ID5130 Parallel Scientific Computing - OpenACC Programming

Kameswararao Anupindi

Department of Mechanical Engineering
Indian Institute of Technology Madras (IITM)

March, 2024

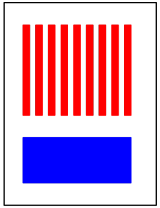
What we will be learning

1. OpenACC
   1.1 Accelerator
   1.2 Hello World Program
   1.3 Compilation and Execution
2. Specification of Parallelism
   2.1 pragmas
   2.2 kernels
   2.3 loops
3. Data movement
   3.1 copy
   3.2 create
   3.3 enter
   3.4 exit
4. Reductions and Atomics
5. Optimizations

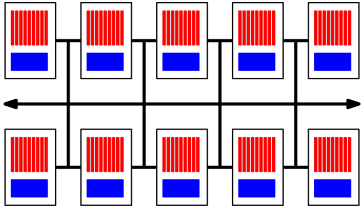
Sources and Acknowledgements

1. Prof. Rupesh Nasre (CS)
2. https://www.openacc.org/
3. OpenACC Specification, Strategies, and Tutorials
4. *OpenACC for Programmers: Concepts and Strategies* by Sunita Chandrasekaran, Guido Juckeland
5. *Parallel Programming with OpenACC* by Rob Farber et al.
6. *Programming Massively Parallel Processors: A Hands-on Approach* by Wen-mei W. Hwu et al.

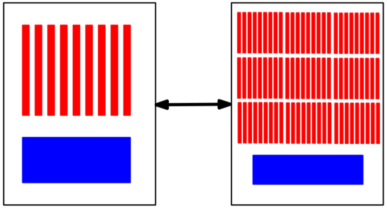
Parallel Programming Paradigms



Multi-core Machine
OpenMP, PosixThreads, Cilk, ...

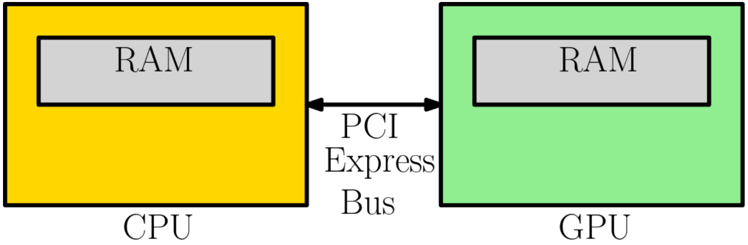


Distributed Memory Machine
MPI, MapReduce, Hadoop, ...



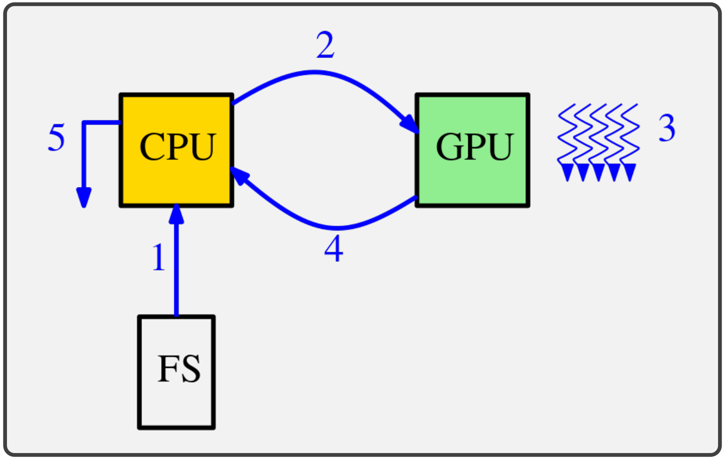
CPU - GPU
OpenACC, OpenCL, CUDA, ...

Different Memories



- CPU and GPU have separate physical memory (RAM)
- A variable residing in CPU memory cannot be directly accessed by a GPU and the other way round.
- OpenACC automatically copies data across CPU-GPU.
- A programmer helps in optimizing this communication.

Typical GPU Program Flow



OpenACC

- Open Accelerator language for C, C++ and FORTRAN
- Directive based implementation (#pragma acc, !$acc)
- OpenACC is designed for performance and portability
- Supported platforms: NVIDIA gpu, AMD gpu, ARM, x86, Sunway, POWER.

Hello World Program

```c
#include <stdio.h>

int main(){

#pragma acc parallel loop
  for(int i = 0; i < 10; i++){
      printf("Hello World %d\n", i);
  }

  return 0;
}
```

Compilation and Execution

```
Compilation: $pgcc hello.c
Execution:   $./a.out
```

Output:

```
Hello World 0
Hello World 1
.
.
.
Hello World 9
```

Compilation and Execution contd...

```
Compilation: $pgcc -acc hello.c
Execution:   $./a.out
```

Output:

```
Hello World 15
Hello World 20
Hello World 62
.
.
Hello World 0
Hello World 1
.
Hello World 93
```

OpenACC Syntax and Advantages

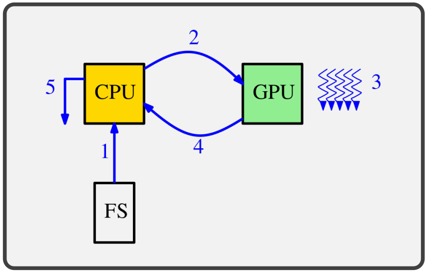
```
#pragma acc directive [clauses]
e.g. #pragma acc parallel loop

!$acc directive [clauses]
e.g. !$acc parallel loop
```

- Incremental parallelization possible!
- Portable, single source code that works across devices.
- Easy to use when compared to OpenCL or CUDA.
- OpenACC goes by *More Science, Less Programming*

```
pgfortran -acc=multicore hello.f90
```

Homework: Read the output of **pgcc -help**

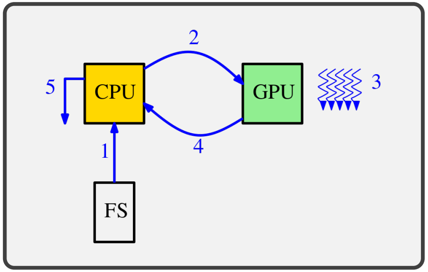
OpenACC Execution Model



Host-directed:

- Attached accelerator (or multiple CPU threads)
- Allocate memory on accelerator
- Initiate data transfer
- Send code to accelerator
- Pass arguments to the compute region
- Queue accelerator code
- Wait for completion
- Transfer results back to the host
- Deallocate memory...

OpenACC Execution Model contd.

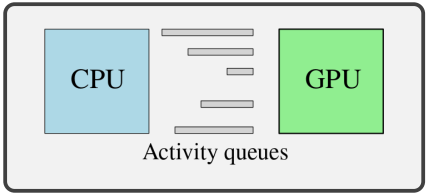


Compute-intensive parts offloaded

- Execute parallel regions (work-sharing loops)
- Execute kernel regions (single or multiple loops)
- Execute serial regions (sequential code)

OpenACC Execution Model contd.

- Host (CPU) can run **asynchronously** with the device (GPU or accelerator).
- Host may en-queue various operations in the **device activity queue**.
- Host thread may query and wait for the queue.
- Multiple queues may be simultaneously active.
- Three levels of parallelism
  - **Gang** (coarse-grain)
  - **Worker** (fine-grain)
  - **Vector** (SIMD)



GPU Computation Hierarchy

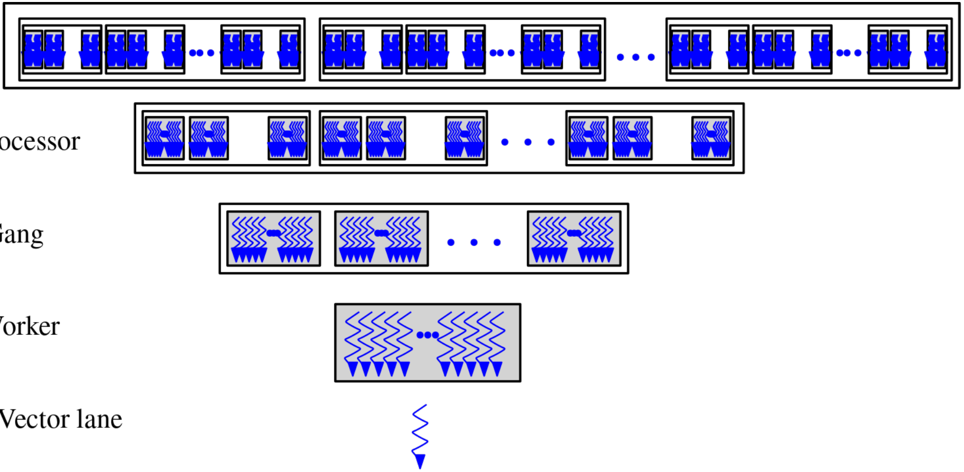
GPU
10^5

Multi-processor
10^4

Block/ Gang
1024

Warp/ Worker
32

Thread/ Vector lane
1



What we will be learning

Primary Compute Constructs

```
#pragma acc parallel [clauses]
#pragma acc kernels [clauses]
#pragma acc serial [clauses]
```

```
!$acc parallel [clauses]
  structured block of code
!$acc end parallel
```

Parallel construct

```c
#pragma acc parallel
{
  printf("How many times am I printed? \n");
}
```

Parallel construct contd...

- On encountering a parallel construct, one or more gangs of workers are created.
- The number of gangs, workers-per-gang, and vector-lanes-per-worker remain fixed in the parallel region.
- Each gang executes the parallel region.

Parallel construct contd...

```c
#pragma acc parallel num_gangs(2)
{
  printf("How many times am I printed? \n");
}
```

Find the output...

```
#pragma acc parallel num_gangs(10) async
{
  printf(" Hello \n");
  printf(" Bye \n");
}
printf(" Host \n");
#pragma acc parallel num_gangs(3)
{
  printf(" One \n");
  printf(" Two \n");
}
```

- Can *Bye* precede *Hello*?
- Can *Host* precede the first *Hello*?
- Can *Bye* follow the last *Two*?

num_gangs(expression)

```c
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char* argv[]){
    int ngangs = atoi(argv[1]);

    #pragma acc parallel num_gangs(ngangs)
    {
        printf("Hello World \n");
        printf("Bye World \n");
    }
    printf("Host \n");
    #pragma acc parallel num_gangs(ngangs/2)
      fun();

    return 0;
}
```

Notes on parallel construct

- No branching in or out of it.
- No dependence on the order of evaluation of the clauses.
- At most one if-clause.
- Supports **private** and **firstprivate**.

What is the output of the following piece of code?

```
#pragma acc parallel if (3 < 2) num_gangs(10)
     printf("Hello World \n");
```

kernels construct

- OpenACC is meant to improve programmer productivity.
- A novice HPC programmer may wishes to specify the code-to-parallelize only.
- Let compiler identify *which parts of the program to be parallelized and how*.
- The **kernels** directives helps one achieve this.

kernels construct contd...

```
#pragma acc kernels
{
  printf("One \n");
  for (int i = 0 ; i < 10; ++i)
      a[i] = i;
  printf("Two: a[9] = %d \n", a[9]);
  for (int i = 0; i < 10; ++i)
      a[i]* = 2;
  printf("Three: a[9] = %d \n", a[9]);
}
```

kernels construct contd...

- The compiler splits the kernels region into a sequence of steps: sequential and parallel.
- Typically, each loop (nested loop) is a different kernel.
- The kernels are executed in order (maintains the order in the original code).

kernels construct contd...

```
#pragma acc kernels
{
  printf("One \n");
  for (int i = 0 ; i < 10; ++i)
  {
     a[i] = i;
     printf("first loop %d \n", i);
  }
  printf("Two: a[9] = %d \n", a[9]);
  for (int i = 0; i < 10; ++i)
  {
      a[i]* = 2;
      printf("second loop %d \n", i);
  }
  printf("Three: a[9] = %d \n", a[9]);
}
```

kernels construct - output

```
One
first loop 0
first loop 1
first loop 2
...
first loop 9
Two: a[9] = 9
second loop 0
second loop 1
second loop 2
...
second loop 9
Three: a[9] = 18
```

kernels construct contd...

```
#pragma acc kernels
{
  printf("One \n");
  for (int i = 0 ; i < 10; ++i)
  {
     a[i] = i;
     printf("first loop %d \n", i);
  }
  printf("Two \n");
  for (int i = 0; i < 10; ++i)
  {
      a[i]* = 2;
      printf("second loop %d \n", i);
  }
  printf("Three \n");
}
```

kernels construct - output

```
One
Two
Three
first loop 0
first loop 1
first loop 2
...
first loop 9
second loop 0
second loop 1
second loop 2
...

second loop 9
```

kernels construct contd...

```
#pragma acc kernels
{
  printf("One \n");
  for (int i = 0 ; i < 10; ++i)
  {
     a[i] = i;
     printf("first loop %d \n", i);
  }
  printf("Two a[9] = %d \n", a[9]);
  for (int i = 0; i < 10; ++i)
  {
      a[i]* = 2;
      printf("second loop %d \n", i);
  }
  printf("Three \n");
}
```

kernels construct - output

```
One
Three
first loop 0
first loop 1
first loop 2
....
first loop 9
Two: a[9] = 9
second loop 0
second loop 1
second loop 2
....
second loop 9
```

Behaviour of kernels

To understand what the compiler does:

```
$pgcc -acc -Minfo=accel kernel_program.c
```

Behaviour of kernels

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int a[10];

#pragma acc kernels
  {
    printf("One \n");
    for (int i = 0; i < 10; i ++)
      {
        a[i] = i;
        printf("in the first loop %d \n", i);
      }
    printf("Two %d \n", a[9]);
```

Behaviour of kernels

```c

    for(int i = 0; i < 10; i++)
      {
        a[i] *= 2;
        printf("in the second loop %d \n", i);
      }
    printf("Three \n");

  }


  return 0;
}
```

Behaviour of kernels

```
$pgcc -acc -Minfo=accel kernels_program.c
main:
      9, Generating implicit copyout(a[:]) [if not already present]
     11, Loop is parallelizable
         Generating NVIDIA GPU code
         11, #pragma acc loop gang, vector(32) /* blockIdx.x
   threadIdx.x */
     16, Accelerator serial kernel generated
         Generating NVIDIA GPU code
     18, Loop is parallelizable
         Generating NVIDIA GPU code
         18, #pragma acc loop gang, vector(32) /* blockIdx.x
   threadIdx.x */
```

Behaviour of kernels

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int a[10];

#pragma acc kernels
  {
    printf("One \n");
    for (int i = 0; i < 10; i ++)
      {
        a[i] = i;
        printf("in the first loop %d \n", i);
      }
```

Behaviour of kernels

```
    // printf("Two %d \n", a[9]);
    printf("Two \n");

    for(int i = 0; i < 100; i++)
      {
        a[i] *= 2;
        printf("in the second loop %d \n", i);
      }
    printf("Three \n");

  }


  return 0;
}
```

Behaviour of kernels

```
$pgcc -acc -Minfo=accel kernels_program.c
main:
      9, Generating implicit copyout(a[:]) [if not already present]
     11, Loop is parallelizable
         Generating NVIDIA GPU code
         11, #pragma acc loop gang, vector(32) /* blockIdx.x
   threadIdx.x */
     19, Loop is parallelizable
         Generating NVIDIA GPU code
         19, #pragma acc loop gang, vector(128) /* blockIdx.x
   threadIdx.x */
```

Notes on kernels

- Without async, kernels end with a barrier.
- All restrictions of parallel also apply to kernels.

Analyze the output of the code using -Minfo flag

```
  int a[10] = {0};
#pragma acc kernels async
  {
    int sum = 0;
    printf("One. \n");
    for (int ii = 0; ii < 10; ii++)
      sum += a[ii] + ii;
    printf("Two: %d \n", sum);
    for (int ii = 0; ii < 10; ii++)
      a[ii] += sum;
    printf("Three . \n");
    for (int ii = 0; ii < 10; ii++)
      printf("a[%d] = %d \n", ii, a[ii]);
    printf("Four. \n");
  }
  printf("Five. \n");
```

Other constructs

```
#pragma acc serial

num_gangs, num_workers, vector_length

private, firstprivate
```

Specifying gangs, workers and threads

```
#pragma acc parallel num_gangs(1024) num_workers(32)
{
  ...
}
```

```
#pragma acc parallel num_gangs(1024)
{
  for (i = 0; i < 2048; i++)
    {
      ...
    }
}
```

Specifying gangs, workers and threads

```
#pragma acc parallel num_gangs(1024)
{
#pragma acc loop gang
  for (i = 0; i < 2048; i++)
    {
      ...
    }
}
```

Specifying gangs, workers and threads

```
#pragma acc parallel num_gangs(1024) num_workers(32)
{
#pragma acc loop gang
  for (i = 0; i < 2048; i++)
    {
#pragma acc loop worker
      for (j = 0; j < 512; j++)
        fun(i, j);
    }
}
```

Specifying gangs, workers and threads

```
#pragma acc parallel num_gangs(1024) num_workers(32) \
vector_length(32)
{
#pragma acc loop gang
  for (i = 0; i < 2048; i++)
    {
#pragma acc loop worker
      for (j = 0; j < 512; j++)
        {
#pragma acc loop vector
          for (k = 0; k < 1024; k++)
            fun(i, j, k);
        }
    }
}
```

Vector addition

```c
int main(){
    int A[N], B[N], C[N];
    init(A, N);
    init(B, N);
    add(C, A, B, N);
    //print(C, N);

    return 0;
}
```

Vector addition

```c
void init(int X[], int n){
#pragma acc parallel loop
    for(int ii=0; ii < n; ++ii)
        X[ii] = ii;
}
```

```c
void add(int C[], int A[], int B[], int n){
#pragma acc parallel loop
    for(int ii=0; ii < n; ++ii)
        C[ii] = A[ii] + B[ii];
}
```

Vector addition - compiler output

```
init:
      6, Generating implicit firstprivate(n,ii)
         Generating NVIDIA GPU code
          8, #pragma acc loop gang, vector(128) /* blockIdx.x
   threadIdx.x */
      6, Generating implicit copyout(X[:n]) [if not already present]
add:
     12, Generating implicit firstprivate(n,ii)
         Generating NVIDIA GPU code
         14, #pragma acc loop gang, vector(128) /* blockIdx.x
   threadIdx.x */
     12, Generating implicit copyin(A[:n]) [if not already present]
         Generating implicit copyout(C[:n]) [if not already present]
         Generating implicit copyin(B[:n]) [if not already present]
```

Data construct

```
#pragma acc data [clauses]
```

- May appear together with parallel, kernels, serial
- Can appear separately
- **copyin**, on entry to the data block
  - checks if a variable is already present on device
  - allocates device memory
  - initiates a copy from host to device
- **copyout**, on exit of the data block
  - checks if a variable is already present on host
  - initiates a copy from device to host
  - deallocates device memory

Vector addition - revisited

```c
int main(){
    int A[N], B[N], C[N];
#pragma acc data copyin(A,B) copyout(C)
    {
    init(A, N);
    init(B, N);
    add(C, A, B, N);
    }
    //print(C, N);
    return 0;
}
```

Modified vector addition - compiler output

```
init:
  6, Generating implicit firstprivate(n,ii)
     Generating NVIDIA GPU code
     8, #pragma acc loop gang, vector(128) /*blockIdx.x threadIdx.x*/
  6, Generating implicit copyout(X[:n]) [if not already present]
add:
 12, Generating implicit firstprivate(n,ii)
     Generating NVIDIA GPU code
    14, #pragma acc loop gang, vector(128) /*blockIdx.x threadIdx.x*/
 12, Generating implicit copyin(A[:n]) [if not already present]
     Generating implicit copyout(C[:n]) [if not already present]
     Generating implicit copyin(B[:n]) [if not already present]
main:
 21, Generating copyin(A[:]) [if not already present]
     Generating copyout(C[:]) [if not already present]
     Generating copyin(B[:]) [if not already present]
```

Vector addition - profiled output

```
export PGI_ACC_TIME=1
```

```
 init  NVIDIA  devicenum=0
   time(us): 34
   6: compute region reached 2 times
       6: kernel launched 2 times
           grid: [8]  block: [128]
           elapsed time(us): total=32 max=20 min=12 avg=16
   6: data region reached 4 times
       9: data copyout transfers: 2
            device time(us): total=34 max=18 min=16 avg=17
```

Vector addition - profiled output

```
add  NVIDIA  devicenum=0
  time(us): 25
  12: compute region reached 1 time
      12: kernel launched 1 time
          grid: [8]  block: [128]
          elapsed time(us): total=12 max=12 min=12 avg=12
  12: data region reached 2 times
      12: data copyin transfers: 2
           device time(us): total=9 max=5 min=4 avg=4
      15: data copyout transfers: 1
           device time(us): total=16 max=16 min=16 avg=16
```

Modified vector addition - profiled output

```
init  NVIDIA  devicenum=0
  time(us): 0
  6: compute region reached 2 times
      6: kernel launched 2 times
          grid: [8]  block: [128]
          elapsed time(us): total=32 max=20 min=12 avg=16
  6: data region reached 4 times

add  NVIDIA  devicenum=0
  time(us): 0
  12: compute region reached 1 time
      12: kernel launched 1 time
          grid: [8]  block: [128]
          elapsed time(us): total=11 max=11 min=11 avg=11
  12: data region reached 2 times
```

Modified vector addition - profiled output

```
main  NVIDIA  devicenum=0
  time(us): 27
  21: data region reached 2 times
      21: data copyin transfers: 2
           device time(us): total=14 max=9 min=5 avg=7
      25: data copyout transfers: 1
           device time(us): total=13 max=13 min=13 avg=13
```

Vector addition - present clause

```
void init(int X[], int n){
#pragma acc parallel loop present(X)
    for(int ii=0; ii < n; ++ii)
        X[ii] = ii;
}
```

```
void add(int C[], int A[], int B[], int n){
#pragma acc parallel loop present(A,B,C)
    for(int ii=0; ii < n; ++ii)
        C[ii] = A[ii] + B[ii];
}
```

Other data clauses

```
copy = copyin (on entry) + copyout (on exit)
#pragma acc data copy(A[:])

create: only create on device but do not copy
#pragma acc data create(B[1:n-2][3:5*m])
The opposite operation is delete

C/C++: array[start:length]
FORTRAN: array[start:end]
```

Unstructured data directives

- Data life-times are not always structured
- The **enter data** directive handles device memory allocation
- **create** or **copyin** clauses may be used for memory allocation
- The **enter data** directive is not the start of a data region, because there can be many enter data directives.

```
#pragma acc enter data [clauses]

Serial and/or Parallel code

#pragma acc exit data [clauses]
```

Unstructured data directives contd...

- The **exit data** directive handles the device memory deallocation.
- Use either the **delete** or the **copyout** clause for memory deallocation.
- For a given array there must be as many **exit data** directives as there are **enter data**.
- They can be in different functions.

```
!$acc enter data [clauses]

Serial and/or Parallel code

!$acc exit data [clauses]
```

Unstructured data directives example

```
#pragma acc enter data copyin(a[0:N],b[0:N]) create(c[0:N])

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

#pragma acc exit data copyout(c[0:N]) delete(a,b)
```

- Can have multiple start/end points.
- Can branch across different functions.
- Memory exists until explicitly deallocated.

Structured data directives example

```
#pragma acc data copyin(a[0:N],b[0:N]) copyout(c[0:N])
{
#pragma acc parallel loop
  for(int i = 0; i < N; i++)
  {
    c[i] = a[i] + b[i];
  }
}
```

- Must have explicit start/end points.
- Must be within a single function.
- Memory exists only within the data region.

Unstructured data directives example - branching across multiple functions

```
int* allocate_array(int N){
  int* ptr = (int *) malloc(N*sizeof(int));
#pragma acc enter data create(ptr[0:N])
return ptr; }

void deallocate_array(int* ptr){
#pragma acc exit data delete(ptr)
  free(ptr); }

int main(){
  int* a = allocate_array(100);
#pragma acc kernels
  fun(a);
  deallocate_array(a);
}
```

Data synchronization

- **update** directive explicitly transfers data between the host and the device.
- Useful when you want to synchronize data in the middle of a data region.
- **self** clause makes host data agree with the device data.
- **device** clause makes device data agree with the host data.

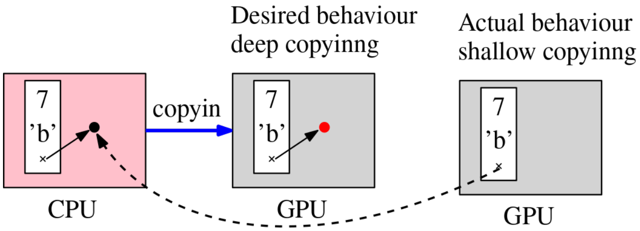
```
#pragma acc update self(a[0:count])
#pragma acc update device(a[0:count])
```

```
!$acc update self(a[0:count])
!$acc update device(a[0:count])
```

Copying Pointers

- OpenACC performs shallow copying of pointers
- Thus, pointer within a structure is simply copied; rather than copying its object.

```
struct mystruct{
int i, char c;
int *ptr;
}s;
*s.ptr
```



Homework: Write a program to test shallow copying.

Loop construct

- Used for work-sharing within a parallel region.
- Can be used within a parallel block, or can be combined with parallel.
- Can declare private variables.
- Supports reduction operation.

```
#pragma acc loop [clauses]
  for loop

!$acc loop [clauses]
do loop
```

Loop construct example

```
#pragma acc parallel
#pragma acc loop
  for(int ii=0; ii < N; ii++)
    a[ii] = ii;
```

```
#pragma acc parallel loop
  for(int ii=0; ii < N; ii++)
    a[ii] = ii;
```

- Work distribution across threads
- Similar to OpenMP.

Nested loops

- Depending on the *pgcc* version, nested parallelism may be allowed.

```
#pragma acc parallel loop
  for(int ii=0; ii < N; ii++)
    #pragma acc parallel loop
      for(int jj=0; jj < N; jj++)
         a[ii][jj] = ii*N + jj;
```

```
NVC++-S-0155-Nested Parallelism is not supported for pragma: acc
   parallel loop (nested_loops.c: 11)
NVC++/x86-64 Linux 22.3-0: compilation completed with severe errors
```

Nested loops contd...

- By default, the loop construct gets associated with the loop that immediately follows.

```
3 #define N 10
...
9 #pragma acc parallel loop
10  for(int ii = 0; ii < N; ii++)
11    for (int jj = 0; jj < N; jj++)
12      a[ii][jj] = ii*N + jj;
```

```
main:
     7, Generating NVIDIA GPU code
        10, #pragma acc loop gang /* blockIdx.x */
        11, #pragma acc loop seq
     7, Generating implicit copyout(a[:][:]) [if not already present]
    11, Loop is parallelizable
```

Nested loops contd...

- To associate **pragma** with nested loops, use collapse.

```
3 #define N 10
...
9 #pragma acc parallel loop collapse
10  for(int ii = 0; ii < N; ii++)
11    for (int jj = 0; jj < N; jj++)
12      a[ii][jj] = ii*N + jj;
```

```
main:
    7, Generating NVIDIA GPU code
       10, #pragma acc loop gang, vector(96) collapse(2) /* blockIdx.
   x threadIdx.x */
       11,   /* blockIdx.x threadIdx.x collapsed */
    7, Generating implicit copyout(a[:][:]) [if not already present]
```

Loop-carried dependence

- Look out for dependence across different iterations.

```
7  int a[N] = {32};
8
9 #pragma acc serial loop
10  for(int ii = 1; ii < N; ii++)
11     a[ii] = a[ii-1];
```

```
main:
      7, Accelerator serial kernel generated
         Generating NVIDIA GPU code
         10, #pragma acc for seq
      7, Generating implicit allocate(a[:]) [if not already present]
         Generating implicit copyin(a[:99]) [if not already present]
         Generating implicit copyout(a[1:]) [if not already present]
```

Loop-carried dependence parallelizing

```
7  int a[N] = {32};
8
9 #pragma acc parallel loop
10  for(int ii = 1; ii < N; ii++)
11    a[ii] = a[ii-1];
```

```
main:
 7, Generating NVIDIA GPU code
  10, #pragma acc loop gang, vector(96) /* blockIdx.x threadIdx.x */
 7, Generating implicit allocate(a[:]) [if not already present]
    Generating implicit copyin(a[:99]) [if not already present]
    Generating implicit copyout(a[1:]) [if not already present]
```

The compiler didn’t say *loop is parallelizable* - but parallelized it! Output: 32 32 0 0 0 0....
Recall that *kernels* was intelligent.

Loop-carried dependence parallelizing contd...

```
7  int a[N] = {32};
8
9 #pragma acc parallel loop auto
10  for(int ii = 1; ii < N; ii++)
11    a[ii] = a[ii-1];
```

```
main:
      7, Generating NVIDIA GPU code
         10, #pragma acc loop seq
      7, Generating implicit allocate(a[:]) [if not already present]
         Generating implicit copyin(a[:99]) [if not already present]
         Generating implicit copyout(a[1:]) [if not already present]
     10, Loop carried dependence of a prevents parallelization
         Loop carried backward dependence of a prevents vectorization
```

Output: 32 32 32 32 32 32....

The Reduction clause

OpenACC reduction clause is similar to OpenMP.

```
8  int x = 0, y = 0;
9 #pragma acc parallel loop reduction(+: x, y)
10  for(int ii = 0; ii < N; ii++){
11     x += 1;
12     y += 2;
13     a[ii] = ii;
14  }
15  printf("x = %d, y = %d \n", x, y);
```

```
main:
8, Generating NVIDIA GPU code
  10, #pragma acc loop gang, vector(96) /* blockIdx.x threadIdx.x */
    Generating reduction(+:y,x)
8, Generating implicit copyout(a[:]) [if not already present]
   Generating implicit copy(y,x) [if not already present]
```

Reduction Forms

Reduction expressions may have different forms.

```
x += 1           // combined operator
x = x + 1        // right relative update
x = 1 + x        // left relative update
++x              // pre-increment
x++              // post-increment
```

Class work

```c
int main()
{
  int a[N], sum;

  #pragma acc parallel loop
  for(int ii = 0; ii < N; ii++)
    a[ii] = ii;

  sum = 1;
#pragma acc parallel loop reduction(+:sum)
  for(int ii = 0; ii < N; ii++)
    sum += a[ii];

  printf("sum = %d \n", sum);
  return 0;
}
```

Other Reduction Operators

```
C/C++:   +, *, max, min, &, |, ^, &&, ||
FORTRAN: +, *, max, min, iand, ior, ieor, .and., .or., .eqv., .neqv.
```

```
#pragma acc parallel loop
  for(int ii = 0; ii < N; ii++)
    a[ii] = ii;

  x = 0;
#pragma acc parallel loop reduction(max: x)
  for(int ii = 0; ii < N; ii++)
    if (a[ii] > x) x = a[ii];

  printf("x = %d \n", sum);
```

Applications: Cryptography

- Implement a substitution ciper
    - Given a string, encrypt it using a function which maps each character to a different character.
    - openacc $\rightarrow$ pqfobdd
- Mapping can be executed in parallel
- Scales well with the input size on many-core accelerators.

Encryption code: serial version

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define N 100
#define OFFSET 1

int main(){
  char inputtext[N], encrypttext[N], decrypttext[N];

  fgets(inputtext, N, stdin);
  encrypt(inputtext, encrypttext);
  decrypt(encrypttext, decrypttext);
  puts(encrypttext);
  puts(decrypttext);
}
```

Encryption code: serial version

```c
void transform(char input[], char output[], int offset){

  for (char *iptr = input, *optr = output; *iptr; iptr++, optr++)
      *optr = 'a' + (*iptr - 'a' + offset)%26;

  return;
}
```

Encryption code: serial version

```
void encrypt(char inputtext[], char encrypttext[]){

  transform(inputtext, encrypttext, OFFSET);

return;
}

void decrypt(char encrypttext[], char decrypttext[]){

  transform(encrypttext, decrypttext, 26-OFFSET);

  return;
}
```

Encryption code: serial version

```
gcc cipher_serial.c
./a.out
Parallel Scientific Computing
QbsbmmfmUTdjfoujgjdU^pnqvujoh
jarallelnmcientificnwomputing
```

Encryption code: serial version, modified transform function

```
void transform(char input[], char output[], int offset){
  char *optr = output;

  for (char *iptr = input; *iptr; iptr++, optr++)
    if ('a' <= *iptr && *iptr <= 'z')
      *optr = 'a' + (*iptr - 'a' + offset)%26;
    else
      *optr = *iptr;

  return;
}
```

Encryption code: serial version

```
gcc cipher_serial.c
./a.out
Parallel Scientific Computing
Pbsbmmfm Sdjfoujgjd Cpnqvujoh
Parallel Scientific Computing
```

Encryption code: parallel version

```
void transform(char input[], char output[], int offset){
  char *optr = output;

#pragma acc parallel loop
  for (char *iptr = input; *iptr; iptr++, optr++)
    if ('a' <= *iptr && *iptr <= 'z')
      *optr = 'a' + (*iptr - 'a' + offset)%26;
    else
      *optr = *iptr;

  return;
}
```

Encryption code: compile and run the parallel version

```
transform:
   13, Generating implicit private(iptr)
       Generating NVIDIA GPU code
       16, #pragma acc loop seq
   13, Generating implicit copyin(input) [if not already present]
   16, Loop without integer trip count will be executed in sequential
     mode
       Complex loop carried dependence of iptr-> prevents
    parallelization
       Loop carried dependence of iptr-> prevents parallelization
       Complex loop carried dependence of optr-> prevents
    parallelization
       Loop carried dependence of optr-> prevents parallelization
       Loop carried backward dependence of optr->,iptr-> prevents
    vectorization
```

Encryption code: compile the parallel version

```
        Loop carried scalar dependence for iptr at line 17,18,20
        Loop carried scalar dependence for optr at line 18,20
        Loop carried backward dependence of iptr-> prevents
    vectorization
        Generating implicit firstprivate(offset,optr,iptr)
        Complex loop carried dependence of optr-> prevents
    parallelization
        Loop carried dependence of optr-> prevents parallelization
        Complex loop carried dependence of optr-> prevents
    parallelization
main:
    49, Generating copyin(inputtext[:]) [if not already present]
        Generating copyout(encrypttext[:],decrypttext[:]) [if not
    already present]
```

Encryption code: run the parallel version

```
Parallel Scientific Computing
Failing in Thread:1
Accelerator Fatal Error: call to cuStreamSynchronize returned error
    700: Illegal address during kernel execution
 File: /home/kacfd-m2/KAnupindi/cipher_parallel.c
 Function: transform:8
 Line: 13
```

Encryption code: parallel version

```
void transform(char input[], char output[], int offset){

#pragma acc parallel loop copyin(input[0:N]) copyout(output[0:N])
  for (char *iptr = input, *optr = output; *iptr; iptr++, optr++)
    if ('a' <= *iptr && *iptr <= 'z')
        *optr = 'a' + (*iptr - 'a' + offset)%26;
      else
        *optr = *iptr;

  return;
}
```

Encryption code: compile and run the parallel version

```
transform:
     13, Generating copyin(input[:10000]) [if not already present]
         Generating copyout(output[:10000]) [if not already present]
         Generating implicit private(optr,iptr)
         Generating NVIDIA GPU code
         25, #pragma acc loop seq
     25, Loop without integer trip count will be executed in
   sequential mode
         Complex loop carried dependence of iptr-> prevents
   parallelization
         Loop carried dependence of iptr-> prevents parallelization
         Complex loop carried dependence of optr-> prevents
   parallelization
         Loop carried dependence of optr->,iptr-> prevents
   parallelization
```

Encryption code: compile the parallel version

```
         Loop carried backward dependence of optr->,iptr-> prevents
    vectorization
         Complex loop carried dependence of optr-> prevents
    parallelization
         Loop carried dependence of optr-> prevents parallelization
         Loop carried scalar dependence for iptr at line 26,27,29
         Loop carried scalar dependence for optr at line 27,29
         Loop carried backward dependence of iptr-> prevents
    vectorization
         Generating implicit firstprivate(offset,optr,iptr)
         Complex loop carried dependence of iptr-> prevents
    parallelization
         Loop carried dependence of iptr->,optr-> prevents
    parallelization
```

Encryption code: run the parallel version

```
Parallel Scientific Computing
Pbsbmmfm Sdjfoujgjd Cpnqvujoh
Parallel Scientific Computing
```

Encryption code: parallel version 2

```
void transform(char input[], char output[], int slength, int offset){

#pragma acc parallel loop copyin(input[0:N]) copyout(output[0:N])
  for (int i = 0; i < slength+1; i++)
    if ('a' <= input[i] && input[i] <= 'z')
      output[i] = 'a' + (input[i] - 'a' + offset)%26;
    else
      output[i] = input[i];

  return;
}
```

Encryption code: compile and run the parallel version 2

```
transform:
      8, Generating copyout(output[:100]) [if not already present]
         Generating copyin(input[:100]) [if not already present]
         Generating implicit firstprivate(i,slength)
         Generating NVIDIA GPU code
         18, #pragma acc loop gang, vector(128) /* blockIdx.x
   threadIdx.x */
     18, Generating implicit firstprivate(offset)
```

```
Parallel Scientific Computing
Pbsbmmfm Sdjfoujgjd Cpnqvujoh
Parallel Scientific Computing
```

Encryption code: on a longer text

```
Tif Ioejbo Iotujuvuf pg Tfdiopmphz Mbesbt jt lopxo cpui obujpobmmz
    boe joufsobujpobmmz gps fydfmmfodf jo ufdiojdbm fevdbujpo, cbtjd
    boe bqqmjfe sftfbsdi, joopwbujpo, fousfqsfofvstijq boe joevtusjbm
     dpotvmubodz. A gbdvmuz pg joufsobujpobm sfqvuf, b ijhimz
    npujwbufe boe csjmmjbou tuvefou dpnnvojuz, fydfmmfou ufdiojdbm
    boe tvqqpsujoh tubgg boe bo fggfdujwf benjojtusbujpo ibwf bmm
    dpousjcvufe up uif qsf-fnjofou tubuvt pg IIT Mbesbt. Tif
    Iotujuvuf jt qspve up cfbs uif mbvsfbuf pg cfjoh Np.1 fohjoffsjoh
     vojwfstjuz jo Ioejb. Mpsf sfdfoumz, IIT Mbesbt ibt cffo hjwfo
    uif ujumf pg Iotujuvuf pg Enjofodf.
```

Homework: Implement substitution with a random permutation of the alphabet.

Application: Matrix Addition

- Implement a matrix addition program
- Element addition can be made in parallel
- Scales well with the input size on many-core accelerators.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix}$$

Matrix addition - serial code

```
#define M 15          /* number of rows */
#define N 15          /* number of columns */

void initialize(int *matrix){
  for(int i = 0; i < M; i++)
    for(int j = 0; j < N; j++)
      matrix[i*N+j] = (i+1)*N+(j+1);
  return;
}

void addmatrices(int *A, int *B, int *C){
  for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
      C[i*N+j] = A[i*N+j] + B[i*N+j];
  return;
}
```

Matrix addition - serial code contd...

```
void printmatrix(int *matrix){

  if (M<=10 && N<=10)
    for (int i = 0; i < M; i++){
      for (int j = 0; j < N; j++)
   printf("%5d ", matrix[i*N+j]);
      printf("\n");
    }
  else
    printf("Skipping printing of large matrix...\n");

  return;
}
```

Matrix addition - serial code contd...

```c
int main(){
  /* allocate memory for matrices */
  int *A = (int *)malloc(M*N*sizeof(int));
  int *B = (int *)malloc(M*N*sizeof(int));
  int *C = (int *)malloc(M*N*sizeof(int));

  initialize(A); initialize(B);
  printf("\n Matrix A is:\n"); printmatrix(A);
  printf("\n Matrix B is:\n"); printmatrix(B);

  addmatrices(A, B, C);
  printf("\n Matrix C is:\n"); printmatrix(C);

  free(A); free(B); free(C);
  return 0;
}
```

Matrix addition - parallel code

```
#define M 100        /* number of rows */
#define N 100        /* number of columns */
void initialize(int *matrix){
#pragma acc parallel loop
  for(int i = 0; i < M; i++)
    for(int j = 0; j < N; j++)
      matrix[i*N+j] = (i+1)*N+(j+1);
  return;
}
void addmatrices(int *A, int *B, int *C){
#pragma acc parallel loop
  for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
      C[i*N+j] = A[i*N+j] + B[i*N+j];
  return;
}
```

Matrix addition - parallel code - compiler output

```
initialize:
9, Generating implicit firstprivate(i)
   Generating NVIDIA GPU code
   11, #pragma acc loop gang /* blockIdx.x */
   12, #pragma acc loop vector(96) /* threadIdx.x */
9, Generating implicit copy(matrix[:10000]) [if not already present]
12, Loop is parallelizable
addmatrices:
18, Generating implicit firstprivate(i)
   Generating NVIDIA GPU code
   20, #pragma acc loop gang, vector(96) /* blockIdx.x threadIdx.x */
   21, #pragma acc loop seq
18, Generating implicit copy(C[:10000]) [if not already pres.]
  Generating implicit copyin(B[:10000],A[:10000])[if not already pr.]
21, Complex loop carried dependence of B->,A->,C-> prevents
    parallelization
```

Matrix addition - parallel code

```
main:
#pragma acc data create(A[0:M*N],B[0:M*N]) copyout(C[0:M*N])
  {
    initialize(A);
    initialize(B);

    printf("\n Matrix A is:\n"); printmatrix(A);
    printf("\n Matrix B is:\n"); printmatrix(B);

    addmatrices(A, B, C);

    printf("\n Matrix C is:\n"); printmatrix(C);
  }
```

Matrix addition - parallel code - compiler output

```
initialize:
9, Generating implicit firstprivate(i)
    Generating NVIDIA GPU code
    11, #pragma acc loop gang /* blockIdx.x */
    12, #pragma acc loop vector(96) /* threadIdx.x */
9, Generating implicit copy(matrix[:10000]) [if not already present]
12, Loop is parallelizable
addmatrices:
18, Generating implicit firstprivate(i)
    Generating NVIDIA GPU code
    20, #pragma acc loop gang, vector(96) /* blockIdx.x threadIdx.x
   */
    21, #pragma acc loop seq
18, Generating implicit copy(C[:10000]) [if not already present]
    Generating implicit copyin(B[:10000],A[:10000]) [if not already
   present]
```

Matrix addition - parallel code - compiler output contd...

```
     21, Complex loop carried dependence of B->,A->,C-> prevents
    parallelization
main:
     50, Generating create(A[:10000]) [if not already present]
         Generating copyout(C[:10000]) [if not already present]
         Generating create(B[:10000]) [if not already present]
```

Matrix addition - parallel code - modified

```
void initialize(int *matrix){
#pragma acc parallel loop present(matrix[0:M*N])
  for(int i = 0; i < M; i++)
    for(int j = 0; j < N; j++)
      matrix[i*N+j] = (i+1)*N+(j+1);

  return;
}
void addmatrices(int *A, int *B, int *C){
#pragma acc parallel loop present(A[0:M*N],B[0:M*N],C[0:M*N])
  for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
      C[i*N+j] = A[i*N+j] + B[i*N+j];

  return;
}
```

Matrix addition - parallel code - modified - compiler output

```
initialize:
      9, Generating present(matrix[:10000])
         Generating implicit firstprivate(i)
         Generating NVIDIA GPU code
         11, #pragma acc loop gang /* blockIdx.x */
         12, #pragma acc loop vector(96) /* threadIdx.x */
     12, Loop is parallelizable
addmatrices:
     18, Generating present(A[:10000],C[:10000],B[:10000])
         Generating implicit firstprivate(i)
         Generating NVIDIA GPU code
         20, #pragma acc loop gang, vector(96) /* blockIdx.x
   threadIdx.x */
         21, #pragma acc loop seq
     21, Complex loop carried dependence of B->,A->,C-> prevents
   parallelization
```

Matrix addition - parallel code - modified - compiler output contd...

```
main:
     50, Generating create(A[:10000]) [if not already present]
         Generating copyout(C[:10000]) [if not already present]
         Generating create(B[:10000]) [if not already present]
```

Matrix addition - parallel code - yet another modification

```
void addmatrices(int *A, int *B, int *C){
#pragma acc parallel loop collapse(2) present(A[0:M*N],B[0:M*N],C[0:M
    *N])
  for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
      C[i*N+j] = A[i*N+j] + B[i*N+j];

  return;
}
```

Matrix addition - parallel code - modified - compiler output

```
initialize:
      9, Generating present(matrix[:10000])
         Generating implicit firstprivate(i)
         Generating NVIDIA GPU code
         11, #pragma acc loop gang /* blockIdx.x */
         12, #pragma acc loop vector(96) /* threadIdx.x */
     12, Loop is parallelizable
addmatrices:
     18, Generating present(A[:10000],C[:10000],B[:10000])
         Generating implicit firstprivate(i)
         Generating NVIDIA GPU code
         20, #pragma acc loop gang, vector(128) collapse(2) /*
   blockIdx.x threadIdx.x */
         21,   /* blockIdx.x threadIdx.x collapsed */
```

Matrix addition - parallel code - modified - compiler output contd...

```
main:
     50, Generating create(A[:10000]) [if not already present]
         Generating copyout(C[:10000]) [if not already present]
         Generating create(B[:10000]) [if not already present]
```

Matrix addition - Timing

Without Collapse

```
initialize  NVIDIA  devicenum=0
   time(us): 9
   9: compute region reached 2 times
       9: kernel launched 2 times
           grid: [300]  block: [128]
   9: data region reached 4 times
 addmatrices  NVIDIA  devicenum=0
   time(us): 135
   18: compute region reached 1 time
       18: kernel launched 1 time
           grid: [3]  block: [128]
   18: data region reached 2 times
 main  NVIDIA  devicenum=0
   time(us): 356
   50: data region reached 2 times
       63: data copyout transfers: 1
```

With Collapse

```
initialize  NVIDIA  devicenum=0
   time(us): 10
   9: compute region reached 2 times
       9: kernel launched 2 times
           grid: [300]  block: [128]
   9: data region reached 4 times
addmatrices  NVIDIA  devicenum=0
   time(us): 10
   18: compute region reached 1 time
       18: kernel launched 1 time
           grid: [704]  block: [128]
   18: data region reached 2 times
main  NVIDIA  devicenum=0
   time(us): 265
   50: data region reached 2 times
       63: data copyout transfers: 1
```

Matrix multiplication

Homework: Write an OpenACC program to perform matrix multiplication and optimize it.