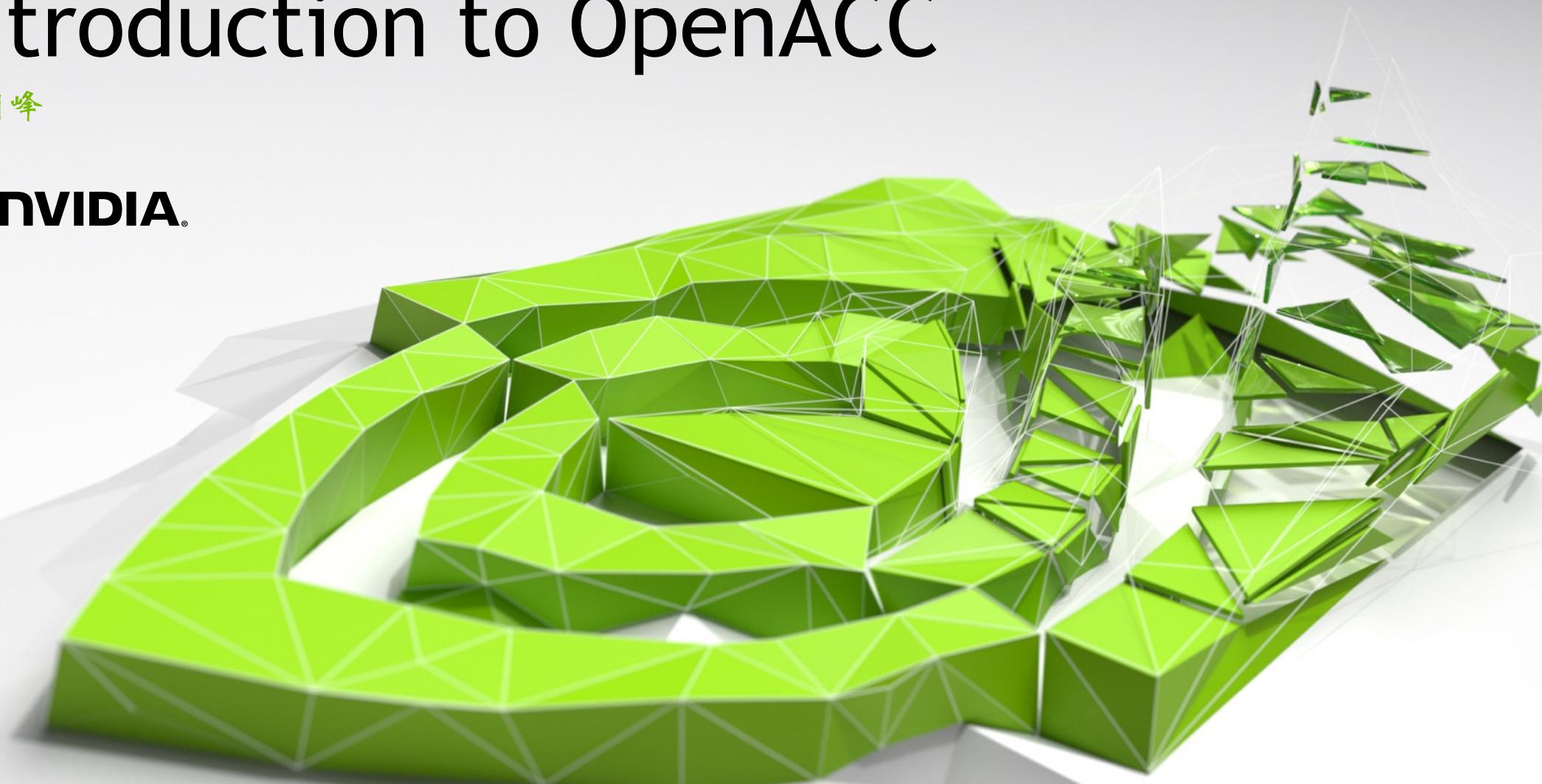


Introduction to OpenACC

周国峰



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

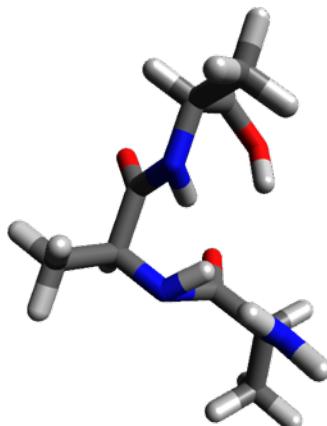


7-8x Speed-Up
5% of Code Modified

8000+
Developers
using OpenACC

LS-DALTON

Large-scale application for calculating high-accuracy molecular energies



“

OpenACC makes GPU computing approachable for domain scientists. Initial OpenACC implementation required only minor effort, and more importantly, no modifications of our existing CPU implementation.

”

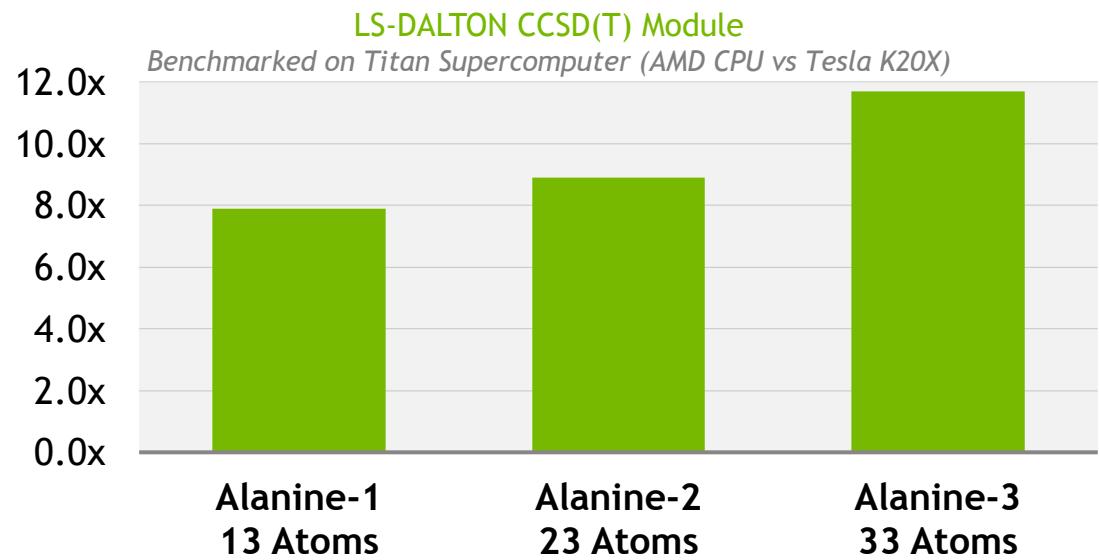
Janus Juul Eriksen, PhD Fellow
qLEAP Center for Theoretical Chemistry, Aarhus University



Minimal Effort

Lines of Code Modified	# of Weeks Required	# of Codes to Maintain
<100 Lines	1 Week	1 Source

Big Performance



OpenACC Directives

```
Manage Data Movement → #pragma acc data copyin(a,b) copyout(c)  
 {  
 ...  
 Initiate Parallel Execution → #pragma acc parallel  
 {  
 #pragma acc loop gang vector  
 for (i = 0; i < n; ++i) {  
 z[i] = x[i] + y[i];  
 ...  
 } }  
 Optimize Loop Mappings → } ...  
 }
```



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

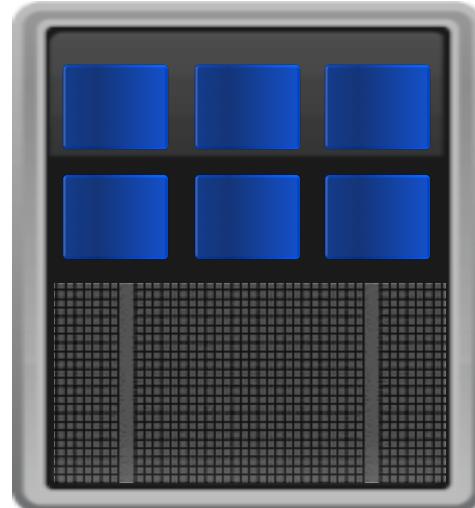
Accelerated Computing Fundamentals

Accelerated Computing

10x Performance & 5x Energy Efficiency for HPC

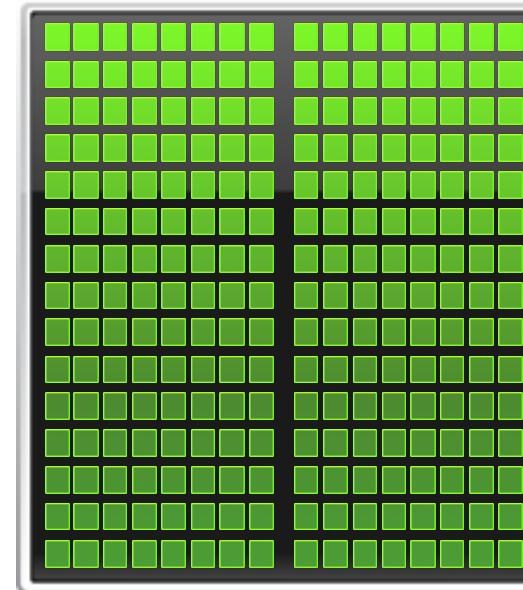
CPU

Optimized for
Serial Tasks

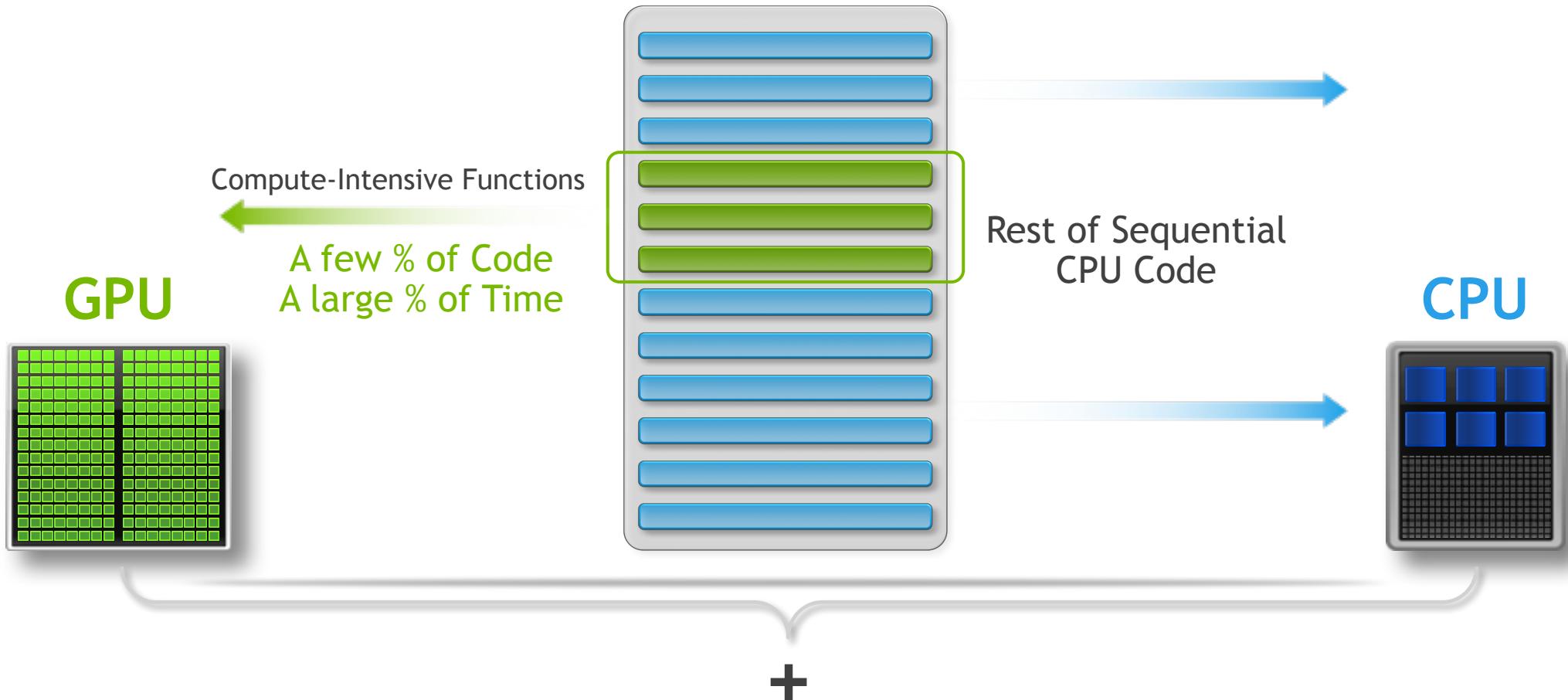


GPU Accelerator

Optimized for
Parallel Tasks

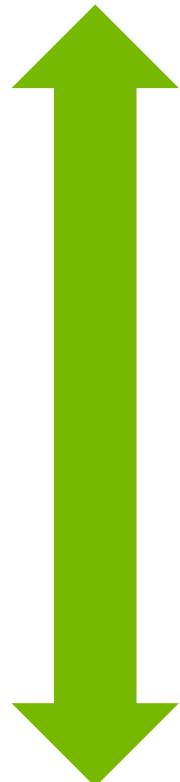


What is Heterogeneous Programming?



Portability & Performance

Portability



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

Performance

Code for Portability & Performance

Libraries

- Implement as much as possible using portable libraries

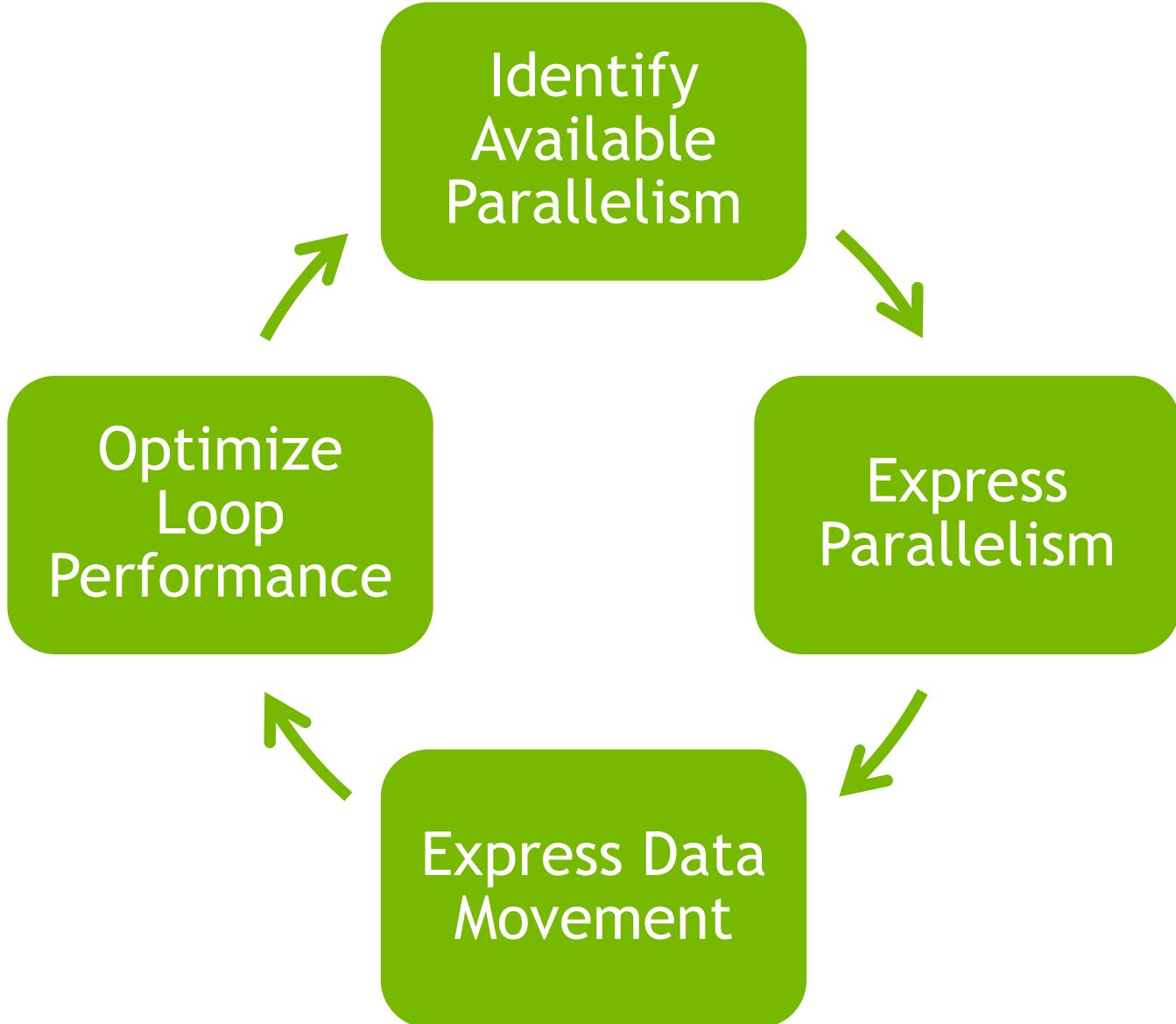
Directives

- Use directives for rapid and portable development

Languages

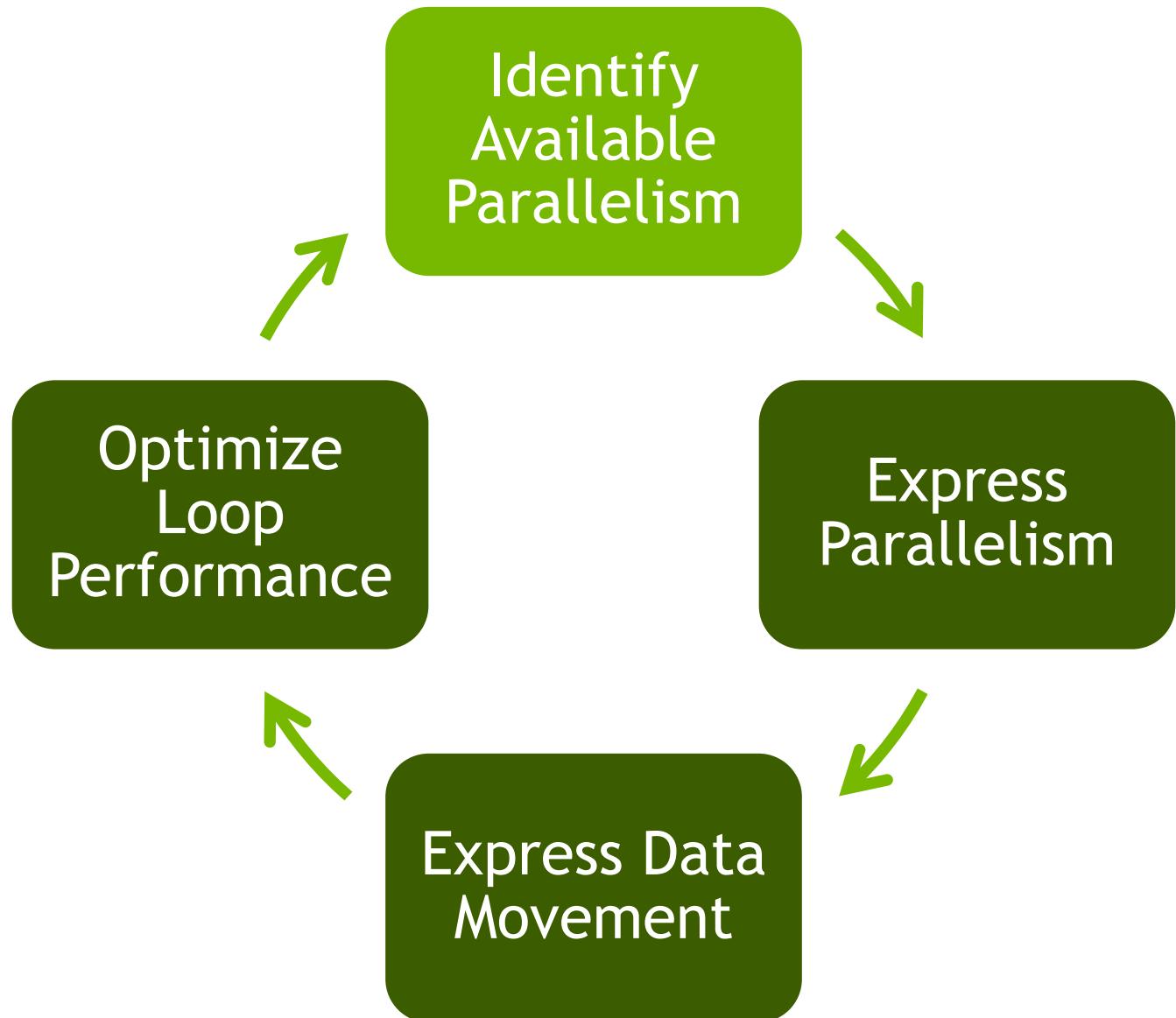
- Use lower level languages for important kernels

OpenACC Programming Cycle



A Simple Example

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define N      (1<<20)
5
6 int main() {
7     int i;
8     int a[N]={0};
9
10    a[0] = 1;
11
12    printf("a[0] = %d\n", a[0]);
13
14    for (i=0; i<N; i++)
15    {
16        a[i] = a[i]+1;
17    }
18
19    printf("a[0] = %d\n", a[0]);
20
21    return 0;
22 }
```

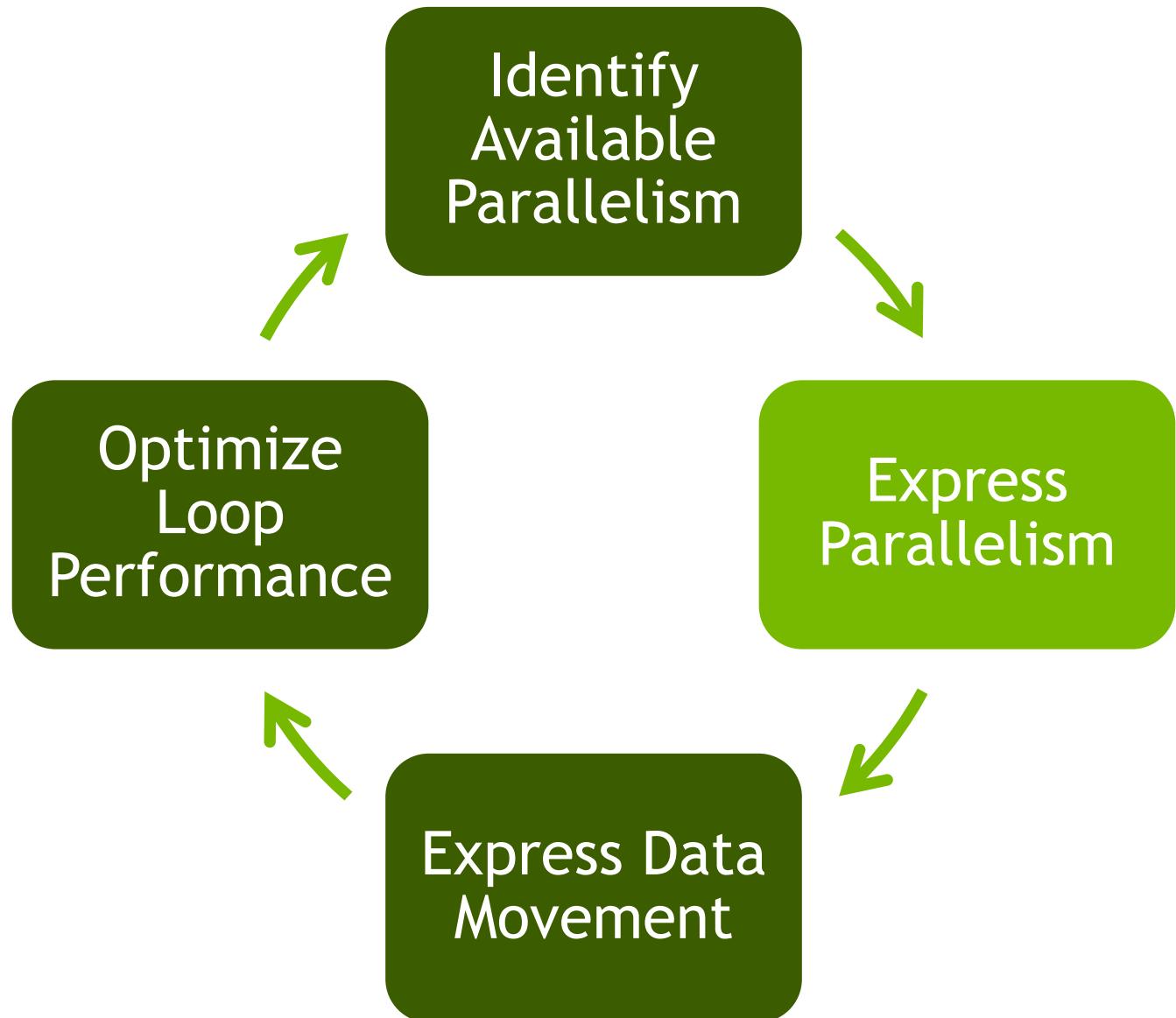


A Simple Example

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define N      (1<<20)
5
6 int main() {
7     int i;
8     int a[N]={0};
9
10    a[0] = 1;
11
12    printf("a[0] = %d\n", a[0]);
13
14    for (i=0; i<N; i++)
15    {
16        a[i] = a[i]+1;
17    }
18
19    printf("a[0] = %d\n", a[0]);
20
21    return 0;
22 }
```



The loop is parallelizable



OpenACC **kernels** Directive

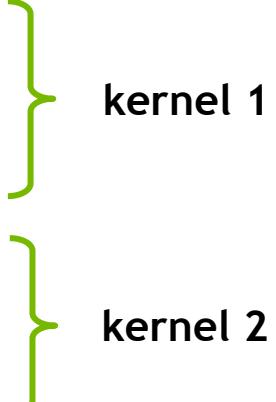
The kernels directive identifies a region that may contain *loops* that the compiler can turn into parallel *kernels*.

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;
    }
}
```



The code snippet shows two nested loops. The first loop is grouped by a brace under the **kernels** directive, labeled "kernel 1". The second loop is also grouped by a brace under the same **kernels** directive, labeled "kernel 2".

The compiler identifies
2 parallel loops and
generates 2 kernels.

A Simple Example

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define N      (1<<20)
5
6 int main() {
7     int i;
8     int a[N]={0};
9
10    a[0] = 1;
11
12    printf("a[0] = %d\n", a[0]);
13
14    for (i=0; i<N; i++)
15    {
16        a[i] = a[i]+1;
17    }
18
19    printf("a[0] = %d\n", a[0]);
20
21    return 0;
22 }
```

A Simple Example

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define N      (1<<20)
5
6 int main() {
7     int i;
8     int a[N]={0};
9
10    a[0] = 1;
11
12    printf("a[0] = %d\n", a[0]);
13
14    #pragma acc kernels
15        for (i=0; i<N; i++)
16        {
17            a[i] = a[i]+1;
18        }
19
20    printf("a[0] = %d\n", a[0]);
21
22    return 0;
23 }
```



- 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.

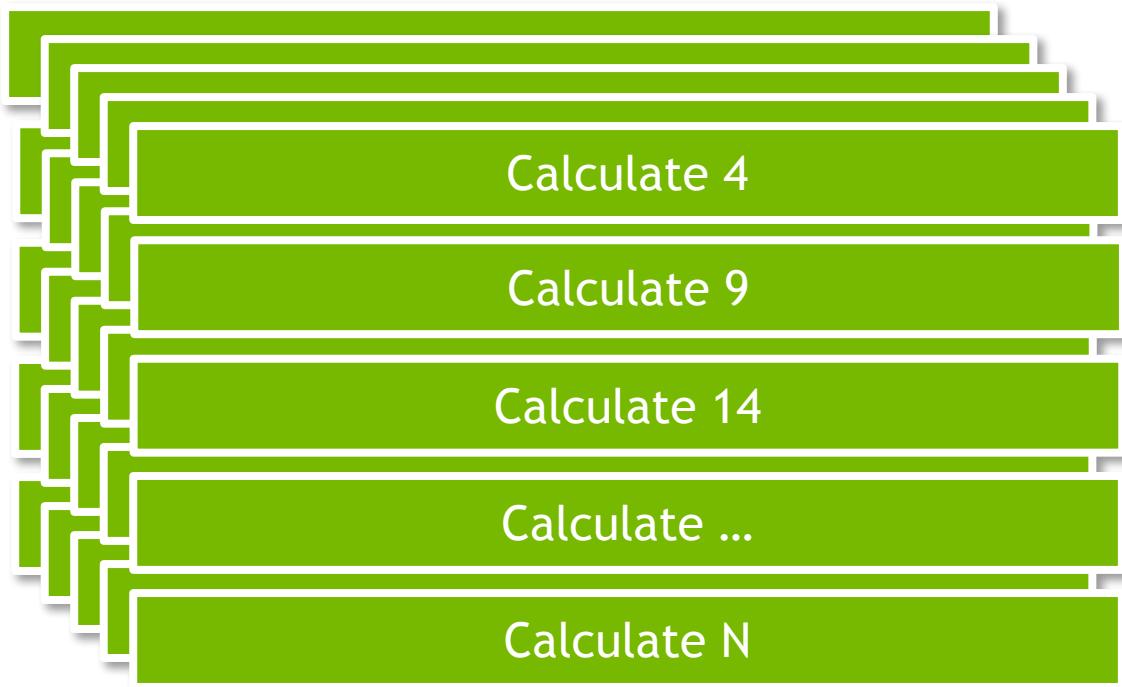
Execution of Serial Loops vs. Parallel Kernels

Calculate 0 - N in order.

Calculate 0

Execution of Serial Loops vs. Parallel Kernels

Calculate 0 - N in order.



What will happen?---Build OpenACC Code

Build the code

- pgcc -acc -Minfo=accel -ta=tesla 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

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define N      (1<<20)
5
6 int main() {
7     int i;
8     int a[N]={0};
9
10    a[0] = 1;
11
12    printf("a[0] = %d\n", a[0]);
13
14    for (i=0; i<N; i++)
15    {
16        a[i] = a[i]+1;
17    }
18
19    printf("a[0] = %d\n", a[0]);
20
21    return 0;
22 }
```

A Simple Example

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define N      (1<<20)
5
6 int main() {
7     int i;
8     int a[N]={0};
9
10    a[0] = 1;
11
12    printf("a[0] = %d\n", a[0]);
13
14    for (i=0; i<N; i++)
15    {
16        a[i] = a[i]+1;
17    }
18
19    printf("a[0] = %d\n", a[0]);
20
21    return 0;
22 }
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define N      (1<<20)
5
6 int main() {
7     int i;
8     int a[N] = {0};
9
10    a[0] = 1;
11
12    printf("a[0] = %d\n", a[0]);
13
14    #pragma acc parallel loop
15        for (i=0; i<N; i++)
16    {
17        a[i] = a[i]+1;
18    }
19
20    printf("a[0] = %d\n", a[0]);
21
22    return 0;
23 }
```

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 compiler 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

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #define N    (1<<20)
4
5 int main() {
6     int i;
7     int a[N]={0};
8
9     a[0] = 1;
10
11    printf("a[0] = %d\n", a[0]);
12
13    for (i=0; i<N; i++)
14    {
15        a[i] = a[i]+1;
16    }
17    for (i=0; i<N; i++)
18    {
19        a[i] = a[i]+1;
20    }
21
22    printf("a[0] = %d\n", a[0]);
23    return 0;
24 }
```

Parallelize with kernels

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #define N    (1<<20)
4
5 int main() {
6     int i;
7     int a[N]={0};
8
9     a[0] = 1;
10
11    printf("a[0] = %d\n", a[0]);
12
13    for (i=0; i<N; i++)
14    {
15        a[i] = a[i]+1;
16    }
17    for (i=0; i<N; i++)
18    {
19        a[i] = a[i]+1;
20    }
21
22    printf("a[0] = %d\n", a[0]);
23    return 0;
24 }
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #define N    (1<<20)
4
5 int main() {
6     int i;
7     int a[N]={0};
8
9     a[0] = 1;
10
11    printf("a[0] = %d\n", a[0]);
12
13 #pragma acc kernels
14 {
15     for (i=0; i<N; i++)
16     {
17         a[i] = a[i]+1;
18     }
19     for (i=0; i<N; i++)
20     {
21         a[i] = a[i]+1;
22     }
23 }
24
25    printf("a[0] = %d\n", a[0]);
26    return 0;
27 }
```

Compiler generates
two kernels for the
region

Parallelize with Parallel loop

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #define N    (1<<20)
4
5 int main() {
6     int i;
7     int a[N]={0};
8
9     a[0] = 1;
10
11    printf("a[0] = %d\n", a[0]);
12
13    for (i=0; i<N; i++)
14    {
15        a[i] = a[i]+1;
16    }
17    for (i=0; i<N; i++)
18    {
19        a[i] = a[i]+1;
20    }
21
22    printf("a[0] = %d\n", a[0]);
23    return 0;
24 }
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #define N    (1<<20)
4
5 int main() {
6     int i;
7     int a[N]={0};
8
9     a[0] = 1;
10
11    printf("a[0] = %d\n", a[0]);
12
13    #pragma acc parallel loop ◀ Kernel 1
14        for (i=0; i<N; i++)
15        {
16            a[i] = a[i]+1;
17        }
18    #pragma acc parallel loop ◀ Kernel 2
19        for (i=0; i<N; i++)
20        {
21            a[i] = a[i]+1;
22        }
23
24    printf("a[0] = %d\n", a[0]);
25    return 0;
26 }
```

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:

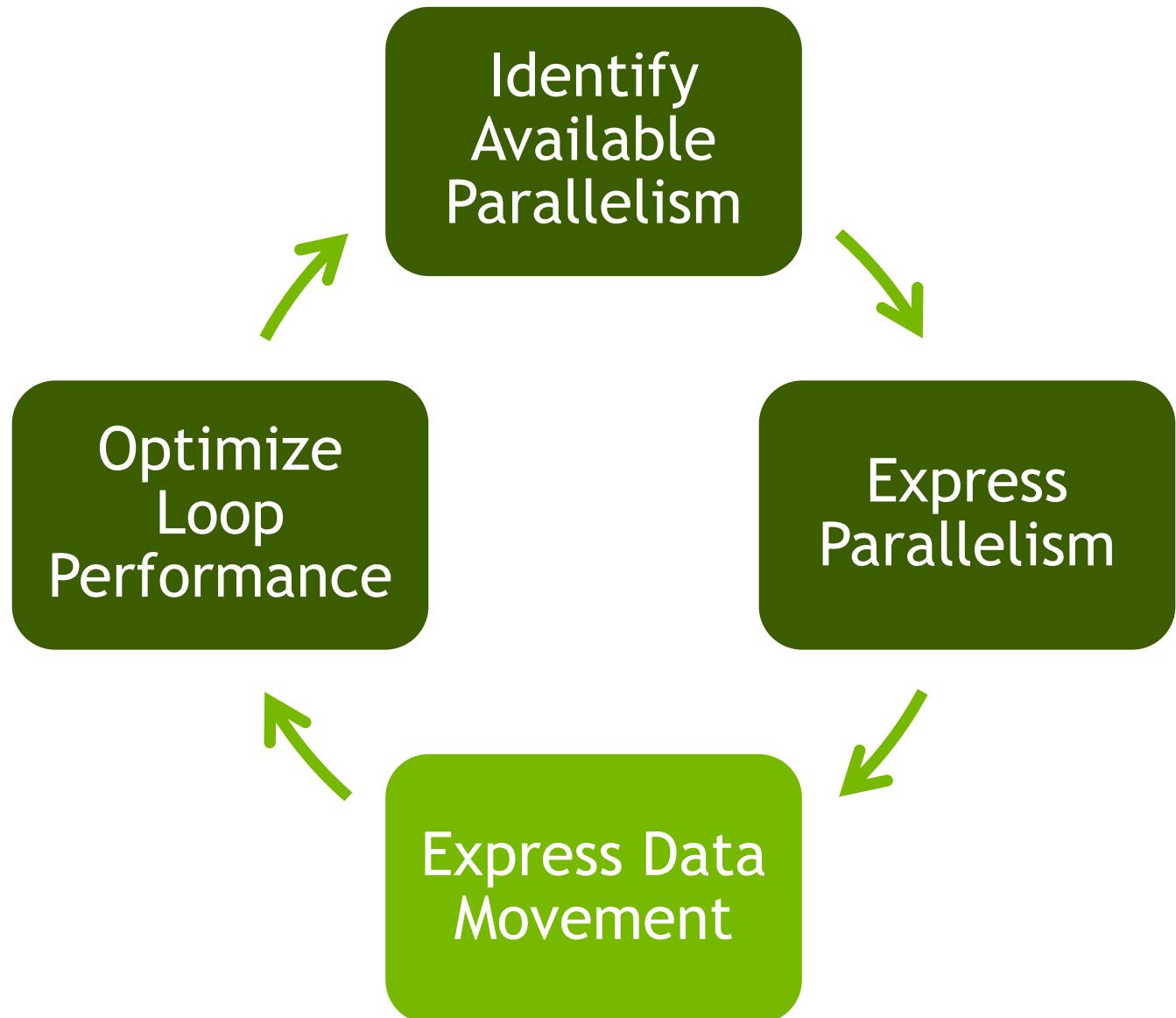
- There is only one pair of copyin (CPU to GPU) and copyout (GPU to CPU)
- Correspond to the single **kernels** directive

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...



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

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #define N      (1<<20)
4
5 int main() {
6     int i;
7     int a[N]={0};
8
9     a[0] = 1;
10
11    printf("a[0] = %d\n", a[0]);
12
13 #pragma acc parallel loop
14     for (i=0; i<N; i++)
15     {
16         a[i] = a[i]+1;
17     }
18 #pragma acc parallel loop
19     for (i=0; i<N; i++)
20     {
21         a[i] = a[i]+1;
22     }
23
24     printf("a[0] = %d\n", a[0]);
25     return 0;
26 }
```

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

Define data region to eliminate unnecessary copy

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #define N      (1<<20)
4
5 int main() {
6     int i;
7     int a[N]={0};
8
9     a[0] = 1;
10
11    printf("a[0] = %d\n", a[0]);
12
13 #pragma acc parallel loop
14     for (i=0; i<N; i++)
15     {
16         a[i] = a[i]+1;
17     }
18 #pragma acc parallel loop
19     for (i=0; i<N; i++)
20     {
21         a[i] = a[i]+1;
22     }
23
24    printf("a[0] = %d\n", a[0]);
25    return 0;
26 }
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #define N      (1<<20)
4
5 int main() {
6     int i;
7     int a[N]={0};
8
9     a[0] = 1;
10
11    printf("a[0] = %d\n", a[0]);
12
13 #pragma acc data copy(a[0:N])
14 {
15 #pragma acc parallel loop
16     for (i=0; i<N; i++)
17     {
18         a[i] = a[i]+1;
19     }
20 #pragma acc parallel loop
21     for (i=0; i<N; i++)
22     {
23         a[i] = a[i]+1;
24     }
25 }
26
27    printf("a[0] = %d\n", a[0]);
28    return 0;
29 }
```

Define data region to eliminate unnecessary copy

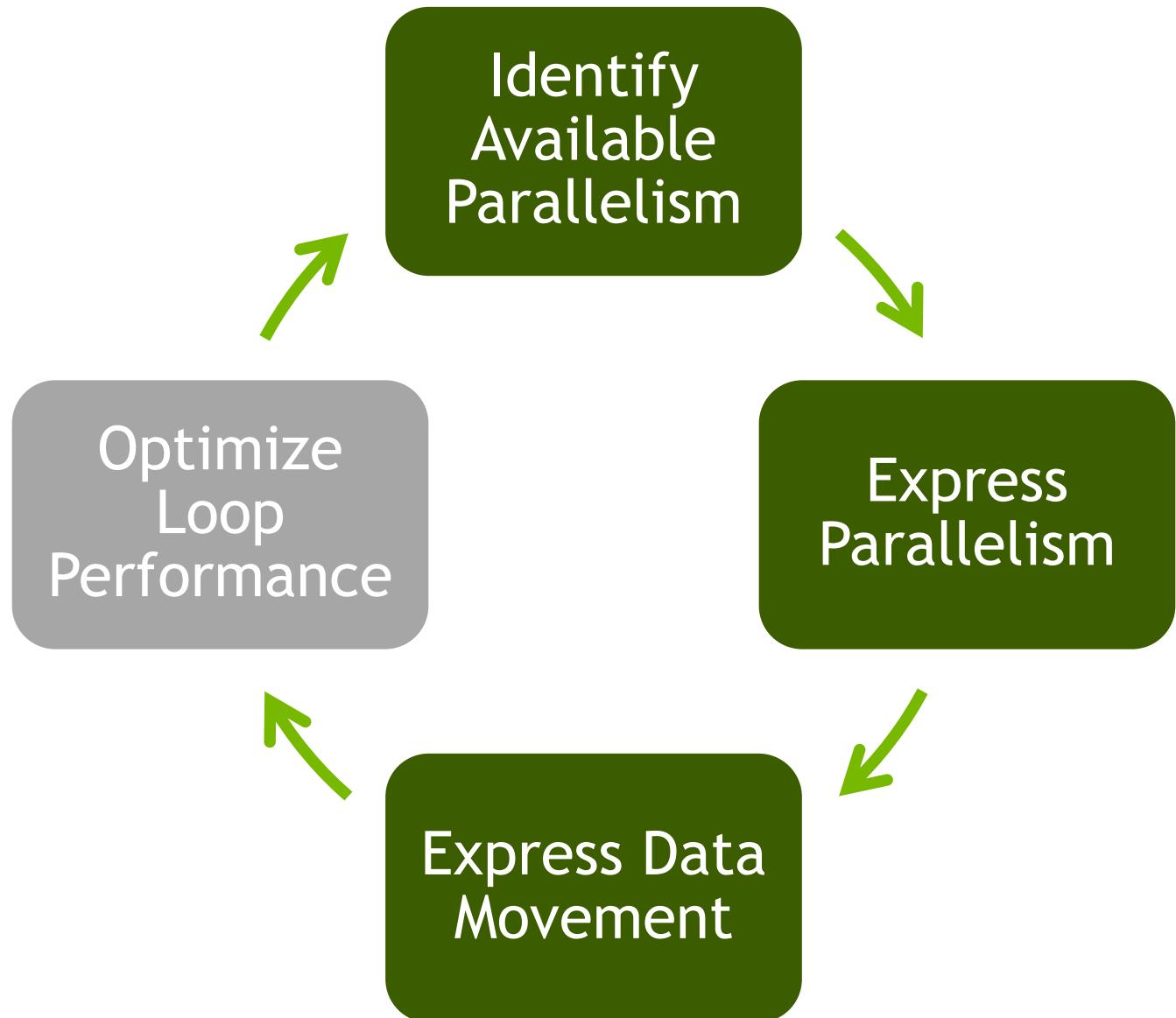
Profile result of **parallel loop** after defined data region

```
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!

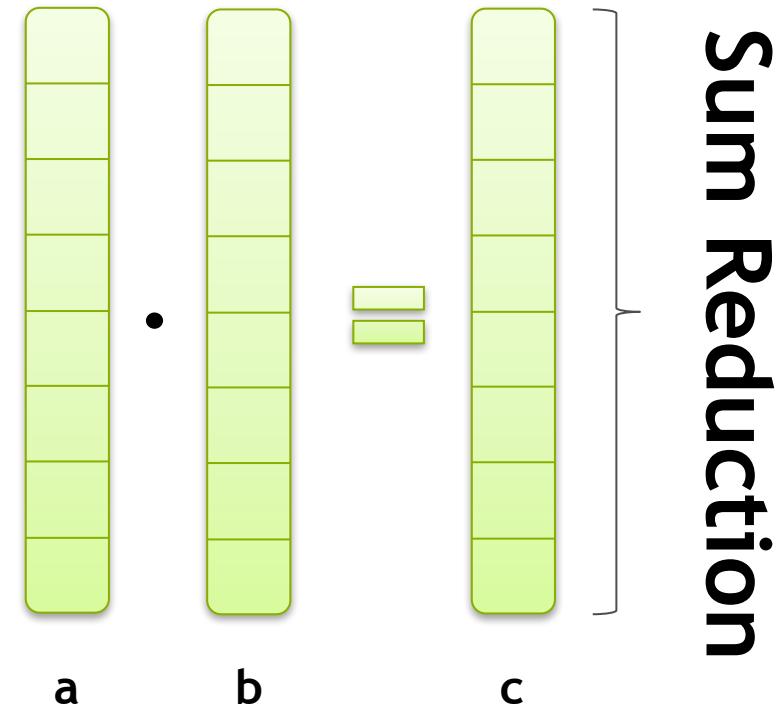


Case Study(1): Vector Dot Product

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

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define N (1<<24)
5
6 int main()
7 {
8     int i, block;
9     double *a, *b, *c;
10    struct timeval start;
11    struct timeval end;
12    double elapsedTime;
13    double sum= 0.0f;
14
15    a = (double *)malloc(sizeof(double) * N);
16    b = (double *)malloc(sizeof(double) * N);
17    c = (double *)malloc(sizeof(double) * N);
18
19    // init a and b
20    for (i=0; i<N; i++)
21    {
22        a[i] = (double)rand()/RAND_MAX;
23        b[i] = (double)rand()/RAND_MAX;
24    }
25
26    // Step 1: component product
27    for (i=0; i<N; i++)
28    {
29        c[i] = a[i] * b[i];
30    }
31
32    // Step 2: reduction
33    for (i=0; i<N; i++)
34    {
35        sum += c[i];
36    }
37
38    free(a);
39    free(b);
40    free(c);
41
42    return 0;
43 }
```

Step 1: Identify Available Parallelism

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define N (1<<24)
5
6  int main()
7  {
8      int i, block;
9      double *a, *b, *c;
10     struct timeval start;
11     struct timeval end;
12     double elapsedTime;
13     double sum= 0.0f;
14
15     a = (double *)malloc(sizeof(double) * N);
16     b = (double *)malloc(sizeof(double) * N);
17     c = (double *)malloc(sizeof(double) * N);
18
19     // init a and b
20     for (i=0; i<N; i++)
21     {
22         a[i] = (double)rand()/RAND_MAX;
23         b[i] = (double)rand()/RAND_MAX;
24     }
25
26     // Step 1: component product
27     for (i=0; i<N; i++)
28     {
29         c[i] = a[i] * b[i];
30     }
31
32     // Step 2: reduction
33     for (i=0; i<N; i++)
34     {
35         sum += c[i];
36     }
37
38     free(a);
39     free(b);
40     free(c);
41
42     return 0;
43 }
```

Step 1: Identify Available Parallelism

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define N (1<<24)
5
6 int main()
7 {
8     int i, block;
9     double *a, *b, *c;
10    struct timeval start;
11    struct timeval end;
12    double elapsedTime;
13    double sum= 0.0f;
14
15    a = (double *)malloc(sizeof(double) * N);
16    b = (double *)malloc(sizeof(double) * N);
17    c = (double *)malloc(sizeof(double) * N);
18
19    // init a and b
20    for (i=0; i<N; i++)
21    {
22        a[i] = (double)rand()/RAND_MAX;
23        b[i] = (double)rand()/RAND_MAX;
24    }
25
26    // Step 1: component product
27    for (i=0; i<N; i++)
28    {
29        c[i] = a[i] * b[i];
30    }
31
32    // Step 2: reduction
33    for (i=0; i<N; i++)
34    {
35        sum += c[i];
36    }
37
38    free(a);
39    free(b);
40    free(c);
41
42    return 0;
43 }
```

1st loop

2nd loop

Step 2: Express the Parallelism by Directives

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

Step 2: Express the Parallelism by Directives

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

```
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 }
```

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 : 260.43 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

```
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 }

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 }
```

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 : 165.91 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 : 165.91 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

OpenACC VS CUDA

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/time.h>
4
5 #define N (1<<24)
6 #define blocksize 1024
7 #define blocknumb (N/blocksize)
8
9 #define checkCudaAPIErrors(F) if ((F) != cudaSuccess) \
10 { printf("Error at line %d in file %s: %s\n", __LINE__, __FILE__, \
11         cudaGetErrorString(cudaGetLastError())); exit(-1); }
12
13 __global__ void vecDot(double *a, double *b, double *sub_sum)
14 {
15     int gid = blockDim.x * blockIdx.x + threadIdx.x;
16     __shared__ double component[blocksize];
17
18     component[threadIdx.x] = a[gid] * b[gid];
19
20     __syncthreads();
21     for (int i=(blocksize>>1); i>0; i=(i>>1))
22     {
23         if (threadIdx.x < i)
24             component[threadIdx.x] += component[threadIdx.x + i];
25         __syncthreads();
26     }
27
28     if (threadIdx.x == 0)
29     {
30         sub_sum[blockIdx.x] = component[0];
31     }
32 }
```

```
34 int main()
35 {
36     int i, device = 0;
37     double *h_a, *h_b, *h_c;
38     double *d_a, *d_b, *d_c;
39     double *h_subSum;
40     double *d_subSum;
41
42     struct timeval start;
43     struct timeval end;
44     double elapsedTime;
45     double sum_cpu = 0.0;
46     double sum_gpu = 0.0;
47     cudaDeviceProp prop;
48
49     h_a = (double *)malloc(sizeof(double) * N);
50     h_b = (double *)malloc(sizeof(double) * N);
51     h_c = (double *)malloc(sizeof(double) * N);
52     h_subSum = (double *)malloc(sizeof(double) * blocknumb);
53
54
55     // init a and b
56     for (i=0; i<N; i++)
57     {
58         h_a[i] = (double)rand()/RAND_MAX;
59         h_b[i] = (double)rand()/RAND_MAX;
60         h_c[i] = h_a[i] * h_b[i];
61
62         sum_cpu += h_c[i];
63     }
64
65     cudaSetDevice(device);
66     cudaGetDeviceProperties(&prop, device);
67     printf("Using gpu %d: %s\n", device, prop.name);
68
```

```
69 // timer begin
70 gettimeofday(&start, NULL);
71
72 cudaMalloc((void**)&d_a, sizeof(double) * N);
73 cudaMalloc((void**)&d_b, sizeof(double) * N);
74 cudaMalloc((void**)&d_c, sizeof(double) * N);
75 cudaMalloc((void**)&d_subSum, sizeof(double) * blocknumb);
76
77 checkCudaAPIErrors(cudaMemcpy(d_a, h_a, sizeof(double) * N, cudaMemcpyHostToDevice));
78 checkCudaAPIErrors(cudaMemcpy(d_b, h_b, sizeof(double) * N, cudaMemcpyHostToDevice));
79
80 vecDot<<<blocknumb, blocksize>>>(d_a, d_b, d_subSum);
81 checkCudaAPIErrors(cudaMemcpy(h_subSum, d_subSum, sizeof(double) * blocknumb, cudaMemcpyDeviceToHost));
82
83 cudaFree(d_a);
84 cudaFree(d_b);
85 cudaFree(d_c);
86 cudaFree(d_subSum);
87
88 for (i=0; i<blocknumb; i++)
89 {
90     sum_gpu += h_subSum[i];
91 }
92 // timer end
93 gettimeofday(&end, NULL);
94
95 elapsedTime = (end.tv_sec - start.tv_sec) * 1000.0;      // sec to ms
96 elapsedTime += (end.tv_usec - start.tv_usec) / 1000.0;    // us  to ms
97
98 printf("the result on GPU is %lf\n", sum_gpu);
99 printf("the result on CPU is %lf\n", sum_cpu);
100 printf("the elapsedTime is %f ms\n", elapsedTime);
101
102 free(h_a);
103 free(h_b);
104 free(h_c);
105
```

OpenACC VS CUDA

Performance

- For the extremely simple case, the performance between OpenACC and CUDA is comparable
 - CUDA: 140.50 ms
 - OpenACC: 165.91 ms

Workload

- CUDA >> OpenACC
 - Almost rewrite the code by 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

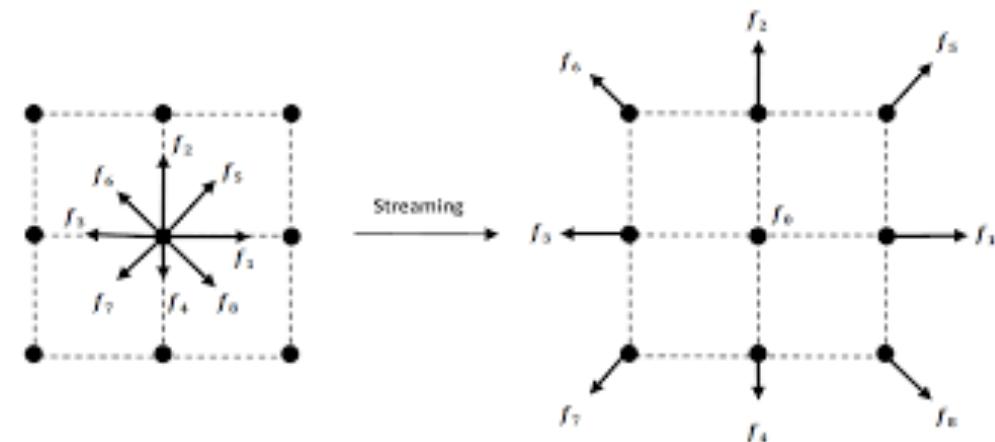
The computation of LBM is divided into two step

- Collide at current node (totally local operation)

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

- Stream to adjacent nodes (D2Q9)

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



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 5130.01 ms, so 19.62X speedup on K40

Accelerate the code with CUDA

Write two kernels: 353 lines code

Performance 1395.04 ms, so 3.68X 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	560	143	3.92
collision	405	112	3.61

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)

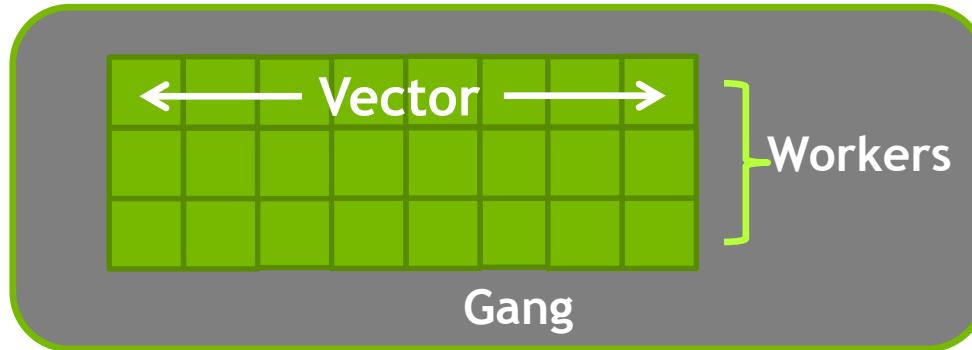
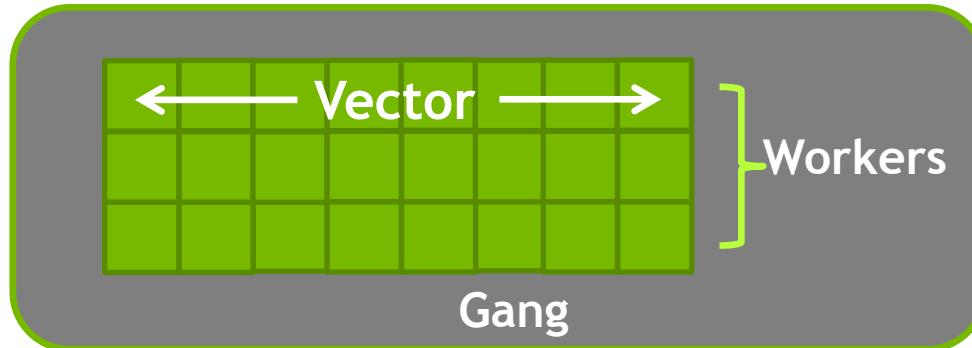
- Registers used by each kernel and occupancy achieved

	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: 2658.74 ms, $5130.01 / 2658.74 = 1.93X$
- Compared with CUDA: $2658.74 / 1395.04 = 1.91X$
- Block and Grid

	OpenACC	CUDA
propagate	$(64, 8, 1) / (\textcolor{red}{32}, 1, 1)$	$(32, 16, 1) / (2, 2, 32)$
collision	$(64, 8, 1) / (\textcolor{red}{32}, 1, 1)$	$(32, 16, 1) / (2, 2, 32)$

- 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 expand loops

Optimize the kernels generated by OpenACC

Use collapse to expand 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: 2242.97 ms, $5130.01 / 2242.97 = 45X$ Speedup
- Compared with CUDA: $2242.97 / 1395.04 = 1.6X$ 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

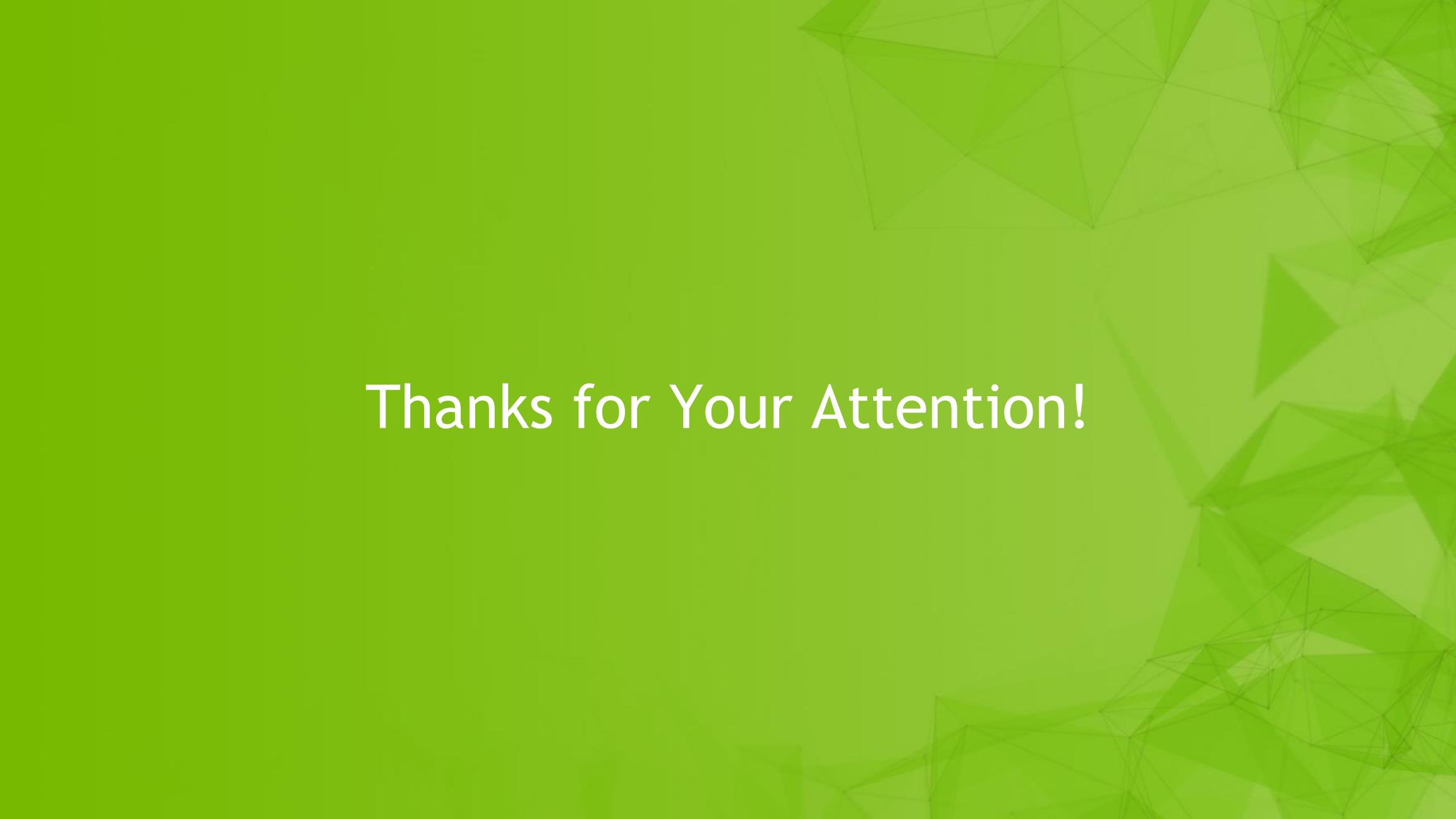
OpenACC Programming Cycle

- 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.



Thanks for Your Attention!