LEHIGH UNIVERSITY

Introduction to OpenACC

2021 HPC Workshop: Parallel Programming

Alexander B. Pacheco

Research Computing

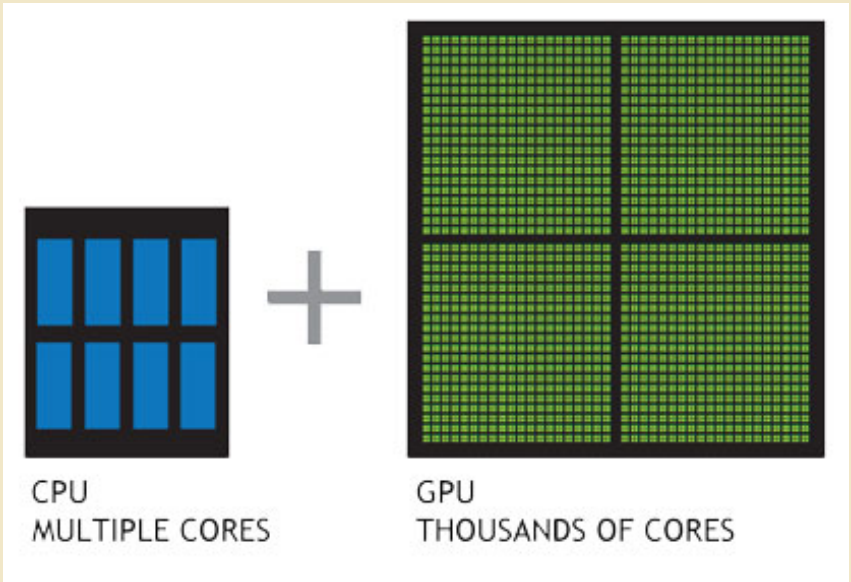
July 13 - 15, 2021

CPU vs GPU

CPU : consists of a few cores optimized for sequential serial processing

GPU : has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously

GPU enabled applications

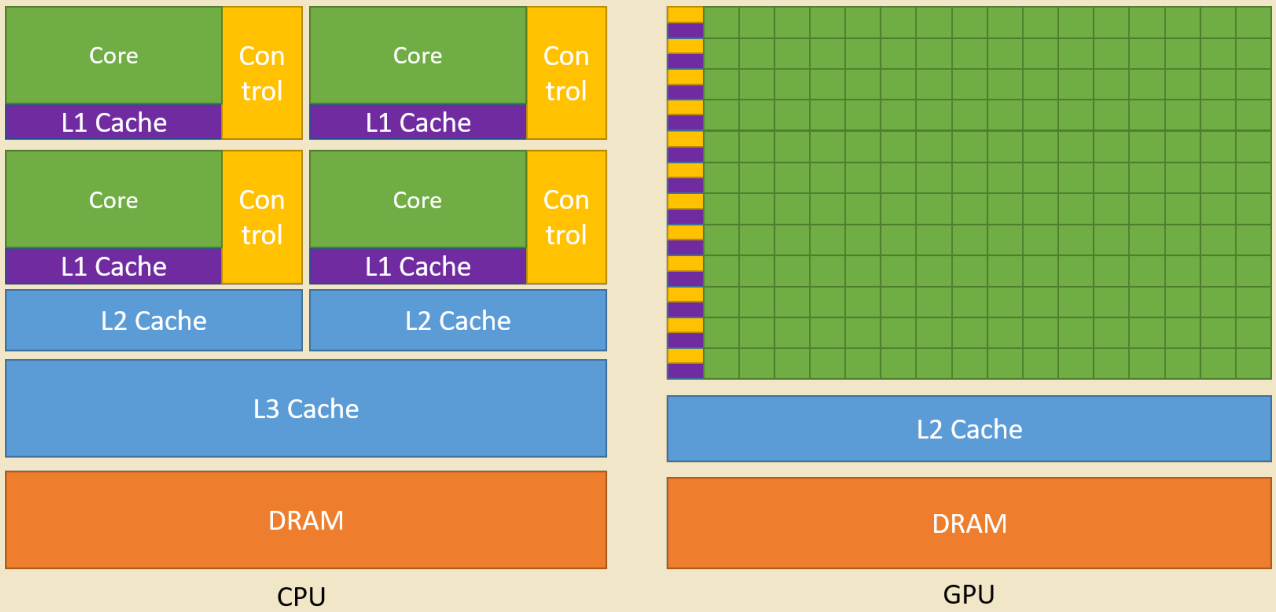


CPU vs GPU

CPU : consists of a few cores optimized for sequential serial processing

GPU : has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously

GPU enabled applications

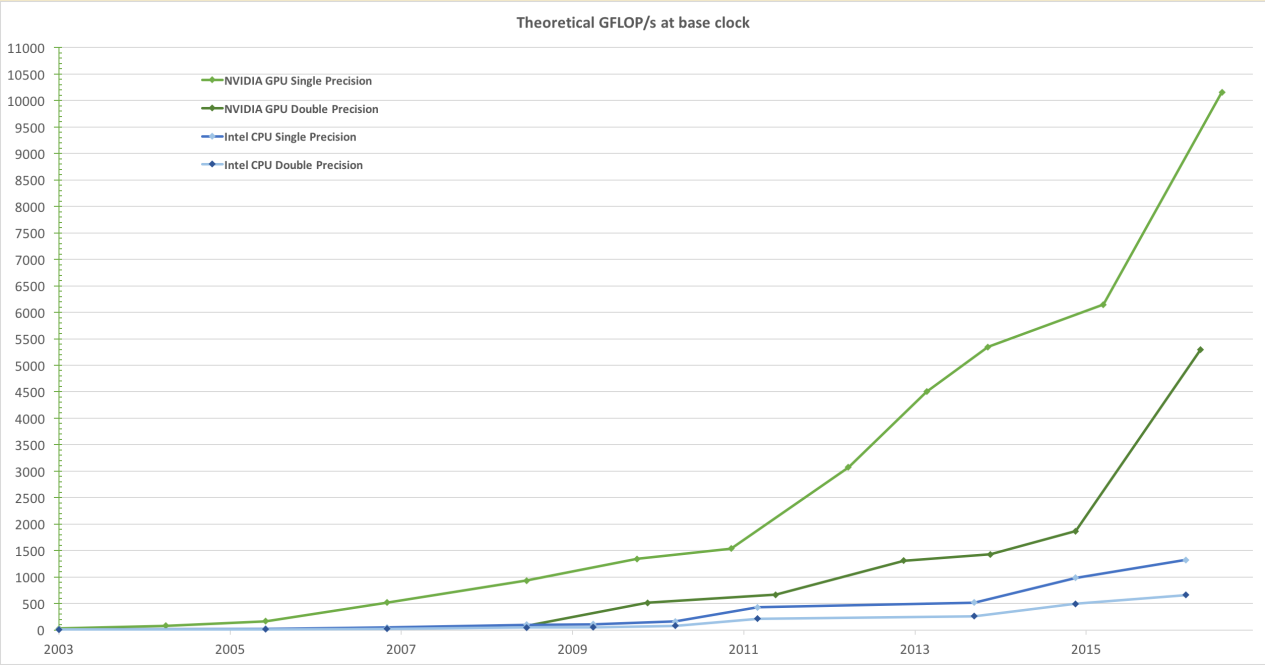


CPU vs GPU

CPU : consists of a few cores optimized for sequential serial processing

GPU : has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously

GPU enabled applications

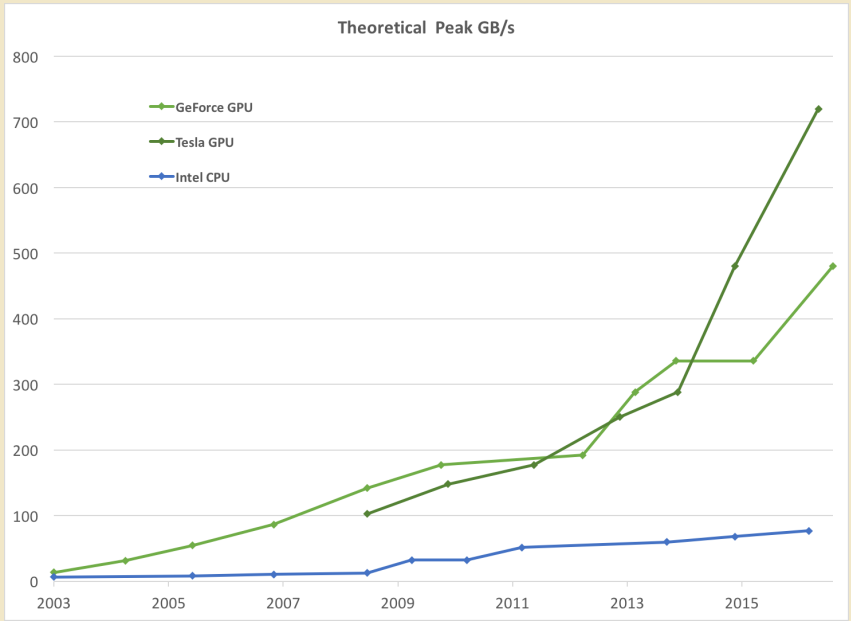


CPU vs GPU

CPU : consists of a few cores optimized for sequential serial processing

GPU : has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously

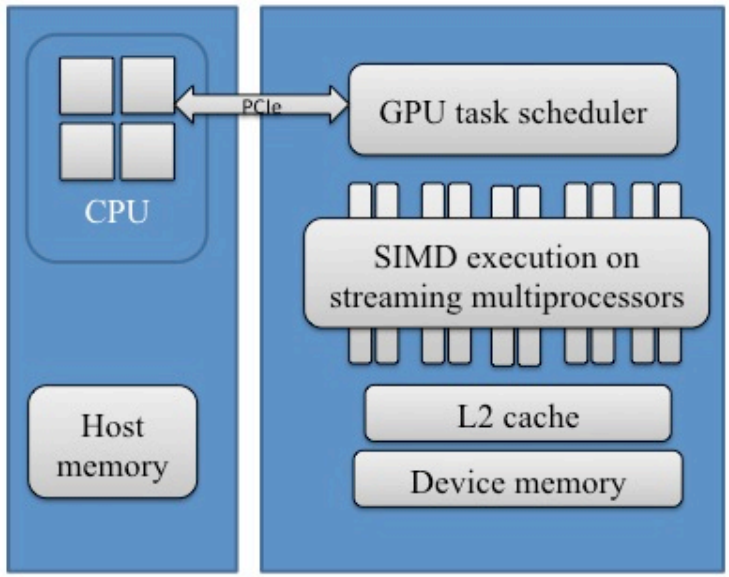
GPU enabled applications



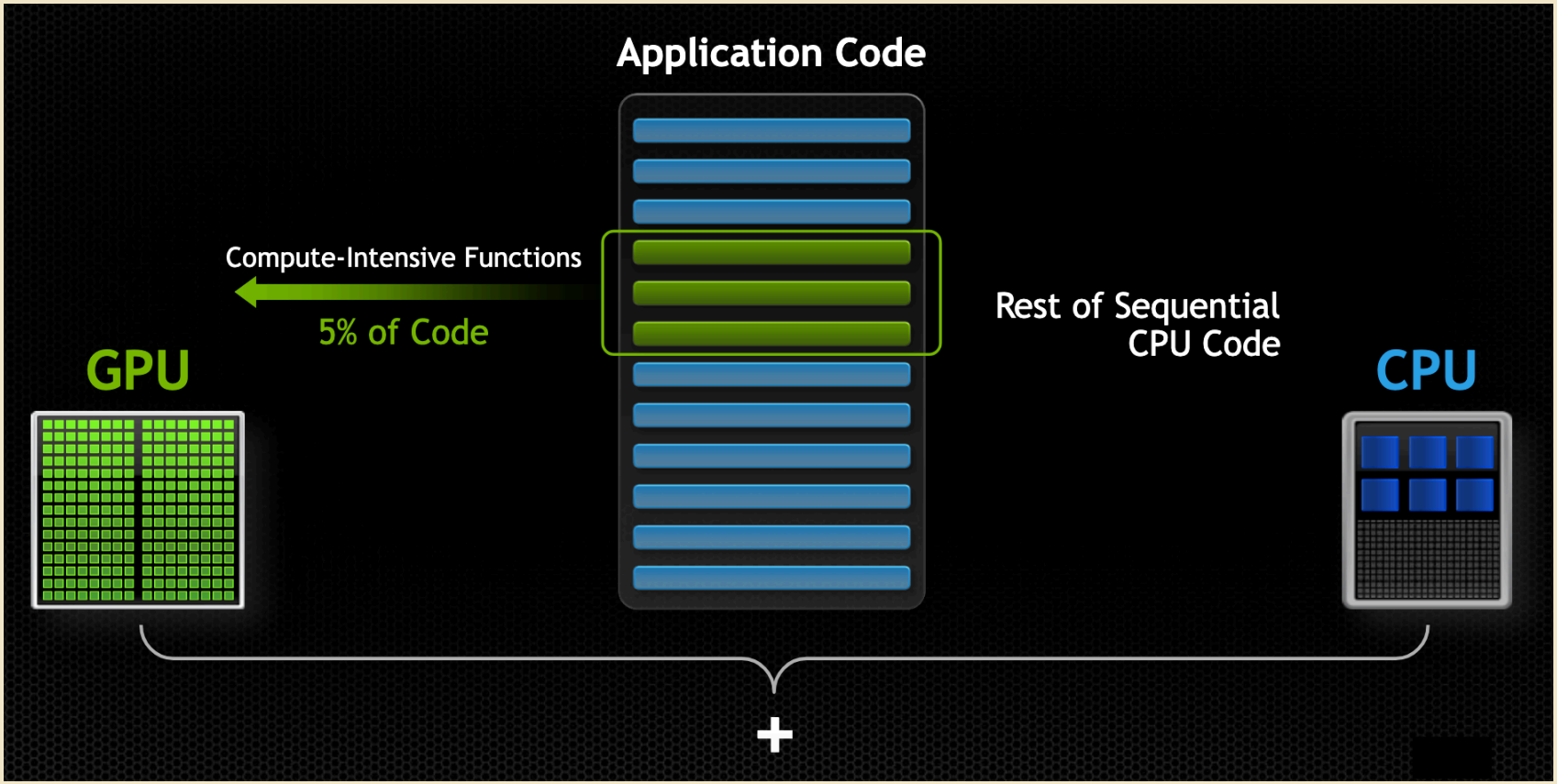
GPU Design

Model GPU design

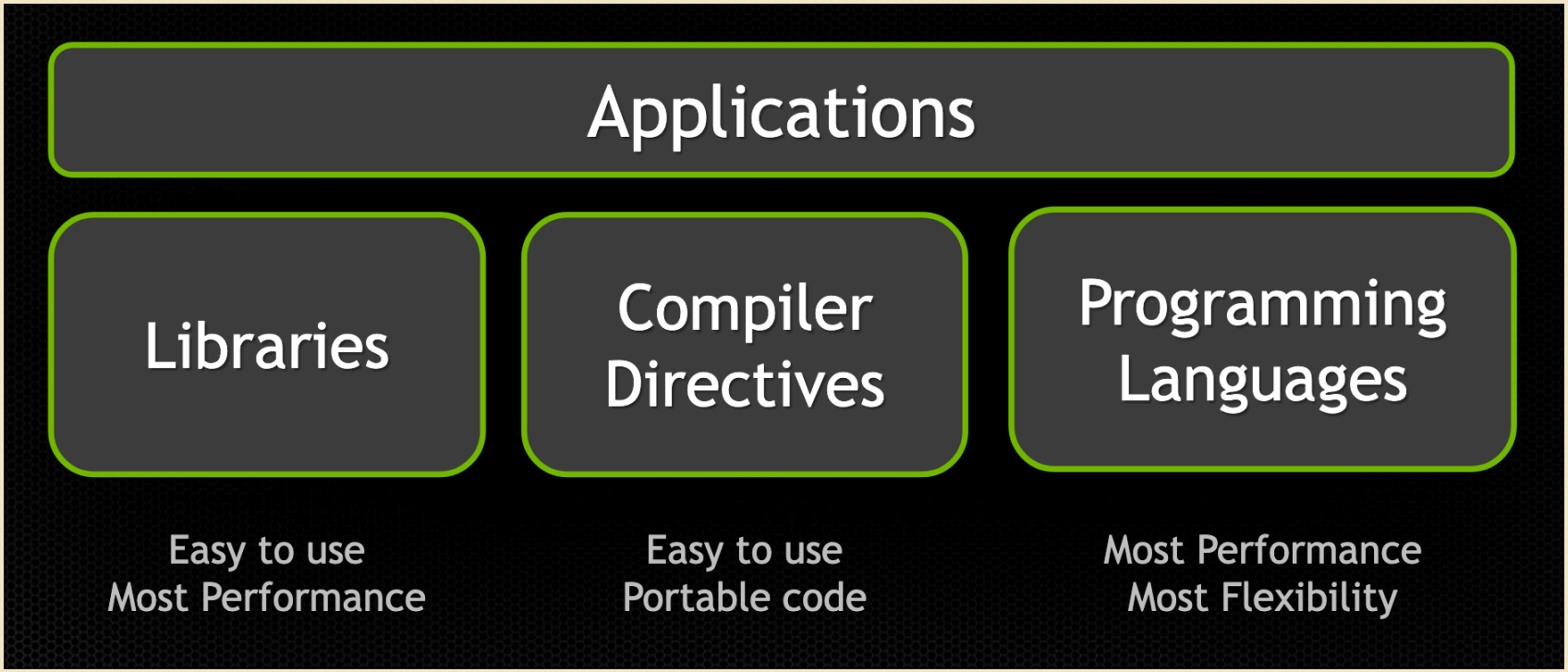
- Large number of cores working in SIMD mode.
- Slow global memory access, high bandwidth.
- CPU communication over PCI bus.
- Warp scheduling and fast switching queue model.



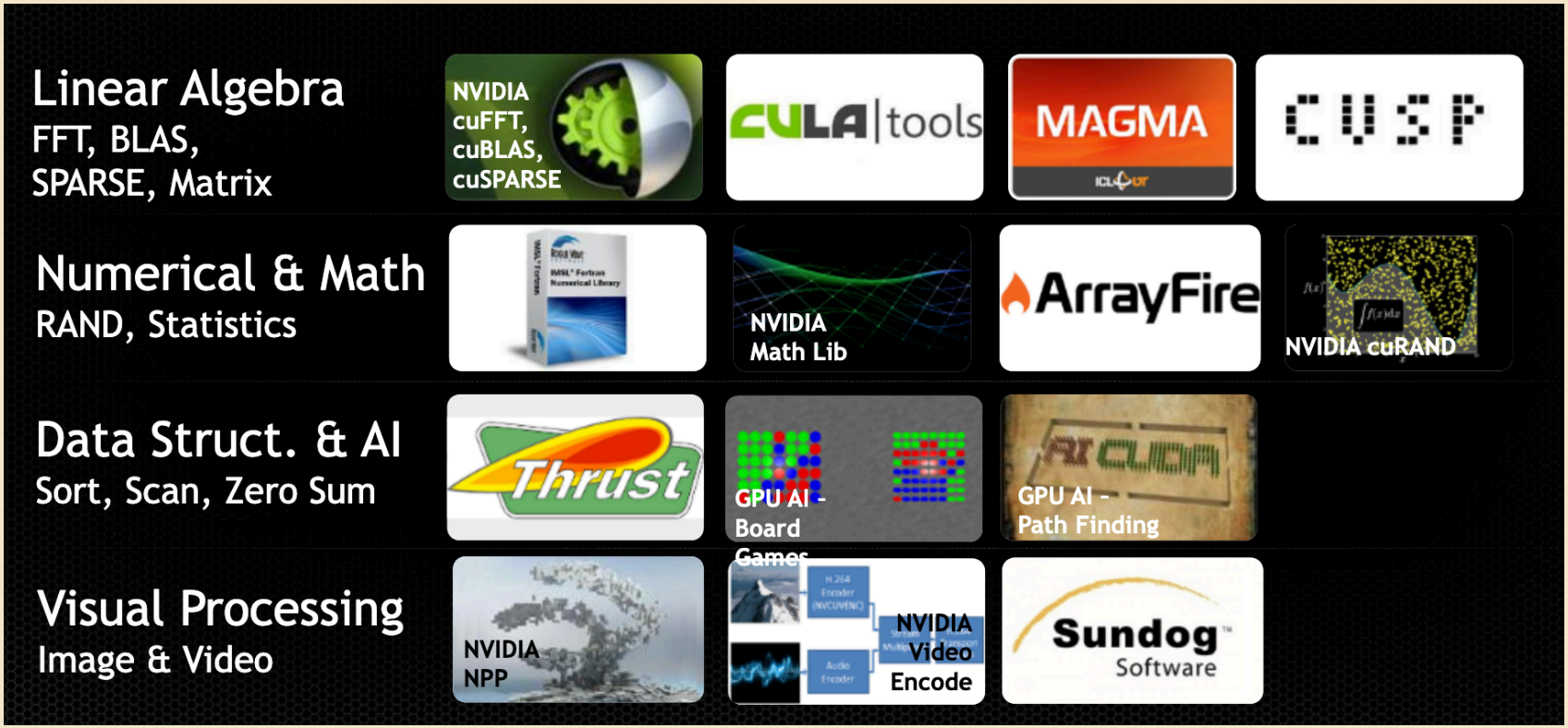
Hetergenous Programming



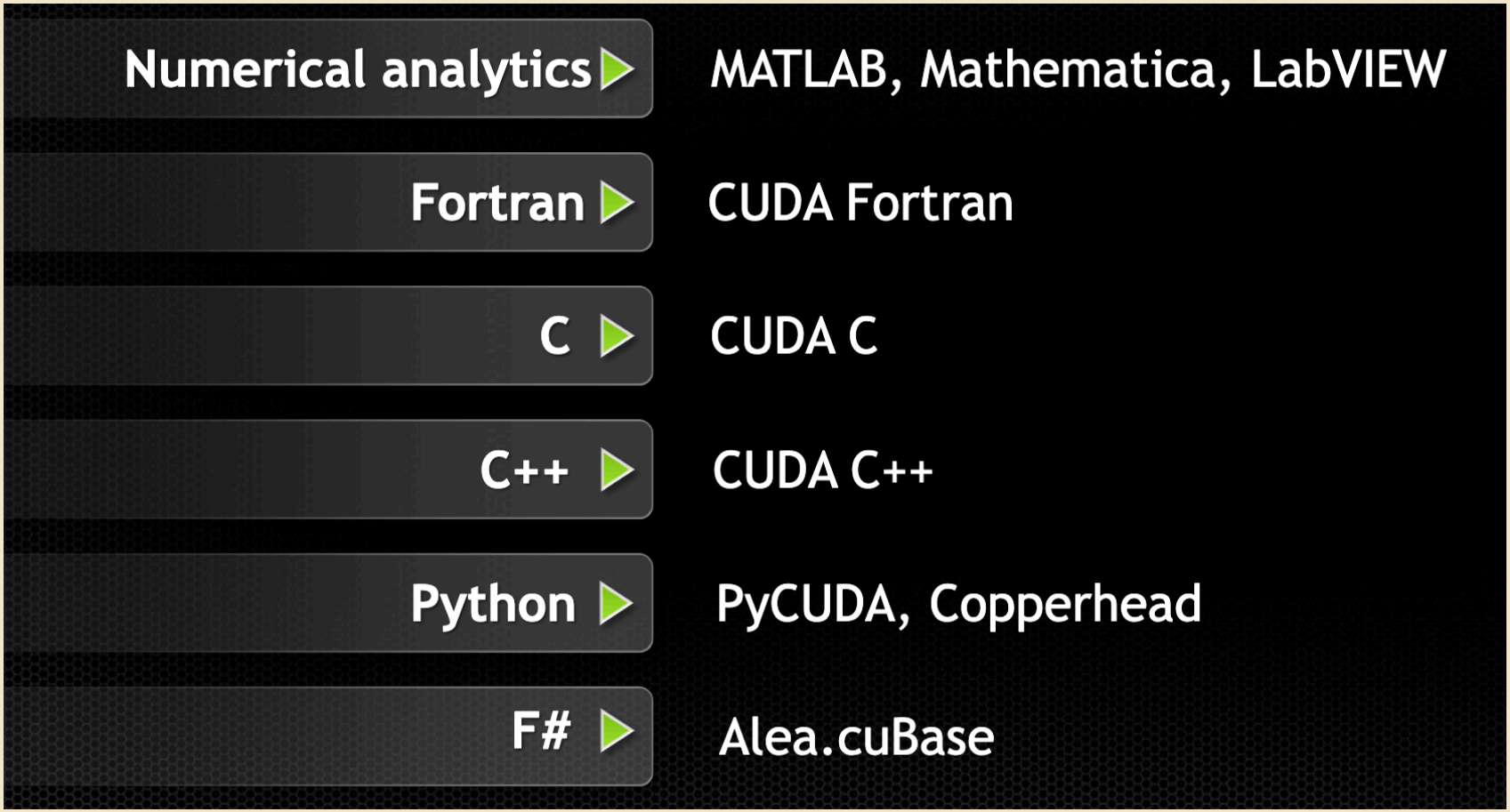
Accelerate Application for GPU



GPU Accelerated Libraries



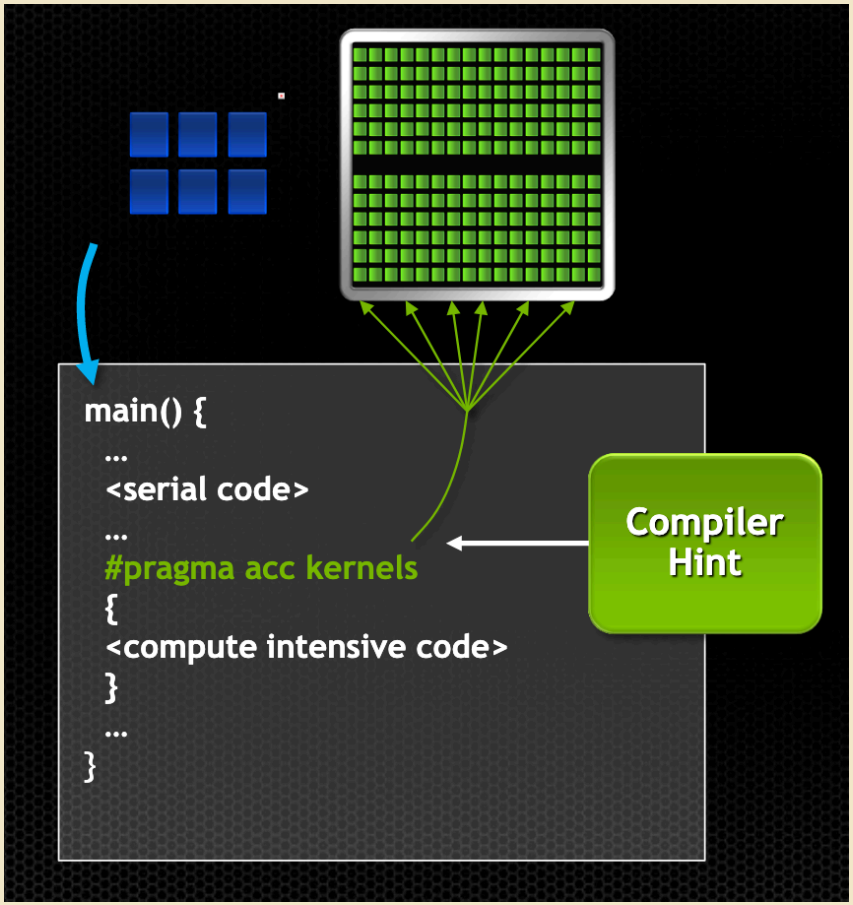
GPU Programming Languages



Accelerator Fundamentals

- We must expose enough parallelism to saturate the device
  - Accelerator threads are slower than CPU threads
  - Accelerators have orders of magnitude more threads
- Fine grained parallelism is good
- Coarse grained parallelism is bad
  - Lots of legacy apps have only exposed coarse grain parallelism
    - i.e. MPI and possibly OpenMP

What is OpenACC?



- Open Standard
- Easy, Compiler-Driven Approach
- portable across host CPUs and accelerators

What is OpenACC?

History

- OpenACC was developed by The Portland Group (PGI), Cray, CAPS and NVIDIA.
- PGI, Cray, and CAPs have spent over 2 years developing and shipping commercial compilers that use directives to enable GPU acceleration as core technology.
- The small differences between their approaches allowed the formation of a group to standardize a single directives approach for accelerators and CPUs.
- Full OpenACC 2.0 Specification available online
  - `http://www.openacc-standard.org/`
  - Implementations available now from PGI and Cray

What is OpenACC?

LEHIGH UNIVERSITY

The Standard for GPU Directives

Simple: Directive are the easy path to accelerate compute intensive applications

Open: OpenACC is an open GPU directives standard, making GPU programming straightforwards and portable across parallel and multi-core processors

Powerful: GPU directives allow complete access to the massive parallel power of a GPU

What is OpenACC?

LEHIGH UNIVERSITY

High Level

- Compiler directives to specify parallel regions in C & Fortran
  - Offload parallel regions
  - Portable across OSes, host CPUs, accelerators, and compilers
- Create high-level heterogenous programs
  - Without explicit accelerator intialization
  - Without explicit data or program transfers between host and accelerator

High Level · · · with low-level access

- Programming model allows programmers to start simple
- Compiler gives additional guidance
  - Loop mappings, data location and other performance details
- Compatible with other GPU languages and libraries
  - Interoperate between CUDA C/Fortran and GPU libraries
  - e.g. CUFFT, CUBLAS, CUSPARSE, etc

Why OpenACC

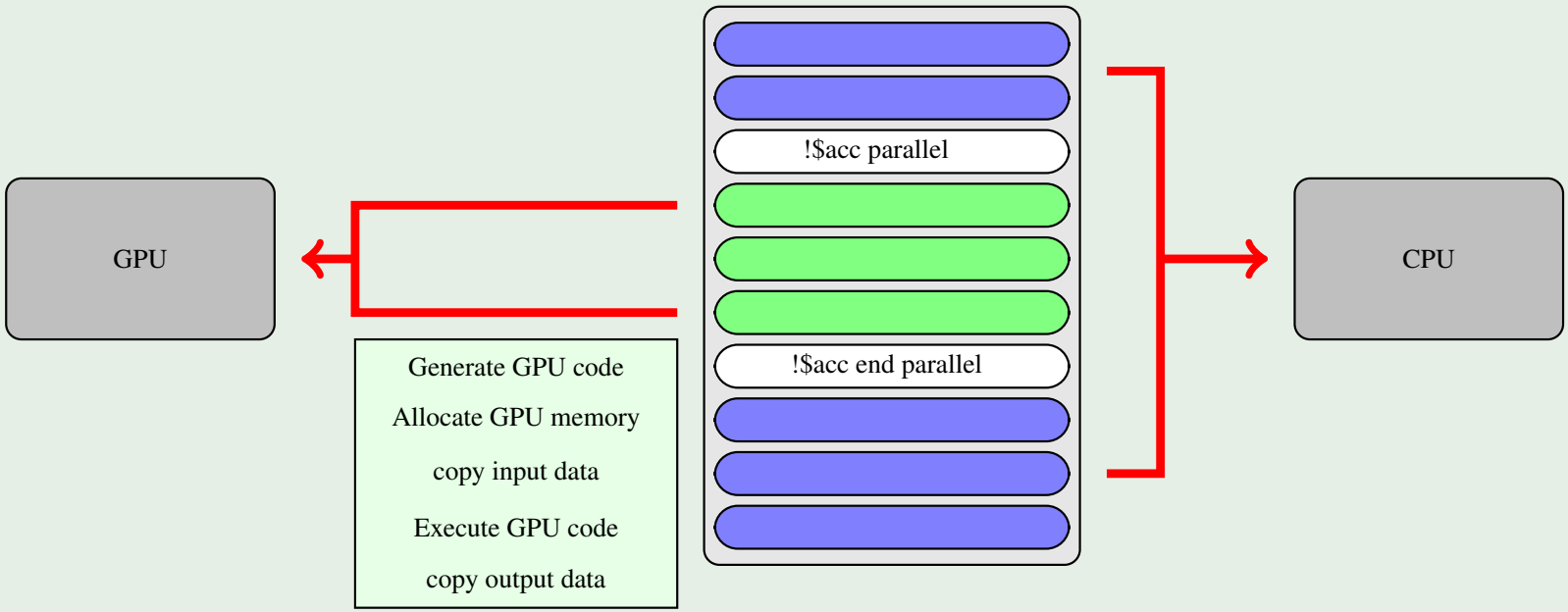
- Directives are easy and powerful.
- Avoid restructuring of existing code for production applications.
- Focus on expressing parallelism.

OpenACC is not GPU Programming

OpenACC is Expressing Parallelism in your code

OpenACC Execution Model

- Application code runs on the CPU (sequential, shared or distributed memory)
- OpenACC directives indicate that the following block of compute intensive code needs to be offloaded to the GPU or accelerator.



OpenACC Directive Syntax

LEHIGH UNIVERSITY

C/C++

#pragma acc directive [clause [,] clause] ...]

... often followed by a structured code block

Fortran

!$acc directive [clause [,] clause] ...]

...often paired with a matching end directive surrounding a structured code block:

!$acc end directive

OpenACC kernels directive

C: `#pragma acc kernels [clause]`

Fortran `!$acc kernels [clause]`

- The kernels directive expresses that a region may contain parallelism and the compiler determines what can be safely parallelized.
- The compiler breaks code in the kernel region into a sequence of kernels for execution on the accelerator device.
- What is a kernel? A function that runs in parallel on the GPU.
- When a program encounters a kernels contruct, it will launch a sequence of kernels in order on the device.

Fortran

```
!$acc kernels
do i = 1, n
   x(i) = 1.0
   y(i) = 2.0
end do

do i = 1, n
   y(i) = y(i) + a * x(i)
end do
!$acc end kernels
```

C/C++

```
#pragma acc kernels
{
  for (i = 0; i < n; i++){
    x[i] = 1.0 ;
    y[i] = 2.0 ;
  }

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

OpenACC Parallel Directive

- The **parallel** directive identifies a block of code as having parallelism.
- Compiler generates a parallel kernel for that loop.

C: `#pragma acc parallel [clauses]`

Fortran: `!$acc parallel [clauses]`

Fortran

```
!$acc parallel
do i = 1, n
   x(i) = 1.0
   y(i) = 2.0
end do

do i = 1, n
   y(i) = y(i) + a * x(i)
end do
!$acc end parallel
```

C/C++

```
#pragma acc parallel
{
  for (i = 0; i < n; i++){
    x[i] = 1.0 ;
    y[i] = 2.0 ;
  }

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

OpenACC Loop Directive

- Loops are the most likely targets for Parallelizing.
- The Loop directive is used within a parallel or kernels directive identifying a loop that can be executed on the accelerator device.

C: `#pragma acc loop [clauses]`

Fortran: `!$acc loop [clauses]`

- The loop directive can be combined with the enclosing parallel or kernels

C: `#pragma acc kernels loop [clauses]`

Fortran: `!$acc parallel loop [clauses]`

- The loop directive clauses can be used to optimize the code. This however requires knowledge of the accelerator device.

Clauses: gang, worker, vector, num_gangs, num_workers

Fortran

```
!$acc loop
do i = 1, n
   y(i) = y(i) + a * x(i)
end do
!$acc end loop
```

C/C++

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

OpenACC parallel vs. kernels

PARALLEL

- Requires analysis by programmer to ensure safe parallelism.
- Straightforward path from OpenMP

KERNELS

- Compiler performs parallel analysis and parallelizes what it believes is safe.
- Can cover larger area of code with single directive
- Gives compiler additional leeway

Both approaches are equally valid and can perform equally well.

Exercise

- Parallelize the saxpy code by adding OpenACC directives - parallel or kernels
- Compile the code using the following flags to the NVIDIA HPC SDK compiler

  ```
  -acc -gpu
  ```

- Run the code and compare timing with serial and openmp code

Serial Code

```
program saxpy

  implicit none
  integer :: i,n
  real,dimension(:),allocatable :: x, y
  real :: a,start_time, end_time

  n = 200000000
  allocate(x(n),y(n))

  x = 1.0d0
  y = 2.0d0
  a = 2.0d0

  call cpu_time(start_time)
  do i = 1, n
     y(i) = y(i) + a * x(i)
  end do
  call cpu_time(end_time)

  deallocate(x,y)

  print '(a,f8.6)', 'SAXPY Time: ', end_time - start_time

end program saxpy
```

OpenMP Code

```
program saxpy

  implicit none
  integer :: i,n,omp_get_max_threads
  real,dimension(:),allocatable :: x, y
  real :: a,start_time, end_time

  n = 200000000
  allocate(x(n),y(n))

  !$omp parallel do default(shared) private(i)
  do i = 1, n
     x(i) = 1.0
     y(i) = 1.0
  end do
  !$omp end parallel do
  a = 2.0

  call cpu_time(start_time)
  !$omp parallel do default(shared) private(i)
  do i = 1, n
     y(i) = y(i) + a * x(i)
  end do
  !$omp end parallel do
  call cpu_time(end_time)

  deallocate(x,y)

  print '(a,i3,2x,a,f8.6)', 'Threads: ', omp_get_max_threads(), &
        'SAXPY Time: ', end_time - start_time

end program saxpy
```

OpenACC Code

```
program saxpy

  implicit none
  integer :: i,n
  real,dimension(:),allocatable :: x, y
  real :: a,start_time, end_time

  n = 200000000
  allocate(x(n),y(n))

  !$acc parallel loop
  do i = 1, n
     x(i) = 1.0
     y(i) = 1.0
  end do
  !$acc end parallel loop
  a = 2.0

  call cpu_time(start_time)
  !$acc parallel loop
  do i = 1, n
     y(i) = y(i) + a * x(i)
  end do
  !$acc end parallel loop
  call cpu_time(end_time)
  deallocate(x,y)

  print '(a,f8.6)', 'SAXPY Time: ', end_time - start_time

end program saxpy
```

CUDA Fortran Code

```
module mymodule
contains
  attributes(global) subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
    attributes(value) :: a, n
    i = threadIdx%x+(blockIdx%x-1)*blockDim%x
    if (i<=n) y(i) = a*x(i)+y(i)
  end subroutine saxpy
end module mymodule

program main
  use cudafor; use mymodule
  integer, parameter :: n = 200000000
  real, device :: x_d(n), y_d(n)
  real, device :: a_d
  real :: start_time, end_time

  x_d = 1.0
  y_d = 2.0
  a_d = 2.0

  call cpu_time(start_time)
  call saxpy<<<4096, 256>>>(n, a, x_d, y_d)
  call cpu_time(end_time)

  print '(a,f15.6,a)', 'SAXPY Time: ', end_time - start_time, 'in secs'
end program main
```

SAXPY: Compile & Run

Compile

```
[alp514.sol](1018): nvc -acc -gpu -o saxpyc_acc saxpy_acc.c
```

- Specify the gpu architecture: `-gpu=cc75`
- Get more information about the compilation: `-Minfo=accel`

```
[alp514.hawk-b625](1005): nvfortran -acc -gpu=cc75 -Minfo=accel -o saxpyf_acc saxpy_acc.f90
saxpy:
     12, Generating Tesla code
         13, !$acc loop vector(128) ! threadidx%x
     12, Generating implicit copyout(x(1:200000000)) [if not already present]
     13, Loop is parallelizable
     15, Generating Tesla code
         16, !$acc loop vector(128) ! threadidx%x
     15, Generating implicit copyout(y(1:200000000)) [if not already present]
     16, Loop is parallelizable
     20, Generating Tesla code
         21, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
     20, Generating implicit copy(y(1:200000000)) [if not already present]
         Generating implicit copyin(x(1:200000000)) [if not already present]
```

SAXPY: Compile & Run

Fortran Timings

Algorithm	Device	Time (s)	Speedup
Serial	Xeon Gold 6230R	0.504534	
OpenMP (12 threads)	Xeon Gold 6230R	0.050300	10.03
OpenMP (24 threads)	Xeon Gold 6230R	0.026623	18.95
OpenMP (48 threads)	Xeon Gold 6230R	0.025263	19.97
OpenACC	Tesla T4	0.517426	0.98
CUDA	Tesla T4	0.007846	64.30

C Timings

Algorithm	Device	Time (s)	Speedup
Serial	Xeon Gold 6230R	0.512128	
OpenMP (12 threads)	Xeon Gold 6230R	0.056454	9.07
OpenMP (24 threads)	Xeon Gold 6230R	0.048442	10.57
OpenMP (48 threads)	Xeon Gold 6230R	0.026348	19.44
OpenACC	Tesla T4	3.434997	0.15
CUDA	Tesla T4	0.000008	64016

What's going with OpenACC code?
Why even bother with OpenACC if performance is so bad?

Analyzing OpenACC Run Time

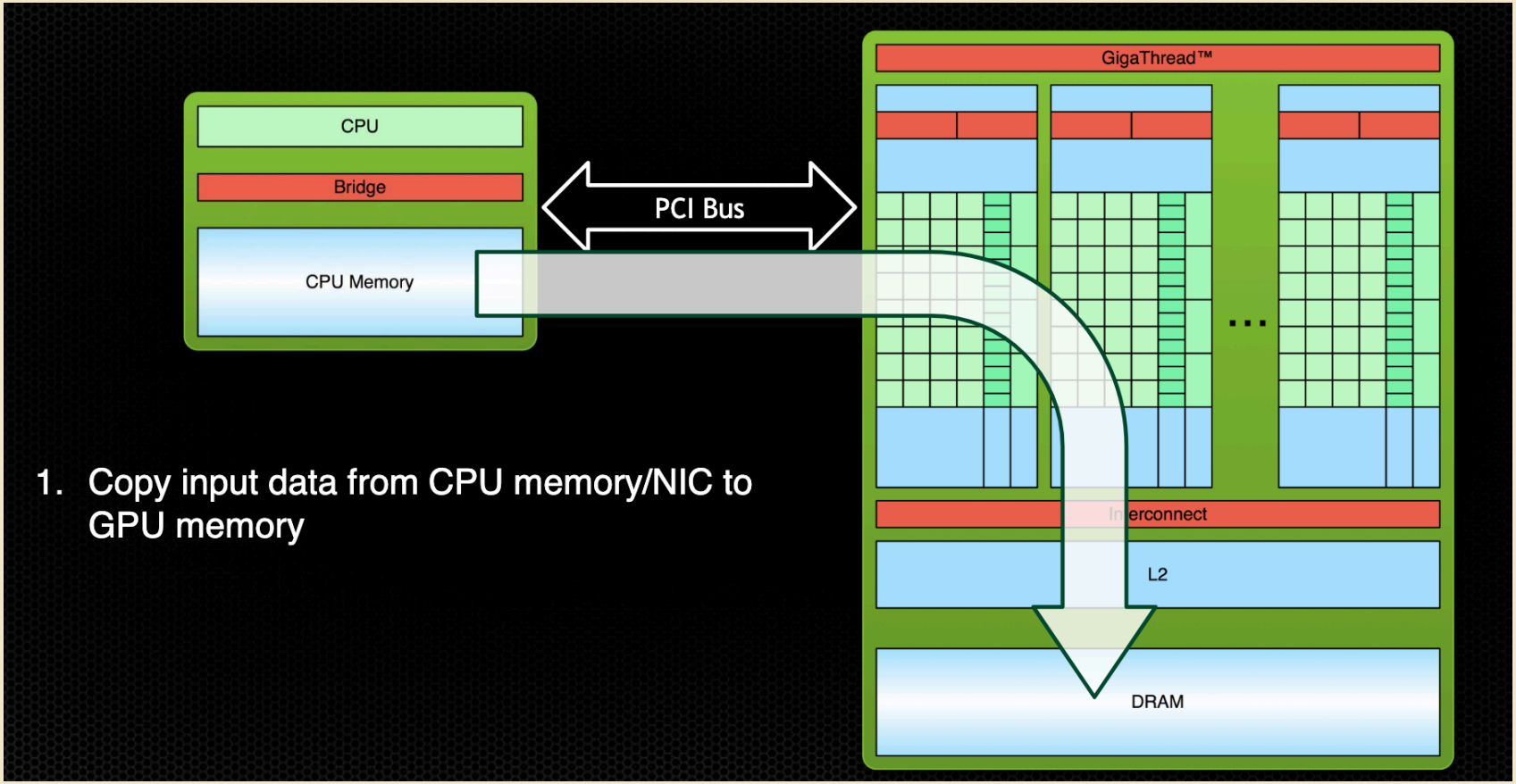
- The NVIDIA HPC SDK compiler provides automatic instrumentation when NV_ACC_TIME=1 at runtime

```
[alp514.hawk-b624](1002): NV_ACC_TIME=1 ./saxpyf_acc-nodata
SAXPY Time: 0.778822

Accelerator Kernel Timing data
/home/alp514/Workshop/2021HPC/parprog/solution/saxpy/saxpy_acc-nodata.f90
  saxpy  NVIDIA  devicenum=0
    time(us): 639,595
    11: compute region reached 1 time
        11: kernel launched 1 time
            grid: [65535]  block: [128]
             device time(us): total=7,824 max=7,824 min=7,824 avg=7,824
            elapsed time(us): total=7,872 max=7,872 min=7,872 avg=7,872
    11: data region reached 2 times
        16: data copyout transfers: 96
             device time(us): total=243,594 max=2,558 min=1,751 avg=2,537
    20: compute region reached 1 time
        20: kernel launched 1 time
            grid: [65535]  block: [128]
             device time(us): total=9,556 max=9,556 min=9,556 avg=9,556
            elapsed time(us): total=9,605 max=9,605 min=9,605 avg=9,605
    20: data region reached 2 times
        20: data copyin transfers: 96
             device time(us): total=256,886 max=2,709 min=1,841 avg=2,675
        24: data copyout transfers: 48
             device time(us): total=121,735 max=2,556 min=1,752 avg=2,536
```

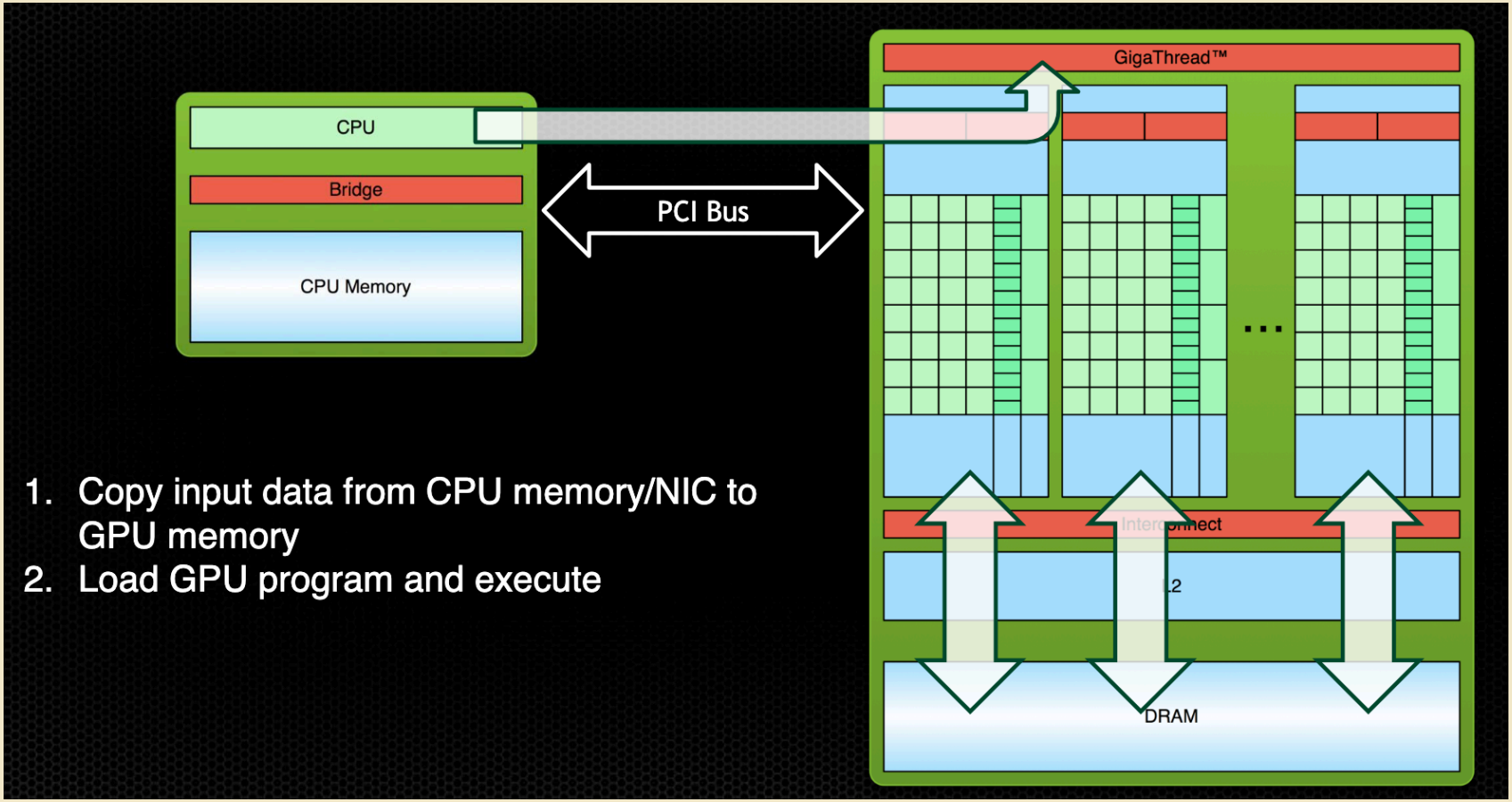
```
[alp514.hawk-b624](1003): NV_ACC_TIME=1 ./saxpyc_acc-nodata
SAXPY Time: 3.984324

Accelerator Kernel Timing data
/home/alp514/Workshop/2021HPC/parprog/solution/saxpy/saxpy_acc-nodata.c
  main  NVIDIA  devicenum=0
    time(us): 1,277,749
    16: compute region reached 1 time
        16: kernel launched 1 time
            grid: [65535]  block: [128]
             device time(us): total=14,513 max=14,513 min=14,513 avg=14,513
            elapsed time(us): total=14,561 max=14,561 min=14,561 avg=14,561
    16: data region reached 2 times
        19: data copyout transfers: 192
             device time(us): total=487,132 max=2,559 min=946 avg=2,537
    22: compute region reached 1 time
        22: kernel launched 1 time
            grid: [65535]  block: [128]
             device time(us): total=19,484 max=19,484 min=19,484 avg=19,484
            elapsed time(us): total=19,518 max=19,518 min=19,518 avg=19,518
    22: data region reached 2 times
        22: data copyin transfers: 192
             device time(us): total=513,260 max=2,713 min=993 avg=2,673
        24: data copyout transfers: 96
             device time(us): total=243,360 max=2,556 min=944 avg=2,535
```

- Fortran: $\sim 17\,ms$ for actual calculation and $\sim 0.6\,s$ for data transfer
- C: $\sim 35\,ms$ for actual calculation and $\sim 1.2\,s$ for data transfer

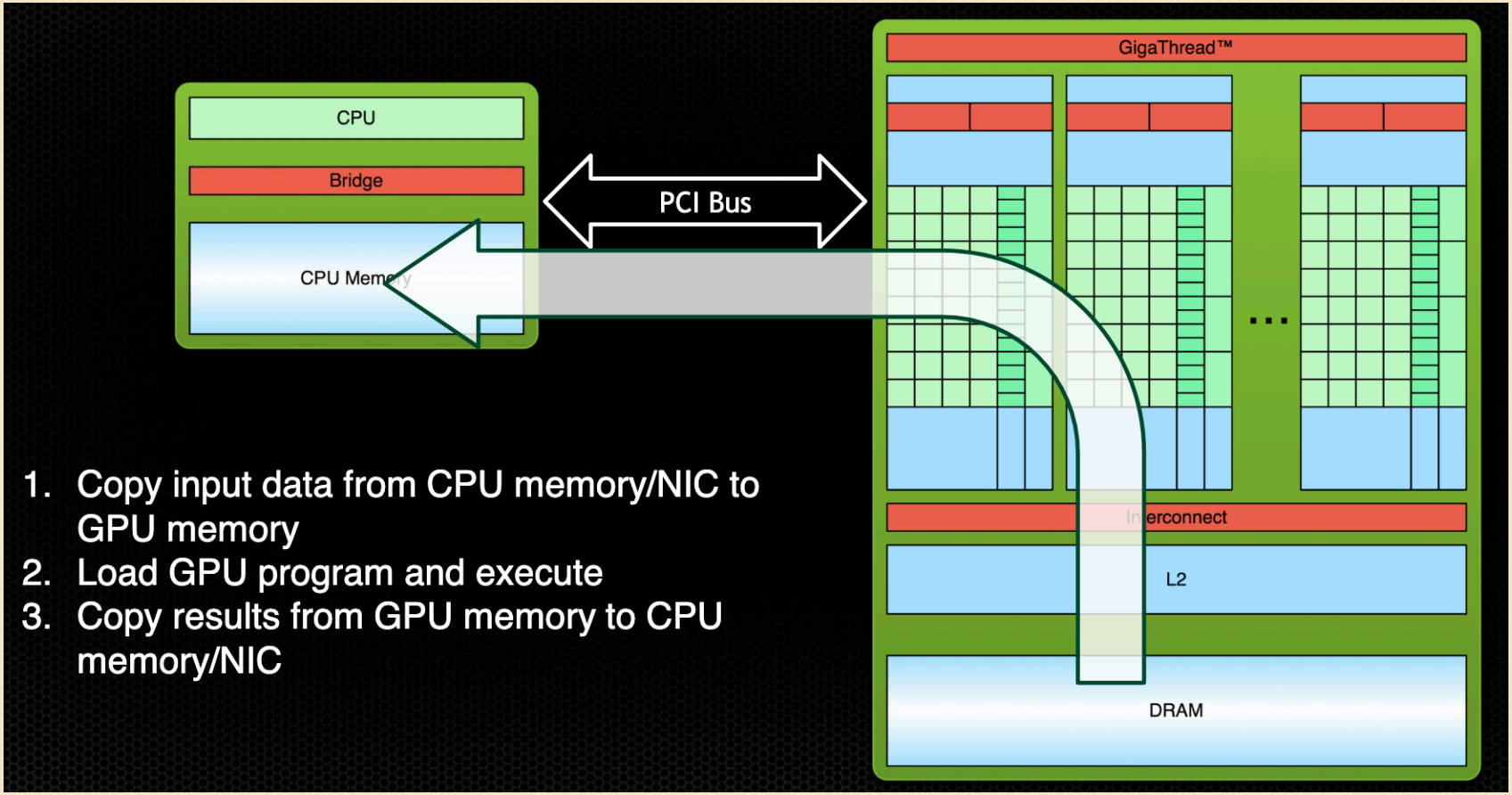
Processing Flow



Processing Flow



Processing Flow



Defining data regions

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

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

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

Data Clauses

copy(list)	Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.
copyin(list)	Allocates memory on GPU and copies data from host to GPU when entering region.
copyout(list)	Allocates memory on GPU and copies data to the host when exiting region.
create(list)	Allocates memory on GPU but does not copy.
present(list)	Data is already present on GPU from another containing data region.

- Other clauses: present_or_copy[in|out], present_or_create, deviceptr.

Array Shaping

- Compiler sometime cannot determine size of arrays
  - Must specify explicitly using the data clauses and array "shape"

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

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

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

Exercise

- Modify the SAXPY code to add a structured data region at the appropriate spot
- How does the compiler output the change?
- Is the code faster now?
- By how much and how does it compare with the serial and openmp code?
- Reprofile the code using NV_ACC_TIME

Fortran Timings

Algorithm	Device	Time (s)	Speedup
Serial	Xeon Gold 6230R	0.504534	
OpenMP (12 threads)	Xeon Gold 6230R	0.050300	10.03
OpenMP (24 threads)	Xeon Gold 6230R	0.026623	18.95
OpenMP (48 threads)	Xeon Gold 6230R	0.025263	19.97
OpenACC	Tesla T4	0.009601	52.55
CUDA	Tesla T4	0.007846	64.30

C Timings

Algorithm	Device	Time (s)	Speedup
Serial	Xeon Gold 6230R	0.512128	
OpenMP (12 threads)	Xeon Gold 6230R	0.056454	9.07
OpenMP (24 threads)	Xeon Gold 6230R	0.048442	10.57
OpenMP (48 threads)	Xeon Gold 6230R	0.026348	19.44
OpenACC	Tesla T4	0.019029	26.91
CUDA	Tesla T4	0.000008	64016

Update Construct

- Used to update existing data after it has changed in its corresponding copy (e.g. upate device copy after host copy changes).
- Move data from GPU to host, or host to GPU.
- Data movement can be conditional and asynchronous.
- Fortran

```
!$acc update [clause ···]
```

- C

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

- Clause
  - `host(list)`
  - `device(list)`
  - `if(expression)`
  - `async(expression)`

OpenACC private Clause

```
#pragma acc parallel loop
  for(int i=0;i<M;i++) {
    for(int jj=0;jj<10;jj++)
      tmp[jj]=jj;
    int sum=0;
    for(int jj=0;jj<N;jj++)
      sum+=tmp[jj];
    A[i]=sum;
  }
```

```
#pragma acc parallel loop
     private(tmp[0:9])
  for(int i=0;i<M;i++) {
    for(int jj=0;jj<10;jj++)
      tmp[jj]=jj;
    int sum=0;
    for(int jj=0;jj<N;jj++)
      sum+=tmp[jj];
    A[i]=sum;
  }
```

- Compiler cannot parallelize because tmp is shared across threads
- Also useful for live-out scalars

OpenACC reduction Clause

- Reduction clause is allowed on *parallel* and *loop* constructs

Fortran

```
!$acc parallel reduction(operation: var)
  structured block with reduction on var
!$acc end parallel
```

C

```
#pragma acc kernels reduction(operation: var) {
  structured block with reduction on var
}
```

Exercise

- Parallelize the Pi Calculation code using the Reduction clause
- Compare timing with serial and openmp code

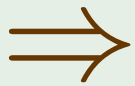
Nested Loops

- Currently we have only exposed parallelism on the outer loop
- We know that both loops can be parallelized
- Lets look at methods for parallelizing multiple loops

OpenACC collapse Clause

collapse(n): Applies the associated directive to the following ntightly nested loops.

```
#pragma acc parallel
#pragma acc loop
    collapse(2)
for(int i=0; i<N; i++)
  for(int j=0; j<N; j
      ++)
    ...
```



```
#pragma acc parallel
#pragma acc loop
for(int ij=0; ij<N*N;
    ij++)
...
```

Loops must be tightly nested

Exercise

- Parallelze the Matmul code using collapse clause
- Compare timings and GFLOPS with serial and openmp code

Further Speedups

- OpenACC gives us more detailed control over parallelization
  - Via **gang**, **worker** and **vector** clauses
- By understanding more about specific GPU on which you're running, using these clauses may allow better performance.
- By understanding bottlenecks in the code via profiling, we can reorganize the code for even better performance.

General Principles: Finding Parallelism in Code

- (Nested) for/do loops are best for parallelization
- Large loop counts are best
- Iterations of loops must be independent of each other
  - To help compiler: restrict keyword (C), independent clause
  - Use subscripted arrays, rather than pointer-indexed arrays
- Data regions should avoid wasted bandwidth
  - Can use directive to explicitly control sizes
- Various annoying things can interfere with accelerated regions.
  - Function calls within accelerated region must be inlineable.
  - No IO

OpenACC: Is it worth it?

- High-level. No involvement of OpenCL, CUDA, etc
- Single source. No forking off a separate GPU code. Compile the same program for accelerators or serial, non-GPU programmers can play along.
- Efficient. Experience shows very favorable comparison to low-level implementations of same algorithms.
- Performance portable. Supports GPU accelreators and co-processors from multiple vendors, current and future versions.
- Incremental. Developers can port and tune parts of their application as resources and profiling dictates. No wholesale rewrite required. Which can be quick.

Further Reading and References

- OpenACC Programming and Best Practices Guide
- OpenACC 2.7 API Reference Card
- Parallel programming with OpenACC - Rob Farber (Libraries Link)
- OpenACC for Programmers: Concepts and Strategies - Guido Juckeland and Sunita Chandrasekaran (Libraries Link)

  Lecture derived from slides and presentations by
- Michael Wolfe, PGI
- Jeff Larkin, NVIDIA
- John Urbanic, PSC

  Search for OpenACC presentations at the GPU Technology Conference Website for further study
  `http://www.gputechconf.com/gtcnew/on-demand-gtc.php`