
NVIDIA
OPENACC MINI-HACKATHON
ORNL – November 2016

GAMING

PRO VISUALIZATION

DATA CENTER

AUTO

World Leader in Visual Computing

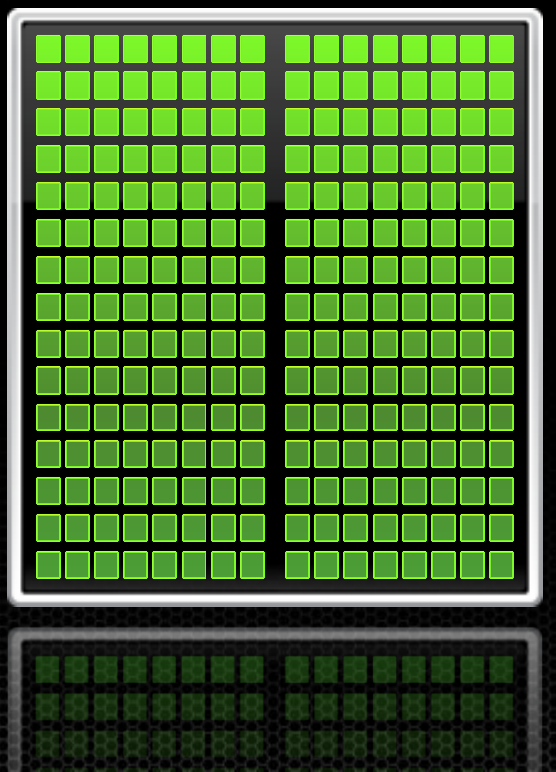
HPC's Biggest Challenge: Power

Accelerated Computing

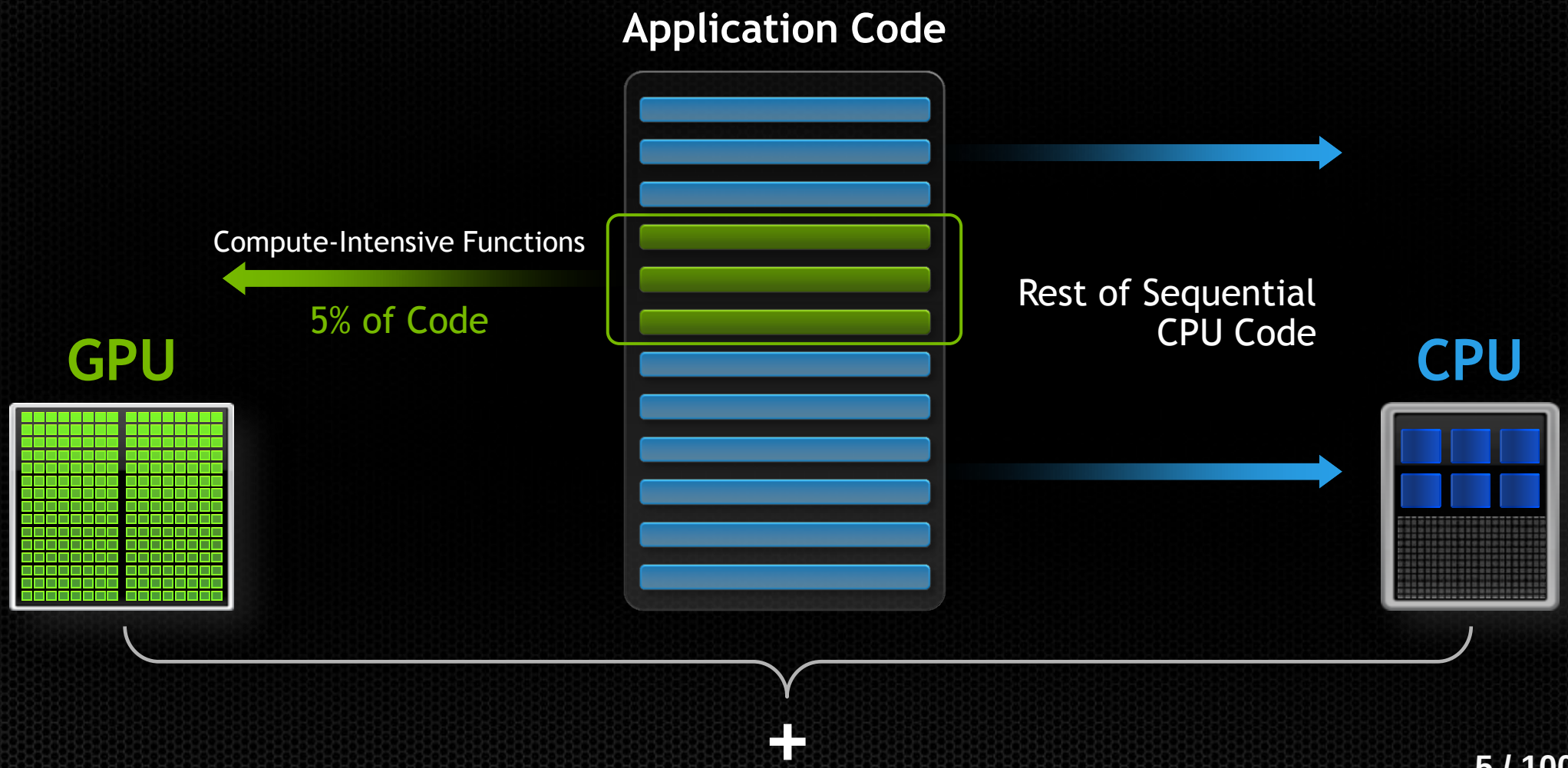
10x Performance & 5x Energy Efficiency for HPC

CPU
Optimized for
Serial Tasks

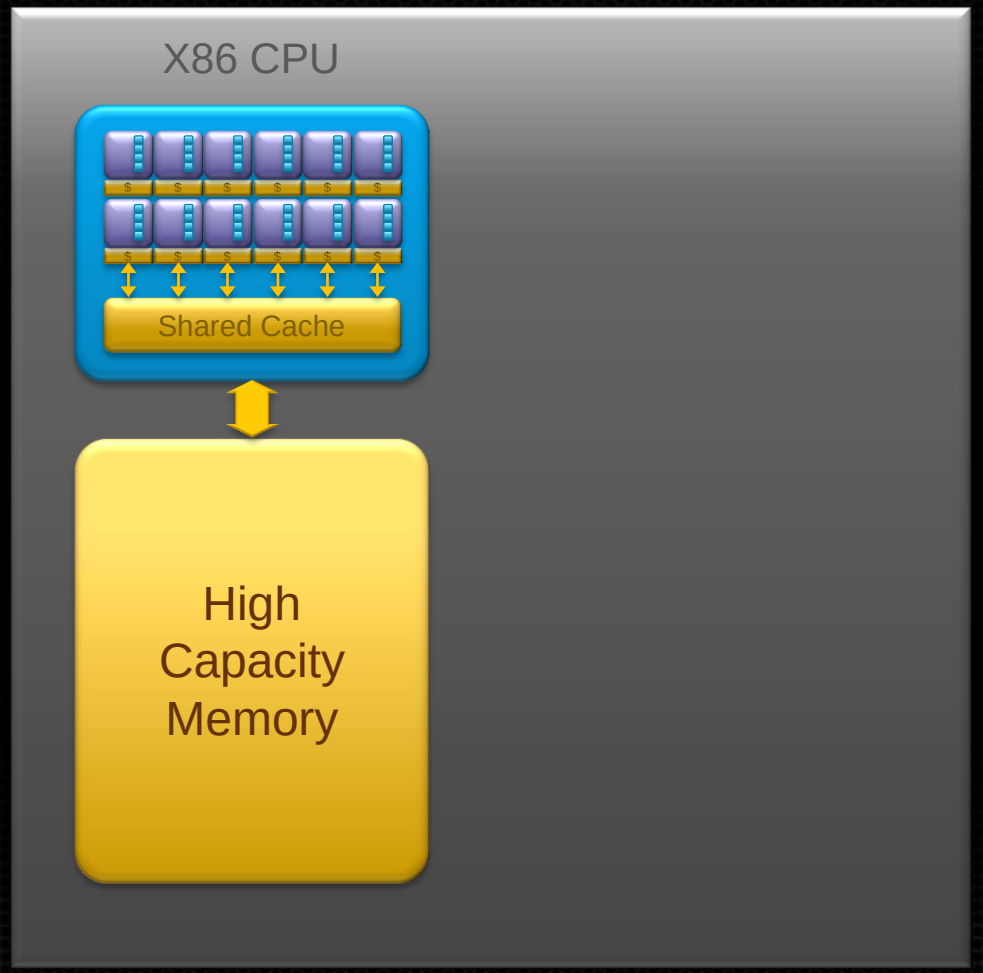
GPU Accelerator
Optimized for
Parallel Tasks



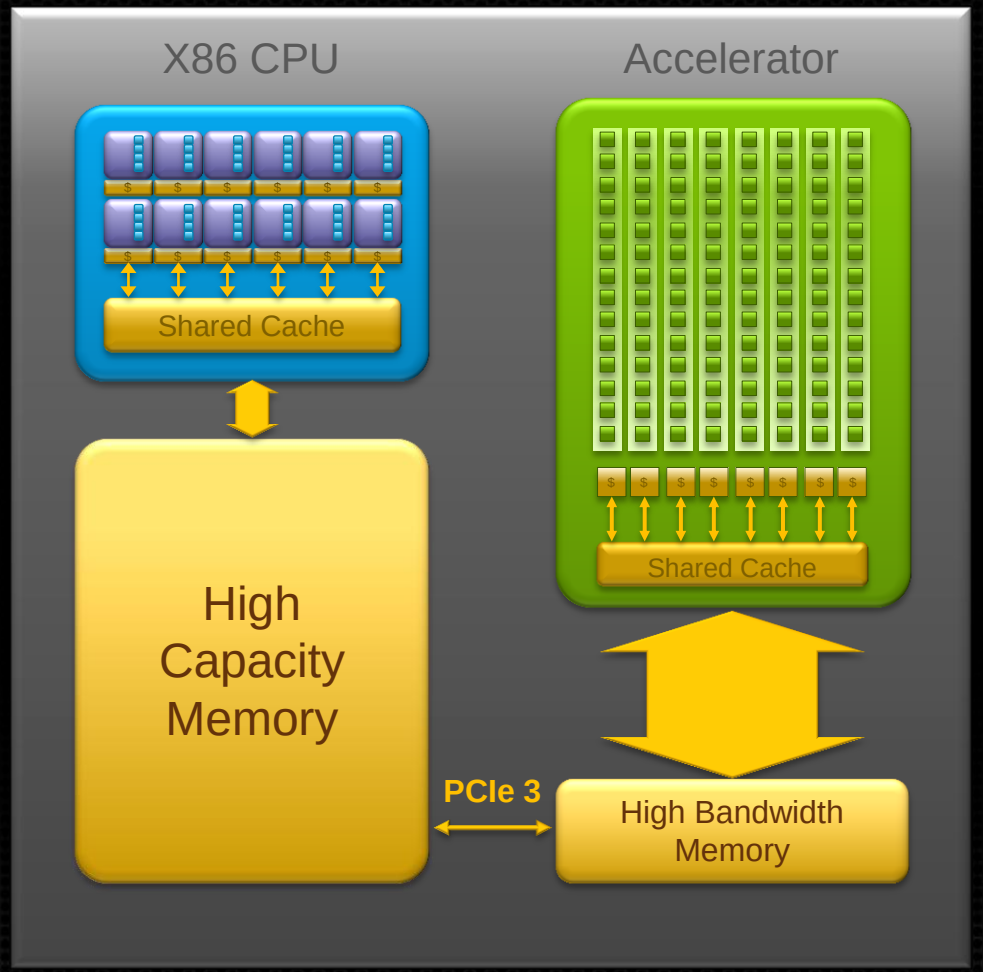
What is Heterogeneous Programming?



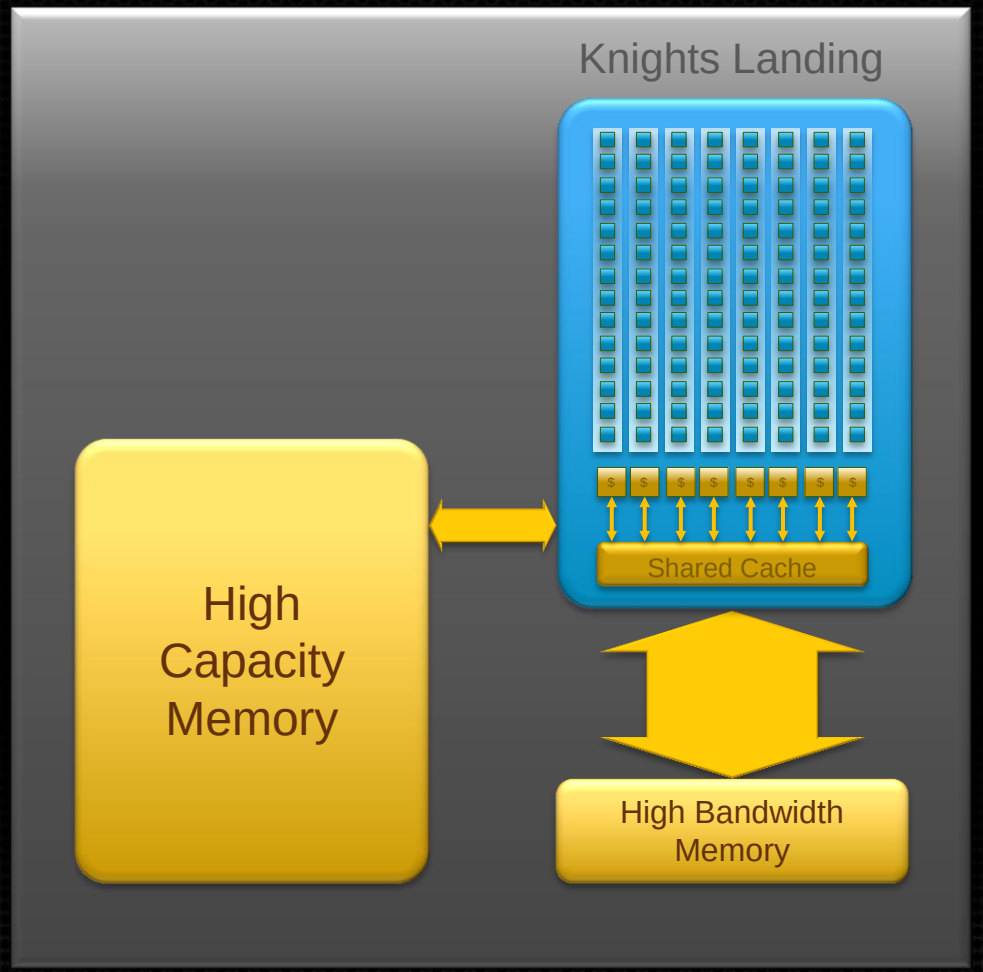
Typical Node Architecture Today



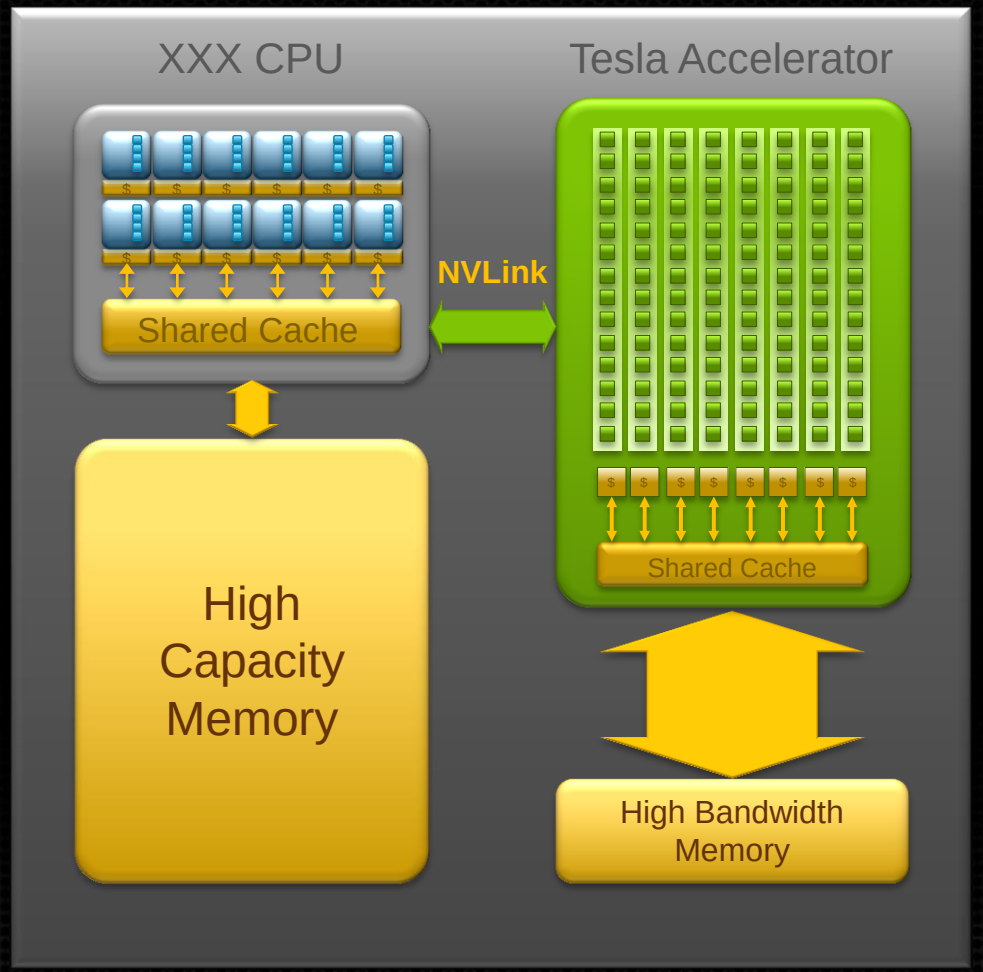
Architecture of an Accelerated Node



Future Architecture 1



Future Architecture 2



NVIDIA Tesla P100 Accelerators

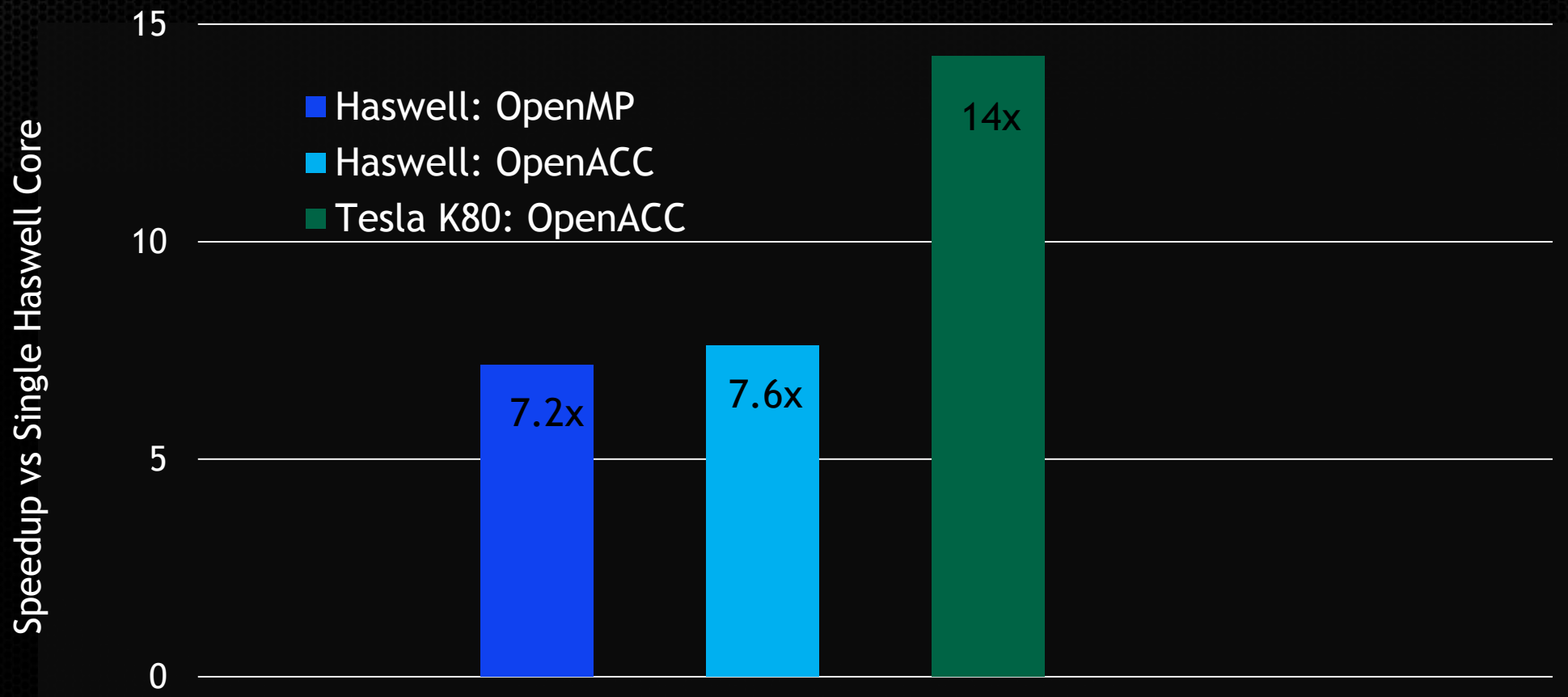
Tesla P100
for NVLink-enabled Servers

5.3 TF DP · 10.6 TF SP · 21 TF HP
720 GB/sec Memory Bandwidth, 16 GB

Tesla P100
for PCIe-Based Servers

4.7 TF DP · 9.3 TF SP · 18.7 TF HP
Config 1: 16 GB, 720 GB/sec
Config 2: 12 GB, 540 GB/sec

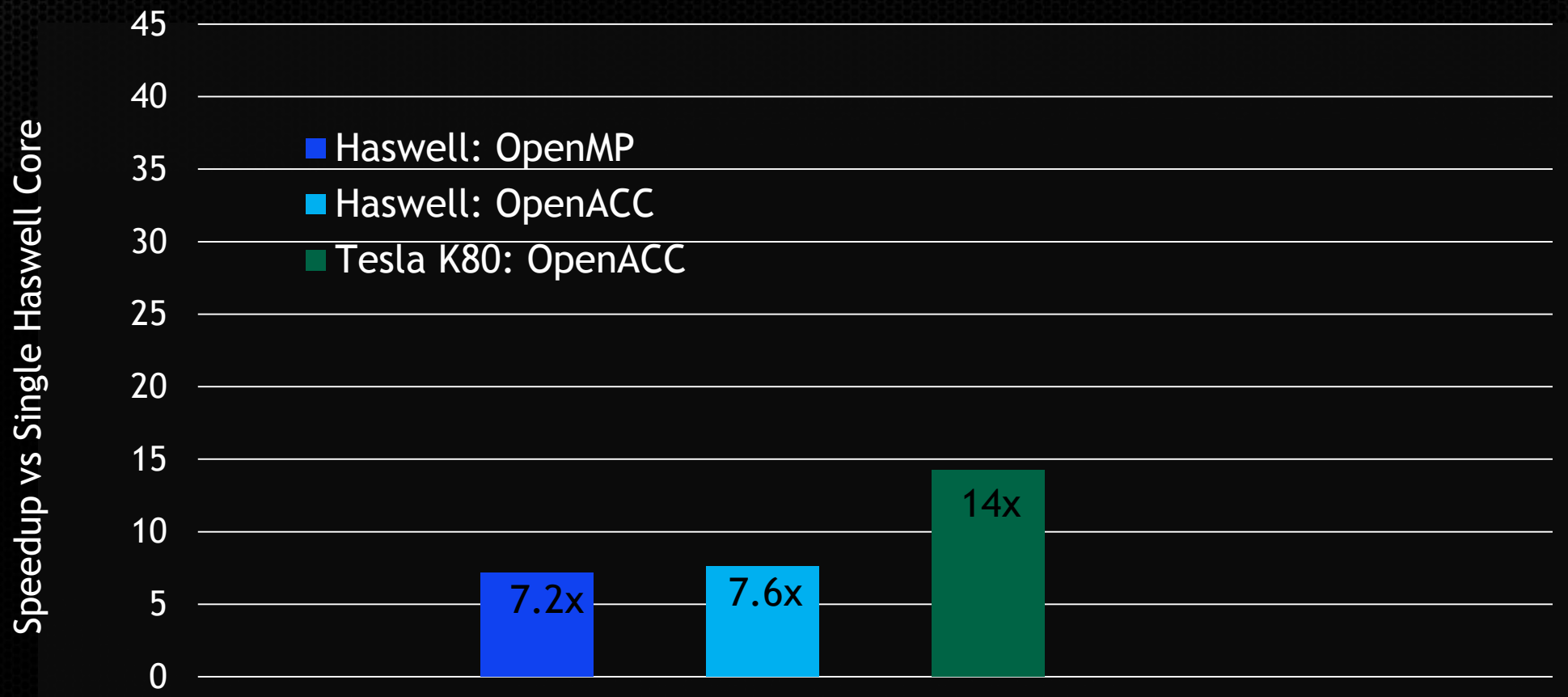
CloverLeaf Performance - Dual Haswell vs Tesla K80



CPU: Intel Xeon E5-2698 v3, 2 sockets, 32 cores, 2.30 GHz, HT disabled
GPU: NVIDIA Tesla K80 (single GPU)
OS: CentOS 6.6, Compiler: PGI 16.5

PGI®

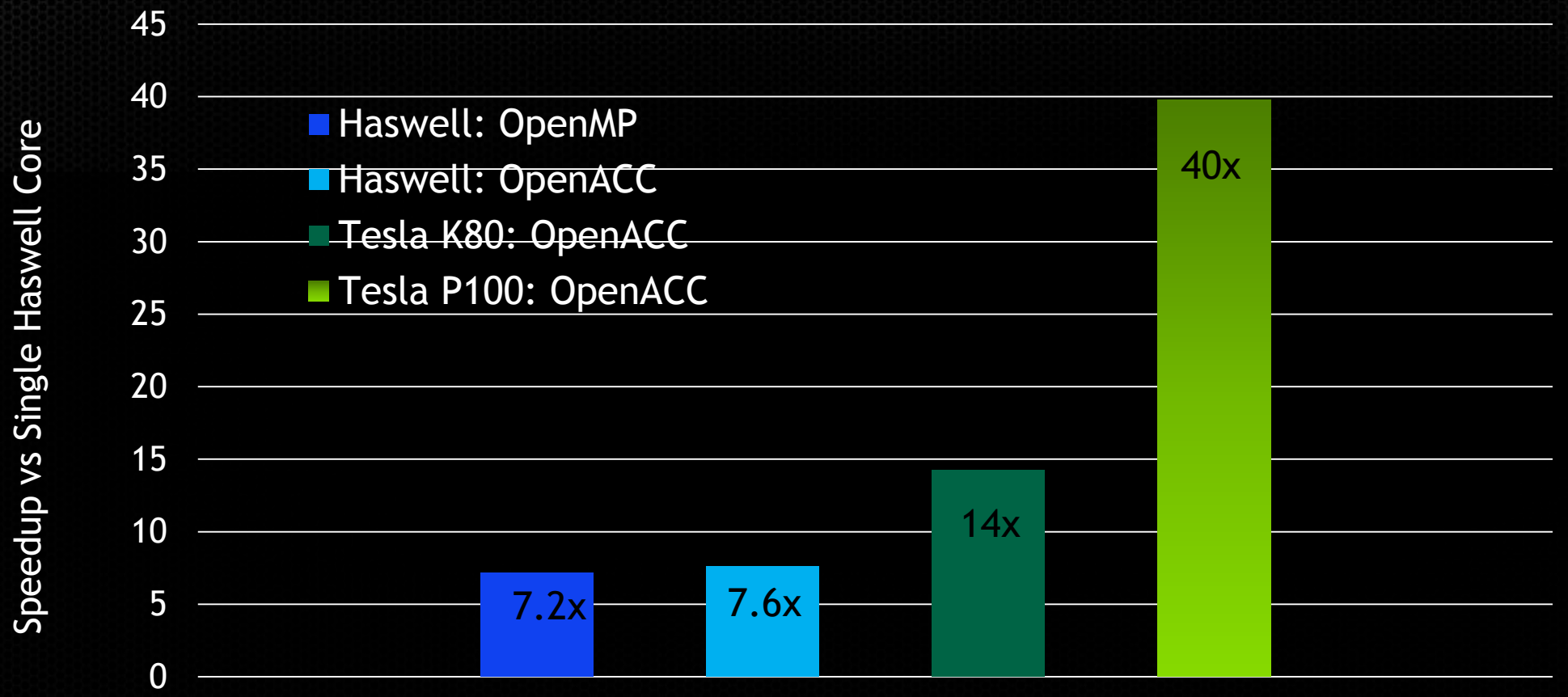
CloverLeaf Performance - Dual Haswell vs Tesla K80



CPU: Intel Xeon E5-2698 v3, 2 sockets, 32 cores, 2.30 GHz, HT disabled
GPU: NVIDIA Tesla K80 (single GPU)
OS: CentOS 6.6, Compiler: PGI 16.5

PGI®

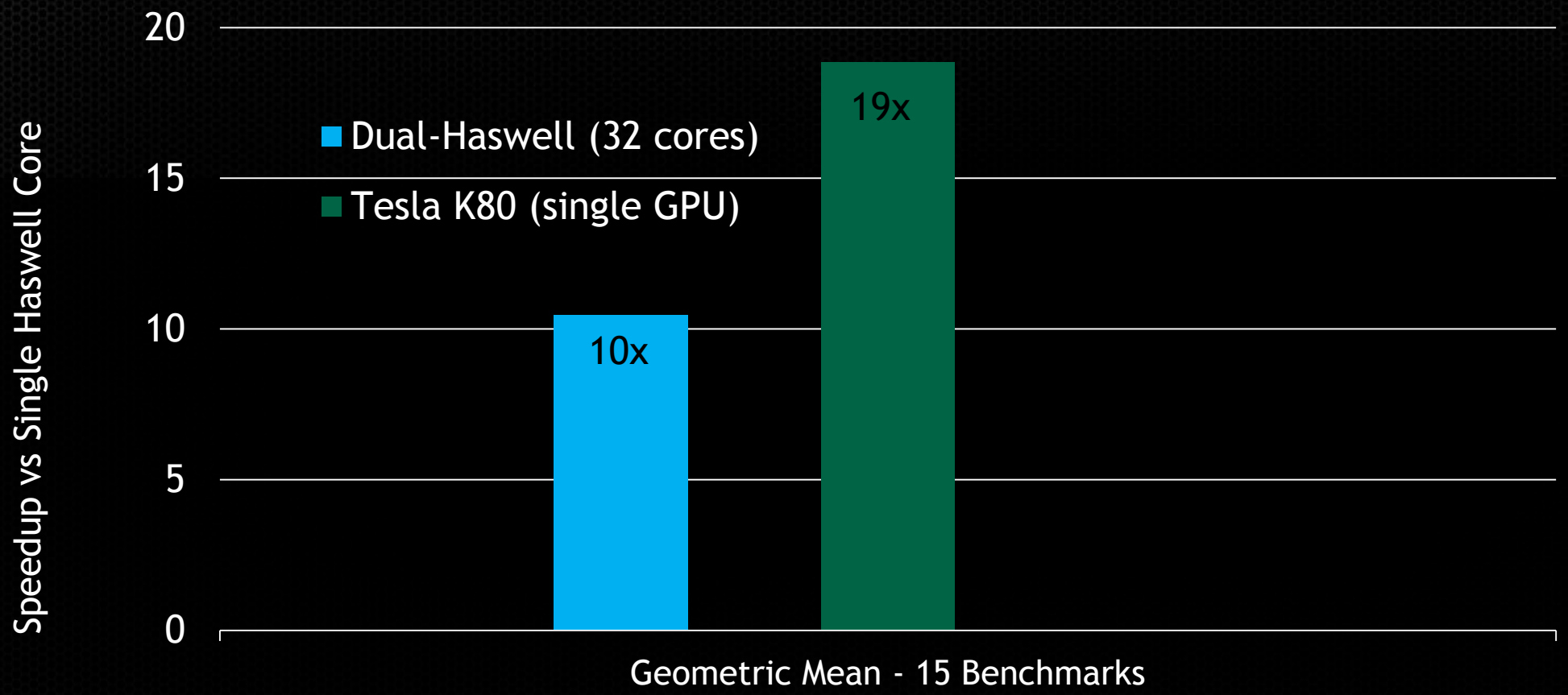
CloverLeaf Performance - Tesla P100 Pascal



CPU: Intel Xeon E5-2698 v3, 2 sockets, 32 cores, 2.30 GHz, HT disabled
GPU: NVIDIA Tesla K80 (single GPU), NVIDIA Tesla P100 (Single GPU)
OS: CentOS 6.6, Compiler: PGI 16.5

PGI®

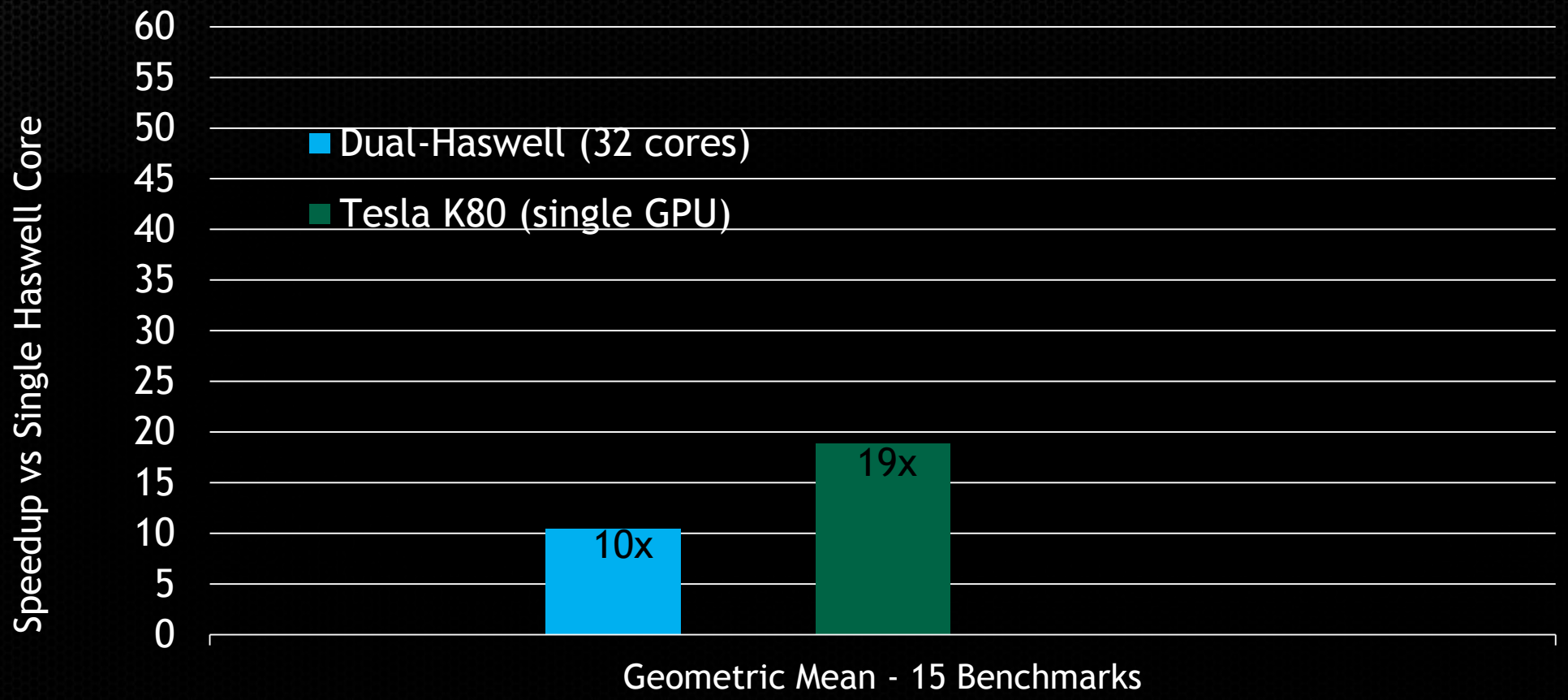
OpenACC SPEC ACCEL Benchmarks – Haswell vs Tesla K80



PGI 16.1 OpenACC Multicore and K80 results from SPEC ACCEL™ measured Feb 2016. PGI 16.7 P100 results measured June 2016.
SPEC® and the benchmark name SPEC ACCEL™ are registered trademarks of the Standard Performance Evaluation Corporation.

PGI®

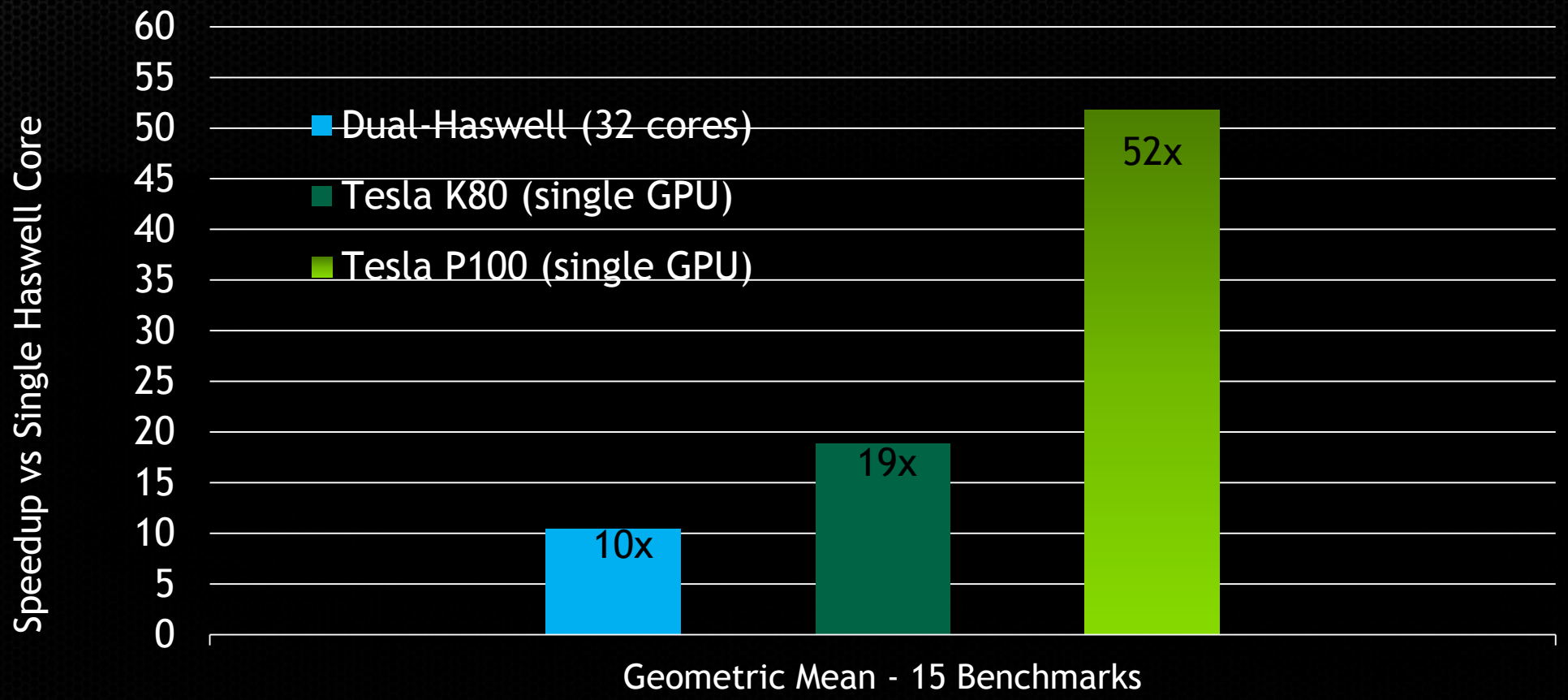
OpenACC SPEC ACCEL Benchmarks – Haswell vs Tesla K80



PGI 16.1 OpenACC Multicore and K80 results from SPEC ACCEL™ measured Feb 2016. PGI 16.7 P100 results measured June 2016.
SPEC® and the benchmark name SPEC ACCEL™ are registered trademarks of the Standard Performance Evaluation Corporation.

PGI®

OpenACC SPEC ACCEL Benchmarks - Tesla P100 Pascal

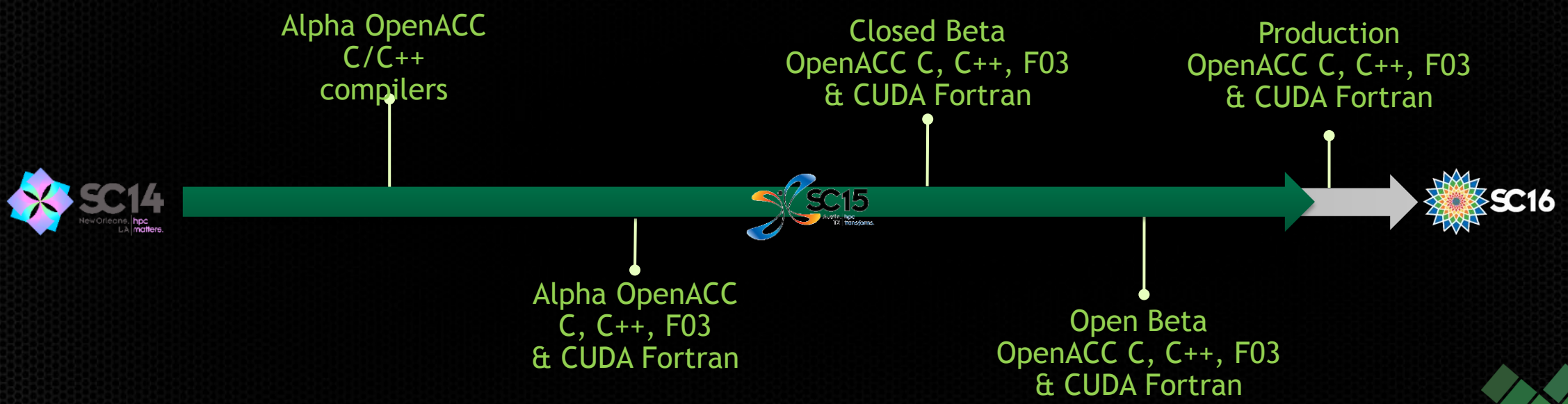


PGI 16.1 OpenACC Multicore and K80 results from SPEC ACCEL™ measured Feb 2016. PGI 16.7 P100 results measured June 2016.
SPEC® and the benchmark name SPEC ACCEL™ are registered trademarks of the Standard Performance Evaluation Corporation.

PGI®

PGI on OpenPOWER+Tesla Timeline

PGI Roadmaps are subject to change without notice



PGI for OpenPOWER+Tesla

- Feature parity with PGI Compilers on Linux/x86+Tesla
- CUDA Fortran, OpenACC, OpenMP, CUDA C/C++ host compiler
- Integrated with IBM's optimized LLVM / OpenPOWER code generator

PGI®

x86

Recompile

Power™

PGI for OpenPOWER+Tesla

- Feature parity with PGI Compilers on Linux/x86+Tesla
- CUDA Fortran, OpenACC, OpenMP, CUDA C/C++ host compiler
- Integrated with IBM's optimized LLVM / OpenPOWER code generator

PGI®

x86

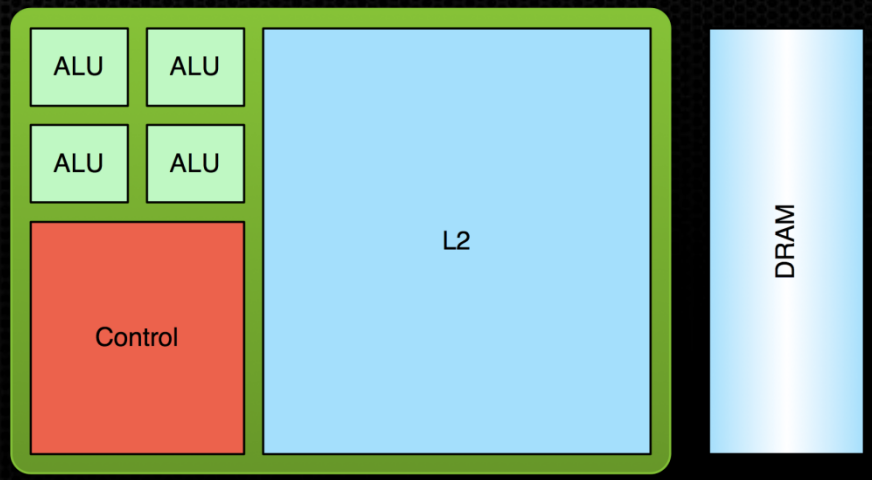
Recompile

Power™

Accelerator Programming Constants

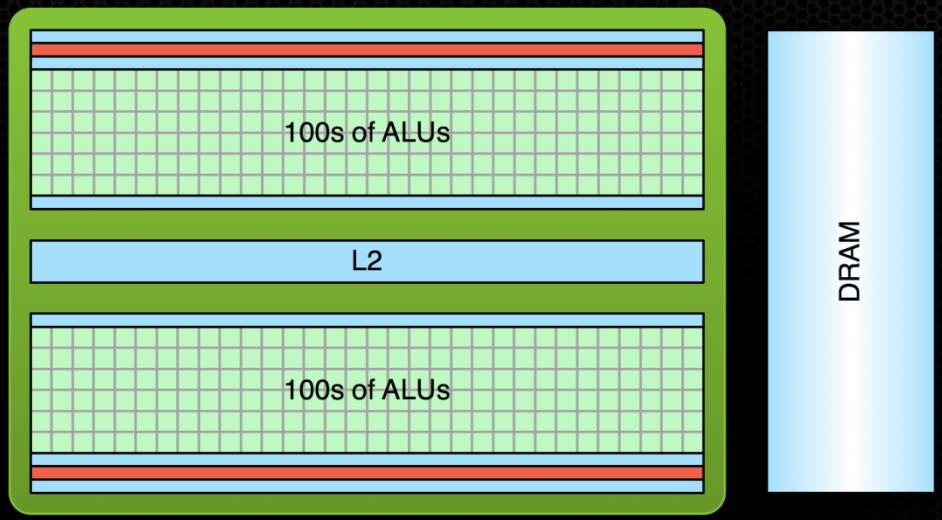
- Allocate data on the GPU
- Move data from host, or initialize data on GPU
- Launch kernel(s)
- Gather results from GPU
- Deallocate data

Low Latency or High Throughput?



CPU

- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution
- 10's of threads

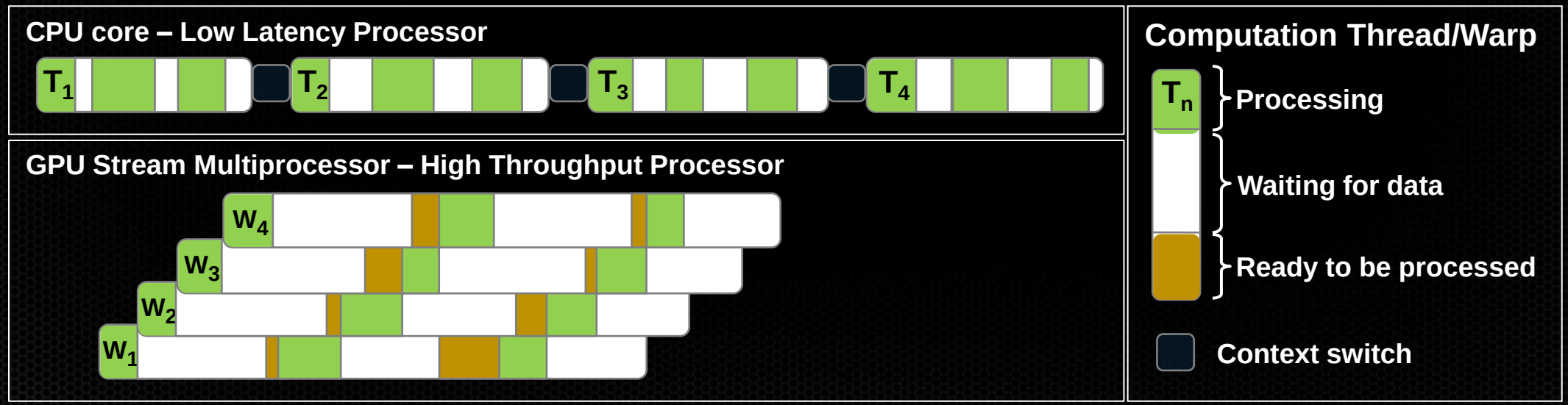


GPU

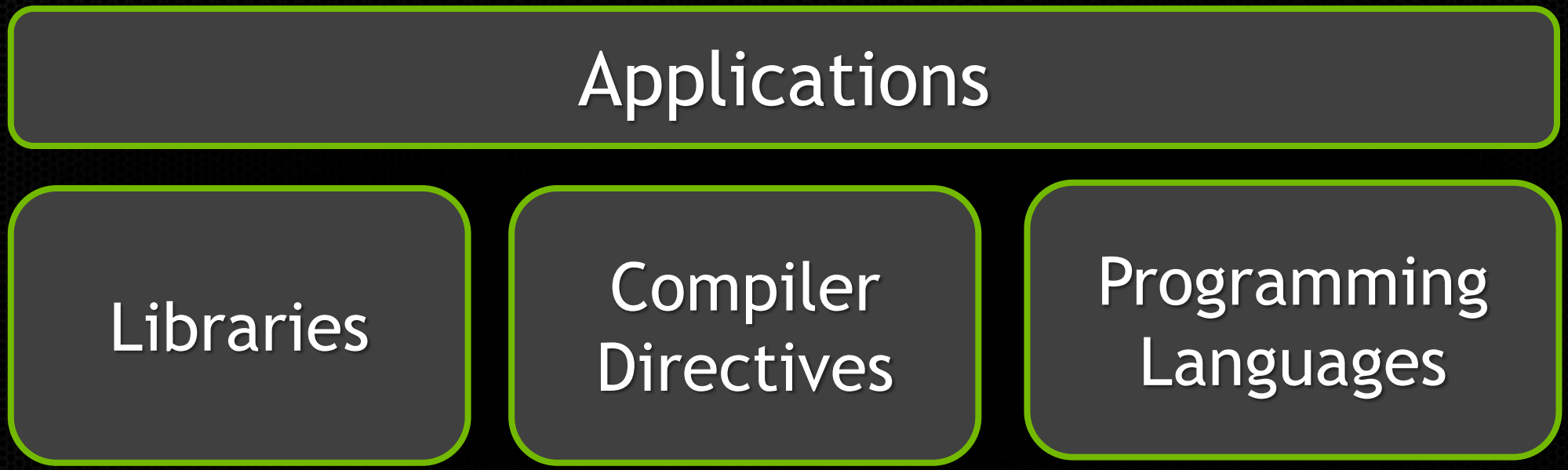
- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation
- 10000's of threads

Low Latency or High Throughput?

- CPU architecture must minimize latency within each thread
- GPU architecture hides latency with computation from other thread warps



3 Ways to Accelerate Applications

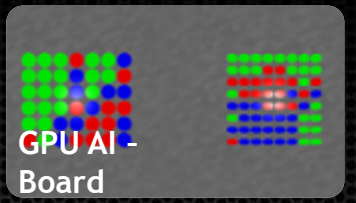


GPU Accelerated Libraries

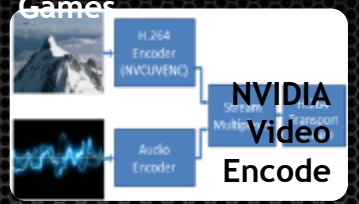
Linear Algebra
FFT, BLAS,
SPARSE, Matrix

Numerical & Math
RAND, Statistics

Data Struct. & AI
Sort, Scan, Zero Sum



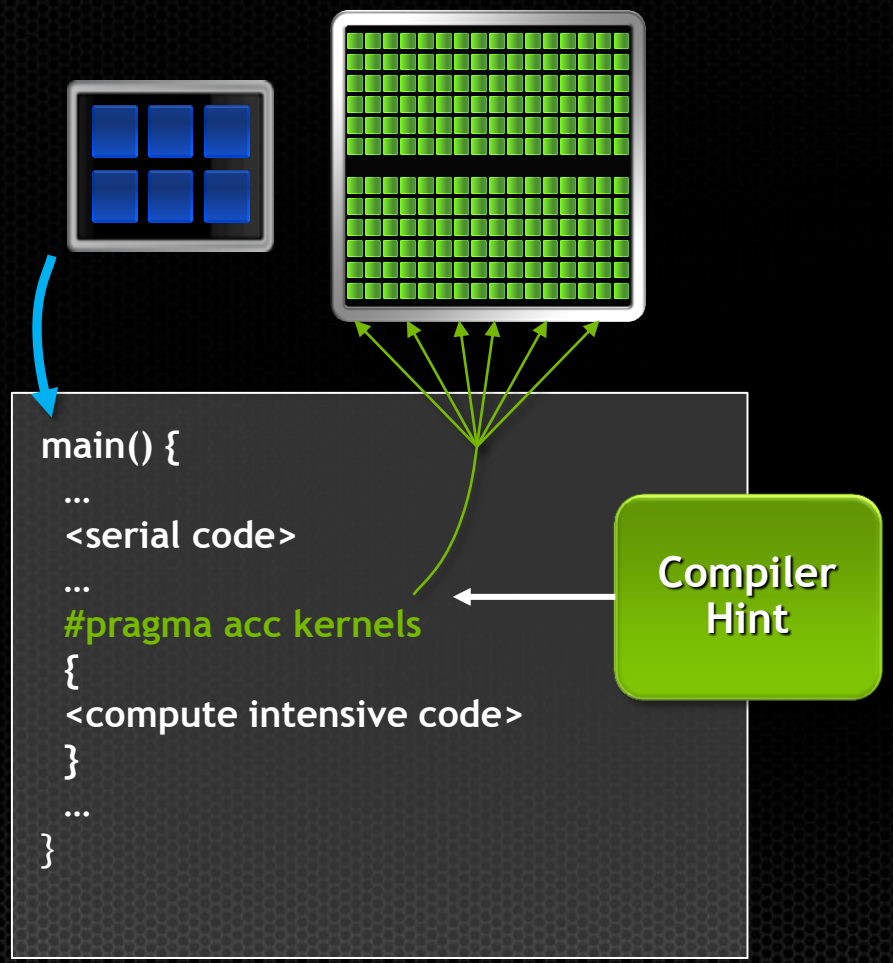
Visual Processing
Image & Video



GPU Programming Languages

Numerical analytics	MATLAB, Mathematica, LabVIEW
Fortran	CUDA Fortran
C	CUDA C
C++	CUDA C++
Python	PyCUDA, Copperhead
F#	Alea.cuBase

OpenACC: Open, Simple, Portable



- Open Standard
- Easy, Compiler-Driven Approach
- Portable on GPUs and Xeon Phi

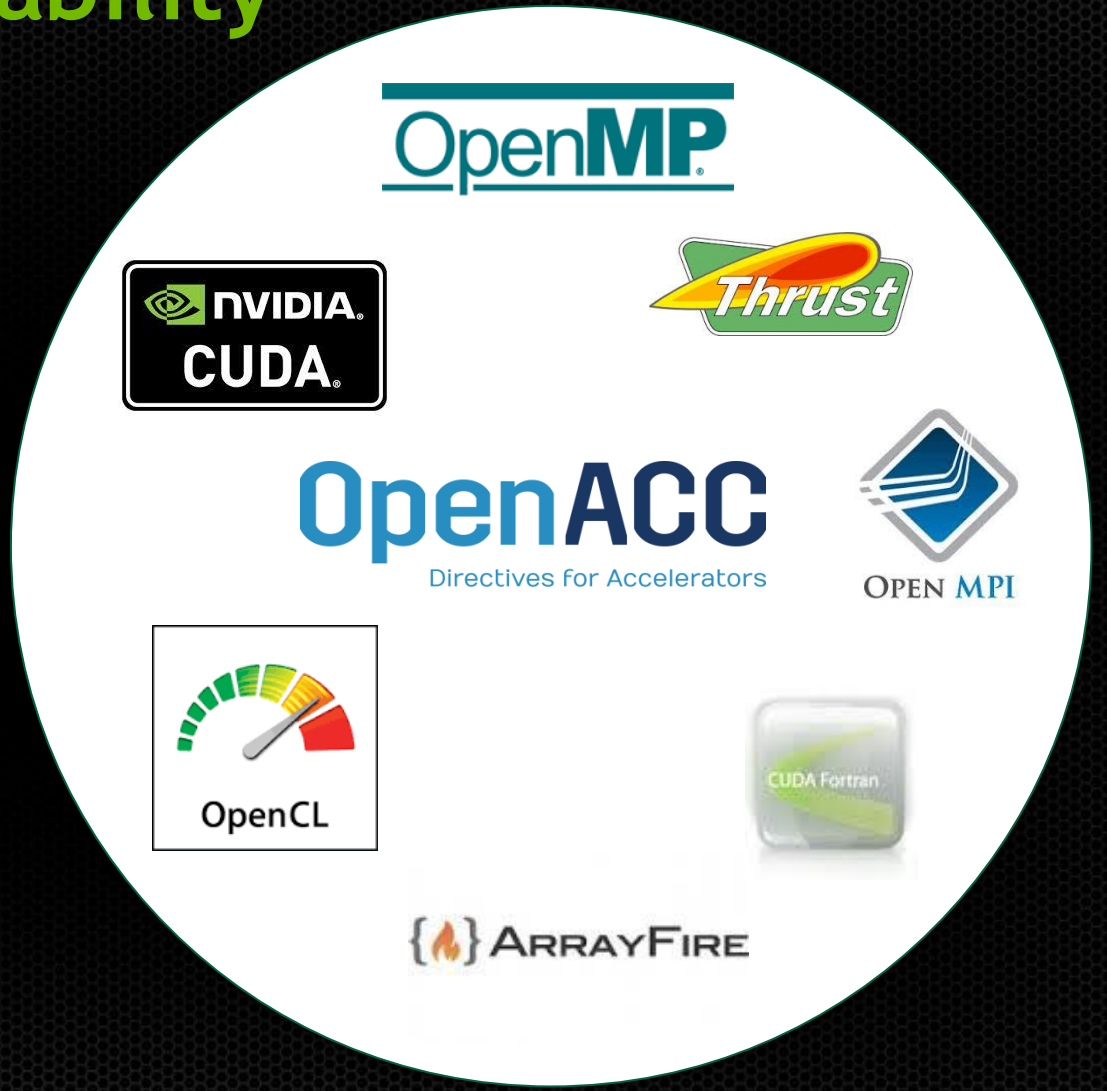
CAM-SE Climate
6x Faster on GPU
Top Kernel: 50% of Runtime

OpenACC
The Standard for GPU Directives

- **Simple:** Directives are the easy path to accelerate compute intensive applications
- **Open:** OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors
- **Powerful:** GPU Directives allow complete access to the massive parallel power of a GPU

OpenACC Interoperability

- OpenACC plays well with others:
  - Add CUDA (C or Fortran), OpenCL, or accelerated libraries to an OpenACC application
  - Write a truly heterogeneous application with MPI, OpenMP and OpenACC
  - Add OpenACC to an existing accelerated application
  - Share data between OpenACC and CUDA

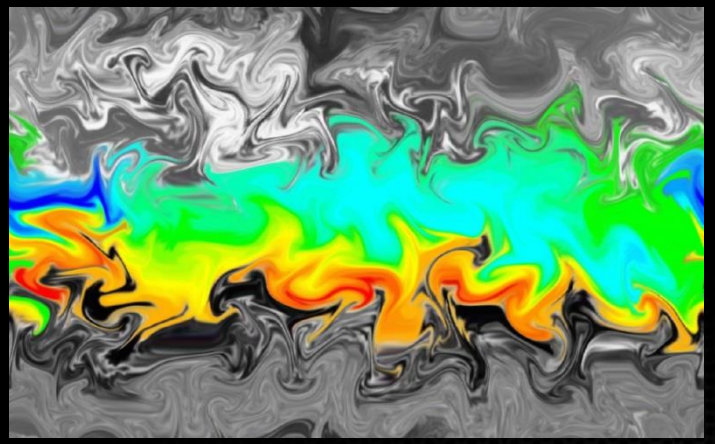


OpenACC Members and Supporters

OpenMP Members

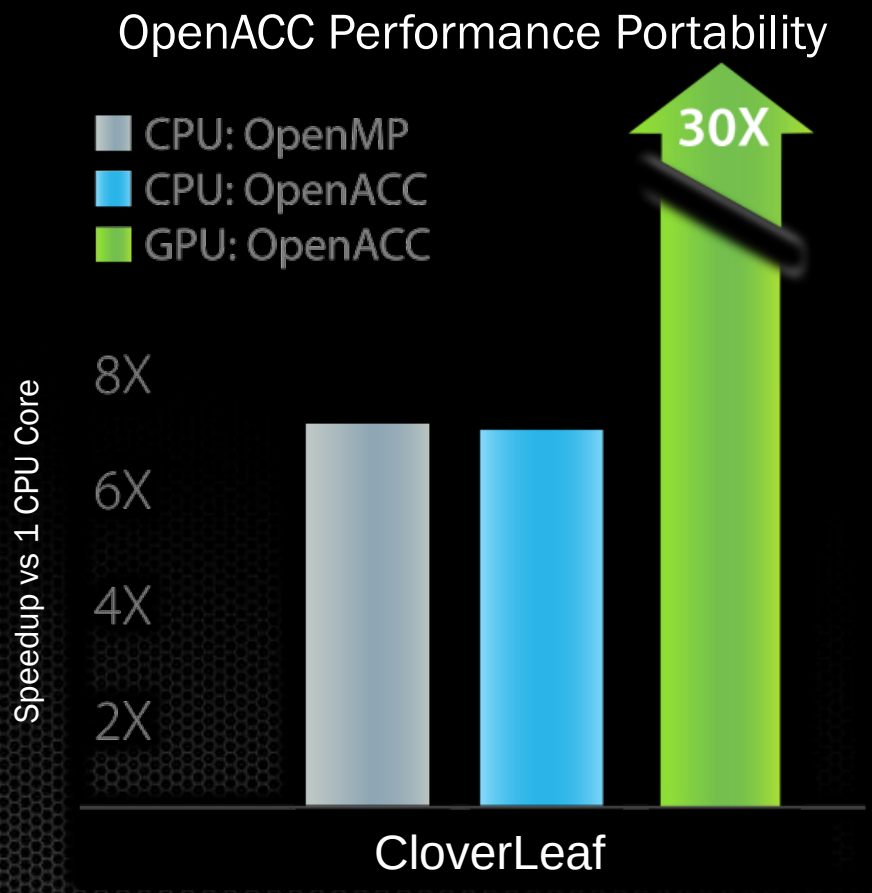
OpenACC Performance Portability: CloverLeaf

Hydrodynamics Application



"We were extremely impressed that we can run OpenACC on a CPU with no code change and get equivalent performance to our OpenMP/MPI implementation"

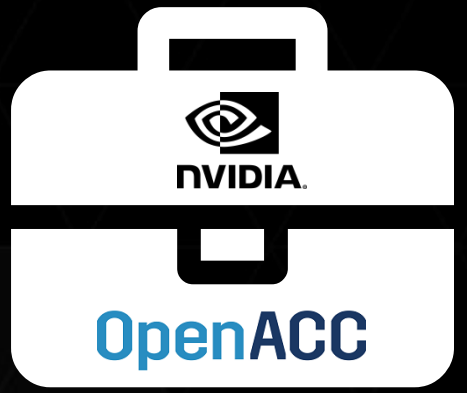
Wayne Gaudin and Oliver Perks
Atomic Weapons Establishment, UK



Benchmarked Intel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz, Accelerator: Tesla K80

Introducing the NVIDIA OpenACC Toolkit

Free Toolkit Offers Simple & Powerful Path to Accelerated Computing



PGI Compiler
Free OpenACC compiler for academia

NVProf Profiler
Easily find where to add compiler directives

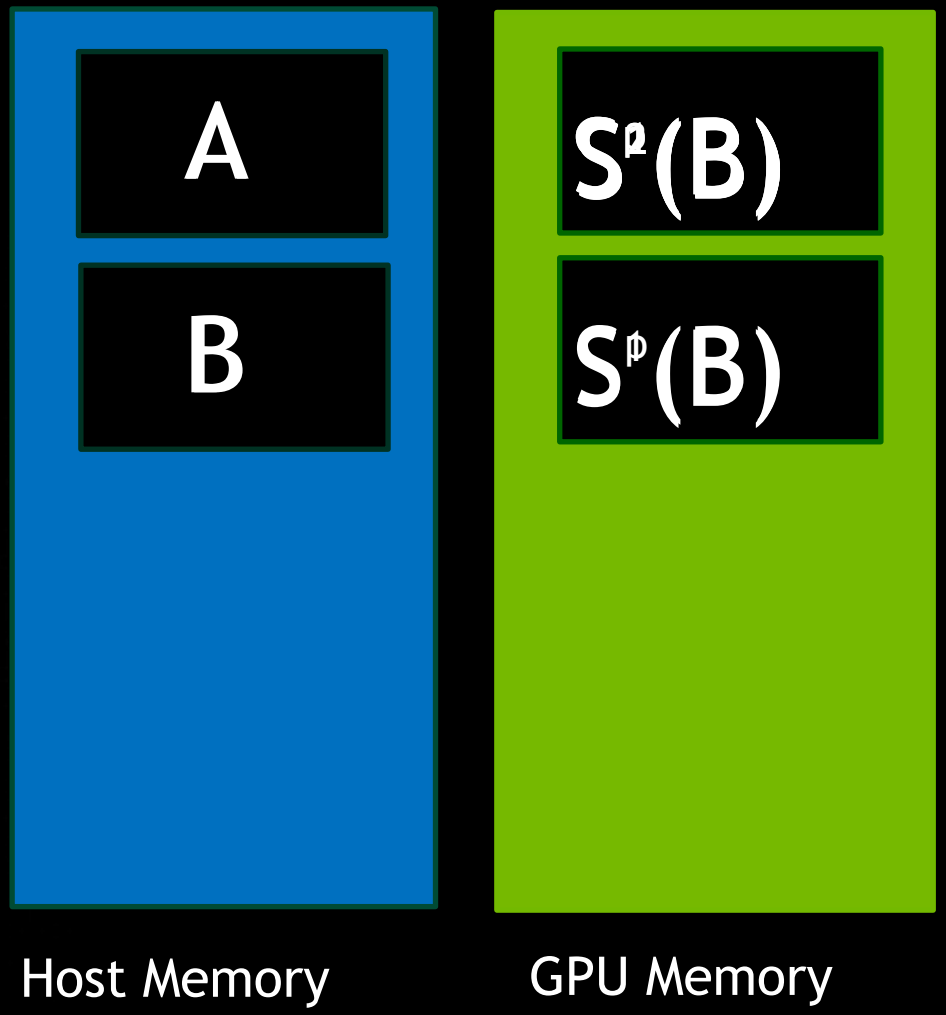
Code Samples
Learn from examples of real-world algorithms

Documentation
Quick start guide, Best practices, Forums

Download at http://www.nvidia.com/openacc

OpenACC Example

```
#pragma acc data \
        copy(b[0:n][0:m]) \
        create(a[0:n][0:m])
{
for (iter = 1; iter <= p; ++iter){
  #pragma acc kernels
  {
  for (i = 1; i < n-1; ++i){
    for (j = 1; j < m-1; ++j){
      a[i][j]=w0*b[i][j]+
              w1*(b[i-1][j]+b[i+1][j]+
                  b[i][j-1]+b[i][j+1])+
              w2*(b[i-1][j-1]+b[i-1][j+1]+
                  b[i+1][j-1]+b[i+1][j+1]);
  } }
  for( i = 1; i < n-1; ++i )
    for( j = 1; j < m-1; ++j )
      b[i][j] = a[i][j];
  }
}
}
```



Basic Syntactic Concepts

Fortran OpenACC directive syntax

- `!$acc` *`directive [clause]...`*
- `&` continuation
- Fortran-77 syntax rules
  - `!$acc` or `C$acc` or `*$acc` in columns 1-5
  - continuation with nonblank in column 6

C/C++ OpenACC directive syntax

- `#pragma acc` *`directive [clause]... eol`*
- continue to next line with backslash

OpenACC compute

- How do you know where to put the directives?
- Are there OpenMP directives already in the code?
- Have you profiled your code?

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

Accelerator Compute 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

Building Accelerator Programs

- `pgfortran -acc a.f90`
- `pgcc -acc a.c`
- `pgc++ -acc a.cpp`
- Other options:
  - `-ta=tesla[:cc2x|cc3x|cc50]`
  - `-ta=tesla[:cuda7.0|cuda7.5]`
  - `-ta=tesla,host` [default with `-acc`]
  - `-ta=radeon`
  - `-ta=tesla,radeon,host`
- *RECOMMENDED:* Enable compiler feedback with `-Minfo` or `-Minfo=accel`

Building Accelerator Programs cont.

- `man pgcc`
- `pgcc -help`
- `pgcc -ta -acc -help`
- `pgcc -Minfo=accel`

- `pgfortran -Mcuda  # Implies CUDA Fortran`
- `pgcc -Mcuda  # Links in CUDA Runtime`
- `pgcc -Mcudalib[=cublas|cufft|curand|cusparse]`

Running your program

- a.out

- No special environment needed
- No dynamic compilation overhead

- pgaccelinfo will show you the accelerators available

Complete, Simple OpenACC Program

```
real a(100), r(100)
a = [ (real(i), i = 1, 100) ]
!$acc kernels loop
do i = 1, 100
    r(i) = a(i) * 2.0
end do
print *,sum(r).eq.100*101
end
```

```
  3, Generating copyout(r(:))
     Generating copyin(a(:))
     Generating Tesla code
  4, Loop is parallelizable
     Accelerator kernel generated
      4, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
```

Hands On Activity (Example 1)

1. Profile the current application using pgprof

```
%> pgc++ -O3 -Minfo=ccff main.cpp -o a.out
%> pgprof -o pgprof.exam1.txt ./a.out
%> pgprof -i pgprof.exam1.txt --cpu-profiling-show-ccff on
```

2. Modify the Makefile to build with OpenACC

-acc	Compile with OpenACC
-ta=tesla	Target NVIDIA GPUS

3. Add parallel loop directives to parallelizable loops
4. Run again:

```
%> time ./a.out
```

Did the application get faster or slower?

Hands On Activity (Example 1)

1. How do we know what happened?
2. Modify the Makefile again

   -Minfo=accel Verbose OpenACC Output

3. Rebuild the application

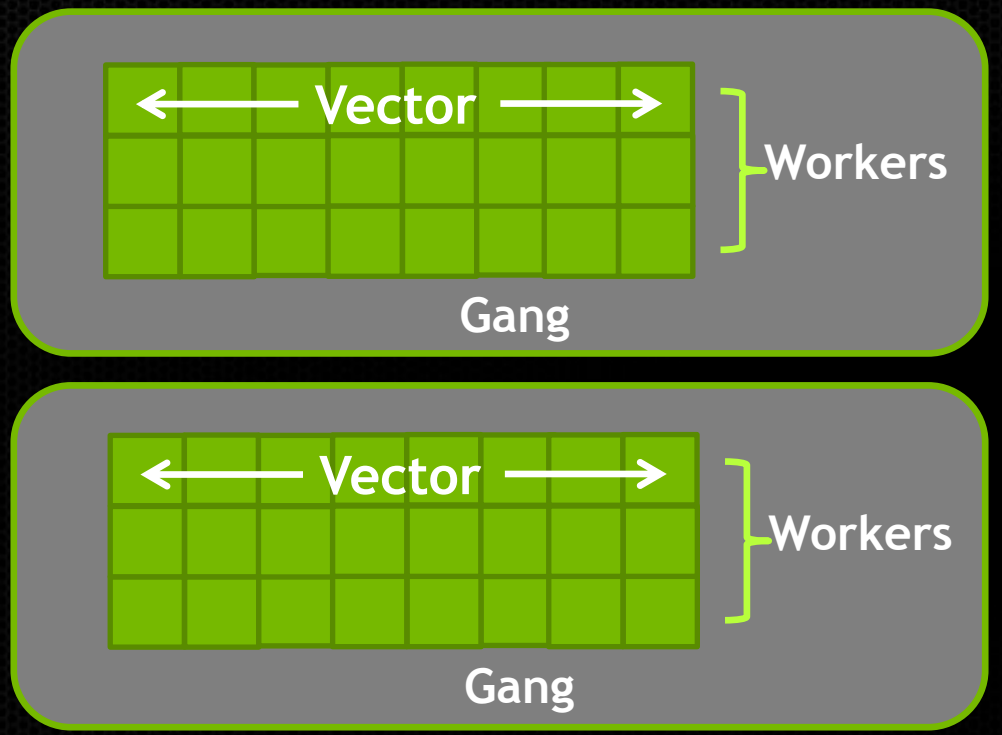
```
% pgc++ -acc -Minfo=accel -ta=tesla main.cpp
main:
     18, Generating copy(b[:10000000])
         Accelerator kernel generated
         Generating Tesla code
         20, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     21, Generating copy(a[:10000000])
         Generating copyin(b[:10000000])
         Accelerator kernel generated
         Generating Tesla code
         23, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     24, Generating copy(a[:10000000],b[:10000000])
         Accelerator kernel generated
         Generating Tesla code
         26, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

Generated 3 Kernels

Accelerator Compute Constructs

- Parallel Construct
- Kernels Construct
- Loop Construct
- Combined Loop Directives
- Other clauses and directives
  - private data
  - reductions
  - collapsing loops
  - caching

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

OpenACC gang, worker, vector Clauses

- gang, worker, and vector can be added to a loop clause
- A parallel region can only specify one of each gang, worker, vector
- Control the size using the following clauses on the parallel region
  - num_gangs(n), num_workers(n), vector_length(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

- GANG:
  - CUDA thread block
  - OpenCL workgroup
  - NVIDIA streaming multiprocessor
  - AMD Radeon vector unit
  - Intel Xeon Phi Coprocessor core
  - multicore chip
- Coarse grain parallelism across ‘core’ or hardware unit
- No synchronization between gangs

gang, worker, vector

- WORKER:
  - CUDA/NVIDIA warp
  - OpenCL subgroup
  - AMD Radeon wavefront
  - IXPC thread
  - multicore core
- fine grain parallelism for latency tolerance, multithreading
- You can usually ignore worker parallelism until you are fine-tuning!

gang, worker, vector

- VECTOR (lane):
  - CUDA thread
  - OpenCL work item
  - Intel Xeon Phi Coprocessor vector lane
  - multicore SSE or AVX lane
- Synchronous parallelism, SIMD parallelism, vector parallelism
- Compiler can remap parallelism to improve performance

OpenACC parallel Directive

- When a parallel region is encountered, a kernel is launched on the device.
- Each kernel can contain one or more gangs, each with one or more workers, where each worker may have a vector containing a fixed number of vector lanes.
- Just as in OpenMP, a parallel directive by itself does no work-sharing, and we say the gang starts executing in *gang-redundant* mode.

Parallel Construct

C

```
#pragma acc parallel num_gangs(100) vector_length(128)
{
    #pragma acc loop gang vector
    for( i = 0; i < n; ++i ) r[i] = a[i]*2.0f;
}
```

Fortran

```
!$acc parallel num_gangs(100) vector_length(128)
      !$acc loop gang vector
      do i = 1,n
          r(i) = a(i) * 2.0
      enddo
!$acc end parallel
```

OpenACC parallel loop Directive

parallel: a parallel region of code. The compiler generates a parallel **kernel** for that region.

loop: identifies a loop that should be distributed across threads

parallel & loop are often placed together

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

Parallel kernel

Kernel:
A function that runs in parallel on the GPU

Parallel Loop Construct

C

```
#pragma acc parallel loop \
    num_gangs(100) vector_length(128) \
    gang vector
    for( i = 0; i < n; ++i ) r[i] = a[i]*2.0f;
```

Fortran

```
!$acc parallel loop &
    num_gangs(100) vector_length(128) gang vector
    do i = 1,n
        r(i) = a(i) * 2.0
    enddo
```

Parallel Construct Clauses

- Data clauses
  - `async( int-expr )`
  - `num_gangs( int-expr )`
  - `num_workers( int-expr )`
  - `vector_length( int-expr )`
  - `if( expr )`
  - `private( var-list ), firstprivate( var-list )`
  - `reduction( operator : var-list )`
- loop clauses for `parallel loop`
  - `gang, worker, vector, seq, auto`
  - `private( var-list )`
  - `reduction( operator : var-list )`

OpenACC Example: SAXPY

SAXPY in C

```
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma acc parallel loop
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}
...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

SAXPY in Fortran

```
subroutine saxpy(n, a, x, y)
  real :: x(n), y(n), a
  integer :: n, i

  !$acc parallel loop
  do i=1,n
    y(i) = a*x(i)+y(i)
  enddo
  !$acc end parallel loop
end subroutine saxpy

...
! Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x, y)
...
```

OpenACC `kernels` construct

The kernels construct expresses that a region may contain parallelism and the compiler determines what can safely be parallelized.

```
#pragma acc kernels
{
  for(int i=0; i<N; i++)
  {
    a[i] = 0.0;
    b[i] = 1.0;
    c[i] = 2.0;
  }

  for(int i=0; i<N; i++)
  {
    a(i) = b(i) + c(i)
  }
}
```

kernel 1

kernel 2

The compiler identifies 2 parallel loops and generates 2 kernels.

Kernels Construct

C

```
#pragma acc kernels
{
    for( i = 0; i < n; ++i ) r[i] = a[i]*2.0f;
}
```

Fortran

```
!$acc kernels
    do i = 1,n
        r(i) = a(i) * 2.0
    enddo
!$acc end kernels
```

Kernels Loop Construct

C

```
#pragma acc kernels loop
    for( i = 0; i < n; ++i ) r[i] = a[i]*2.0f;
```

Fortran

```
!$acc kernels loop
    do i = 1,n
        r(i) = a(i) * 2.0
    enddo
```

In this Fortran case, `!$acc end kernels loop` is optional

Kernels Construct Clauses

- Data clauses
  - `async( int-expr )`
  - `if( expr )`
- loop clauses for kernels loop
  - `gang [( num_gangs )]`
  - `worker [( num_workers )]`
  - `vector [( vector_length )]`
  - `seq`
  - `auto`
  - `private( var-list )`
  - `reduction( operator: var-list )`
  - `independent`

OpenACC `parallel loop` vs. `kernels`

PARALLEL LOOP

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

KERNELS

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

Both approaches are equally valid and can perform equally well.

Loop Directive

C

```
#pragma acc loop clause...
    for( i = 0; i < n; ++i ){
    ....
    }
```

Fortran

```
!$acc loop clause...
    do i = 1, n
    . . .
    end do
```

Loop Directive

```
#pragma acc parallel
{
    #pragma acc loop gang/worker/vector/seq/auto
    for( i = 0; i < n; ++i )
      …
    #pragma acc loop gang/worker/vector/seq/auto
    for( i = 0; i < n; ++i )
       …
}
// gang outermost, worker middle, vector innermost
// auto may choose
```

Loop Directive

```
#pragma acc kernels
{
    #pragma acc loop gang/worker/vector/seq/auto
    for( i = 0; i < n; ++i )
      …
    #pragma acc loop gang/worker/vector/seq/auto
    for( i = 0; i < n; ++i )
      …
}
// gang outermost, worker middle, vector innermost
// auto may choose, auto is default in kernels
```

OpenACC reduction Clause

reduction: specifies a reduction operation and variables for which that operation needs to be applied

```
int sum=0;
#pragma acc parallel loop reduction(+:sum)
for(int i=0; i<N; i++)
{
  ...
  sum+=…
}
```

Reduction clause in OpenMP

```
change = tolerance + 1.0
!$omp parallel shared(change)
do while(change > tolerance)
    change = 0
    !$omp do reduction(max:change) private(i,j)
    do i = 2, m-1
      do j = 2, n-1
        newa(i,j) = w0*a(i,j) + &
            w1 * (a(i-1,j)+a(i,j-1)+a(i+1,j)+a(i,j+1))
        change = max(change,abs(newa(i,j)-a(i,j)))
      enddo
    enddo
    !$omp do private(i,j)
    do i = 2, m-1
      do j = 2, n-1
        a(i,j) = newa(i,j)
      enddo
    enddo
enddo
!$omp end parallel
```

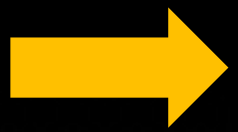
Reduction clause in OpenACC

```
!$acc data create(newa(1:m,1:n)) copy(a(1:m,1:n))
do while(change > tolerance)
    change = 0
    !$acc parallel reduction(max:change)
    !$acc loop
    do i = 2, m-1
      do j = 2, n-1
        newa(i,j) = w0*a(i,j) + &
           w1 * (a(i-1,j)+a(i,j-1)+a(i+1,j)+a(i,j+1))
        change = max(change,abs(newa(i,j)-a(i,j)))
      enddo
    enddo
    !$acc loop
    do i = 2, m-1
      do j = 2, n-1
        a(i,j) = newa(i,j)
      enddo
    enddo
   !$acc end parallel
enddo
!$acc end data
```

OpenACC collapse Clause

collapse(n): Applies the associated directive to the following *n* tightly 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++)
    ...
```

Aliasing Rules Prevent Parallelization

```
23, Loop is parallelizable
        Accelerator kernel generated
        23, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    25, Complex loop carried dependence of 'b->' prevents parallelization
        Loop carried dependence of 'a->' prevents parallelization
        Loop carried backward dependence of 'a->' prevents vectorization
        Accelerator scalar kernel generated
    27, Complex loop carried dependence of 'a->' prevents parallelization
        Loop carried dependence of 'b->' prevents parallelization
        Loop carried backward dependence of 'b->' prevents vectorization
        Accelerator scalar kernel generated
```

OpenACC independent clause

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

```
#pragma acc kernels
{
#pragma acc loop independent
for(int i=0; i<N; i++)
{
  a[i] = 0.0;
  b[i] = 1.0;
  c[i] = 2.0;
}
#pragma acc loop independent
for(int i=0; i<N; i++)
{
  a[i] = b[i] + c[i]
}
}
```

kernel 1

kernel 2

The compiler identifies 2 parallel loops and generates 2 kernels.

C99: `restrict` Keyword

- Declaration of intent given by the programmer to the compiler
  - Applied to a pointer, e.g.

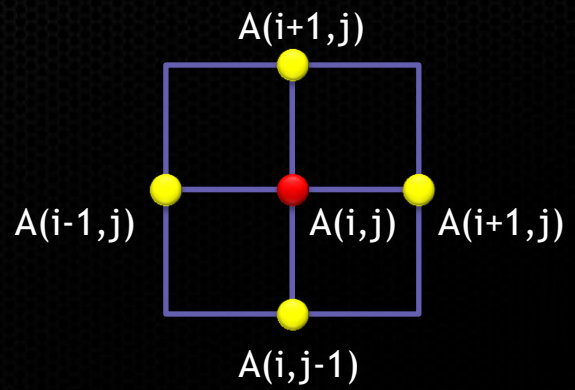
```
float *restrict ptr
```

  - Meaning: “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”*

- Limits the effects of pointer aliasing
- OpenACC compilers often require `restrict` to determine independence
  - Otherwise the compiler can’t parallelize loops that access `ptr`
  - Note: if programmer violates the declaration, behavior is undefined

http://en.wikipedia.org/wiki/Restrict

Hands on Activity (Example 2)



$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

- Given a 2D grid
  - Set every vertex equal to the average of neighboring vertices
    - Repeat until converged
  - Common algorithmic pattern

Hands on Activity (Example 2)

1. Build & Run
2. Use pgprof to identify the largest bottlenecks
3. Use what you have learned to parallelize the largest function
4. Can the second largest function be parallelized?
    1. Use the reduction clause to parallelize the error function

Hands On Activity (Example 2)

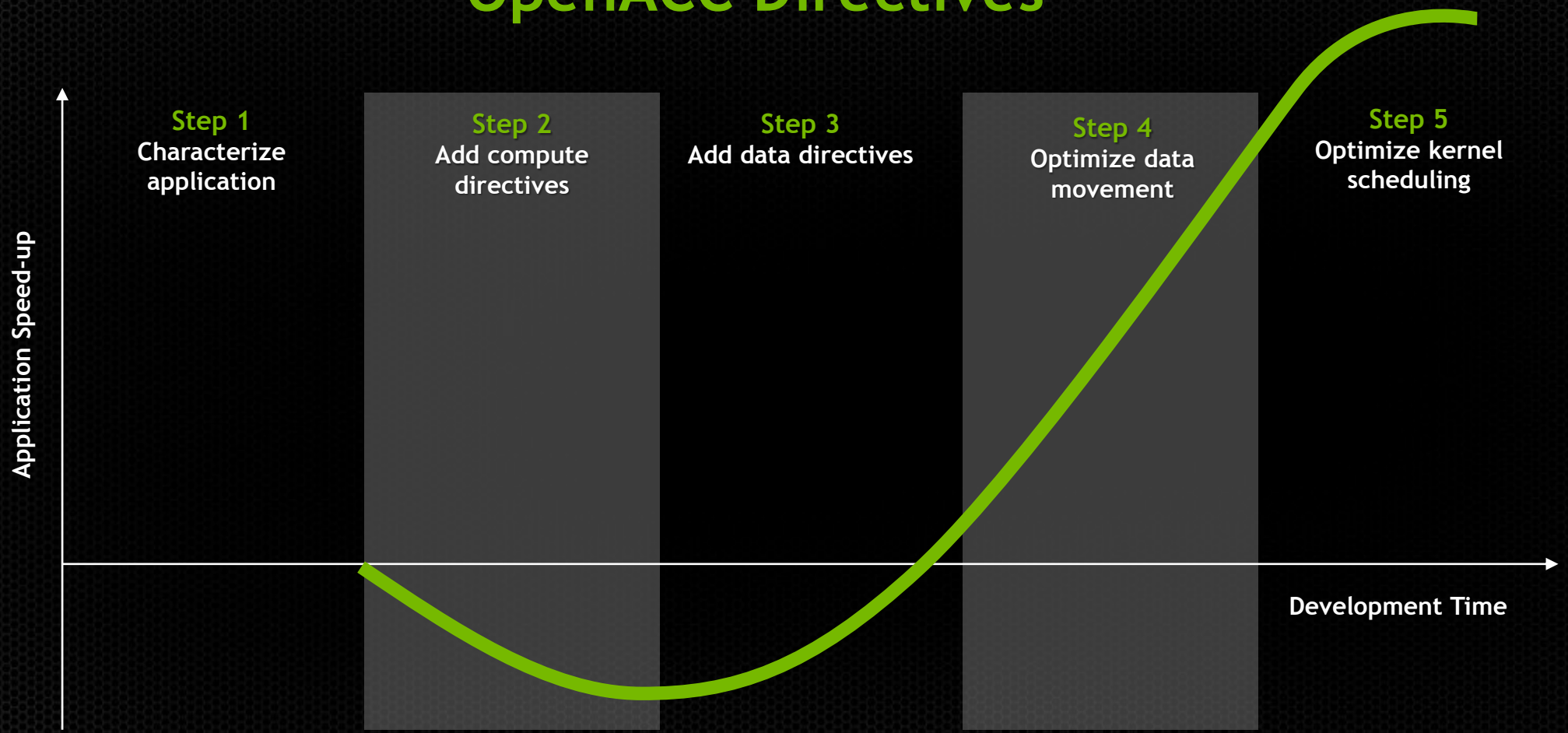
1. Use the collapse clause to parallelize the inner and outer loops
   - Did you see any performance increase?

Note that you can use pgprof to see both the GPU and CPU performance

Hands On Activity (Example 2)

1. Replace collapse clause with some combination of gang/worker/vector
2. Experiment with different sizes using num_gangs, num_workers, and vector_length
   - What is the best configuration that you have found?

Typical Porting Experience with OpenACC Directives



The Need for Data Management

- The compiler can discover the need for data movement automatically in some cases and insert the appropriate runtime calls
- The shapes and sizes of arrays or data structures may not be apparent to the compiler
- Definite performance advantages to move data regions “out” and avoid data transfer at every kernel launch
- Scalars are generally handled optimally; loop variables are private by default, scalars are firstprivate
- Arrays, even very small ones, are global (shared) by default
- Most OpenACC implementations use a present table

OpenACC Data Management

- Data constructs, data regions, data lifetimes
  - data construct is the lexical scope of ‘acc data’
  - data region is dynamic scope of ‘acc data’
  - data lifetime is dynamic scope from device allocate to deallocate
- data clauses
  - copy, copyin, copyout, create
    - no data allocate or movement(!) if already present
    - Otherwise allocate at data construct entry, add to present table
    - Then deallocate at data construct exit, remove from present table
    - copyin+copyout dependent on clause
  - present
    - must appear in the present table

Performance Goals

- Data movement between Host and Accelerator
  - minimize amount of data
  - minimize number of data moves
  - minimize frequency of data moves
  - optimize data allocation in device memory
- Use the –Minfo compiler option and scan output to compare your reasoning to the compiler's.

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.

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

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

Data Region

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

Data Clauses

Starting in PGI 15.1, all copy and create clauses behave as their "present_or" variants

`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.

OpenACC enter exit Directives

enter: Defines the start of an unstructured data region

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

exit: Defines the end of an unstructured data region

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

- Used to define data regions when scoping doesn't allow the use of normal data regions (e.g. The constructor/destructor of a class).
- Every variable in an enter clause should also appear in an exit clause

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

Dynamic Data Lifetimes

- C

```
#pragma acc enter data copyin( list )
#pragma acc enter data create( list )
```

- Fortran

```
!$acc enter data copyin( list )
!$acc enter data create( list )
```

- Starts a data lifetime (not a data region)
- appears in present table
- `exit data delete( list )  copyout( list )`

OpenACC update Directive

update: Explicitly transfers data between the host and the device

Useful when you want to update data in the middle of a data region

Clauses:

device: copies from the host to the device

host: copies data from the device to the host

```
#pragma acc update host(x[0:count])
MPI_Send(x,count,datatype,dest,tag,comm);
```

Update Directive

- C

```
#pragma acc update host( list )
#pragma acc update device( list )
```

- Fortran

```
!$acc update host( list )
!$acc update device( list )
```

  - data must be in a data clause for an enclosing data region
  - may be noncontiguous
  - implies `present( list )`
  - both may be on a single line
    - `update host( list ) device( list )`

Array Shaping

- Compiler sometimes cannot determine size of arrays
  - Must specify explicitly using 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:end)), copyout(b(s/4:3*s/4))
```

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

Fortran Array Sections

- `a(1:n,1:m) … copy(a(1:n,1:m))`
  - allocates full array (1:n,1:m)
  - copies full array (1:n,1:m)
- `a(1:n,1:m)…` contiguous subarray `copy(a(1:n,2:m-1))`
  - allocates subarray (1:n,2:m-1)
  - copies subarray (1:n,2:m-1)
  - `a(1:n,1:m)` is NOT present , `a(1:n,2:m-1)` is present
- `a(1:n,1:m)…` noncontiguous `copy(a(2:n-1,2:m-1))`
  - allocates contiguous bounding subarray (1:n,2:m-1)
  - copies subarray (2:n-1,2:m-1)
  - `a(1:n,2:m-1)` IS present, but may not be up to date
  - data copies may take longer

C Array Sections

- `float r[100][200];  copy(r[0:100][0:200])`
  - allocates rectangular array [100][200]
  - copies subarray [0:100][0:200]
- `float r[100][200];  copy(r[1:n][0:200])`
  - allocates subarray [1:n][200] ([n-1][200])
  - copies subarray [1:n][0:200]
- `float r[100][200];  copy(r[1:n][1:m])`
  - allocates bounding subarray [1:n][200]
  - copies subarray [1:n][1:m]

C Array Sections

- `float *r[100]; copy(r[0:100][0:200])`
  - allocates vector of pointers [100]
  - allocates rectangular array [100][200]
  - fills in pointers [100]
  - copies subarray [0:100][0:200]
- `float *r[100]; copy(r[1:n][0:200])`
  - allocates vector of pointers [n-1]
  - allocates rectangular subarray [1:n][200]
  - fills in pointers [1:n]
  - copies subarray [1:n][0:200]
- `float *r[100]; copy(r[1:n][1:m])`
  - allocates vector of pointers [n-1]
  - allocates rectangular subarray [1:n][0:m]
  - fills in pointers [1:n]
  - copies subarray [1:n][1:m]

C Array Sections

- `float **r;  copy(r[0:100][0:200])`
  - allocates vector of pointers [100]
  - allocates rectangular array [100][200]
  - fills in pointers [100]
  - copies subarray [0:100][0:200]
- `float **r; copy(r[1:n][0:200])`
  - allocates vector of pointers [n-1]
  - allocates rectangular subarray [1:n][200]
  - fills in pointers [1:n]
  - copies subarray [1:n][0:200]
- `float **r; copy(r[1:n][1:m])`
  - allocates vector of pointers [n-1]
  - allocates rectangular subarray [1:n][0:m]
  - fills in pointers [1:n]
  - copies subarray [1:n][1:m]

Hands On Activity (Example 2)

1. Modify the code to add use enter/exit data instead of a structured data region

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[0:10]
  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

Fortran - Privatization of Local Arrays

```
!$acc kernels
do i = 1, M
  do j = 1, N
    do jj = 1, 10
      tmp(jj) = jj
    end do
    A(i,j) = sum(tmp)
  end do
end do
!$acc end kernels
```

```
% pgf90 -ta=nvidia -Minfo=accel  private.f90
privatearr:
     10, Generating copyout(tmp(1:10))
         Generating compute capability 1.0 binary
         Generating compute capability 1.3 binary
     11, Parallelization would require privatization of array 'tmp(1:10)'
     13, Parallelization would require privatization of array 'tmp(1:10)'
         Sequential loop scheduled on host
     14, Loop is parallelizable
         Accelerator kernel generated
         14, !$acc do parallel, vector(10)
     17, Loop is parallelizable
         Accelerator kernel generated
         17, !$acc do parallel, vector(10)
             Sum reduction generated for tmp$r
```

Compiler automatically privatizes scalars, but not arrays.

Privatization of Local Arrays - cont.

```
!$acc kernels
do i = 1, M
  !$acc loop private(tmp)
  do j = 1, N
    do jj = 1, 10
      tmp(jj) = jj
    end do
    A(i,j) = sum(tmp)
  end do
end do
!$acc end kernels
```

```
% pgf90 -ta=nvidia,time -Minfo=accel private2.f90
privatearr2:
     10, Generating copyout(a(1:1024,1:1024))
         Generating compute capability 1.0 binary
         Generating compute capability 1.3 binary
     11, Loop is parallelizable
     13, Loop is vectorizable
         Accelerator kernel generated
         11, !$acc do parallel, vector(16)
         13, !$acc do vector(16)
     14, Loop is parallelizable
     17, Loop is parallelizable
```

You need to privatize local temporary arrays. The default is to assume they are shared.

Nested Private

```
float x, y[10], p, q[10];

#pragma acc parallel private(x, y)
{
   // x and y are private to the gang
   #pragma acc loop gang
   for( i = 0; i < n; ++i ){
     // iterations shared across gangs
     // x and y are private to the gang

     #pragma acc loop vector private(p, q)
     for( j = 0; j < m; ++j ){
       // p and q are private to vector lane
     }
   }
}
```

“live-out” Variables

```
!$acc kernels
   do i = 1, M
      do j = 1, N
         idx = i+j
         A(i,j) = idx
      enddo
   enddo
!$acc end kernels
print *, idx, A(1,1), A(M,N)
```

```
% pgf90 -ta=nvidia,time -Minfo=accel  liveout.f90
liveout:
     11, Generating copyout(a(1:1024,1:1024))
     12, Loop is parallelizable
         Accelerator kernel generated
         12, !$acc do parallel, vector(256)
     13, Inner sequential loop scheduled on accelerator
     14, Accelerator restriction: induction variable live-out from loop: idx
     15, Accelerator restriction: induction variable live-out from loop: idx
```

```
!$acc kernels
   do i = 1, M
!$acc loop private(idx)
      do j = 1, N
         idx = i+j
         A(i,j) = idx
      enddo
   enddo
!$acc end kernels
print *, idx, A(1,1), A(M,N)
```

```
% pgf90 -ta=nvidia,time -Minfo=accel liveout2.f90
liveout2:
     10, Generating copyout(a(1:1024,1:1024))
     11, Loop is parallelizable
     13, Loop is parallelizable
         Accelerator kernel generated
         11, !$acc do parallel, vector(16)
         13, !$acc do parallel, vector(16)
```

Global data

```
float a[1000000];

extern void matvec(…);
…

for (i = 0; i < m; ++i) {
   matvec(v, x, i, n);
}
```

```
extern float a[];

void matvec (float* v, float* x,
int I, int n) {
   for (int j = 0; n < n; ++j)
      x[i] += a[i*n+j] * v[j];
}
```

declare create

```
float a[1000000];
#pragma acc declare create(a)

#pragma acc routine worker
extern void matvec(…);
…

#pragma acc parallel loop
for (i = 0; i < m; ++i) {
   matvec(v, x, i, n);
}
```

```
extern float a[];
#pragma acc declare create(a)

#pragma acc routine worker
void matvec (float* v, float* x,
int I, int n) {
   #pragma acc loop worker
   for (int j = 0; n < n; ++j)
      x[i] += a[i*n+j] * v[j];
}
```

declare device_resident

```
float a[1000000];
#pragma acc declare \
device_resident(a)

#pragma acc routine worker
extern void matvec(…);
…

#pragma acc parallel loop
for (i = 0; i < m; ++i) {
   matvec(v, x, i, n);
}
```

```
extern float a[];
#pragma acc declare \
device_resident(a)

#pragma acc routine worker nohost
void matvec (float* v, float* x,
int I, int n) {
   #pragma acc loop worker
   for (int j = 0; n < n; ++j)
      x[i] += a[i*n+j] * v[j];
}
```

declare link

```
float a[1000000];
#pragma acc declare link(a)

#pragma acc routine worker
extern void matvec(…);
…

#pragma acc parallel loop \
            copyin(a[0:n*m])
for (i = 0; i < m; ++i) {
   matvec(v, x, i, n);
}
```

```
extern float a[];
#pragma acc declare link(a)

#pragma acc routine worker
void matvec (float* v, float* x,
int I, int n) {
   #pragma acc loop worker
   for (int j = 0; n < n; ++j)
      x[i] += a[i*n+j] * v[j];
}
```

Global Data Review

```
float x[1000];
#pragma acc declare create(x)
// static allocation, host + device

float y[1000];
#pragma acc declare device_resident(y)
// static allocation, device-only

float z[10000];
#pragma acc declare link(z)
// static allocation on host, dynamic allocation on device
```

Fortran Global Data

```
module m
 real, allocatable :: x(:)
 !$acc declare create(x)
 !  x dynamically allocated on host+device

 real, allocatable :: y(:)
 !$acc declare device_resident(y)
 !  y dynamically allocated on device only

end module
```

OpenACC host_data directive

host_data use_device(list):

makes the address of the device data available on the host

Useful for GPU aware libraries (e.g. MPI, CUBLAS, etc)

```
#pragma acc data copy(x)
{
  // x is a host pointer here
  #pragma acc host_data use_device(x)
  {
    // x is a device pointer here
    MPI_Send(x,count,datatype,dest,tag,comm)
  }
  // x is a host pointer here
}
```

CUDA Library Calls via Host_Data

OpenACC Main Calling CUBLAS

```
int N = 1<<20;
float *x, *y
// Allocate & Initialize X & Y
...
cublasInit();
#pragma acc data copyin(x[0:N]) copy(y[0:N])
{
  #pragma acc host_data use_device(x,y)
  {
    // Perform SAXPY on 1M elements
    cublasSaxpy(N, 2.0, x, 1, y, 1);
  }
}
cublasShutdown();
```

OpenACC can interface with existing GPU-optimized libraries (from C/C++ or Fortran).

This includes…

- CUBLAS
- Libsci_acc
- CUFFT
- MAGMA
- CULA
- Thrust
- …

http://www.pgroup.com/lit/articles/insider/v5n2a2.htm