Introduction to OpenACC

周国峰 chandlerz@nvidia.com

NVIDIA

Agenda

Why OpenACC?

Accelerated Computing Fundamentals

OpenACC Programming Cycle

2 Cases Study

Agenda

Why OpenACC?

Accelerated Computing Fundamentals

OpenACC Programming Cycle

2 Cases Study

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

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];
          ...
      }
  }
  ...
}
```

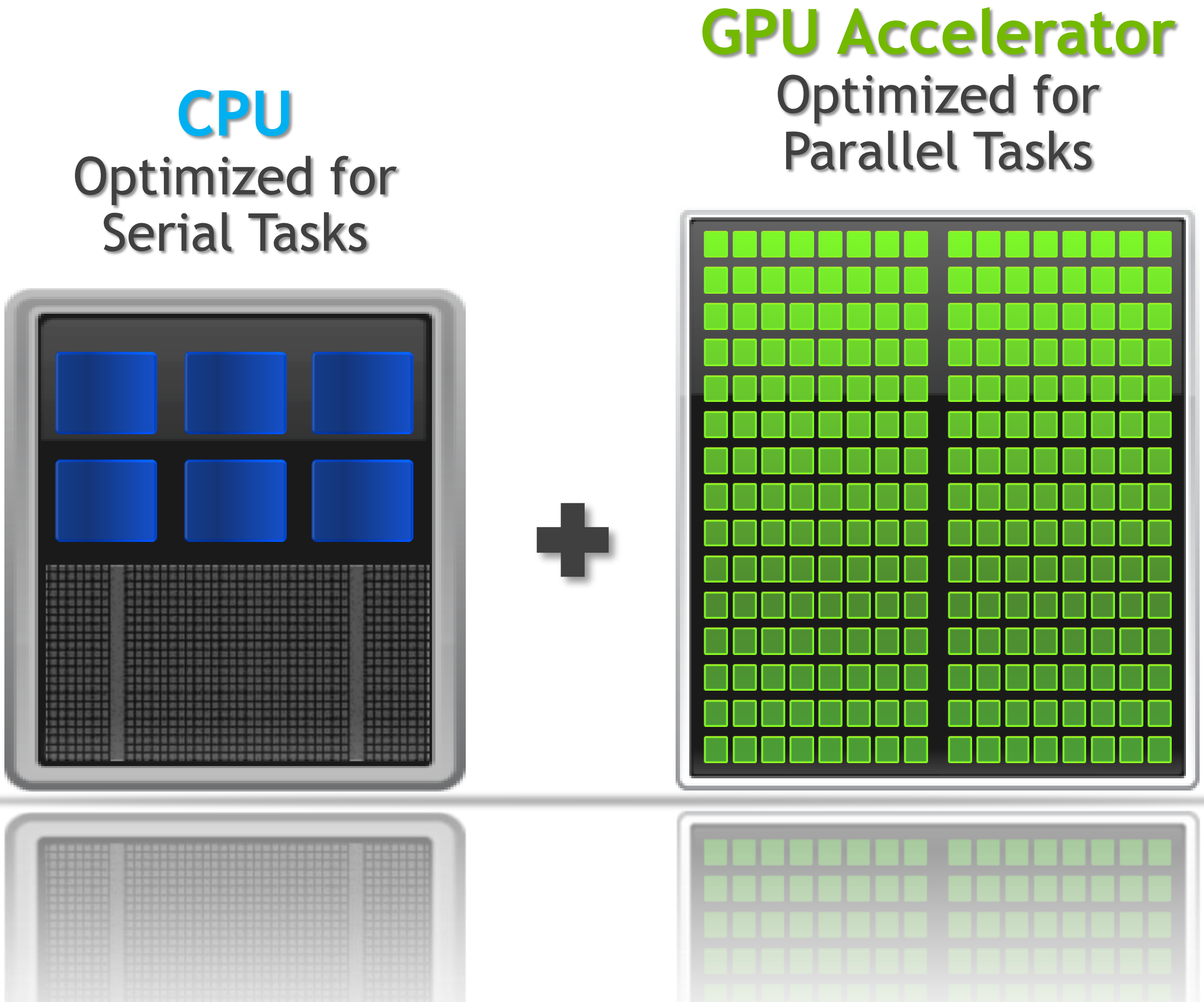
OpenACC
Directives for Accelerators

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

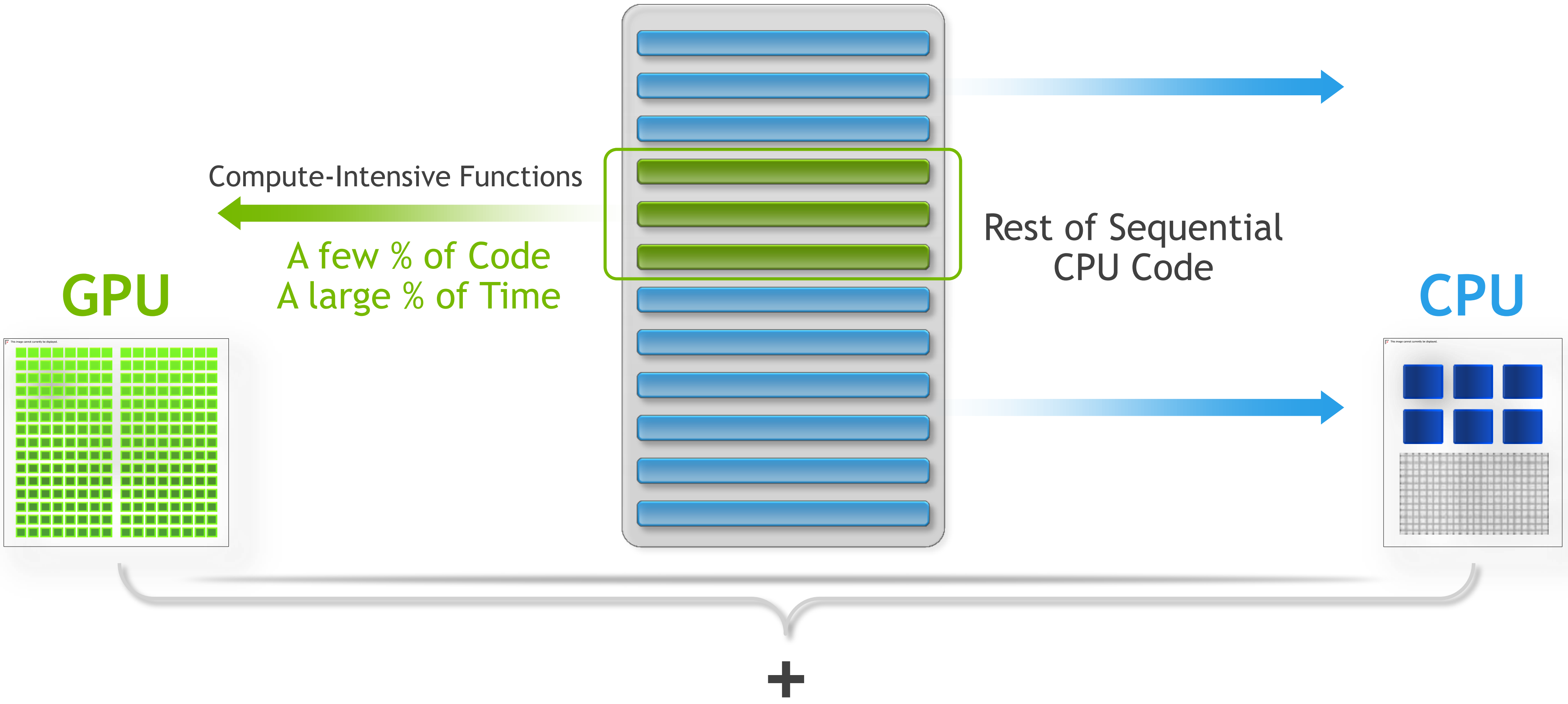
Accelerated Computing Fundamentals

Accelerated Computing

10x Performance & 5x Energy Efficiency for HPC



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

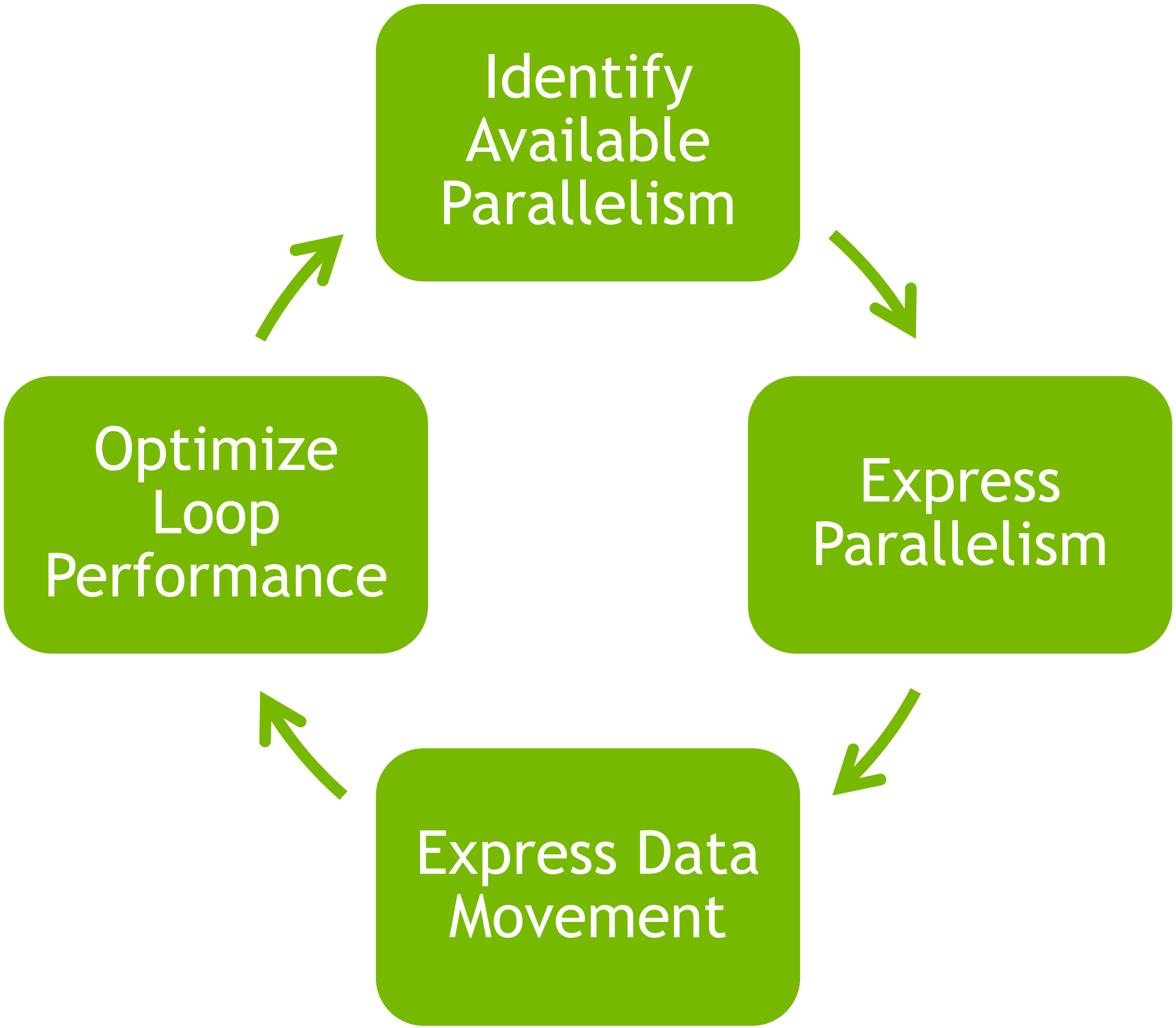
Directives

- Use directives for rapid and portable development

Languages

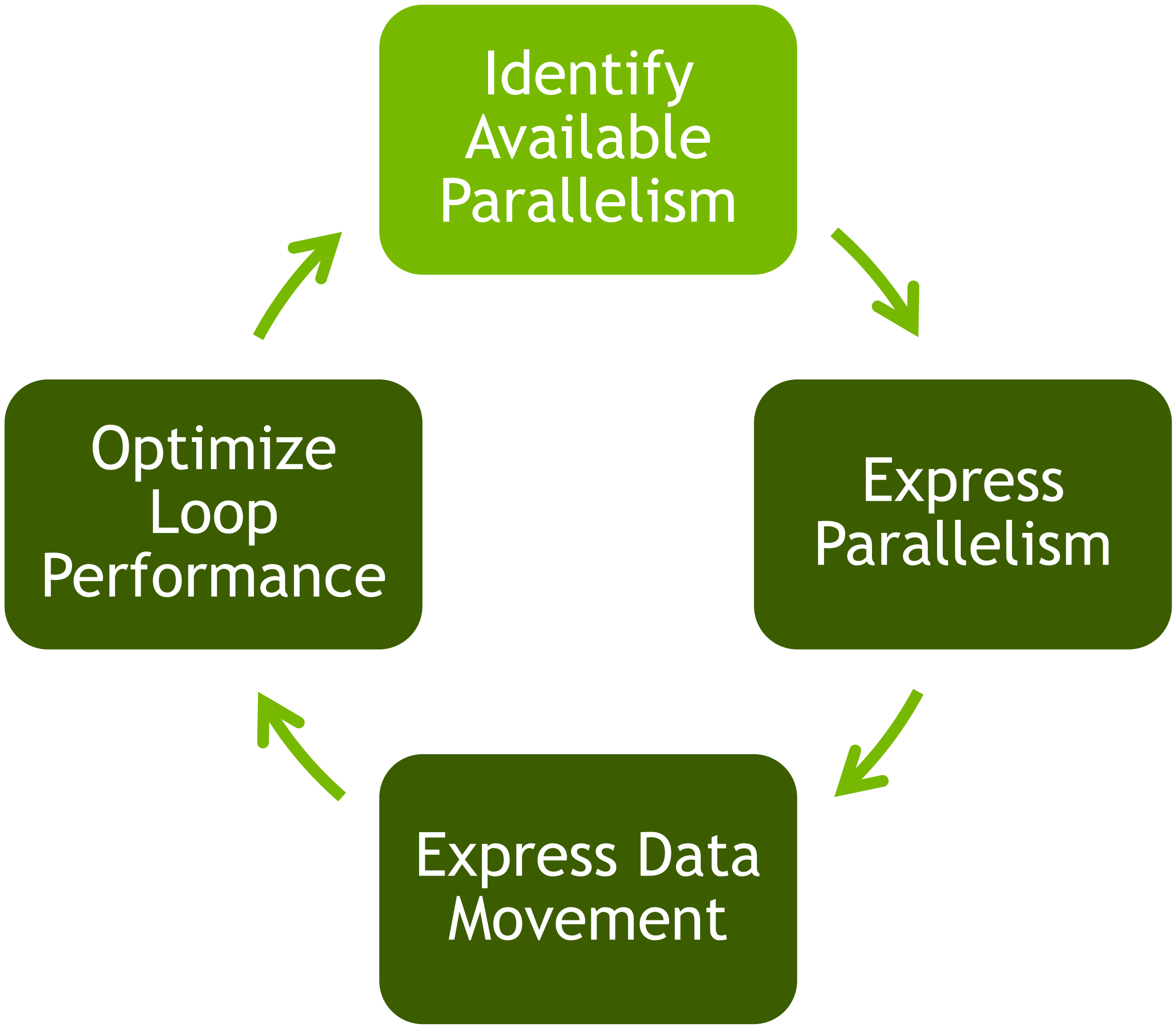
- Use lower level languages for important kernels

OpenACC Programming Cycle


Identify Available Parallelism
Express Parallelism
Express Data Movement
Optimize Loop Performance

A Simple Example

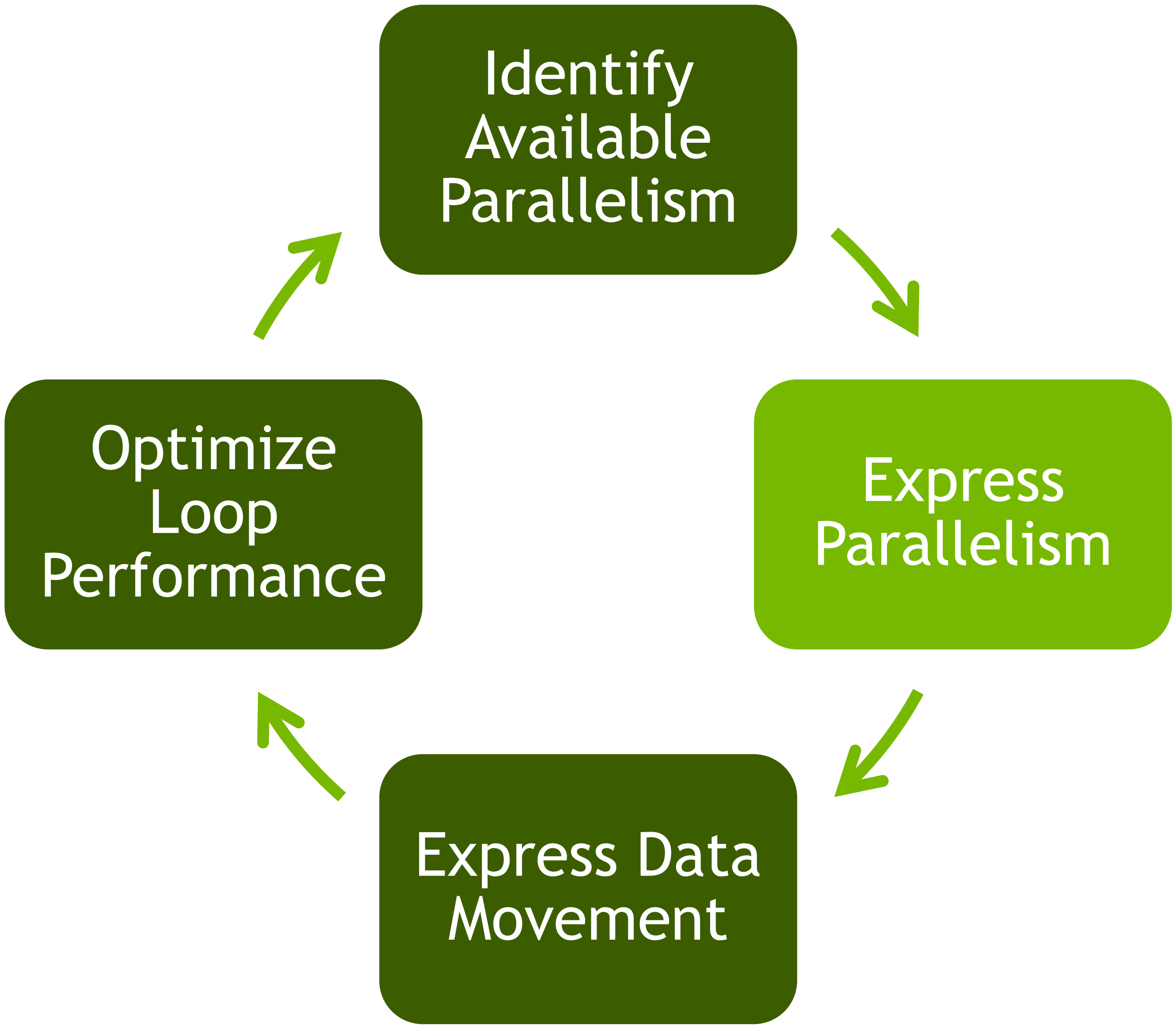
```
#include <stdio.h>
#include <stdlib.h>

#define N   (1<<20)

int main() {
    int i;
    int a[N]={0};

    a[0] = 1;

    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]={0};

     a[0] = 1;

     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 [clause]

```
#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]={0};

     a[0] = 1;

     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]={0};

     a[0] = 1;

     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 GPU
- The compiler will parallel the loop
- And a kernel will be generated

Execution of Serial Loops vs. Parallel Kernels

Calculate 0 - N in order.

Execution of Serial Loops vs. Parallel Kernels

Calculate 0 - N in order.

Calculate 0

Execution of Serial Loops vs. Parallel Kernels

What will happen?---Build OpenACC Code

Build the code

- pgcc -acc -Minfo=accel -ta=tesla -Mcuda=cc60 main.c
- -acc: OpenAcc Directives
- -Minfo = accel: output compiling message
- -ta = tesla for NVIDIA GPU (or radeon for AMD GPU)
- Other flags
  - set PGI_ACC_TIME=1 to output profiling result
  - -Mcuda=cc35 specify GPU arch
  - -Mcuda=keepgpu keep kernel source files

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 OpenACC Program

Profile Result

```
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

Runtime

- 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 in parallel
- 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

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

#define N   (1<<20)

int main() {
    int i;
    int a[N]={0};

     a[0] = 1;

     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]={0};

     a[0] = 1;

     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] ={0};

     a[0] = 1;

     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 parallelism

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 sake of 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 may be dependence between loop iterations
- The 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? 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

Independent clause

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

How to fix the issue? 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 aliasing

- 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 compilers often require `restrict` to determine independence

How to fix the issue? C99 restrict keyword

Kernels: pointer aliasing prevents parallelization

restrict : forbidding pointer aliasing

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

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

int main() {
    int i;
    int a[N]={0};

    a[0] = 1;

    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

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

int main() {
    int i;
    int a[N]={0};

    a[0] = 1;

    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;
}
```

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

int main() {
    int i;
    int a[N]={0};

    a[0] = 1;

    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]={0};

    a[0] = 1;

    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]={0};

    a[0] = 1;

    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:

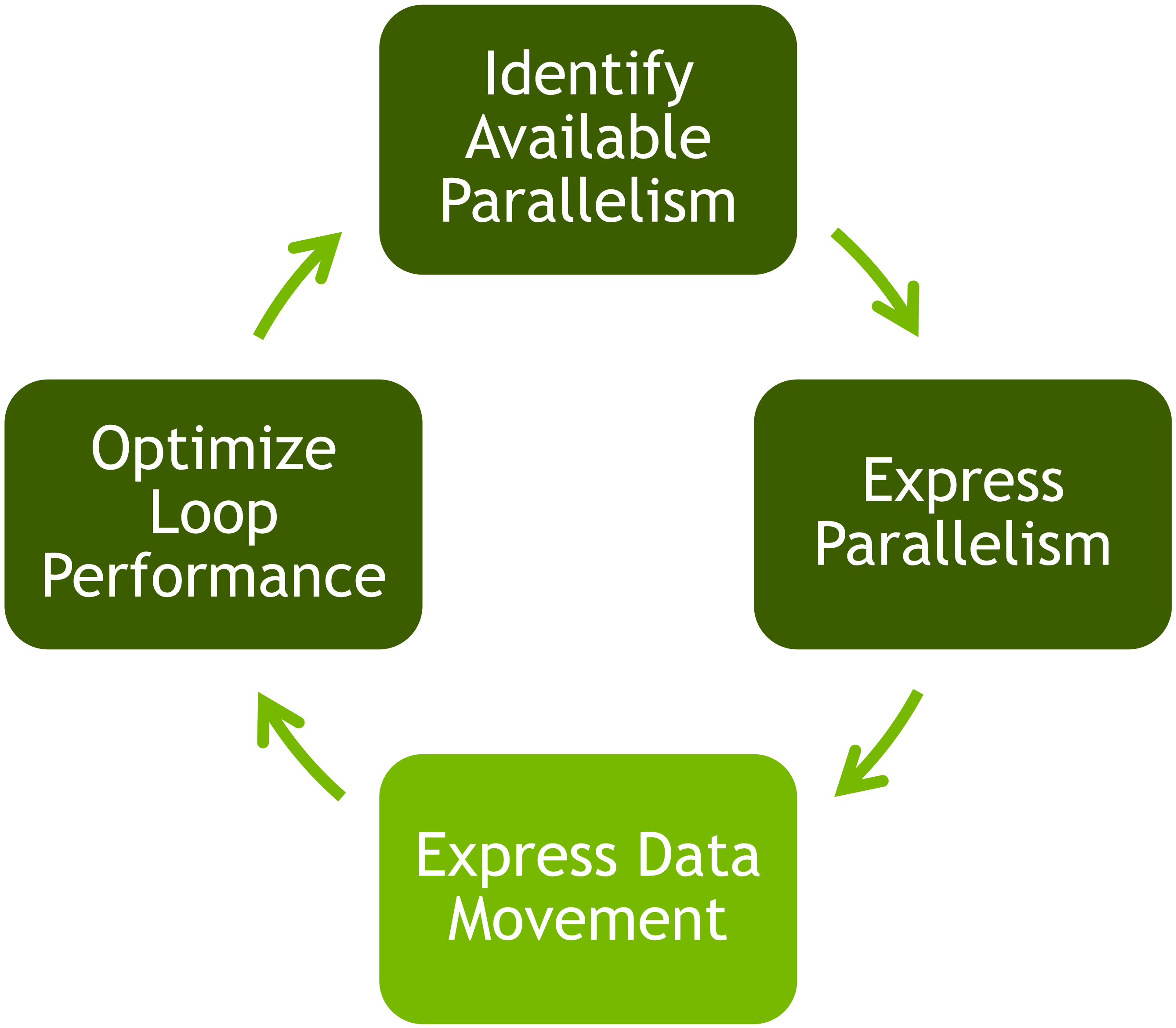
- There is only one pair of copyin (CPU to GPU) and copyout (GPU to CPU)
- Correspond to the single **kernels** directive

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?

Yes, Let us go to the next topic…


Identify Available Parallelism
Express Parallelism
Express Data Movement
Optimize Loop Performance

Structured Data Regions

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.

Unstructured Data Directives

Used to define data regions when scoping doesn't allow the use of normal data regions (e.g. the constructor/destructor of a class).

`enter data` Defines the start of an unstructured data lifetime

- clauses: `copyin(list)`, `create(list)`

`exit data` Defines the end of an unstructured data lifetime

- clauses: `copyout(list)`, `delete(list)`

```
#pragma acc enter data copyin(a)
...
#pragma acc exit data delete(a)
```

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"

C/C++

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

Fortran

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

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

Define data region to eliminate unnecessary copy

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

int main() {
    int i;
    int a[N]={0};

    a[0] = 1;

    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

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

int main() {
    int i;
    int a[N]={0};

    a[0] = 1;

    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;
}
```

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

int main() {
    int i;
    int a[N]={0};

    a[0] = 1;

     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

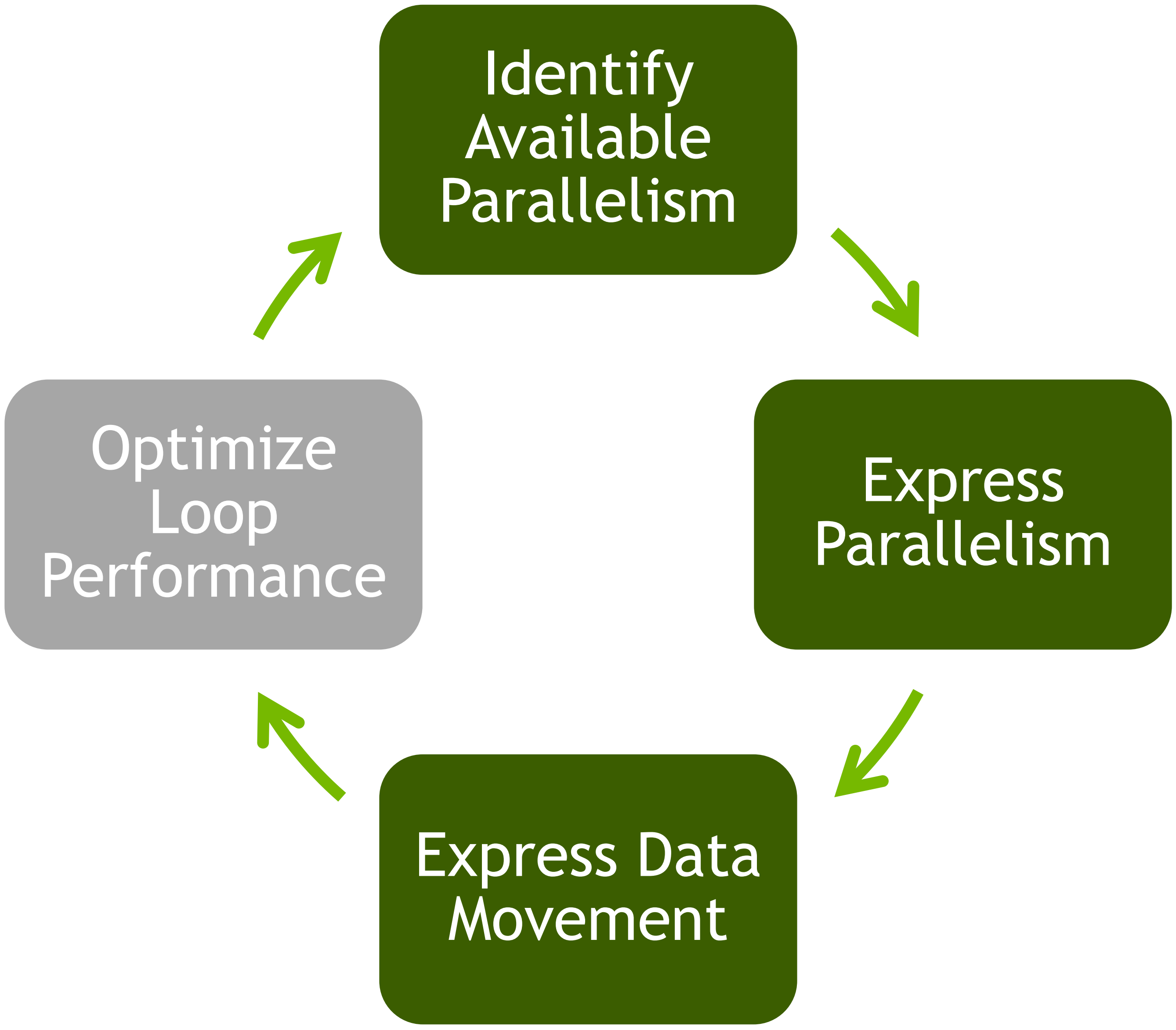
Profile result of parallel loop after defined data region

```
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

Recap Last Course and Questions

Question(1): Synchronous VS Asynchronous

CUDA VS OpenACC

CUDA : Kernel launch is asynchronous

- Which means the control returns to the CPU thread before the device completes the execution of the kernel

OpenACC: Kernel launch is synchronous or asynchronous?

- Use clause **async**() to implement asynchronous
- Often paired with **wait**() clause
- More details please refer to OpenACC specifications

Question(2): Parallelize nested loop Case4

Example:

- A 2-layer nested loop
- Use parallel loop to parallelize it

The loop directives can be nested

- loop:identifies a loop that should be distributed across threads
- parallel & loop are often placed together

```
#include <stdio.h>
#include <stdlib.h>
#define M   (1<<8)
#define N   (1<<8)

int main() {

    int i, j, s;
    float *restrict x = (float*)malloc(M*N*sizeof(float));

    for (j=0; j<N; j++)
    {
        int indexBegin = j*M;

        for (i=0; i<M; i++)
        {
            x[i + indexBegin] = 1.0;
        }
    }

    printf("x[0] = %f, x[last] = %f\n", x[0], x[M*N-1]);

    free(x);

    return 0;
}
```

Question(3): Parallelize loop with function or subroutine

Before OpenACC 2.0

- Inline all functions within parallel region, OR
- Not parallelize loops containing function calls at all

With OpenACC 2.0 and beyond

- Use **routine** directive to give compiler the necessary information about the function and its loops
- Add at the function definition informing the compiler of the level of parallelism
- Example case5

Question(4): Can we share CUDA context between CUDA and NV Codec, and How?

The answer is YES! We can share CUDA context between CUDA and NV Codec

- Use cuCtxGetCurrent to get the CUDA primary context
- cuCtxPushCurrent: Binds the specified CUDA context to the calling CPU thread
- cuCtxPopCurrent: Pops the current CUDA context from the current CPU thread

(CUDA) Context

- A CUDA context is analogous to a CPU process

Other questions

OpenACC将来可能支持python?

Don’t know the roadmap. But I guess not!

OpenACC对所有英伟达的 gpu 都支持吗？对tx1支持吗?

No. There is no need to support TX1. Only CUDA C/C++ support TX1.

copyin, copyout操作会增加内存空间的使用吗？ 尤其是数据量特别大的情况下?

I don’t think so since copyin/copyout won’t extra allocate memory. It should be pointed out they will induce memcpy between CPU and GPU, so we should avoid it by explicitly managing memcpy by using data directives and its clauses

OpenACC 中是否有指定使用 shared memory 的 Directives 或者 Clauses

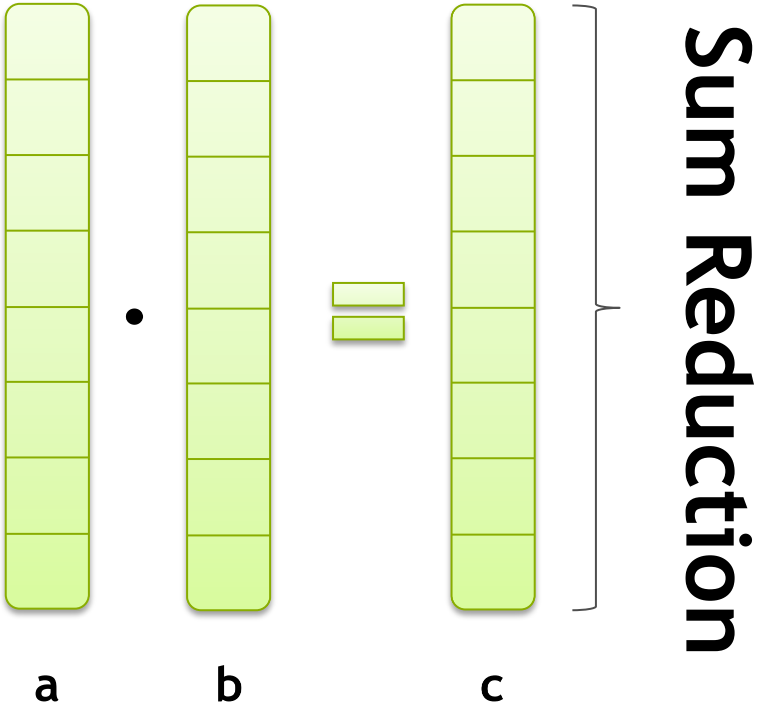
Yes. We can use the cached directives to cache data [on GPU through shared memory]

Case Study(1): Vector Dot Product

Case Study(1): Vector Dot Product

For the sake of parallel implementation, we divide dot product into two sub-steps

- First sub-step is computing the component product and storing the result into array C
- Secondly, reduce the array C



Vector Dot Product: Serial Code

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

#define N (1<<24)

int main()
{
    int i, block;
    double *a, *b, *c;
    struct timeval start;
    struct timeval end;
    double elapsedTime;
    double sum= 0.0f;

    a = (double *)malloc(sizeof(double) * N);
    b = (double *)malloc(sizeof(double) * N);
    c = (double *)malloc(sizeof(double) * N);

    // init a and b
    for (i=0; i<N; i++)
    {
        a[i] = (double)rand()/RAND_MAX;
        b[i] = (double)rand()/RAND_MAX;
    }

    // Step 1: component product
    for (i=0; i<N; i++)
    {
        c[i] = a[i] * b[i];
    }

    // Step 2: reduction
    for (i=0; i<N; i++)
    {
        sum += c[i];
    }

    free(a);
    free(b);
    free(c);

    return 0;
}
```

Step 1: Identify Available Parallelism

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

#define N (1<<24)

int main()
{
    int i, block;
    double *a, *b, *c;
    struct timeval start;
    struct timeval end;
    double elapsedTime;
    double sum= 0.0f;

    a = (double *)malloc(sizeof(double) * N);
    b = (double *)malloc(sizeof(double) * N);
    c = (double *)malloc(sizeof(double) * N);

    // init a and b
    for (i=0; i<N; i++)
    {
        a[i] = (double)rand()/RAND_MAX;
        b[i] = (double)rand()/RAND_MAX;
    }

    // Step 1: component product
    for (i=0; i<N; i++)
    {
        c[i] = a[i] * b[i];
    }

    // Step 2: reduction
    for (i=0; i<N; i++)
    {
        sum += c[i];
    }

    free(a);
    free(b);
    free(c);

    return 0;
}
```

Step 1: Identify Available Parallelism

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

#define N (1<<24)

int main()
{
    int i, block;
    double *a, *b, *c;
    struct timeval start;
    struct timeval end;
    double elapsedTime;
    double sum= 0.0f;

    a = (double *)malloc(sizeof(double) * N);
    b = (double *)malloc(sizeof(double) * N);
    c = (double *)malloc(sizeof(double) * N);

    // init a and b
    for (i=0; i<N; i++)
    {
        a[i] = (double)rand()/RAND_MAX;
        b[i] = (double)rand()/RAND_MAX;
    }

    // Step 1: component product
    for (i=0; i<N; i++)
    {
        c[i] = a[i] * b[i];
    }

    // Step 2: reduction
    for (i=0; i<N; i++)
    {
        sum += c[i];
    }

    free(a);
    free(b);
    free(c);

    return 0;
}
```

1st loop

2nd loop

Step 2: Express the Parallelism by Directives

```
    // Step 1: component product
    for (i=0; i<N; i++)
    {
        c[i] = a[i] * b[i];
    }

    // Step 2: reduction
    for (i=0; i<N; i++)
    {
        sum += c[i];
    }

    free(a);
    free(b);
    free(c);

    return 0;
}
```

Step 2: Express the Parallelism by Directives

```c
    // Step 1: component product
    for (i=0; i<N; i++)
    {
        c[i] = a[i] * b[i];
    }

    // Step 2: reduction
    for (i=0; i<N; i++)
    {
        sum += c[i];
    }

    free(a);
    free(b);
    free(c);

    return 0;
}
```

```c
    // Step 1: component multiply
#pragma acc kernels
#pragma acc loop independent
    for (i=0; i<N; i++)
    {
        c[i] = a[i] * b[i];
    }

    // Step 2: reduction
#pragma acc kernels
    for (i=0; i<N; i++)
    {
        sum += c[i];
    }

    free(a);
    free(b);
    free(c);

    return 0;
}
```

Step 2: Express the Parallelism by Directives

```
26      // Step 1: component multiply
27  #pragma acc kernels
28  #pragma acc loop independent
29      for (i=0; i<N; i++)
30      {
31          c[i] = a[i] * b[i];
32      }
33
34      // Step 2: reduction
35  #pragma acc kernels
36      for (i=0; i<N; i++)
37      {
38          sum += c[i];
39      }
40
41      free(a);
42      free(b);
43      free(c);
44
45      return 0;
46  }
```

执行时间：310.43 ms

```
grid: [1]  block: [256]
         device time(us): total=102 max=102 min=102 avg=102
        elapsed time(us): total=118 max=118 min=118 avg=118
35: data region reached 2 times
    35: data copyin transfers: 8
         device time(us): total=22,230 max=2,794 min=2,775
avg=2,778
    35: data copyout transfers: 9
         device time(us): total=20,392 max=2,564 min=14
avg=2,265
41: data region reached 1 time
```

Step 3: Express Data Movement

```
    // Step 1: component multiply
    #pragma acc kernels
    #pragma acc loop independent
    for (i=0; i<N; i++)
    {
        c[i] = a[i] * b[i];
    }

    // Step 2: reduction
    #pragma acc kernels
    for (i=0; i<N; i++)
    {
        sum += c[i];
    }

    free(a);
    free(b);
    free(c);

    return 0;
}
```

```
#pragma acc data copyin(a[0:N], b[0:N]), create(c[0:N])
{
    // Step 1: component multiply
    #pragma acc kernels
    #pragma acc loop independent
    for (i=0; i<N; i++)
    {
        c[i] = a[i] * b[i];
    }

    // Step 2: reduction
    #pragma acc kernels
    for (i=0; i<N; i++)
    {
        sum += c[i];
    }
}

  free(a);
  free(b);
  free(c);

  return 0;
}
```

Step 3: Express Data Movement

```
26   #pragma acc data copyin(a[0:N], b[0:N]), create(c[0:N])
27   {
28        // Step 1: component multiply
29        #pragma acc kernels
30        #pragma acc loop independent
31        for (i=0; i<N; i++)
32        {
33             c[i] = a[i] * b[i];
34        }
35
36        // Step 2: reduction
37        #pragma acc kernels
38        for (i=0; i<N; i++)
39        {
40             sum += c[i];
41        }
42   }
43
44      free(a);
45      free(b);
46      free(c);
47
48      return 0;
49  }
```

执行时间：230.12 ms

```
26: data region reached 1 time
    26: data copyin transfers: 16
         device time(us): total=44,550 max=2,801 min=2,779 avg=2,784
29: compute region reached 1 time
    31: kernel launched 1 time
        grid: [65535]  block: [128]
         device time(us): total=5,367 max=5,367 min=5,367 avg=5,367
        elapsed time(us): total=5,402 max=5,402 min=5,402 avg=5,402
37: compute region reached 1 time
    38: kernel launched 1 time
        grid: [65535]  block: [128]
         device time(us): total=2,668 max=2,668 min=2,668 avg=2,668
        elapsed time(us): total=2,687 max=2,687 min=2,687 avg=2,687
    38: reduction kernel launched 1 time
        grid: [1]  block: [256]
         device time(us): total=102 max=102 min=102 avg=102
        elapsed time(us): total=119 max=119 min=119 avg=119
44: data region reached 1 time
```

Step 3: Express Data Movement

```
26   #pragma acc data copyin(a[0:N], b[0:N]), create(c[0:N])
27   {
28          // Step 1: component multiply
29          #pragma acc kernels
30          #pragma acc loop independent
31          for (i=0; i<N; i++)
32          {
33                 c[i] = a[i] * b[i];
34          }
35
36          // Step 2: reduction
37          #pragma acc kernels
38          for (i=0; i<N; i++)
39          {
40                 sum += c[i];
41          }
42   }
43
44       free(a);
45       free(b);
46       free(c);
47
48       return 0;
49   }
```

执行时间： 230.12 ms

```
26: data region reached 1 time
    26: data copyin transfers: 16
         device time(us): total=44,550 max=2,801 min=2,779 avg=2,784
29: compute region reached 1 time
    31: kernel launched 1 time
        grid: [65535]  block: [128]
         device time(us): total=5,367 max=5,367 min=5,367 avg=5,367
        elapsed time(us): total=5,402 max=5,402 min=5,402 avg=5,402
37: compute region reached 1 time
    38: kernel launched 1 time
        grid: [65535]  block: [128]
         device time(us): total=2,668 max=2,668 min=2,668 avg=2,668
        elapsed time(us): total=2,687 max=2,687 min=2,687 avg=2,687
    38: reduction kernel launched 1 time
        grid: [1]  block: [256]
         device time(us): total=102 max=102 min=102 avg=102
        elapsed time(us): total=119 max=119 min=119 avg=119
44: data region reached 1 time
```

Data copy occurs only at the entrance of data region

Parallelize the case with parallel loop

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

#define N (1<<24)

int main()
{
    int i, block;
    double *a, *b, *c;
    struct timeval start;
    struct timeval end;
    double elapsedTime;
    double sum= 0.0f;

    a = (double *)malloc(sizeof(double) * N);
    b = (double *)malloc(sizeof(double) * N);
    c = (double *)malloc(sizeof(double) * N);

    // init a and b
    for (i=0; i<N; i++)
    {
        a[i] = (double)rand()/RAND_MAX;
        b[i] = (double)rand()/RAND_MAX;
    }

    // Step 1: component product
    for (i=0; i<N; i++)
    {
        c[i] = a[i] * b[i];
    }

    // Step 2: reduction
    for (i=0; i<N; i++)
    {
        sum += c[i];
    }

    free(a);
    free(b);
    free(c);

    return 0;
}
```

Step 1: Identify Available Parallelism

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

#define N (1<<24)

int main()
{
    int i, block;
    double *a, *b, *c;
    struct timeval start;
    struct timeval end;
    double elapsedTime;
    double sum= 0.0f;

    a = (double *)malloc(sizeof(double) * N);
    b = (double *)malloc(sizeof(double) * N);
    c = (double *)malloc(sizeof(double) * N);

    // init a and b
    for (i=0; i<N; i++)
    {
        a[i] = (double)rand()/RAND_MAX;
        b[i] = (double)rand()/RAND_MAX;
    }

    // Step 1: component product
    for (i=0; i<N; i++)
    {
        c[i] = a[i] * b[i];
    }

    // Step 2: reduction
    for (i=0; i<N; i++)
    {
        sum += c[i];
    }

    free(a);
    free(b);
    free(c);

    return 0;
}
```

1st loop

2nd loop

Step 2: Express the Parallelism with parallel loop

kernels

```
    // Step 1: component multiply
#pragma acc kernels
#pragma acc loop independent
    for (i=0; i<N; i++)
    {
        c[i] = a[i] * b[i];
    }

    // Step 2: reduction
#pragma acc kernels
    for (i=0; i<N; i++)
    {
        sum += c[i];
    }

    free(a);
    free(b);
    free(c);

    return 0;
}
```

parallel loop

```
    // Step 1: component multiply
#pragma acc parallel loop
    for (i=0; i<N; i++)
    {
        c[i] = a[i] * b[i];
    }

    // Step 2: reduction
#pragma acc parallel
#pragma acc loop reduction(+:sum)
    for (i=0; i<N; i++)
    {
        sum += c[i];
    }

    free(a);
    free(b);
    free(c);

    return 0;
}
```

Step 3: Express Data Movement

```
        // Step 1: component multiply
 #pragma acc parallel loop
        for (i=0; i<N; i++)
        {
            c[i] = a[i] * b[i];
        }

        // Step 2: reduction
#pragma acc parallel
#pragma acc loop reduction(+:sum)
        for (i=0; i<N; i++)
        {
            sum += c[i];
        }

        free(a);
        free(b);
        free(c);

        return 0;
    }
```

```
 #pragma acc data copyin(a[0:N], b[0:N]), create(c[0:N])
 {
        // Step 1: component multiply
#pragma acc parallel loop
        for (i=0; i<N; i++)
        {
            c[i] = a[i] * b[i];
        }

        // Step 2: reduction
#pragma acc parallel
#pragma acc loop reduction(+:sum)
        for (i=0; i<N; i++)
        {
            sum += c[i];
        }
 }

      free(a);
      free(b);
      free(c);

      return 0;
}
```

OpenACC VS CUDA

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

#define N (1<<24)
#define blocksize 1024
#define blocknumb (N/blocksize)

#define checkCudaAPIErrors(F) if ((F) != cudaSuccess) \
{ printf("Error at line %d in file %s: %s\n", __LINE__, __FILE__, \
       cudaGetErrorString(cudaGetLastError())); exit(-1); }

__global__ void vecDot(double *a, double *b, double *sub_sum)
{
    int gid = blockDim.x * blockIdx.x + threadIdx.x;
    __shared__ double component[blocksize];

    component[threadIdx.x] = a[gid] * b[gid];

    __syncthreads();
    for (int i=(blocksize>>1); i>0; i=(i>>1))
    {
        if (threadIdx.x < i)
            component[threadIdx.x] += component[threadIdx.x + i];
        __syncthreads();
    }

    if (threadIdx.x == 0)
    {
        sub_sum[blockIdx.x] = component[0];
    }
}
```

```
int main()
{
    int i, device = 0;
    double *h_a, *h_b, *h_c;
    double *d_a, *d_b, *d_c;
    double *h_subSum;
    double *d_subSum;

    struct timeval start;
    struct timeval end;
    double elapsedTime;
    double sum_cpu = 0.0;
    double sum_gpu = 0.0;
    cudaDeviceProp prop;

    h_a = (double *)malloc(sizeof(double) * N);
    h_b = (double *)malloc(sizeof(double) * N);
    h_c = (double *)malloc(sizeof(double) * N);
    h_subSum = (double *)malloc(sizeof(double) * blocknumb);


    // init a and b
    for (i=0; i<N; i++)
    {
        h_a[i] = (double)rand()/RAND_MAX;
        h_b[i] = (double)rand()/RAND_MAX;
        h_c[i] = h_a[i] * h_b[i];

        sum_cpu += h_c[i];
    }

    cudaSetDevice(device);
    cudaGetDeviceProperties(&prop, device);
    printf("Using gpu %d: %s\n", device, prop.name);

```

```
    // timer begin
    gettimeofday(&start, NULL);

    cudaMalloc((void**)&d_a, sizeof(double) * N);
    cudaMalloc((void**)&d_b, sizeof(double) * N);
    cudaMalloc((void**)&d_c, sizeof(double) * N);
    cudaMalloc((void**)&d_subSum, sizeof(double) * blocknumb);

    checkCudaAPIErrors(cudaMemcpy(d_a, h_a, sizeof(double) * N, cudaMemcpyHostToDevice));
    checkCudaAPIErrors(cudaMemcpy(d_b, h_b, sizeof(double) * N, cudaMemcpyHostToDevice));

    vecDot<<<blocknumb, blocksize>>>(d_a, d_b, d_subSum);
    checkCudaAPIErrors(cudaMemcpy(h_subSum, d_subSum, sizeof(double) * blocknumb, cudaMemcpyDeviceToHost));

    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    cudaFree(d_subSum);

    for (i=0; i<blocknumb; i++)
    {
        sum_gpu += h_subSum[i];
    }
    // timer end
    gettimeofday(&end, NULL);

    elapsedTime  = (end.tv_sec - start.tv_sec) * 1000.0;    // sec to ms
    elapsedTime += (end.tv_usec - start.tv_usec) / 1000.0;  // us  to ms

    printf("the result on GPU is %lf\n", sum_gpu);
    printf("the result on CPU is %lf\n", sum_cpu);
    printf("the elapsedTime is %f ms\n", elapsedTime);

    free(h_a);
    free(h_b);
    free(h_c);
```

OpenACC VS CUDA

Performance

- For the exstramely simple case, the performance between OpenACC and CUDA is comparable
  - CUDA: 184.01 ms
  - OpenACC: 230.12 ms

Workload

- CUDA >> OpenACC
  - Almost rewrite the code with CUDA
  - Add a few lines by OpenACC

Case Study(2): Lattice Boltzmann Method (LBM)

Introduction to LBM

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

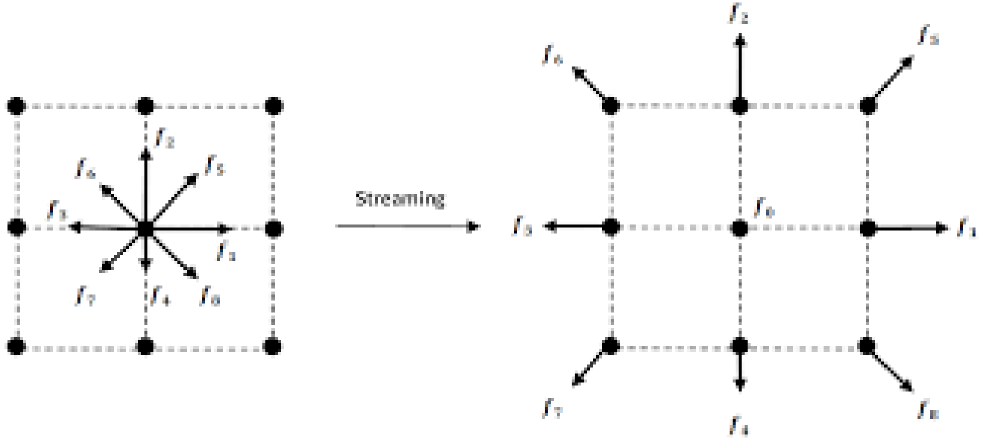
The computation of LBM is divided into two step

- 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

Task Description

Code: 560 lines

Domain size: 64x32x32, D3Q15

Time step: 5000

CPU Time: 100 685.18 ms

Accelerate the code with OpenACC

Step 1: Find the hotspot

- the main loop (t_max = 5000)

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

Accelerate the code with OpenACC

Step 2: Parallelize the two functions in the loop: 3 lines are added

```
void propagete()
{
      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++)
                  {
                        if (FLUID==flag[z][y][x])
                        {
                              k=0;
                              xp=x-e[k][0];
                              yp=y-e[k][1];
                              zp=z-e[k][2];
                              f0temp[z][y][x]=f0[zp][yp][xp];
```

Accelerate the code with OpenACC

Step 2: Parallelize the two functions in the loop: 4 lines are added

```
void collision()
{
      int x, y, z;
      double rho,vx,vy,vz;
      double
f_eq0,f_eq1,f_eq2,f_eq3,f_eq4,f_eq5,f_eq6,f_eq7,f_eq8,f_eq9,f_eq10,f_eq11,f_eq12,f_eq13,f_eq14;
      double square, tau_inv, dummy, product;

      tau_inv=1.0/tau;

#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++)
                  {
                        f0[z][y][x]=f0temp[z][y][x];f1[z][y][x]=f1temp[z][y][x];f2[z][y][x]=f2temp[z][y][x];
                        f3[z][y][x]=f3temp[z][y][x];f4[z][y][x]=f4temp[z][y][x];f5[z][y][x]=f5temp[z][y][x];
                        f6[z][y][x]=f6temp[z][y][x];f7[z][y][x]=f7temp[z][y][x];f8[z][y][x]=f8temp[z][y][x];
```

Accelerate the code with OpenACC

Step 3: Manage date movement: 3 lines are 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++)
       {
           propagete();
           collision();
           if (t%hop==0)
           {
               cout<<"timestep = "<<t<<endl;
           }
       }
   }
```

Accelerate the code with OpenACC

In summary, 14 lines are added

Performance 4891.01 ms, so 20.58X speedup on K40

Accelerate the code with CUDA

Write two kernels: 353 lines code

Performance 1455.74 ms, so 3.36X speedup compared with OpenACC parallelization on K40

Why?

Why CUDA is much faster than OpenACC

Data Communication

- The date transferred is the same
- The transfer speed is almost the same ~10.0 GB/s

Kernels Execution

- There are two kernels

	OpenACC (us)	CUDA (us)	Speedup
propagate	500	143	3.42
collision	360	112	3.21

Why CUDA is much faster than OpenACC

Kernels Configuration

- Block and Grid

	OpenACC	CUDA
propagate	(128, 1, 1)/(**32**, 1, 1)	(32, 16, 1)/(2, 2, 32)
collision	(128, 1, 1)/(**32**, 1, 1)	(32, 16, 1)/(2, 2, 32)

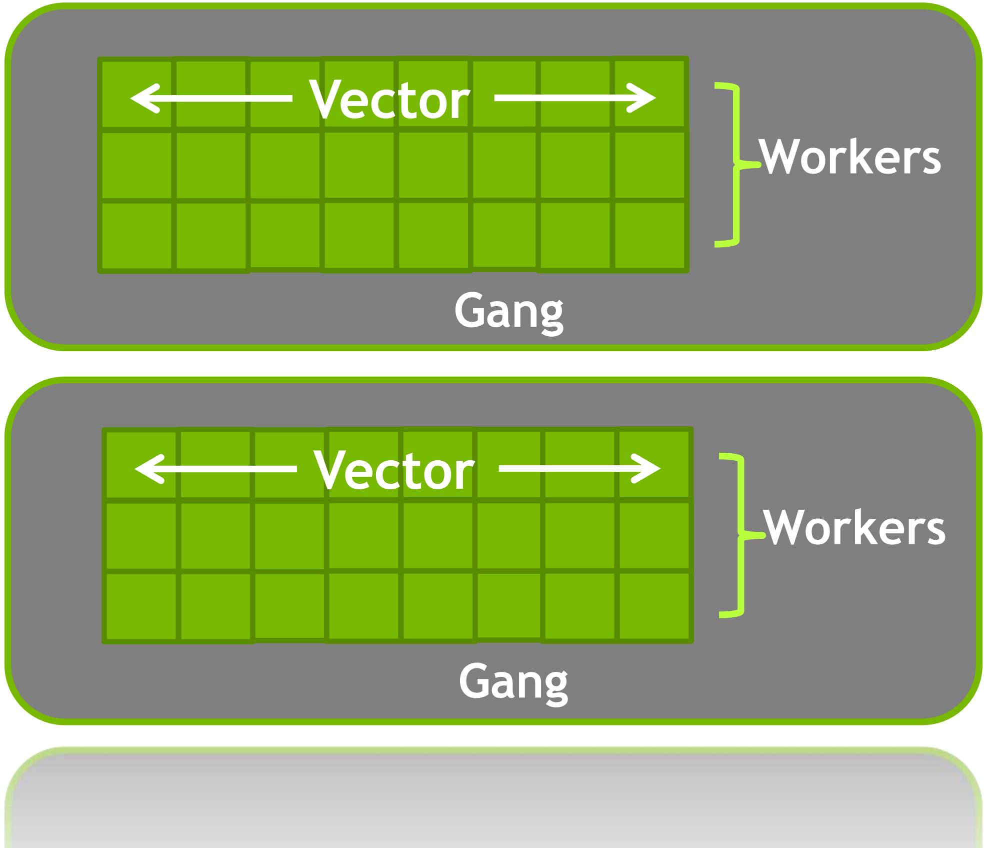
- Registers used by each kernel and occupancy achieved

	OpenACC	CUDA (us)
propagate	96/**6.5%**	48/**45%**
collision	151/**6.6%**	110/23.4%

- Too few threads to fully utilize the device

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 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

Optimize the kernels generated by OpenACC

Step 4: Optimize the loop on the specified device (K40)

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

#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) gang worker num_worker(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++)
                  {
                        if (FLUID==flag[z][y][x])
                        {
                              k=0;
                              xp=x-e[k][0];
                              yp=y-e[k][1];
                              zp=z-e[k][2];
                              f0temp[z][y][x]=f0[zp][yp][xp];
```

Optimize the kernels generated by OpenACC

Step 4: Optimize the loop on the specified device (K40)

```
void collision()
{
      int x, y, z;
      double rho,vx,vy,vz;
      double
f_eq0,f_eq1,f_eq2,f_eq3,f_eq4,f_eq5,f_eq6,f_eq7,f_eq8,f_eq9,f_eq10,f_eq11,f_eq12,f_eq13,f_eq14;
      double square, tau_inv, dummy, product;

      tau_inv=1.0/tau;

#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) gang worker num_worker(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++)
                  {
                        f0[z][y][x]=f0temp[z][y][x];f1[z][y][x]=f1temp[z][y][x];f2[z][y][x]=f2temp[z][y][x];
                        f3[z][y][x]=f3temp[z][y][x];f4[z][y][x]=f4temp[z][y][x];f5[z][y][x]=f5temp[z][y][x];
                        f6[z][y][x]=f6temp[z][y][x];f7[z][y][x]=f7temp[z][y][x];f8[z][y][x]=f8temp[z][y][x];
```

Optimize the kernels generated by OpenACC

Performance

- After optimization: 2668.80 ms, 4891.01 / 2668.80 = 1.83X
- Compared with CUDA: 2668.80 / 1455.74 = 1.83X
- Block and Grid

	OpenACC	CUDA
propagate	(64, 8, 1)/(32, 1, 1)	(32, 16, 1)/(2, 2, 32)
collision	(64, 8, 1)/(32, 1, 1)	(32, 16, 1)/(2, 2, 32)

- Registers used by each kernel and occupancy achieved

	OpenACC	CUDA (us)
propagate	95/24 %	48/45%
collision	128/24%	110/23.4%

- Still need more blocks to fully utilize the device. Use collapse to expend loops

Optimize the kernels generated by OpenACC

Use collapse to expend loops. And use the same method to optimize the other function

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

#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)        \
#pragma acc loop collapse(3)
      for (z=0;z<Nz;z++)
      {
            for (y=0;y<Ny;y++)
            {
                  for (x=0;x<Nx;x++)
                  {
                        if (FLUID==flag[z][y][x])
                        {
                              k=0;
                              xp=x-e[k][0];
                              yp=y-e[k][1];
                              zp=z-e[k][2];
                              f0temp[z][y][x]=f0[zp][yp][xp];
```

Optimize the kernels generated by OpenACC

Performance

- After optimization step 2: 2331.95 ms, 4891.01 / 2331.95 = 2.10X Speedup
- Compared with CUDA: 2331.95 / 1455.74 = 1.6X Slowdown
- Block and Grid

	OpenACC	CUDA
propagate	(128, 1, 1)/(**512**, 1, 1)	(32, 16, 1)/(2, 2, 32)
collision	(128, 1, 1)/(**512**, 1, 1)	(32, 16, 1)/(2, 2, 32)

- Now, each thread deals with one node, and result in better performance.
- Registers used by each kernel and occupancy achieved

	OpenACC	CUDA (us)
propagate	**86**/28.5 %	48/45%
collision	**149**/17.6%	110/23.4%

Summary

Summary(1)

Why OpenACC?

- Open, Simple and Portable

Accelerated Computation

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

OpenACC Programming Cycle

- Only cover three stages
    - Parallelism analysis
    - Express parallelism to compiler with directive
    - Define data region to eliminate unnecessary data copy

Summary(1)

OpenACC Directive

- 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
- data and its clauses
  - Define data region to eliminate unnecessary data copy for the sake of performance.
  - clauses: copy, copyin, copyout, create et al.

Summary(2)

OpenACC Case Study

- **Some other clauses of loop**
    - Reduction reduction(operation : var)
    - Collapse
- **4 parallelism levels**
    - Gang
    - Worker
    - Vector
    - seq

Thanks for Your Attention!