Introduction to OpenACC

chandlerz@nvidia.com 周国峰

Wuhan University 2017/10/13

NVIDIA

Agenda

Why OpenACC?

Accelerated Computing Fundamentals

OpenACC Programming Cycle

Case Study – Lattice Boltzmann Method (LBM)

Why OpenACC?

OpenACC

Simple | Powerful | Portable

Fueling the Next Wave of Scientific Discoveries in HPC

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

University of Illinois
PowerGrid- MRI Reconstruction

70x Speed-Up
2 Days of Effort

RIKEN Japan
NICAM- Climate Modeling

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

8000+

Developers

using OpenACC

http://www.cray.com/sites/default/files/resources/OpenACC_213462.12_OpenACC_Cosmo_CS_FNL.pdf
http://www.hpcwire.com/off-the-wire/first-round-of-2015-hackathons-gets-underway
http://on-demand.gputechconf.com/gtc/2015/presentation/S5297-Hisashi-Yashiro.pdf
http://www.openacc.org/content/experiences-porting-molecular-dynamics-code-gpus-cray-xk7

What Are Compiler Directives

```
program myScience

   ... serial code...
   !$acc parallel loop
   do j = 1, n1
      do i = 1, n2
        ...
       enddo
   enddo
   ...
end program myScience
```

Complier Directive

- Insert portable compiler directives by programmers
- Compiler parallelizes code and manages data movement in default way
- Programmer optimizes incrementally
- Designed for multi-core CPUs, GPUs & many-core accelerators

OpenACC Directives

Manage Data Movement

Initiate Parallel Execution

Optimize Loop Mappings

```
#pragma acc data copyin(a,b) copyout(c)
{
  ...
  #pragma acc parallel
  {
  #pragma acc loop gang vector
      for (i = 0; i < n; ++i) {
          z[i] = x[i] + y[i];
          ...
      }
  }
  ...
}
```

OpenACC Directives for Accelerators

- Incremental
- Multi-Platform, Single source
- Interoperable, CUDA OpenCL
- Performance portable
- CPU, GPU, MIC (In furture)

OpenACC VS OpenMP

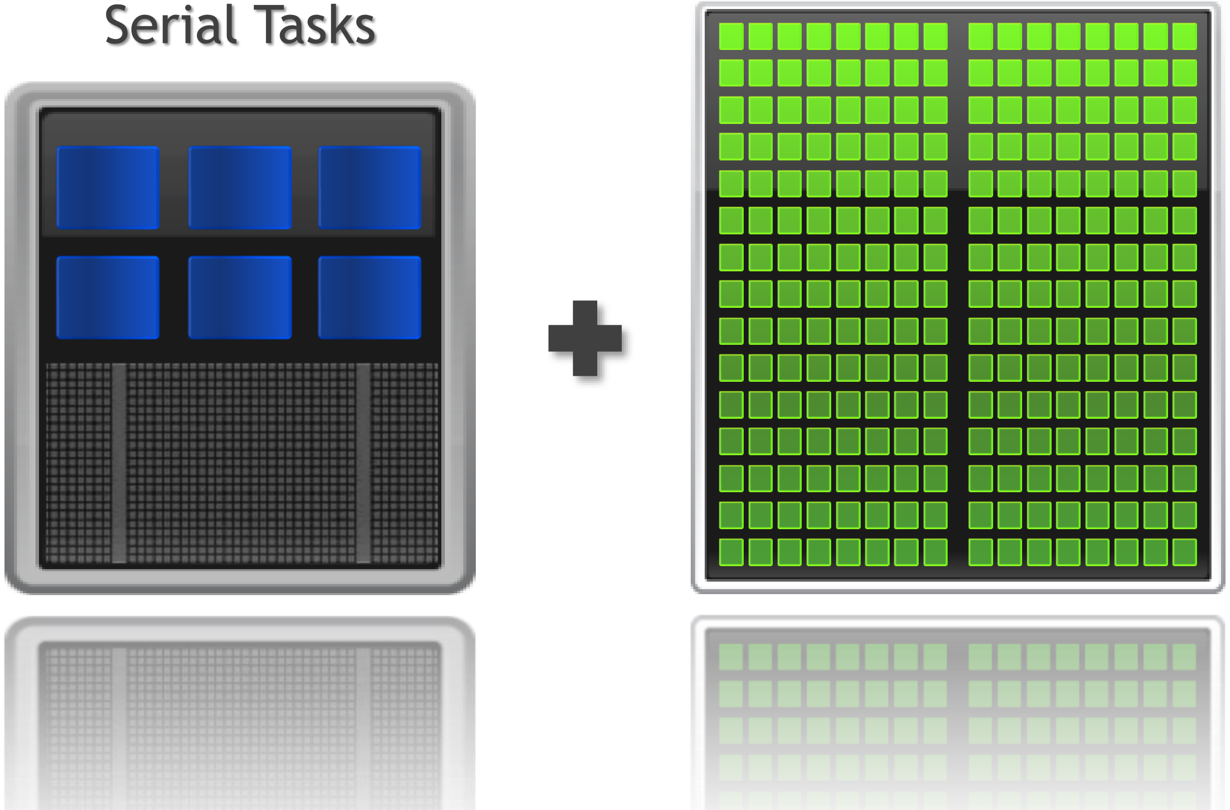
OpenACC	OpenMP
Initially designed for accelerators. Separate host and accelerator memories	Initially designed for shared memory processors. Also support offloading to accelerators
Focused on accelerated computing	General purpose parallelism
More agile	More measured
Performance Portability	Functional Portability
Descriptive	*Prescriptive*
Extensive interoperability	Limited interoperability
More mature for accelerators	More mature for multi-core

Accelerated Computing Fundamentals

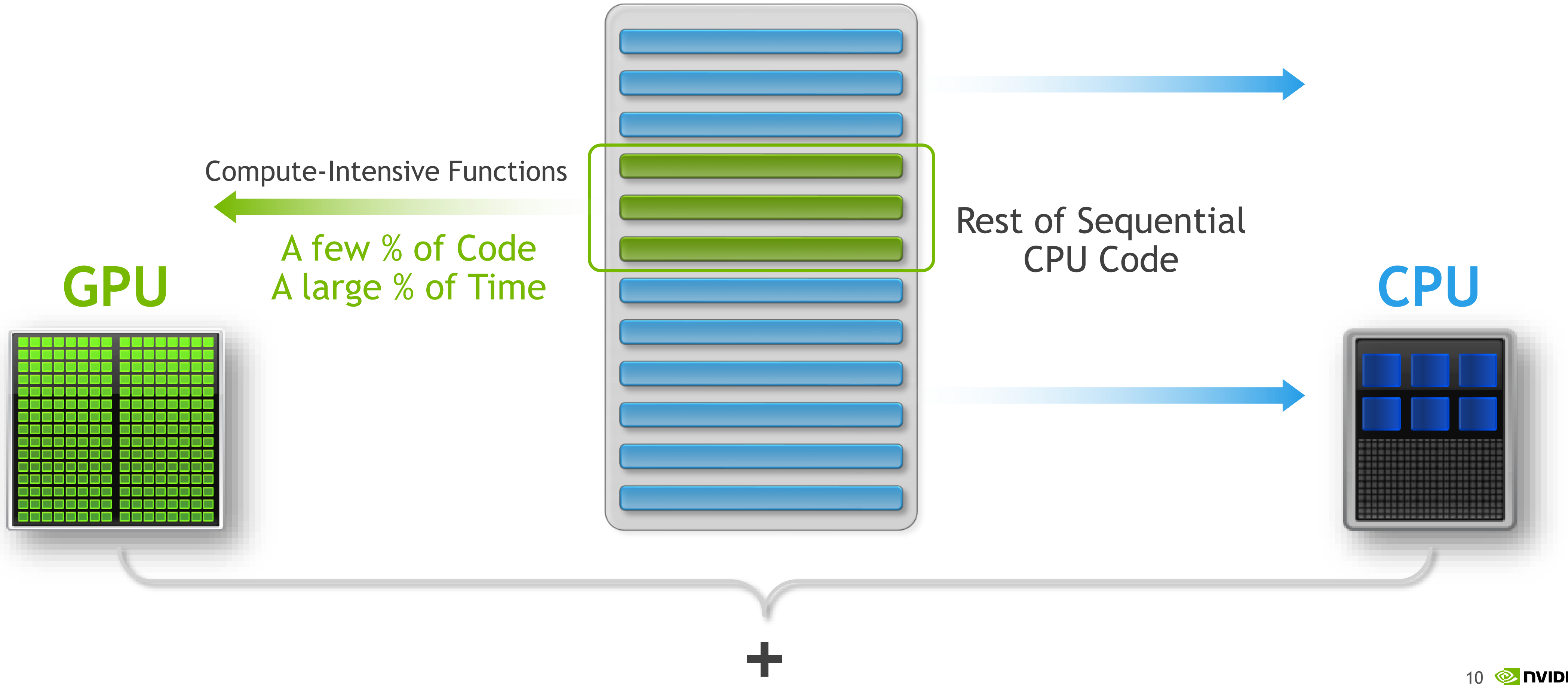
Accelerated Computing

CPU
Optimized for
Serial Tasks

GPU Accelerator
Optimized for
Parallel Tasks



What is Heterogeneous Programming?



Portability & Performance

Portability

Performance

Accelerated Libraries

High performance with little or no code change

Limited by what libraries are available

Compiler Directives

High Level: Based on existing languages; simple, familiar, portable

High Level: Performance may not be optimal

Parallel Language Extensions

Greater flexibility and control for maximum performance

Often less portable and more time consuming to implement

Code for Portability & Performance

Libraries

- Implement as much as possible using portable libraries

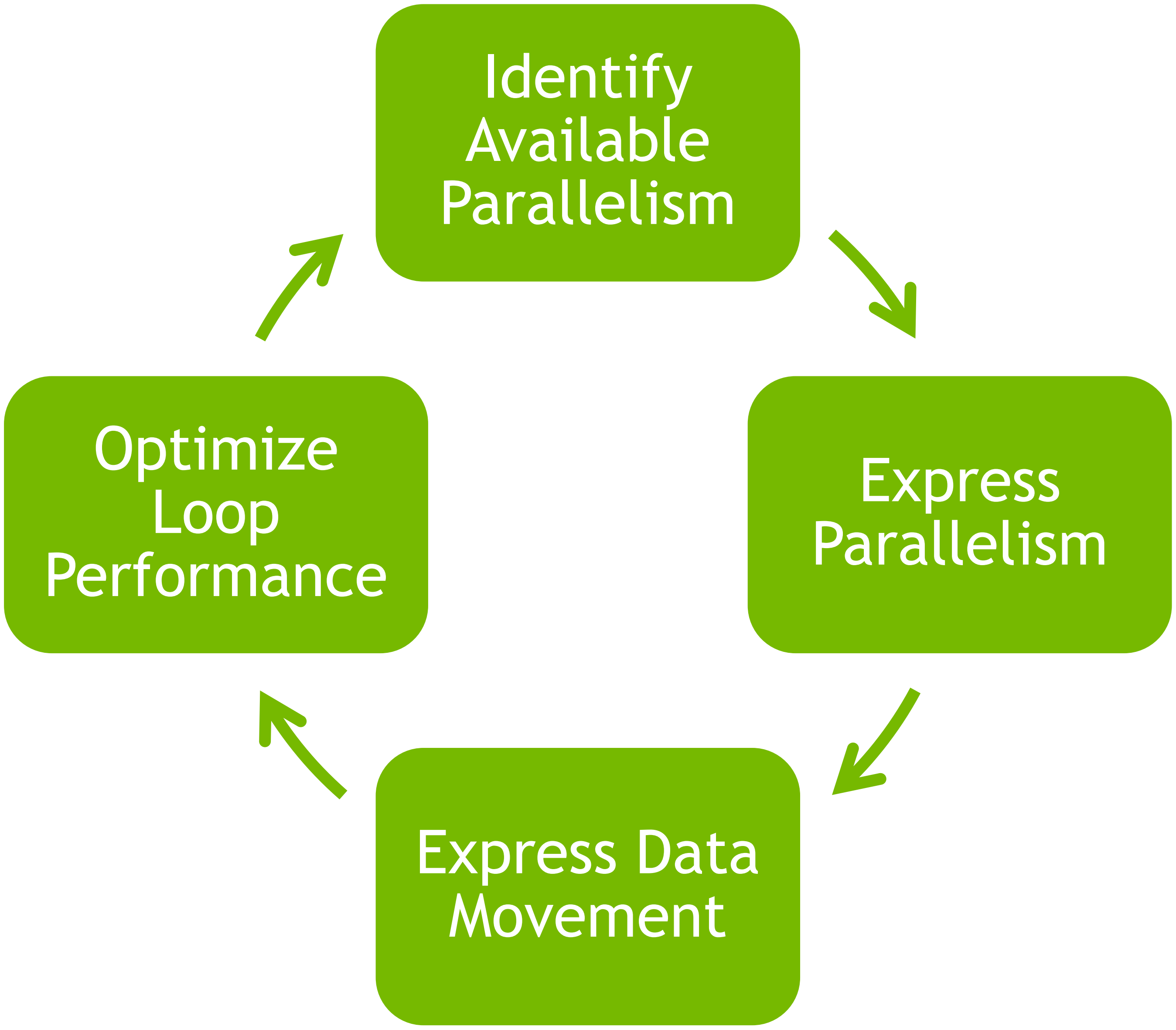
Directives

- Use directives for rapid and portable development

Languages

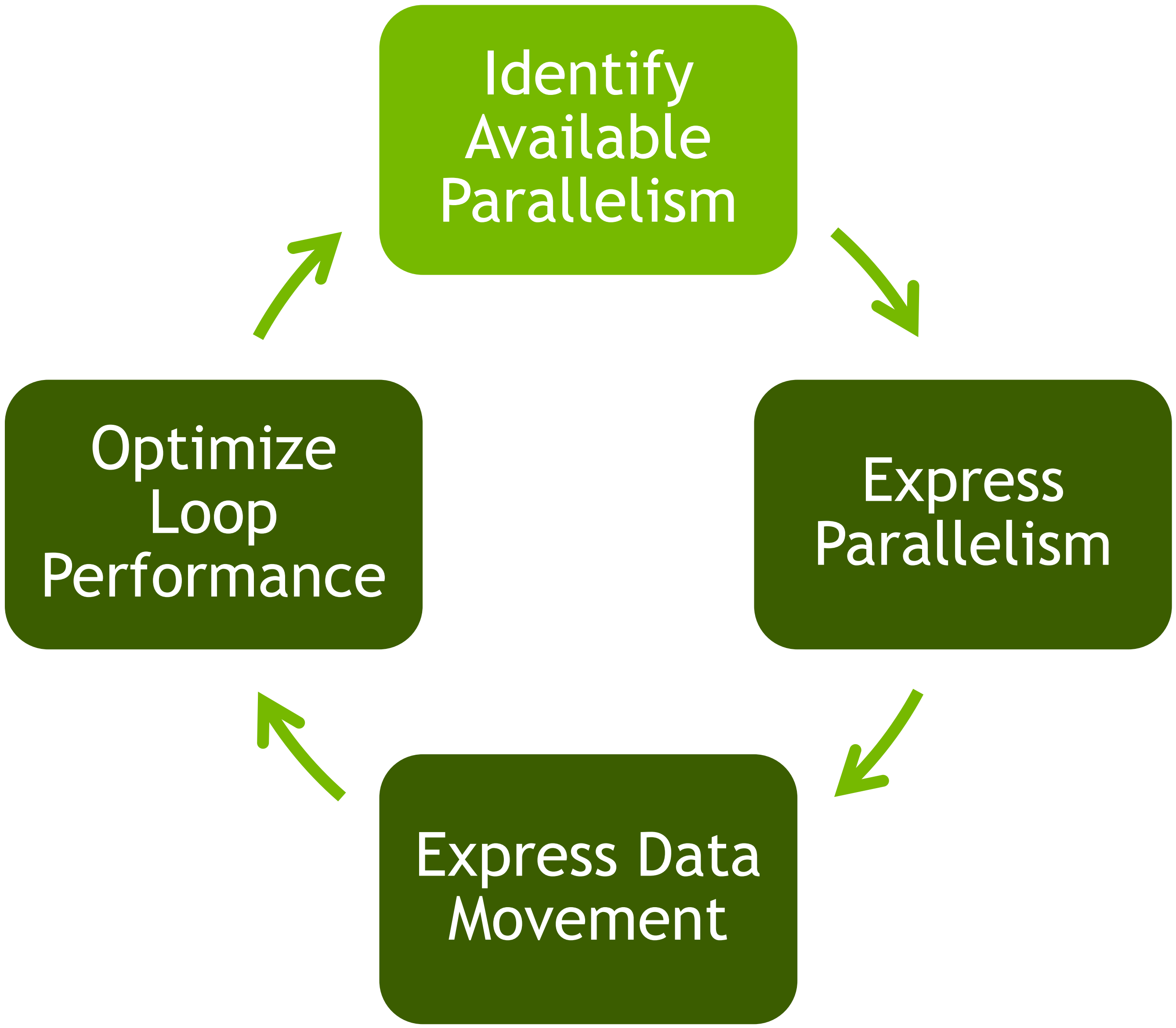
- Use lower level languages for important kernels

OpenACC Programming Cycle


Identify Available Parallelism
Express Parallelism
Express Data Movement
Optimize Loop Performance

A Simple Example

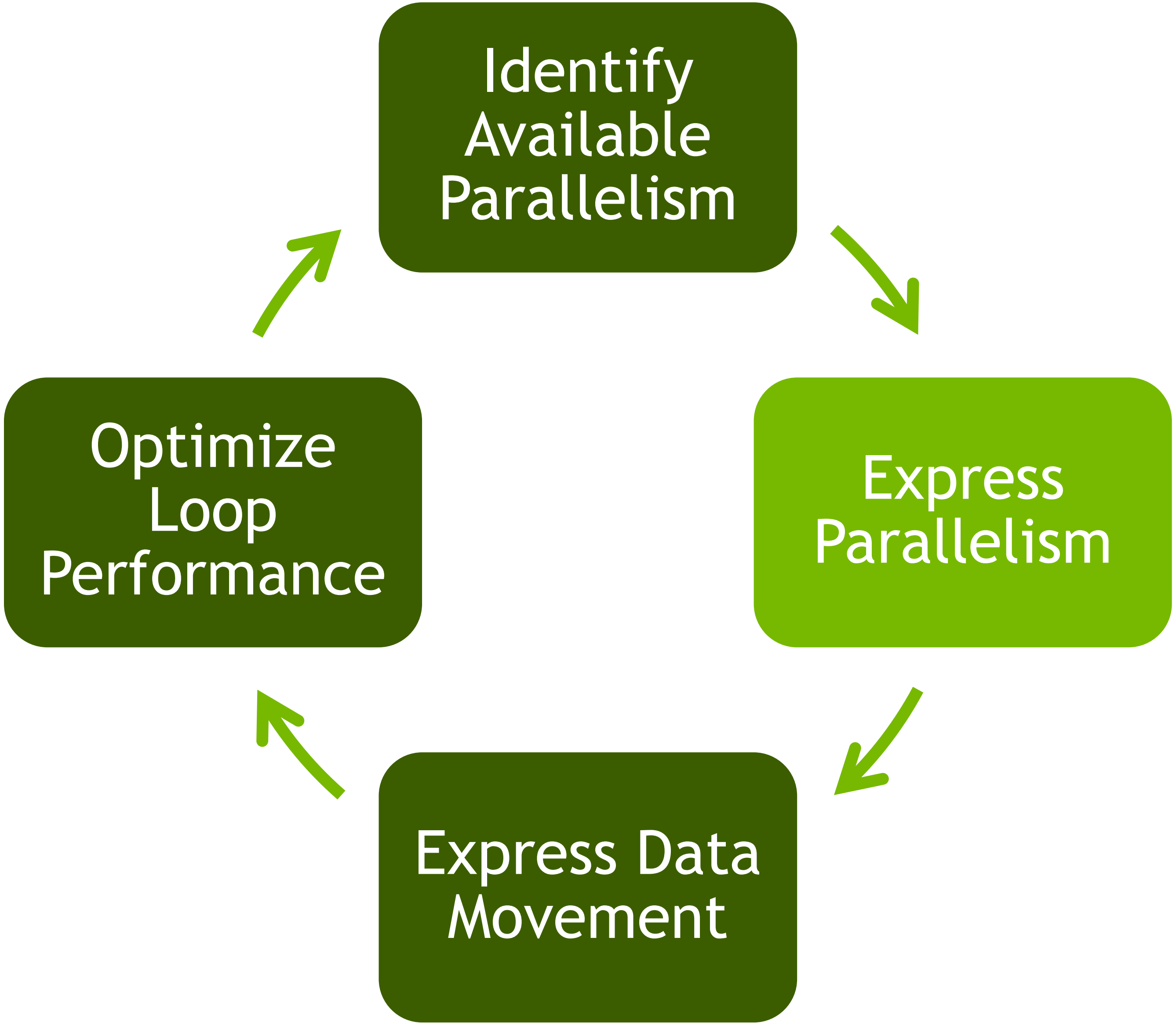
```
#include <stdio.h>
#include <stdlib.h>

#define N   (1<<20)

int main() {
    int i;
    int a[N];

    a[0] = 0;

    printf("a[0] = %d\n", a[0]);

    for (i=0; i<N; i++)
    {
        a[i] = a[i]+1;
    }

    printf("a[0] = %d\n", a[0]);

    return 0;
 }
```


Identify Available Parallelism
Express Parallelism
Express Data Movement
Optimize Loop Performance

A Simple Example

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

#define N   (1<<20)

int main() {
    int i;
    int a[N];

     a[0] = 0;

     printf("a[0] = %d\n", a[0]);

     for (i=0; i<N; i++)
     {
         a[i] = a[i]+1;
     }

     printf("a[0] = %d\n", a[0]);

     return 0;
 }
```

The loop is parallelizable


Identify Available Parallelism
Express Parallelism
Express Data Movement
Optimize Loop Performance

OpenACC Kernels Directive

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

Kernels Usage:

#pragma acc kernels [cluase]

```
#pragma acc kernels
{
    for(int i=0; i<N; i++)
    {
      x[i] = 1.0;
    }

    for(int i=0; i<N; i++)
    {
      y[i] = 2.0;
    }
}
```

kernel 1

kernel 2

The compiler identifies 2 parallel loops and generates 2 kernels.

A Simple Example

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

#define N   (1<<20)

int main() {
    int i;
    int a[N];

     a[0] = 0;

     printf("a[0] = %d\n", a[0]);

     for (i=0; i<N; i++)
     {
         a[i] = a[i]+1;
     }

     printf("a[0] = %d\n", a[0]);

     return 0;
 }
```

A Simple Example

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

#define N   (1<<20)

int main() {
    int i;
    int a[N];

     a[0] = 0;

     printf("a[0] = %d\n", a[0]);

 #pragma acc kernels
     for (i=0; i<N; i++)
     {
         a[i] = a[i]+1;
     }

     printf("a[0] = %d\n", a[0]);

     return 0;
 }
```

- The only change to the code
- The compiler will parallel the loop
- And a kernel will be generated

Execution of Serial Loops vs. Parallel Kernels

```
for (int i = 0; i < 16384; i++)
{
  C[i] = A[i] + B[i];
}
```

```
function loopBody(A, B, C, i)
{
  C[i] = A[i] + B[i];
}
```

Execution of Serial Loops vs. Parallel Kernels

```
for (int i = 0; i < 16384; i++)
{
  C[i] = A[i] + B[i];
}
```

```
function loopBody(A, B, C, i)
{
  C[i] = A[i] + B[i];
}
```

Calculate 0 -16383 in order.

Execution of Serial Loops vs. Parallel Kernels

```
for (int i = 0; i < 16384; i++)
{
  C[i] = A[i] + B[i];
}
```

Calculate 0 -16383 in order.

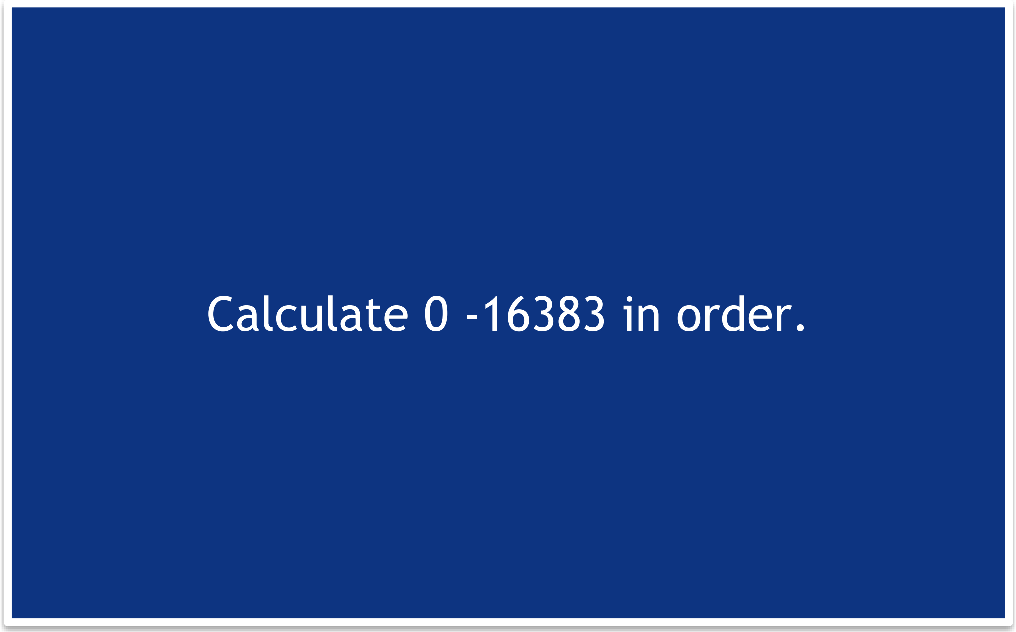
```
function loopBody(A, B, C, i)
{
  C[i] = A[i] + B[i];
}
```

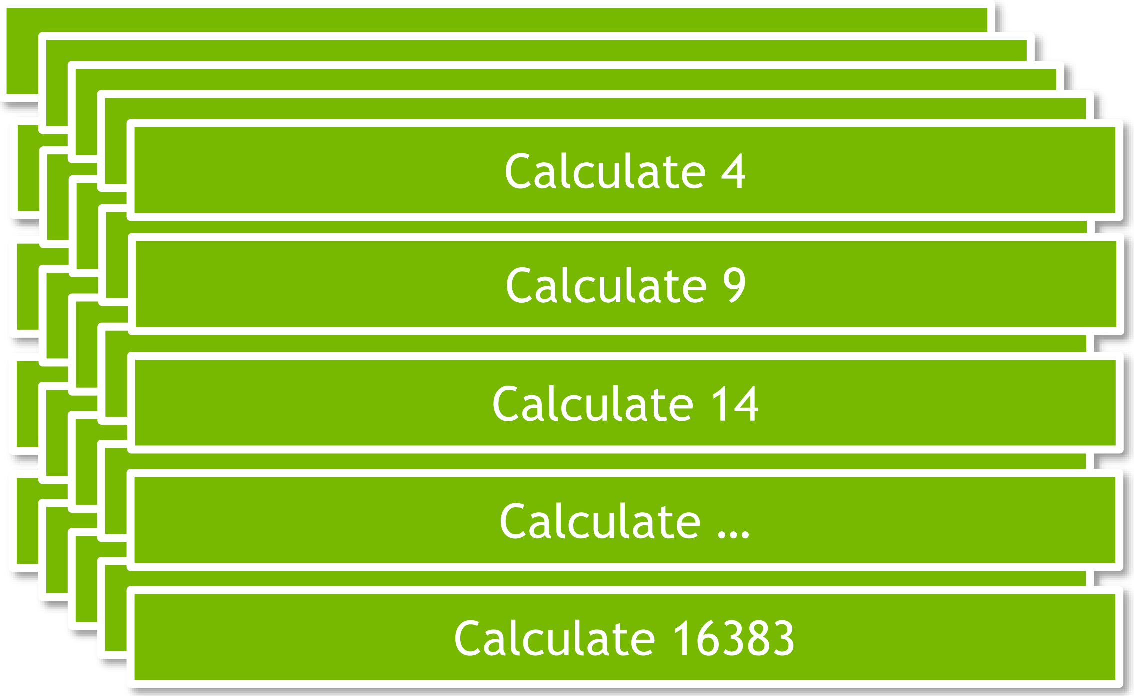
Calculate 0

Execution of Serial Loops vs. Parallel Kernels

```
for (int i = 0; i < 16384; i++)
{
  C[i] = A[i] + B[i];
}
```

```
function loopBody(A, B, C, i)
{
  C[i] = A[i] + B[i];
}
```





What Will happen? - Build OpenACC Code

Build the code: pgcc -acc -Minfo=accel -ta=tesla main.c

- -acc: enable OpenAcc directives
- -Minfo = accel: output openacc compiling message
- -ta = tesla: specify the target accelerator is NVIDIA Tesla GPU

Compiler output:

```
main:
     14, Generating copy(a[:])
     15, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         15, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

What Will happen? - Execute the Program

Running with profiling information: PGI_ACC_TIME=1 ./a.out

```
14: compute region reached 1 time
    15: kernel launched 1 time
        grid: [8192]  block: [128]
         device time(us): total=48 max=48 min=48 avg=48
        elapsed time(us): total=266 max=266 min=266 avg=266
14: data region reached 1 time
    14: data copyin transfers: 1
         device time(us): total=700 max=700 min=700 avg=700
20: data region reached 1 time
    20: data copyout transfers: 1
         device time(us): total=647 max=647 min=647 avg=647
```

What Will Happen by Default?

Compiling

- Compiler analyzes the data dependency of the marked region
- Compiler generates a kernel for the marked region

Running

- When entering the parallel region, allocates memory on GPU and copies data from CPU to GPU, corresponding the copyin at line 14
- Execute the generated kernel, corresponding the execution at line 15
- When exiting the parallel region, copies data from GPU to CPU and free the memory on GPU, corresponding the copyout at line 20

OpenACC Parallel Loop Directive

parallel - Programmer identifies a block of code containing parallelism. Compiler generates a *kernel*.

loop - Programmer identifies a loop that can be parallelized within the kernel.

NOTE: parallel & loop are often placed together

```
#pragma acc parallel loop
for(int i=0; i<N; i++)
{
  x[i] = 1;
  y[i] = 1;
}
```

Generates a Parallel Kernel

A Simple Example

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

#define N   (1<<20)

int main() {
    int i;
    int a[N];

     a[0] = 0;

     printf("a[0] = %d\n", a[0]);

     for (i=0; i<N; i++)
     {
         a[i] = a[i]+1;
     }

     printf("a[0] = %d\n", a[0]);

     return 0;
 }
```

A Simple Example

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

#define N   (1<<20)

int main() {
    int i;
    int a[N];

     a[0] = 0;

     printf("a[0] = %d\n", a[0]);

     for (i=0; i<N; i++)
     {
         a[i] = a[i]+1;
     }

     printf("a[0] = %d\n", a[0]);

     return 0;
 }
```

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

#define N   (1<<20)

int main() {
    int i;
    int a[N];

     a[0] = 0;

     printf("a[0] = %d\n", a[0]);

 #pragma acc parallel loop
     for (i=0; i<N; i++)
     {
         a[i] = a[i]+1;
     }

     printf("a[0] = %d\n", a[0]);

     return 0;
 }
```

What Will Happen? - Build OpenACC Code

Build the code

Compiler output for parallel loop:

```
main:
     14, Generating copy(a[:])
     14, Accelerator kernel generated
         Generating Tesla code
         15, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

Compare Compiler Output

Compiler output for parallel loop:

```
main:
     14, Generating copy(a[:])
     14, Accelerator kernel generated
         Generating Tesla code
         15, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

Compiler output for kernels:

```
main:
     14, Generating copy(a[:])
     15, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         15, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

Kernels VS Parallel Loop (1)

Kernels

- Kernels is a hint to the compiler.
- Notify the compiler there may be parallelism in the code marked by kernels
- Compiler takes charge of analyzing the code and guarantees the safe parallelsim

parallel loop:

- Parallel is an assertion to the compiler
- Notify the compiler there is parallelism in the code marked by parallel, and please parallelizes the code in spite of the safety
- It's the programmer's responsibility to ensure safe parallelism

So…

Kernels VS Parallel Loop (1)

Compiler output for parallel loop:

```
main:
     14, Generating copy(a[:])
     14, Accelerator kernel generated
         Generating Tesla code
         15, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

The programmer guarantees it is safe to parallelize

Compiler output for kernels:

```
main:
     14, Generating copy(a[:])
     15, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         15, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

The complier thinks it is safe to parallelize

Kernels VS Parallel Loop (2)

Kernels: pointer aliasing prevents parallelization

Kernels

- It’s the compiler responsibility to ensure safety
- In some cases the compiler may not have enough information to determine whether it is safe to parallelize a loop at compile time
- So, it will not parallelize the loop for the correctness

Example:

```
for(int i=0; i<N; i++)
{
  x[i] = 1.0;
  y[i] = x[i];
}
```

Kernels VS Parallel Loop (2)

Kernels: pointer aliasing prevents parallelization

Example:

```
for(int i=0; i<N; i++)
{
  x[i] = 1.0;
  y[i] = x[i];
}
```

```
#pragma acc kernels
for(int i=0; i<N; i++)
{
  x[i] = 1.0;
  y[i] = x[i];
}
```

```
#pragma acc parallel loop
for(int i=0; i<N; i++)
{
  x[i] = 1.0;
  y[i] = x[i];
}
```

Kernels VS Parallel Loop (2)

Kernels: pointer aliasing prevents parallelization

Example:

```
#pragma acc kernels
for(int i=0; i<N; i++)
{
  x[i] = 1.0;
  y[i] = x[i];
}
```

```
#pragma acc parallel loop
for(int i=0; i<N; i++)
{
  x[i] = 1.0;
  y[i] = x[i];
}
```

Compiling output for **kernels**:

```
Complex loop carried dependence of x-> prevents parallelization
Loop carried dependence of y-> prevents parallelization
Loop carried backward dependence of y-> prevents vectorization
Accelerator scalar kernel generated
```

- The dependence is caused by **pointer aliasing**
- Compiler thinks there is dependence between loop iterations
- Its region isn't parallelized. A **scalar kernel** is generated

Kernels VS Parallel Loop (2)

Kernels: pointer aliasing prevents parallelization

Example:

```
#pragma acc kernels
for(int i=0; i<N; i++)
{
  x[i] = 1.0;
  y[i] = x[i];
}
```

```
#pragma acc parallel loop
for(int i=0; i<N; i++)
{
  x[i] = 1.0;
  y[i] = x[i];
}
```

Compiling output for **parallel loop**:

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

- Compiler parallelizes the region directly without analyzing safety
- A **parallel kernel** is generated

How to Fix the Issue? The *Independent* Clause

Kernels: pointer aliasing prevents parallelization

Example:

```
#pragma acc kernels
for(int i=0; i<N; i++)
{
  x[i] = 1.0;
  y[i] = x[i];
}
```

Need to give compiler additional information to make the compiler can safely parallelize the region

The **Independent** clause

- Specifies that loop iterations are data independent. It overrides any compiler dependency analysis

How to Fix the Issue? The *Independent* Clause

Kernels: pointer aliasing prevents parallelization

Using **independent** clause:

```
#pragma acc kernels
for(int i=0; i<N; i++)
{
  x[i] = 1.0;
  y[i] = x[i];
}
```

```
#pragma acc kernels
#pragma acc loop independent
for(int i=0; i<N; i++)
{
  x[i] = 1.0;
  y[i] = x[i];
}
```

Rebuild the code

```
Loop is parallelizable
Accelerator kernel generated
Generating Tesla code
37, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

Finally the compiler safely parallelizes the code with the additional information

How to Fix the Issue? C99 **Restrict** Keyword

Kernels: pointer aliasing prevents parallelization

restrict : forbidding pointer alaising

- 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
- Usage: `float *restrict ptr`
- OpenACC compiler often requires *restrict* to determine independence

How to Fix the Issue? C99 **Restrict** Keyword

Kernels: pointer aliasing prevents parallelization

restrict : forbidding pointer alaising

```
int *restrict x = (int *)malloc(…)
int *restrict y = (int *)malloc(…)

#pragma acc kernels
for(int i=0; i<N; i++)
{
  x[i] = 1.0;
  y[i] = x[i];
}
```

Rebuild the code

```
Loop is parallelizable
Accelerator kernel generated
Generating Tesla code
37, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

Finally the compiler safely parallelizes the code with the additional information

Kernels VS Parallel Loop (3)

Kernels: A single **kernels** directive can parallelize larger area of code and generate multi ***kernels****(kernels executing on GPU)

Parallel loop: A single **parallel loop** directive only parallelizes one loop and generates one ***kernel***

Example with Two Loops

```
#include <stdio.h>
#include <stdlib.h>
#define N   (1<<20)

int main() {
    int i;
    int a[N];

    a[0] = 0;

    printf("a[0] = %d\n", a[0]);

    for (i=0; i<N; i++)
    {
        a[i] = a[i]+1;
    }
    for (i=0; i<N; i++)
    {
        a[i] = a[i]+1;
    }

    printf("a[0] = %d\n", a[0]);
    return 0;
}
```

Parallelize with Kernels

```c
#include <stdio.h>
#include <stdlib.h>
#define N   (1<<20)

int main() {
    int i;
    int a[N];

    a[0] = 0;

    printf("a[0] = %d\n", a[0]);

    for (i=0; i<N; i++)
    {
        a[i] = a[i]+1;
    }
    for (i=0; i<N; i++)
    {
        a[i] = a[i]+1;
    }

    printf("a[0] = %d\n", a[0]);
    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#define N   (1<<20)

int main() {
    int i;
    int a[N];

    a[0] = 0;

    printf("a[0] = %d\n", a[0]);

#pragma acc kernels
    {
    for (i=0; i<N; i++)
    {
        a[i] = a[i]+1;
    }
    for (i=0; i<N; i++)
    {
        a[i] = a[i]+1;
    }
    }

    printf("a[0] = %d\n", a[0]);
    return 0;
}
```

Compiler generates two kernels for the region

Parallelize with Parallel Loop

```c
#include <stdio.h>
#include <stdlib.h>
#define N   (1<<20)

int main() {
    int i;
    int a[N];

    a[0] = 0;

    printf("a[0] = %d\n", a[0]);

    for (i=0; i<N; i++)
    {
        a[i] = a[i]+1;
    }
    for (i=0; i<N; i++)
    {
        a[i] = a[i]+1;
    }

    printf("a[0] = %d\n", a[0]);
    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#define N   (1<<20)

int main() {
    int i;
    int a[N];

    a[0] = 0;

    printf("a[0] = %d\n", a[0]);

#pragma acc parallel loop
    for (i=0; i<N; i++)
    {
        a[i] = a[i]+1;
    }
#pragma acc parallel loop
    for (i=0; i<N; i++)
    {
        a[i] = a[i]+1;
    }

    printf("a[0] = %d\n", a[0]);
    return 0;
}
```

Kernel 1

Kernel 2

Kernels VS Parallel Loop (3): Profile Results

Profile result of **kernels**

```
13: compute region reached 1 time
        15: kernel launched 1 time
            grid: [8192]  block: [128]
             device time(us): total=48 max=48 min=48 avg=48
            elapsed time(us): total=256 max=256 min=256 avg=256
        19: kernel launched 1 time
            grid: [8192]  block: [128]
             device time(us): total=46 max=46 min=46 avg=46
            elapsed time(us): total=63 max=63 min=63 avg=63
13: data region reached 1 time
        13: data copyin transfers: 1
             device time(us): total=703 max=703 min=703 avg=703
25: data region reached 1 time
        25: data copyout transfers: 1
             device time(us): total=647 max=647 min=647 avg=647
```

Execute kernel 1

Execute kernel 2

One Copyin

One Copyout

Kernels VS Parallel Loop (3): Profile Results

Profile result of parallel loop

```
13: compute region reached 1 time
        13: kernel launched 1 time
            grid: [8192]  block: [128]
             device time(us): total=48 max=48 min=48 avg=48
            elapsed time(us): total=257 max=257 min=257 avg=257
13: data region reached 1 time
        13: data copyin transfers: 1
             device time(us): total=702 max=702 min=702 avg=702
18: compute region reached 1 time
        18: kernel launched 1 time
            grid: [8192]  block: [128]
             device time(us): total=46 max=46 min=46 avg=46
            elapsed time(us): total=68 max=68 min=68 avg=68
18: data region reached 2 times
        18: data copyin transfers: 1
             device time(us): total=692 max=692 min=692 avg=692
        18: data copyout transfers: 1
             device time(us): total=647 max=647 min=647 avg=647
24: data region reached 1 time
        24: data copyout transfers: 1
             device time(us): total=644 max=644 min=644 avg=644
```

- Execute kernel 1
- One Copyin
- Execute kernel 2
- One Copyin
- One Copyout
- One Copyout

Kernels VS Parallel Loop (3): Profile Results

Kernels:

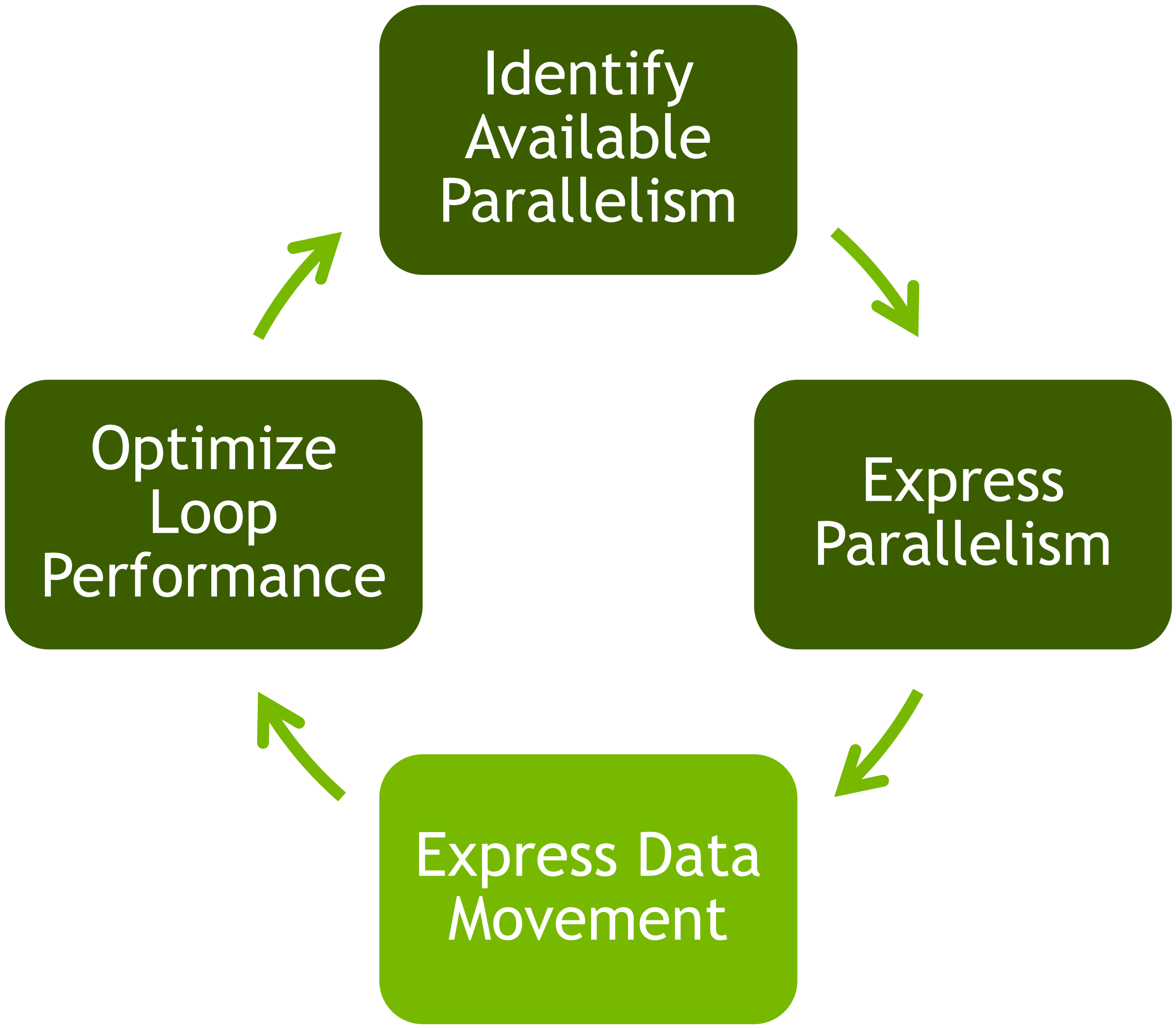
- There is only one pair of copyin (CPU to GPU) and copyout (GPU to CPU)
- Correspond to the single **kernels** directive

Parallel loop

- There are two pairs of copyin and copyout
- Correspond to the two **Parallel loop** directives
- The copyout and copyin between the two kernels aren’t necessary

Given the *PCIe* transfer is slow, which will definitely harms the performance. Can we eliminate the unnecessary data copy?

The answer is Yes! Let us go to the next topic…


Identify Available Parallelism
Express Parallelism
Express Data Movement
Optimize Loop Performance

Data Region

The `data` directive defines a region of code in which GPU arrays remain on the GPU and are shared among all kernels in that region.

```
#pragma acc data
{
#pragma acc kernels/parallel loop
...

#pragma acc kernels/parallel loop
...
}
```

Data Region

Arrays used within the data region will remain on the GPU until the end of the data region.

Data Clauses

`copy ( list )`	Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.
`copyin ( list )`	Allocates memory on GPU and copies data from host to GPU when entering region.
`copyout ( list )`	Allocates memory on GPU and copies data to the host when exiting region.
`create ( list )`	Allocates memory on GPU but does not copy.
`present ( list )`	Data is already present on GPU from another containing data region.
`deviceptr( list )`	The variable is a device pointer (e.g. CUDA) and can be used directly on the device.

Array Shaping

Compiler sometimes cannot determine size of arrays

- Must specify explicitly using data clauses and array "shape"

<u>C/C++</u>

```
#pragma acc data copyin(a[0:nelem]) copyout(b[s/4:3*s/4])
```

<u>Fortran</u>

```
!$acc data copyin(a(1:end)) copyout(b(s/4:3*s/4))
```

Note: data clauses can be used on `data`, `parallel`, or `kernels`

Define Data Region to Eliminate Unnecessary Copy

```
#include <stdio.h>
#include <stdlib.h>
#define N   (1<<20)

int main() {
    int i;
    int a[N];

    a[0] = 0;

    printf("a[0] = %d\n", a[0]);

#pragma acc parallel loop
    for (i=0; i<N; i++)
    {
        a[i] = a[i]+1;
    }
#pragma acc parallel loop
    for (i=0; i<N; i++)
    {
        a[i] = a[i]+1;
    }

    printf("a[0] = %d\n", a[0]);
    return 0;
}
```

Array a isn’t used by host code, so the copy is unnecessary

Define Data Region to Eliminate Unnecessary Copy

```
#include <stdio.h>
#include <stdlib.h>
#define N   (1<<20)

int main() {
    int i;
    int a[N];

    a[0] = 0;

    printf("a[0] = %d\n", a[0]);

#pragma acc parallel loop
    for (i=0; i<N; i++)
    {
        a[i] = a[i]+1;
    }
#pragma acc parallel loop
    for (i=0; i<N; i++)
    {
        a[i] = a[i]+1;
    }

    printf("a[0] = %d\n", a[0]);
    return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#define N   (1<<20)

int main() {
    int i;
    int a[N];

    a[0] = 0;

     printf("a[0] = %d\n", a[0]);

 #pragma acc data copy(a[0:N])
     {
 #pragma acc parallel loop
         for (i=0; i<N; i++)
         {
             a[i] = a[i]+1;
         }
 #pragma acc parallel loop
         for (i=0; i<N; i++)
         {
             a[i] = a[i]+1;
         }
     }

     printf("a[0] = %d\n", a[0]);
     return 0;
 }
```

Define Data Region to Eliminate Unnecessary Copy

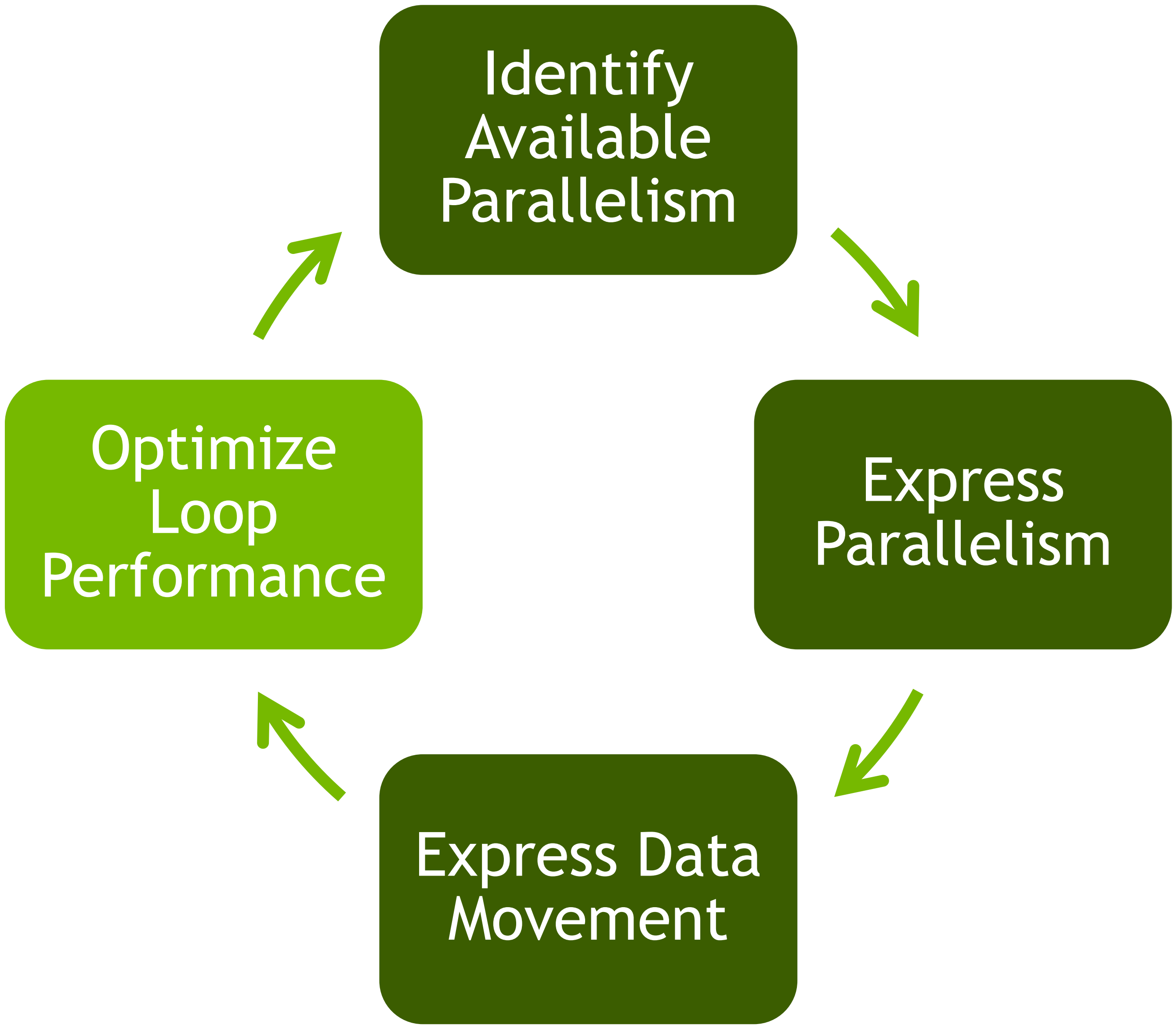
Profile result of parallel loop after defined data region

```
13: data region reached 1 time
    13: data copyin transfers: 1
         device time(us): total=412 max=412 min=412 avg=412
15: compute region reached 1 time
    15: kernel launched 1 time
        grid: [8192]  block: [128]
         device time(us): total=55 max=55 min=55 avg=55
        elapsed time(us): total=254 max=254 min=254 avg=254
20: compute region reached 1 time
    20: kernel launched 1 time
        grid: [8192]  block: [128]
         device time(us): total=52 max=52 min=52 avg=52
        elapsed time(us): total=69 max=69 min=69 avg=69
27: data region reached 1 time
    27: data copyout transfers: 1
         device time(us): total=412 max=412 min=412 avg=412
```

One Copyin

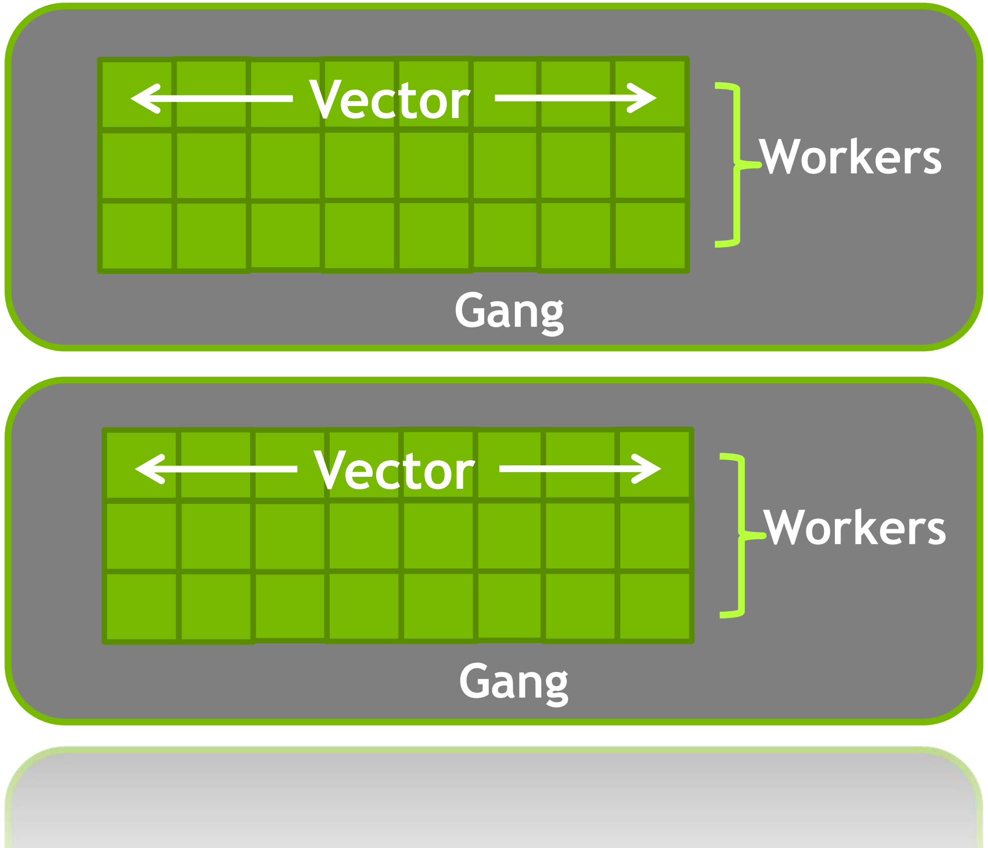
One Copyout

Only one pair of copyin and copyout is left!


Identify Available Parallelism
Express Parallelism
Express Data Movement
Optimize Loop Performance

Optimize the Kernels Generated by OpenACC

OpenACC: 3 Levels of Parallelism



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

Mapping the Parallelism from OpenACC to CUDA

The compiler is free to do what they want

In general

- gang: mapped to blocks (COARSE GRAIN)
- worker: mapped threads (.y) (FINE GRAIN)
- vector: mapped to threads (.x) (FINE SIMD)

Exact mapping is compiler dependent

Performance Tips:

- Use a vector size that is divisible by 32
- Block size is num_workers * vector_length

How to Use gang, worker, vector Clauses?

gang, worker, and vector can be added to a loop clause

Control the size using the following clauses on the parallel region

- parallel: num_gangs(n), num_workers(n), vector_length(n)
- Kernels: gang(n), worker(n), vector(n)

```
#pragma acc parallel loop gang
for (int i = 0; i < n; ++i)
  #pragma acc loop worker
  for (int j = 0; j < n; ++j)
   ...
```

```
#pragma acc parallel vector_length(32)
#pragma acc loop gang
for (int i = 0; i < n; ++i)
  #pragma acc loop vector
  for (int j = 0; j < n; ++j)
    ...
```

⚠ gang, worker, vector appear once per parallel region

Case Study - Lattice Boltzmann Method (LBM)

Introduction to LBM

LBM is a class of computational fluid dynamics (CFD) methods for fluid simulation

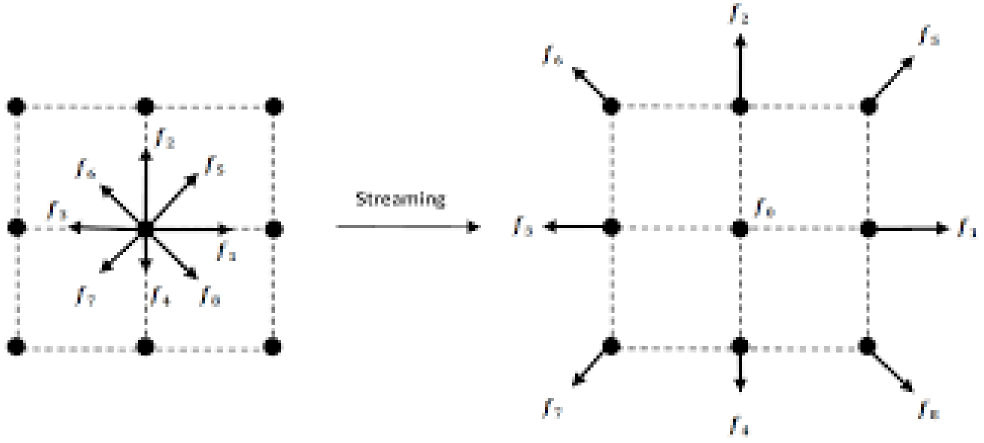
The computation of LBM is divided into two step

- Collide at current node (totally local operation)

$$f_{\mathrm{i}}^{'}(\mathbf{x},t) = f_i(\mathbf{x},t) + \frac{1}{\tau}[f_i^{eq} - f_i]$$

- Stream to adjacent nodes (D2Q9)

$$f_i(\mathbf{x} + e_i \Delta t, t + \Delta t) = \overline{f}_i(\mathbf{x},t)$$



In this case, I take D3Q15 for example

Code Description

Code: 560 lines

Two main functions: propagate and collision

Baseline performance

- Domain Size: 128x64x64, D3Q15
- Time step: 500
- *Time with CPU (E5-2698 v3 @ 2.30 GHz): 118.5 s

** We haven't parallelize the code on CPU. Only a single CPU core is used*

Propagate

```
void propagate()
{
    int x, y, z;
    int xp, yp, zp;
    int k;

    for (z=0;z<Nz;z++)
    {
        for (y=0;y<Ny;y++)
        {
            for (x=0;x<Nx;x++)
            {
                // computing code
                ...
            }
        }
    }
}
```

Collision

```
void collision()
{
    int x, y, z;
    int xp, yp, zp;
    int k;

    for (z=0;z<Nz;z++)
    {
        for (y=0;y<Ny;y++)
        {
            for (x=0;x<Nx;x++)
            {
                // computing code
                ...
            }
        }
    }
}
```

Accelerate the Code with OpenACC

Step 1: Find the hotspot

- the main loop (timeStep = 500)

```
for (int t=1; t<=timeStep; t++)
{
    propagete();
    collision();
    if (t%hop==0)
    {
        cout<<"timestep = "<<t<<endl;
    }
}
```

Accelerate the Code with OpenACC

Step 2-1: Parallelize the nested loop in the propagate function

- only one single line code is added

```
void propagate()
{
	int x, y, z;
	int xp, yp, zp;
	int k;

#pragma acc parallel loop present(f0, f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12, f13, f14)    \
                          present(f0temp, f1temp, f2temp,  f3temp,  f4temp,  f5temp,  f6temp, f7temp, \
                          f8temp, f9temp, f10temp, f11temp, f12temp, f13temp, f14temp)
	for (z=0;z<Nz;z++)
	{
		for (y=0;y<Ny;y++)
		{
			for (x=0;x<Nx;x++)
			{

				// computing code
				...
			}
		}
	}
}
```

Accelerate the Code with OpenACC

Step 2-2: Parallelize the nested loop in the collision function

- only one single line code is added

```
void collision()
{
     int x, y, z;
     int xp, yp, zp;
     int k;

#pragma acc parallel loop present(f0, f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12, f13, f14)    \
                          present(f0temp, f1temp, f2temp,  f3temp,  f4temp,  f5temp,  f6temp, f7temp, \
                          f8temp, f9temp, f10temp, f11temp, f12temp, f13temp, f14temp)
     for (z=0;z<Nz;z++)
     {
          for (y=0;y<Ny;y++)
          {
               for (x=0;x<Nx;x++)
               {

                    // computing code
                    ...
               }
          }
     }
}
```

Accelerate the Code with OpenACC

Step 3: Manage date movement by copying the necessary data to GPU before entering the main loop

- only one single line code is added

```
#pragma acc data copyin(flag) \
                 copy(f0, f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12, f13, f14)      \
                 create(f0temp, f1temp, f2temp,  f3temp,  f4temp,  f5temp,  f6temp, f7temp, \
                        f8temp, f9temp, f10temp, f11temp, f12temp, f13temp, f14temp)
{
    for (int t=1;t<=t_max;t++)
    {
        propagate();
        collision();
        if (t%hop==0)
        {
            cout<<"timestep = "<<t<<endl;
        }
    }
}
```

Accelerate the Code with OpenACC

By far, only 3 lines of code are added into the original code

Performance on P100: 1.275 s

- Speedup: 118.500 / 1.275 = 93X

Continue to optimize the code by specifying the 3 levels of parallelism

- the outmost loop: gang
- the middle loop: worker
- the inner loop: vector

Accelerate the Code with OpenACC

Step 4-1: Use the 3 levels of parallelism to optimize the loop performance

- **propagate**: only 4 lines of code are added

```
void propagate()
{
     int x, y, z;

#pragma acc parallel present(f0, f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12, f13, f14)     \
                     present(f0temp, f1temp, f2temp,  f3temp,  f4temp,  f5temp,  f6temp, f7temp, \
                             f8temp, f9temp, f10temp, f11temp, f12temp, f13temp, f14temp)         \
                     device_type(nvidia) num_workers(8) vector_length(Nx)
#pragma acc loop device_type(nvidia) gang
     for (z=0;z<Nz;z++)
     {
#pragma acc loop device_type(nvidia) worker
          for (y=0;y<Ny;y++)
          {
#pragma acc loop device_type(nvidia) vector
               for (x=0;x<Nx;x++)
               {
                    // computing code
                    ...
               }
          }
     }
}
```

Accelerate the Code with OpenACC

Step 4-2: Use the 3 levels of parallelism to optimize the loop performance

- **collision**: only 4 lines of code are added

```
void collision()
{
     int x, y, z;

#pragma acc parallel present(f0, f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12, f13, f14)     \
                     present(f0temp, f1temp, f2temp,  f3temp,  f4temp,  f5temp,  f6temp, f7temp, \
                             f8temp, f9temp, f10temp, f11temp, f12temp, f13temp, f14temp)        \
                     device_type(nvidia) num_workers(8) vector_length(Nx)
#pragma acc loop device_type(nvidia) gang
     for (z=0;z<Nz;z++)
     {
#pragma acc loop device_type(nvidia) worker
          for (y=0;y<Ny;y++)
          {
#pragma acc loop device_type(nvidia) vector
               for (x=0;x<Nx;x++)
               {
                    // computing code
                    ...
               }
          }
     }
}
```

Accelerate the Code with OpenACC

Finally, only 11 lines of code are added into the original code

Final performance on P100 with OpenACC optimization: 1.135 s

- Speedup: 118.500 / 1.135 = 104X

Use CUDA to rewrite the two functions

- Workload: 353 lines of code
- Performance: 0.644 s
- Speedup over OpenACC: 1.135 / 0.644 = 1.76X

Summary

Summary (1)

Why OpenACC?

- Open, Simple and Portable

Accelerated Computation

- Accelerate computation needs accelerator, typically the CPU+GPU heterogeneous system

OpenACC Programming Cycle

- Parallelism analysis
- Express parallelism to compiler with directive
- Define data region to eliminate unnecessary data copy
- Optimize the loop performance according to the architecture

Summary (2)

OpenACC Directives

- **kernels**
    - Tell compiler there may be parallelism
    - Compiler analyzes the dependency and determines how to parallelize the code
    - Compiler's responsibility to ensure safe parallelism
    - Need more information in order to guarantee parallelizing, eg. **independent** or **restrict**
- **Parallel loop**
    - An assertion to compiler that there is parallelism in the loop marked by parallel loop
    - Compiler must generate a kernel for the loop
    - Programmer's responsibility to ensure safe parallelism

Summary (3)

OpenACC Directive

- **data and its clauses**
    - Define data region to eliminate unnecessary data copy for the sake of performance.
    - Clauses: copy, copyin, copyout, create et al.
- **the three levels of parallelism**
    - gang
    - worker
    - vector
    - gang, worker, vector appear once per parallel region

Thanks for Your Attention!