.o: laplace2d.c
```

```
$(PGCC) -c -o laplace2d_acc_pgi.o $<
```

```
caps: laplace2d_acc_caps.o cudaFun.o
```

```
$(CAPSMC) $(GCC) laplace2d_acc_caps.o cudaFun.o $(CUDA_FLAGS) -o laplace2d_acc_caps_withcuda
```

```
laplace2d_acc_caps.o: laplace2d.c
```

```
$(CAPSMC) $(GCC) -c -o laplace2d_acc_caps.o $<
```

```
cudaFun.o: cudaFun.cu
```

```
nvcc -c cudaFun.cu -o cudaFun.o
```



Exercise 7: Performance

2 eight-core Intel(R) Xeon(R) CPU E5-2687W @ 3.10GHz

GPU: Nvidia Tesla K20s

| Execution | Time (s) - PGI | Time (s) - CAPS |
|----------------------|----------------|-----------------|
| CPU 1 OpenMP thread | 21.9 | 17.1 |
| CPU 8 OpenMP threads | 3.7 | 4.0 |
| OpenACC GPU-parallel | 30 | 33 |
| OpenACC GPU-collapse | 30 | 30 |
| OpenACC GPU-data | 0.9 | 1.3 |
| OpenACC GPU-withcuda | 0.8 | 1.3 |

no significant
improvement for
this simple
case



Exercise 8

- Up to now, we used statically allocated memory
 - Using OpenACC for dynamically allocated data is possible
 - data extents need to be explicitly specified
- It may be more difficult for the compiler to assess the independence of iterations
 - this is obviously crucial adopting the kernels directive
 - but, it may be important even using parallel directive, for the internal loops
 - best practice: use explicit loop directive or collapse clause to strongly control the parallelization of each loop
- Try to accelerate the code in the directory 009-laplace2D-dynamic
 - **using parallel directive**
 - **using kernel directive**



Exercise 8 – Solution excerpts

- Data extents must be specified
`#pragma acc data copy(T[0:n2*n2]), create(Tnew[0:n2*n2])`
- Parallel directive may be used unchanged
`#pragma acc parallel loop collapse(2)`
- If using kernels, loop and collapse have to be specified, otherwise the compiler probably decides to serialize the loop
`#pragma acc kernels loop collapse(2)`

What about performances? Very good



OpenACC and C++

- C++ is supported according to the OpenACC standard
 - but compilers may significantly lag behind!
- In **data** constructs
 - if a variable or array of struct or class is specified, all the data members of the struct or class are allocated and copied, as appropriate
 - if a struct or class member is a pointer type, the data addressed by that pointer are not implicitly copied
- At present, using PGI or CAPS, the code needs to be adapted recovering a C-like style
 - PGI has its own C++ compiler while CAPS relies on an underlying compiler
 - use **_OPENACC** macro to differentiate the code
 - it clearly limits the code maintainability



Exercise 9

- Implement OpenACC acceleration to the Laplace C++ code
 - the code employs vector containers and multi-index-like overloading
 - you need to extract the pointer to the data, overloading the () could be an idea

```
class field {  
    int n;    int m;    vector<double> vec;  
public:  
    field(int nn, int mm) : n(nn), m(mm) { vec.assign(nn*mm,0.); }  
    double& operator() (int i, int j) { return vec[m*i+j]; }  
    double* operator() () { return &(vec[0]); }  
};
```

- to access vector indexes, a macro could be employed
#define IDX(i,j) ((i)*(NM)+(j))



Exercise 9 - Solution

```
double *A_p, *Anew_p;
.....
field A(n,m); field Anew(n,m);
.....
A_p = A(); Anew_p = Anew();
#pragma acc data copy(A_p[0:NN*NM]), create(Anew_p[0:NN*NM])
{
while ( error > tol && iter < iter_max ) {
error = 0.0;
#pragma acc parallel loop collapse(2) reduction(max: error)
for( j = 1; j < n-1; j++) {
for( i = 1; i < m-1; i++) {
#ifdef _OPENACC
Anew_p[IDX(j,i)] = 0.25*( A_p[IDX(j,i+1)]+A_p[IDX(j,i-1)]+A_p[IDX(j-1,i)]+A_p[IDX(j+1,i)]);
error = fmax( error, fabs(Anew_p[IDX(j,i)] - A_p[IDX(j,i)]));
#else
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)));
#endif
} }
.....
```



OpenACC and MPI

- OpenACC may be used to accelerate the codes featuring MPI parallelization
 - each MPI process may be accelerated, just like what happens when adding OpenMP simple constructs to MPI codes
- When dealing with MPI communications, obvious data synchronizations must be performed
 - update host before sending data
 - update device after receiving data
- In the simplest approach, each MPI process controls a different GPU
 - again, use runtime library routines
 - possibly, some MPI processes use GPUs while other processes use the host as the computing unit



Exercise 10

- Add OpenACC directives to the MPI laplace code in the directory 010-laplace2D-mpi/
 - the baseline features a 2D MPI Cartesian decomposition
 - before evolving the field data, halo data are copied into buffers and MPI exchanges are performed
- The association of MPI processes to GPUs is already prepared considering a machine with a defined number of GPUs
 - #define NGPU_PER_NODE 4**
 - mygpu = rank%NGPU_PER_NODE;**
 - employ MPI colours to make it more general
- Remember to specify the array extents to be created or updated
 - when dealing with MPI, dynamic memory is a common choice



Exercise 10 – Solution excerpt

```
#pragma acc data copy(T[0:stride_x*stride_y]), create(Tnew[0:stride_x*stride_y]), \  
create(buffer_s_rl[0:mymysize_y]), create(buffer_s_lr[0:mymysize_y]), \  
create(buffer_s_tb[0:mymysize_x]), create(buffer_s_bt[0:mymysize_x]), \  
create(buffer_r_rl[0:mymysize_y]), create(buffer_r_lr[0:mymysize_y]), \  
create(buffer_r_tb[0:mymysize_x]), create(buffer_r_bt[0:mymysize_x])
```

.....

```
#pragma acc kernels
```

```
#pragma acc loop independent
```

```
for(j = 1; j<=mymysize_y; j++)  
    buffer_s_rl[j-1] = T[stride_y+j];
```

```
#pragma acc update host(buffer_s_rl[0:mymysize_y])
```

.....

```
#pragma acc update device(buffer_r_rl[0:mymysize_y])
```

```
#pragma acc parallel loop collapse(2) reduction(max:myvar)
```

```
for (i=1; i<=mymysize_x; ++i) {  
    for (j=1; j<=mymysize_y; ++j) {
```

use CAPS
compiler, PGI
does not handle
reductions over
nested loops
when dynamic
memory
allocation
occurs



Exercise 10: Performance

2 eight-core Intel(R) Xeon(R) CPU E5-2687W @ 3.10GHz

GPU: Nvidia Tesla K20s

| Execution | Time (s) - PGI | Time (s) - CAPS |
|-------------------------|----------------|-----------------|
| CPU 8 OpenMP threads | 3.7 | 4.0 |
| OpenACC GPU-parallel | 30 | 33 |
| OpenACC GPU-collapse | 30 | 30 |
| OpenACC GPU-data | 0.9 | 1.3 |
| OpenACC GPU-withcuda | 0.8 | 1.3 |
| OpenACC GPU 2 MPI procs | 0.5 | 0.7 |

significant
scalability



Optimizing OpenACC parallelization



Cache Directive

- The cache directive may appear at the top of (inside of) a loop
 - it specifies array elements or subarrays that should be fetched into the highest level of the cache for the body of the loop
 - obviously the actual performance gain strongly depends on the analyzed loop

```
#pragma acc parallel loop collapse(2) reduction(max:error)
for( j = 1; j < n-1; j++) {
    for( i = 1; i < m-1; i++ ) {
        #pragma acc cache(A[i-1:2][j-1:2])
        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]));
    }
}
```



Parallel Construct

Fortran

```
!$acc parallel [clause ...]  
    structured block  
!$acc end parallel
```

Clauses

```
if( condition )  
async( expression )  
num_gangs( expression )  
num_workers( expression )  
vector_length( expression )
```

C

```
#pragma acc parallel [clause ...]  
    { structured block }
```

```
private( list )  
firstprivate( list )  
reduction( operator:list )
```

Also any data clause



Parallel Clauses

- `num_gangs (expression)` Controls how many parallel gangs are created (CUDA `gridDim`).
- `num_workers (expression)` Controls how many workers are created in each gang (CUDA `blockDim`).
- `vector_length (list)` Controls vector length of each worker (SIMD execution).
- `private(list)` A copy of each variable in *list* is allocated to each gang.
- `firstprivate (list)` `private` variables initialized from host.
- `reduction(operator:list)` `private` variables combined across gangs.



Loop Construct

Fortran

```
!$acc loop [clause ...]  
  loop  
!$acc end loop
```

C

```
#pragma acc loop [clause ...]  
  { loop }
```

Combined directives

```
!$acc parallel loop [clause ...]  
!$acc kernels loop [clause ...]
```

Detailed control of the parallel execution of the following loop.



Loop Clauses

`collapse(n)`

Applies directive to the following n nested loops.

`seq`

Executes the loop sequentially on the GPU. Often, something went wrong...

`private(list)`

A copy of each variable in list is created for each iteration of the loop. It is default behavior

`reduction(operator:list)` `private` variables combined across iterations.



Loop Clauses Inside parallel Region

gang

Shares iterations across the gangs of the parallel region.

worker

Shares iterations across the workers of the gang.

vector

Execute the iterations in SIMD mode



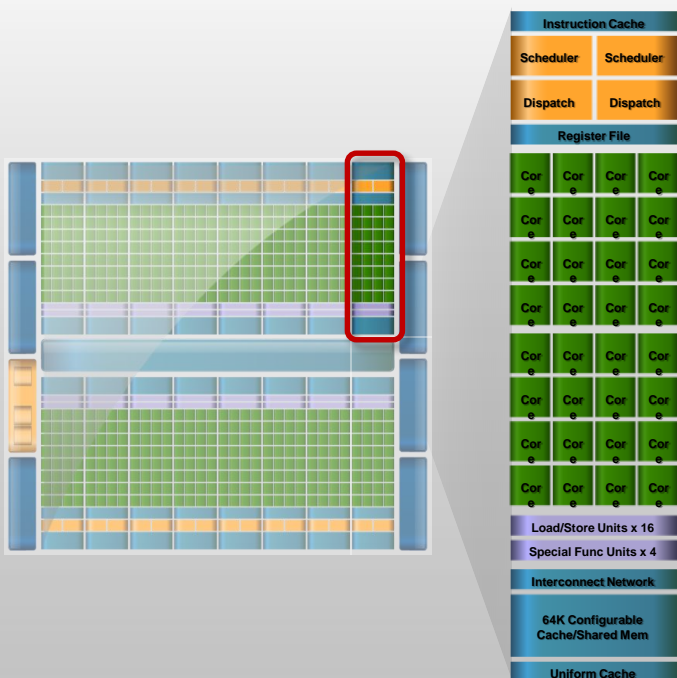
Loop Clauses Inside kernels Region

- `gang [(num_gangs)]` Shares iterations across at most `num_gangs` gangs.
- `worker [(num_workers)]` Shares iterations across at most `num_workers` of a single gang.
- `vector [(vector_length)]` Executes the iterations in SIMD mode with maximum `vector_length`.
- `independent` Specifies that the loop iterations are independent.

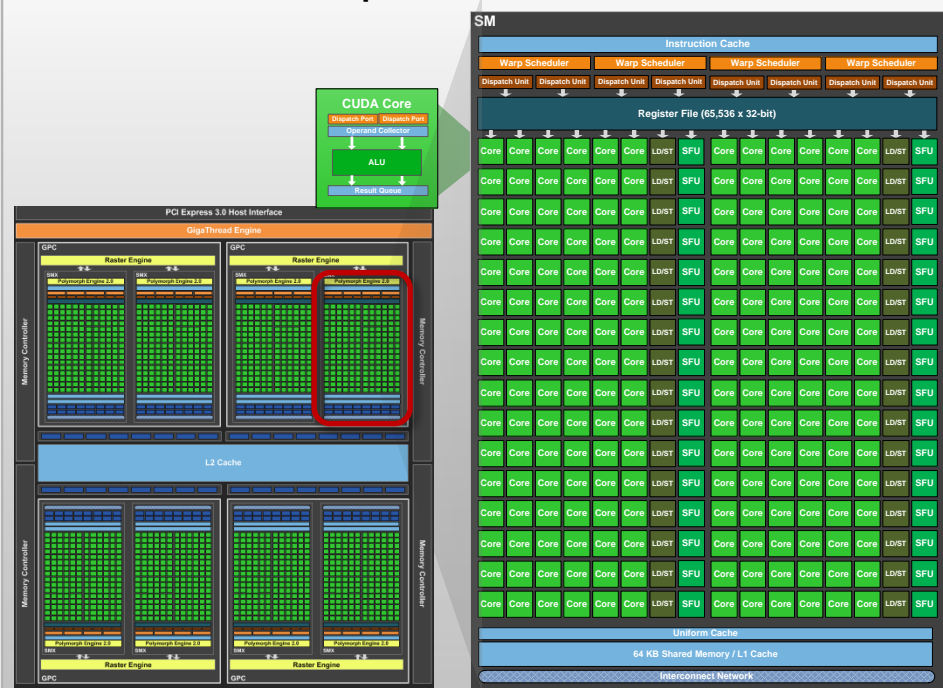


Fermi and Kepler architectures sketches

Fermi



Kepler



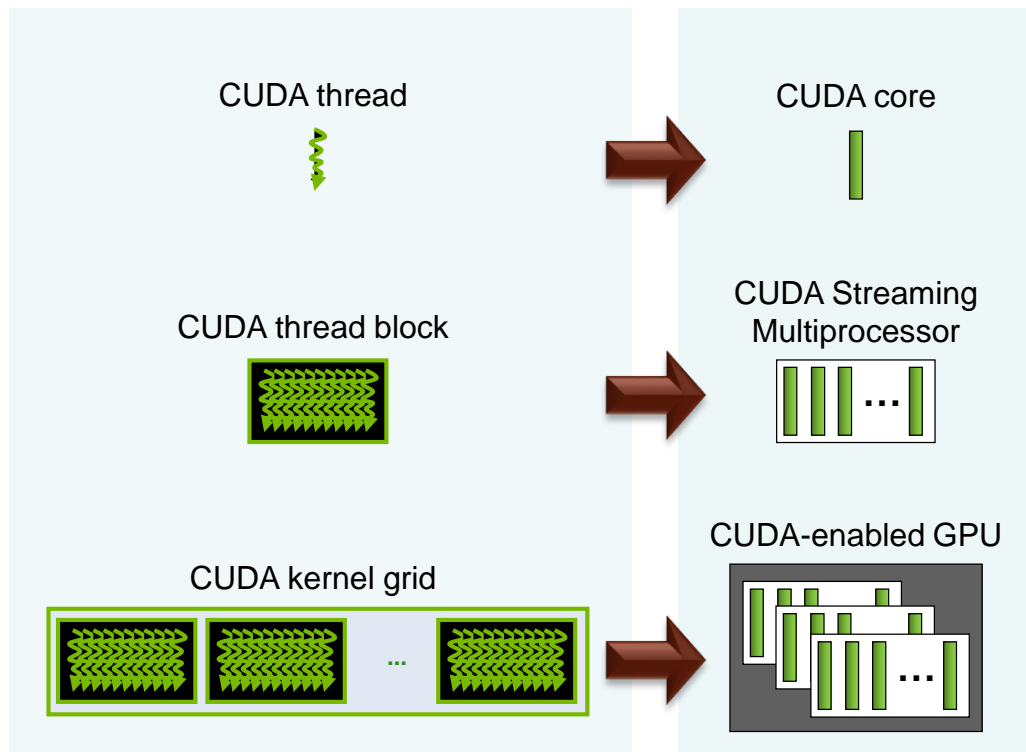


OpenACC and targets

- OpenACC on NVIDIA GPUs compiles to target the CUDA platform
 - CUDA is a parallel computing platform and programming model invented by NVIDIA.
- OpenACC may potentially target different architectures
 - NVIDIA GPU, AMD GPU, Intel MIC, many-cores and CPUs, too
 - CAPS compiler allows for producing OpenCL code instead of CUDA code
 - PGI is testing OpenACC on AMD Radeon cards
- The mapping between OpenACC parallel levels and target is performed by the compiler



CUDA and Kernel Execution



- Each thread is executed by a core
- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks' memory requirements and the SM's memory resources
- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time



OpenACC execution Model

- The OpenACC execution model has three levels:
 - gang, worker and vector
- This is supposed to map to an architecture that is a collection of Processing Elements (PEs)
 - Each PE is multithreaded and each thread can execute vector instructions
- For GPUs one possible mappings could be
 - **gang==block, worker==warp, vector==threads in a warp**
 - **omit “worker” and just have gang==block, vector==threads of a block**
- Depends on what the compiler thinks is the best mapping for the problem



Mapping OpenACC to CUDA threads and blocks

`#pragma acc kernels`

```
for( int i = 0; i < n; ++i ) y[i] += a*x[i];
```

Let the compiler decide: probably
16 blocks, 256 threads each

`#pragma acc kernels loop gang(100) vector(128)`

```
for( int i = 0; i < n; ++i ) y[i] += a*x[i];
```

100 thread blocks, each with 128
threads, each thread executes
one iteration of the loop, using
kernels

`#pragma acc parallel num_gangs(100) vector_length(128)`

```
{  
  #pragma acc loop gang vector  
  for( int i = 0; i < n; ++i ) y[i] += a*x[i]; }
```

100 thread blocks, each with 128
threads, each thread executes
one iteration of the loop, using
parallel



Mapping OpenACC to CUDA threads and blocks

```
#pragma acc parallel num_gangs(100)
```

```
{  
  for( int i = 0; i < n; ++i ) y[i] += a*x[i]; }
```

100 thread blocks, each with
apparently 1 thread, each thread
redundantly executes the loop

```
#pragma acc parallel num_gangs(100)
```

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

compiler can notice that only
'gangs' are being created, so it
might decide to create threads
instead, say 2 thread blocks of 50
threads.



Mapping OpenACC to CUDA threads and blocks

```
#pragma acc kernels loop gang(100) vector(128)
```

```
for( int i = 0; i < n; ++i ) y[i] += a*x[i];
```

100 thread blocks, each with 128 threads, each thread executes one iteration of the loop, using kernels

```
#pragma acc kernels loop gang(50) vector(128)
```

```
for( int i = 0; i < n; ++i ) y[i] += a*x[i];
```

50 thread blocks, each with 128 threads. Each thread does two elements worth of work

Doing multiple iterations per thread can improve performance by amortizing the cost of setup



Tuning loops

- Selecting the optimal strategy to match the 3 OpenACC layers to the code loops is usually a compiler job
 - not easy to manually optimize a code selecting *magic* numbers
 - In case you need to optimize more, probably the use of CUDA and/or CUDA libraries is the best choice for specific sections of code
 - but using OpenACC allows for a much greater portability
 - `device_type` clause may be employed to have multiple tuning

```
void matvecmul( float* x, float* a,  
               float* v, int m, int n ){  
#pragma acc parallel loop \  
    device_type(nvidia) num_gangs(200) \  
    device_type(radeon) num_ganges(400)  
for( int i = 0; i < m; ++i )  
{  
    .....  
}
```



Advanced parallelization

- On the other hand, a control of the OpenACC loop layers may be crucial when the loop structure to parallelize is not trivial
 - i and xx are **privates** as for the **gang** level
 - but xx is **reduction** as for the **worker** level

```
void matvecmul( float* x, float* a,
               float* v, int m, int n ){
    #pragma acc parallel loop gang
    for( int i = 0; i < m; ++i ){
        float xx = 0.0;
        #pragma acc loop worker reduction(+:xx)
        for( int j = 0; j < n; ++j )
            xx += a[i*n+j]*v[j];

        x[i] = xx;
    }
}
```

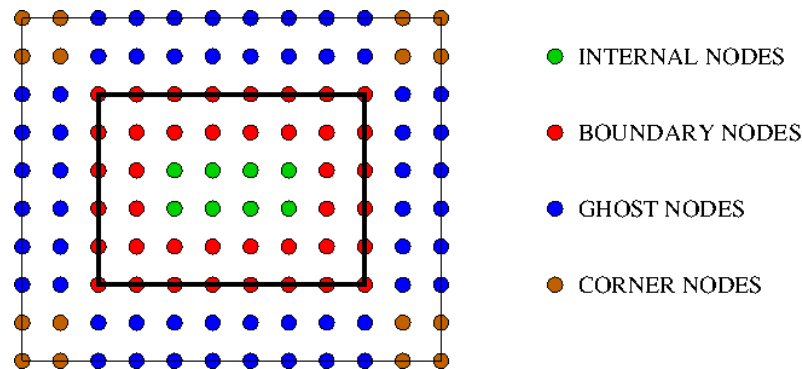


CPU: MPI and Asynchronism

- For CPU parallel MPI programs, employing the asynchronism may significantly improve performances
 - the main idea is to overlap communications to computations
 - the overlapping between computing and communications may be achieved using non-blocking MPI calls

• A basic pattern suitable for one iteration of the Laplace program is based on splitting the domain into bulk, boundary and halo points

- boundary updating
- MPI Send/Recv non-blocking calls to exchange halos (extra-boundary points)
- bulk updating
- MPI Wait calls





CPU+device, MPI and Asynchronism

- For GPU-enabled codes, the asynchronism has additional features
 - the programming model has two main actors: host – coprocessor
 - usually, the basic objective is still the overlapping of tasks
- **CPU and GPU computations may be overlapped**
 - but, the computing power of the GPU is often larger and decomposing the work-load is not trivial
 - however, special tasks may be better executed by the CPU
- **As for overlapping communications, CPU-device copies have to be taken into account**
 - different patterns are required
 - achieving overlapping becomes more important (CPU-GPU copies are expensive)



OpenACC async and wait

- OpenACC allows for asynchronous compute regions and data updates with the **async** clause
- It is also possible to have a number of **async** queues by adding a value expression to the **async** clause
 - activities with the same **async** value will be executed as they are enqueued by the host thread

```
#pragma acc parallel async  
{ ..... }  
#pragma acc update device(...) async  
{ ..... }  
#pragma acc wait
```

```
#pragma acc parallel async(1)  
{ ..... }  
#pragma acc parallel async(2)  
{ ..... }  
#pragma acc wait(1)  
{ ..... }  
#pragma acc wait(2)
```



A CPU-GPU asynchronous pattern for Laplace

- A basic pattern to hide communications between GPU and GPU considering GPU->host and host -> GPU intermediate copies
 - update boundary – synchronous
 - update bulk – asynchronous
 - MPI halo exchange - blocking or not blocking with MPI waits
 - OpenACC wait for update bulk
- Use CUDA-streams or OpenACC **async** to implement it



What we left out

- A few directives, e.g.:
 - **atomic**
 - **enter data/exit data**
- More on targeting platforms
 - e.g., compiler extensions
- Some clauses, e.g., **tile**
- (Much) more on the run-time library



Perspectives

- The main reference: www.openacc-standard.org
- PGI and CAPS full implementations of OpenACC 2.0
- Future improvements of the standard
 - at present, no standard way to support arrays inside C++ classes, C structs or Fortran derived data types
- Multi-platform implementations and tests:
 - AMD GPUs, Intel MICs,...
- **Open-Source OpenACC implementations**
 - OpenACC extension will be supported in mainstream GCC compilers



OpenMP 4.0 and accelerators

- OpenMP – de-facto standard for Shared-Memory Parallelization – from 4.0 release includes directives to handle accelerators
- Some ideas follow the OpenACC approach
- Accelerator
 - different functionality (optimized for something special)
 - different instruction set
 - each accelerator attached to one host device
 - it may or may not share memory with the device
- Execution model:
 - host-centric



OpenMP target data management

- **target data** construct
 - creates a device data environment for the extent of the region
 - map clause: map a variable from the current task's data environment to the device data environment associated with the construct
alloc/to/from/tofrom
- **target update**: synchronize host and device copies
- **declare target**: for variable and functions!



OpenMP target and teams

```
#pragma omp target device(0) map(tofrom:B)
#pragma omp parallel for
for (i=0; i<N; i++)
    B[i] += sin(B[i]);
```

OpenMP 4.0 for
Intel Xeon Phi

```
#pragma omp target device(0) map(tofrom:B)
#pragma omp teams num_teams(num_blocks) num_threads(bsize)
#pragma omp distribute
for (i=0; i<N; i += num_blocks)
#pragma omp parallel for
for (b = i; b < i+num_blocks; b++)
    B[b] += sin(B[b]);
```

OpenMP
4.0 for
Nvidia GPU

```
#pragma acc parallel copy(B[0:N]) num_gangs(numblocks) \
    vector_length(bsize)
#pragma acc loop gang vector
for (i=0; i<N; ++i)
    B[i] += sin(B[i]);
```

OpenACC for
NVIDIA GPU



OpenMP 4.0 implementation

- Intel compiler 2015 implements OpenMP 4.0 target directives but presently supports only Intel Xeon Phi devices
- GNU compiler 5.0 is going to provide support to different type of accelerators
 - keep updated!



References

- OpenACC standard
 - http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf
- OpenACC technical report on structured types
 - <http://www.openacc.org/sites/default/files/TR-14-1.pdf>
- PGI
 - <https://www.pgroup.com/resources/accel.htm>
- OpenMP
 - <http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf>



Credits

Mark Harris - NVIDIA

John Urbanic - Pittsburgh Supercomputing Center

Sarah Tariq - NVIDIA

Jeff Larkin - NVIDIA

Christian Terboven