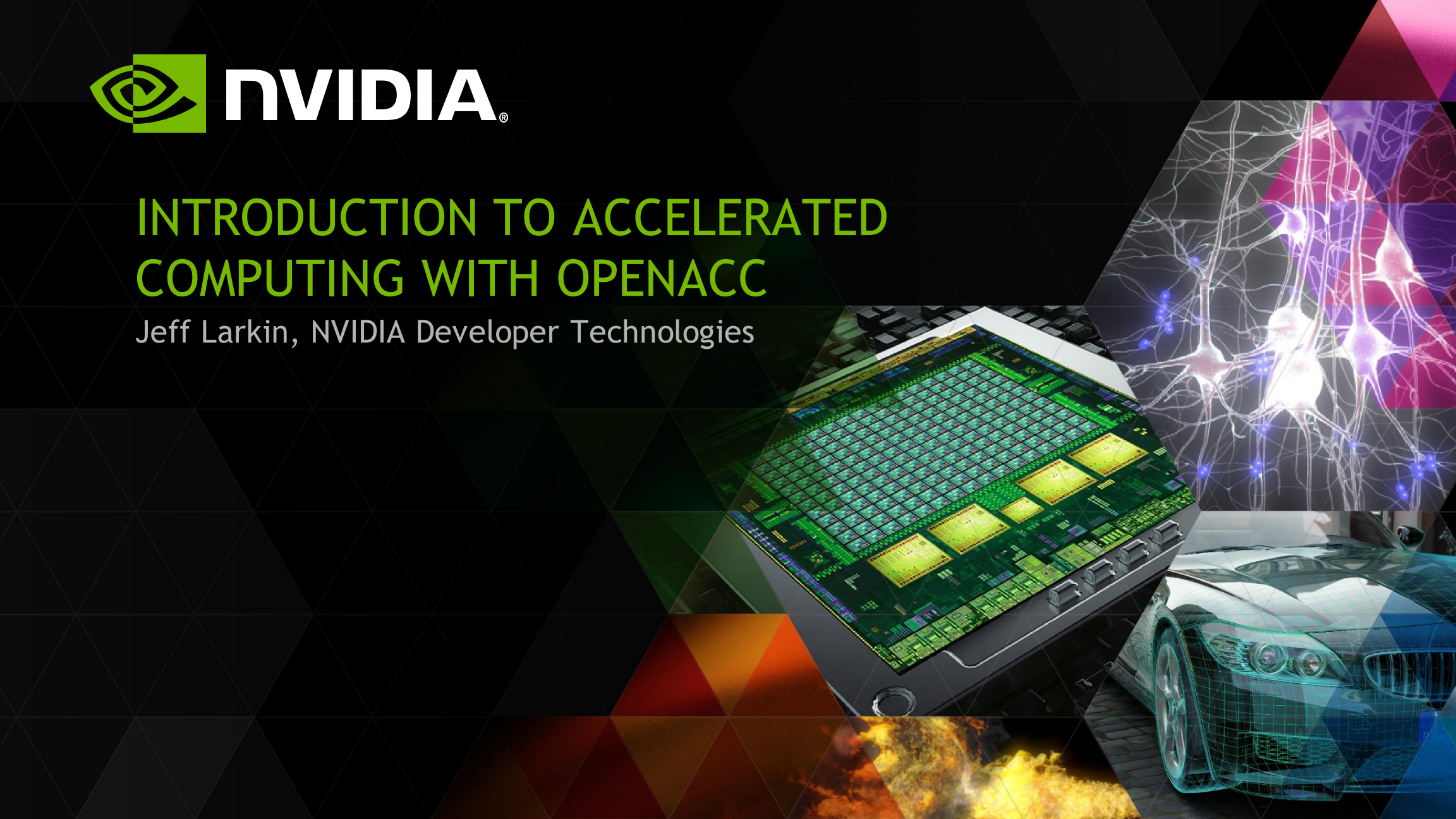


INTRODUCTION TO ACCELERATED COMPUTING WITH OPENACC

Jeff Larkin, NVIDIA Developer Technologies

AGENDA

- NVIDIA Introduction
- Accelerated Computing Basics
- What are Compiler Directives?
- Accelerating Applications with OpenACC
  - Identifying Available Parallelism
  - Exposing Parallelism
  - Optimizing Data Locality
- Next Steps

NVIDIA Introduction

GAMING

DESIGN

ENTERPRISE VIRTUALIZATION

HPC & CLOUD SERVICE PROVIDERS

AUTONOMOUS MACHINES

PC

DATA CENTER

MOBILE

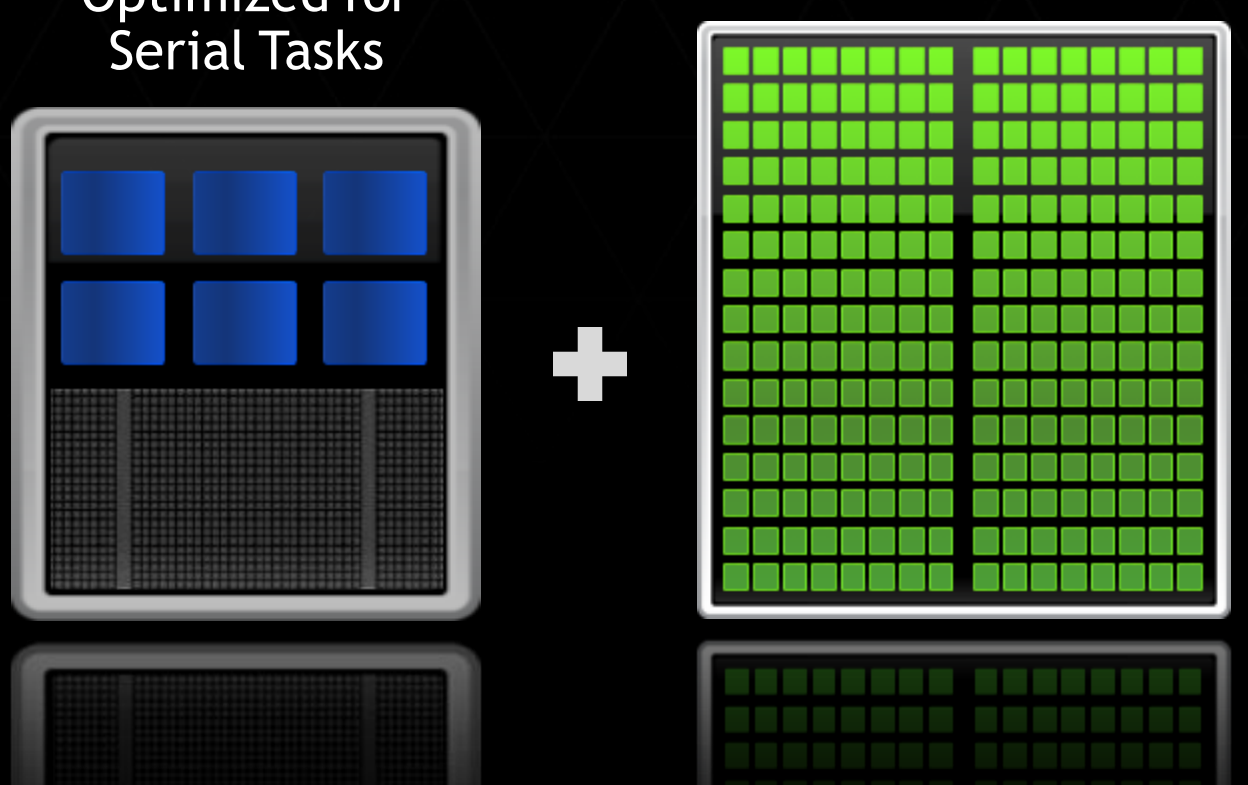
The World Leader in Visual Computing

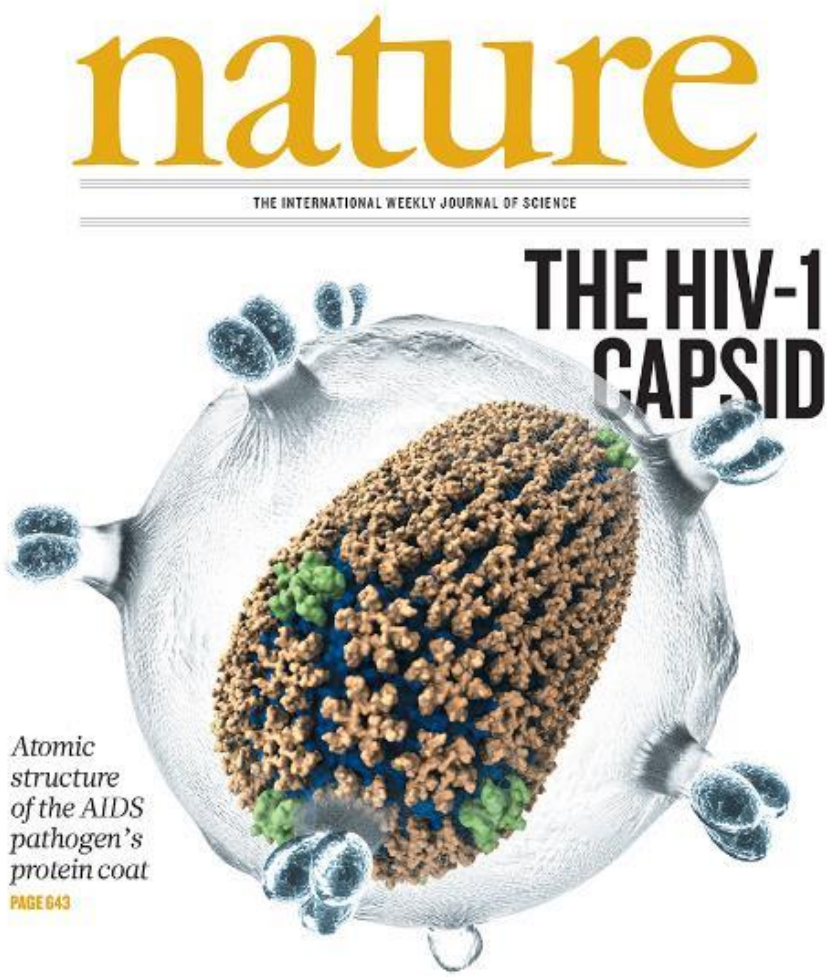
ACCELERATED COMPUTING

10X PERFORMANCE & 5X ENERGY EFFICIENCY

CPU
Optimized for
Serial Tasks

GPU Accelerator
Optimized for
Parallel Tasks





ACCELERATING DISCOVERIES

USING A SUPERCOMPUTER POWERED BY 3,000 TESLA PROCESSORS, UNIVERSITY OF ILLINOIS SCIENTISTS PERFORMED THE FIRST ALL-ATOM SIMULATION OF THE HIV VIRUS AND DISCOVERED THE CHEMICAL STRUCTURE OF ITS CAPSID — "THE PERFECT TARGET FOR FIGHTING THE INFECTION."

WITHOUT GPU, THE SUPERCOMPUTER WOULD NEED TO BE 5X LARGER FOR SIMILAR PERFORMANCE.

TESLA ACCELERATED COMPUTING PLATFORM

Data Center Infrastructure

System Solutions

Communication

Infrastructure Management

Development

Programming Languages

Development Tools

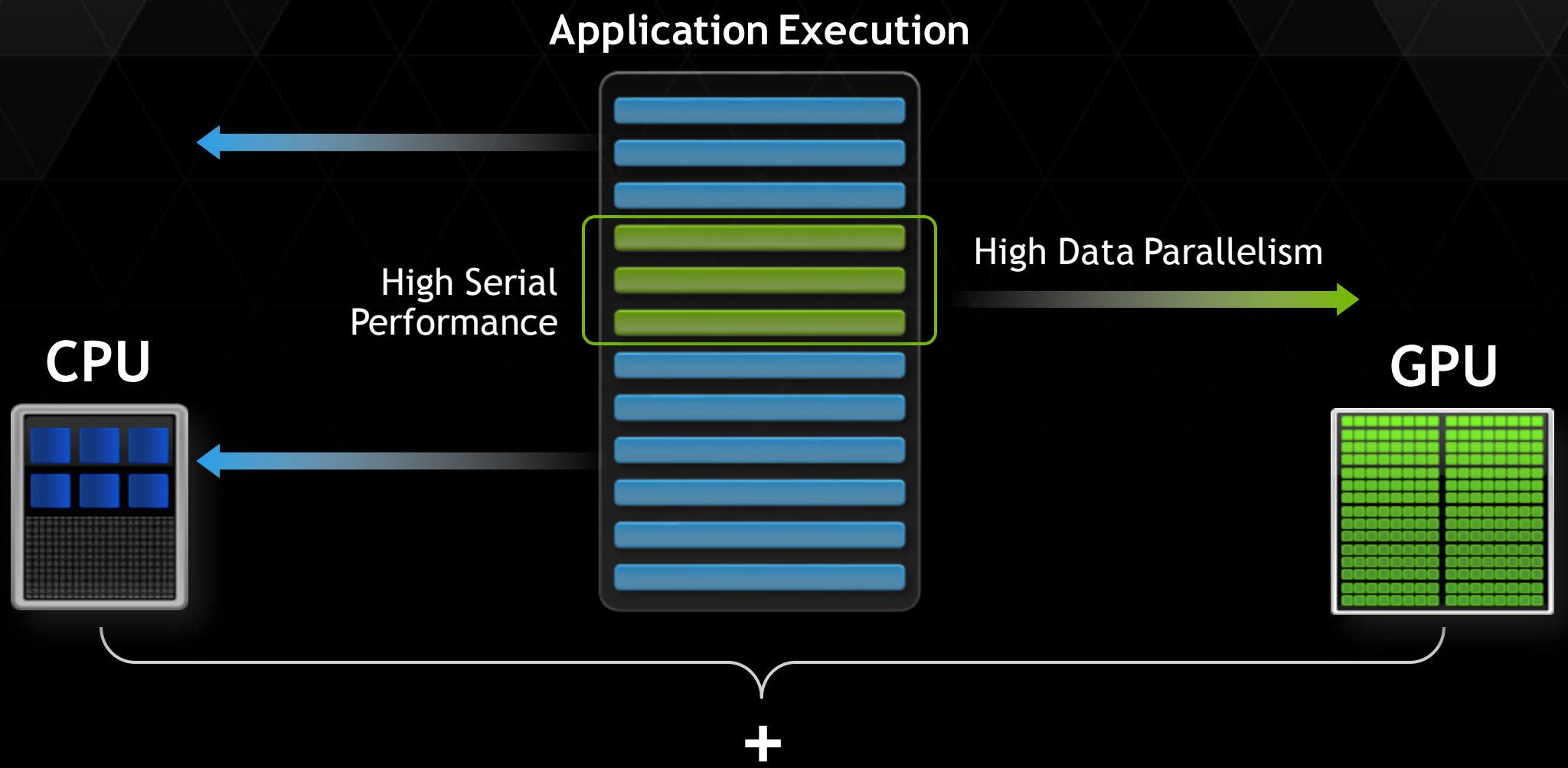
Software Solutions

GPU Accelerators	Interconnect	System Management	Compiler Solutions	Profile and Debug	Accelerated Libraries
GPU Boost	*GPU Direct* *NVLink*	*NVML*	*LLVM*	*CUPTI*	*cuBLAS*

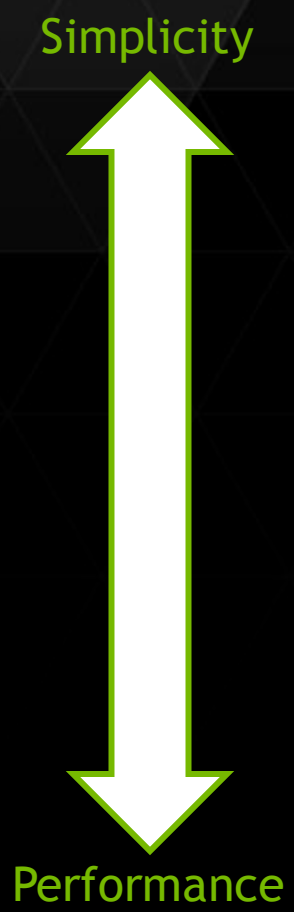
Enterprise Services Support & Maintenance

ACCELERATED COMPUTING BASICS

WHAT IS ACCELERATED COMPUTING?

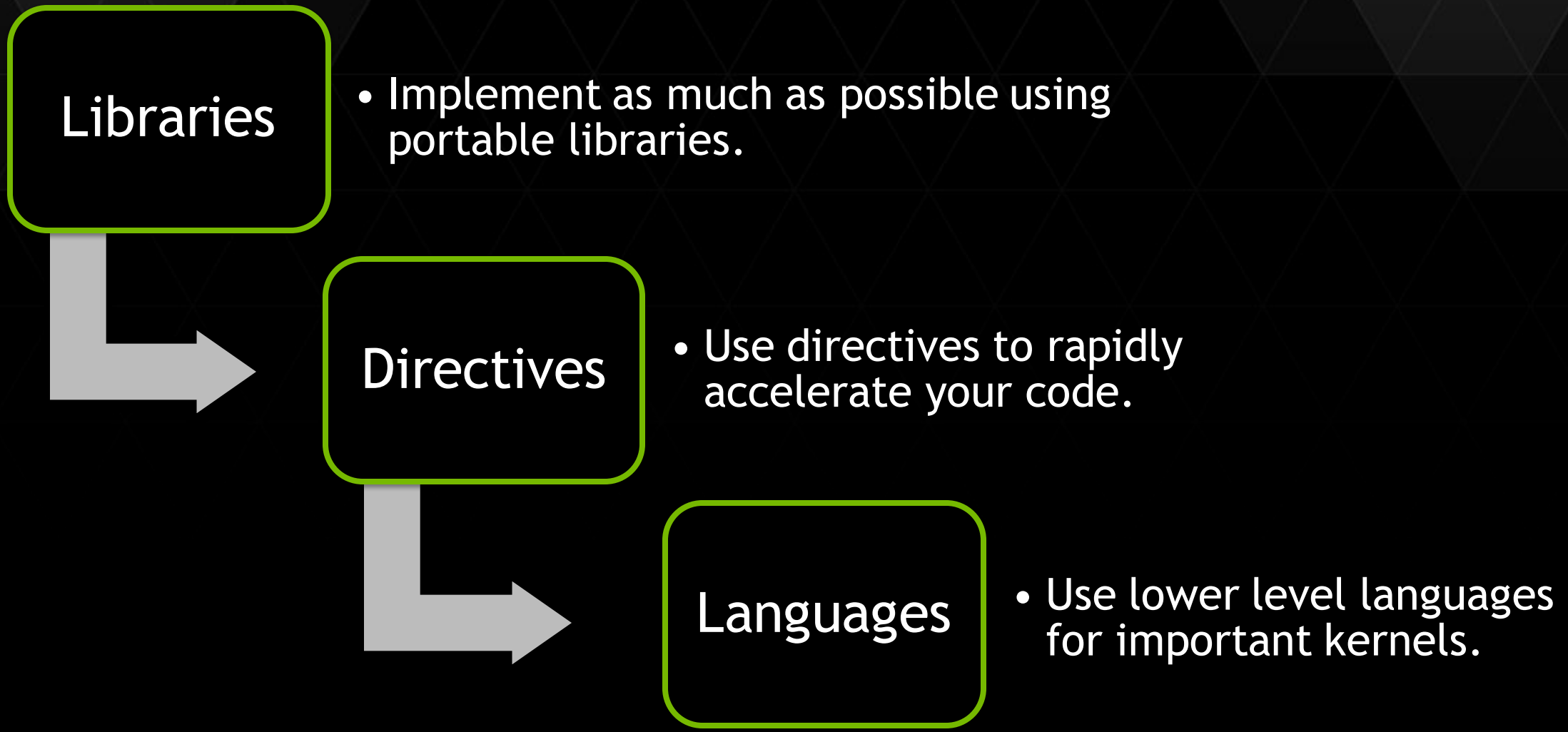


SIMPLICITY & PERFORMANCE



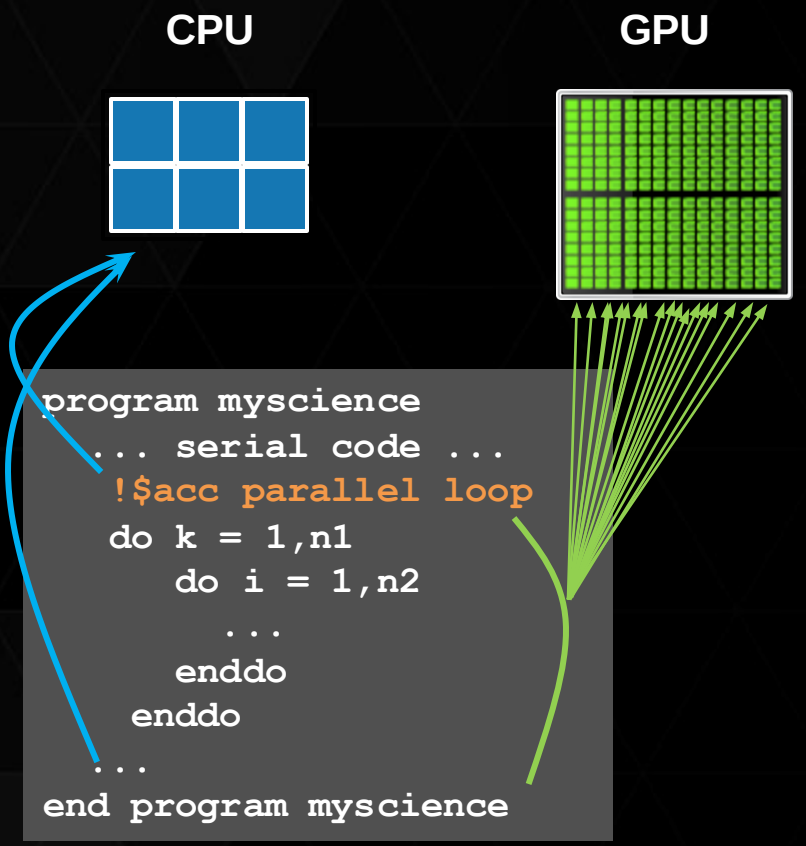
- Accelerated Libraries
  - Little or no code change for standard libraries; high performance
  - Limited by what libraries are available
- Compiler Directives
  - High Level: Based on existing languages; simple and familiar
  - High Level: Performance may not be optimal
- Parallel Language Extensions
  - Expose low-level details for maximum performance
  - Often more difficult to learn and more time consuming to implement

CODE FOR SIMPLICITY & PERFORMANCE



WHAT ARE COMPILER DIRECTIVES?

WHAT ARE COMPILER DIRECTIVES?



- Insert portable compiler directives
- Compiler parallelizes code and manages data movement
- Programmer optimizes incrementally
- Designed for multi-core CPUs, GPUs & many-core Accelerators

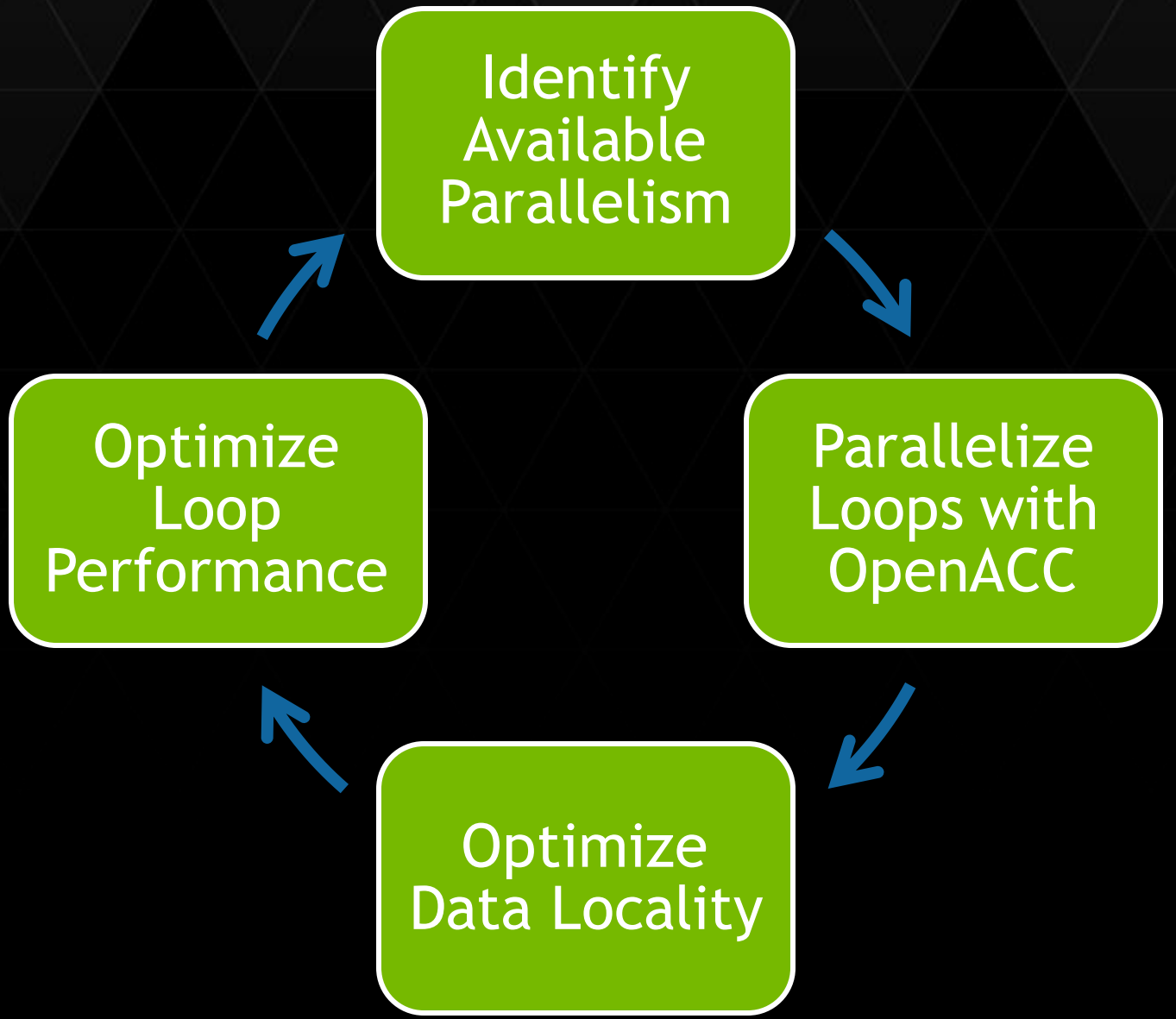
OPENACC: THE STANDARD FOR GPU DIRECTIVES

- **Simple:** Easy path to accelerate compute intensive applications
- **Open:** Open standard that can be implemented anywhere
- **Portable:** Represents parallelism at a high level making it portable to any architecture

OpenACC
Directives for Accelerators

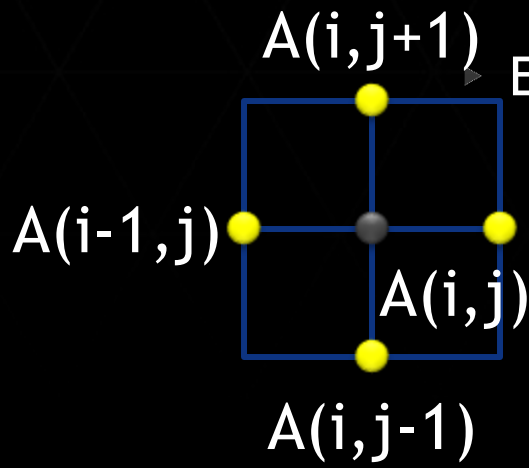
OPENACC MEMBERS AND PARTNERS

ACCELERATING APPLICATIONS WITH OPENACC


Identify Available Parallelism
Parallelize Loops with OpenACC
Optimize Data Locality
Optimize Loop Performance

EXAMPLE: JACOBI ITERATION

- Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.
  - Common, useful algorithm
  - Example: Solve Laplace equation in 2D: $\nabla^2 f(x, y) = 0$

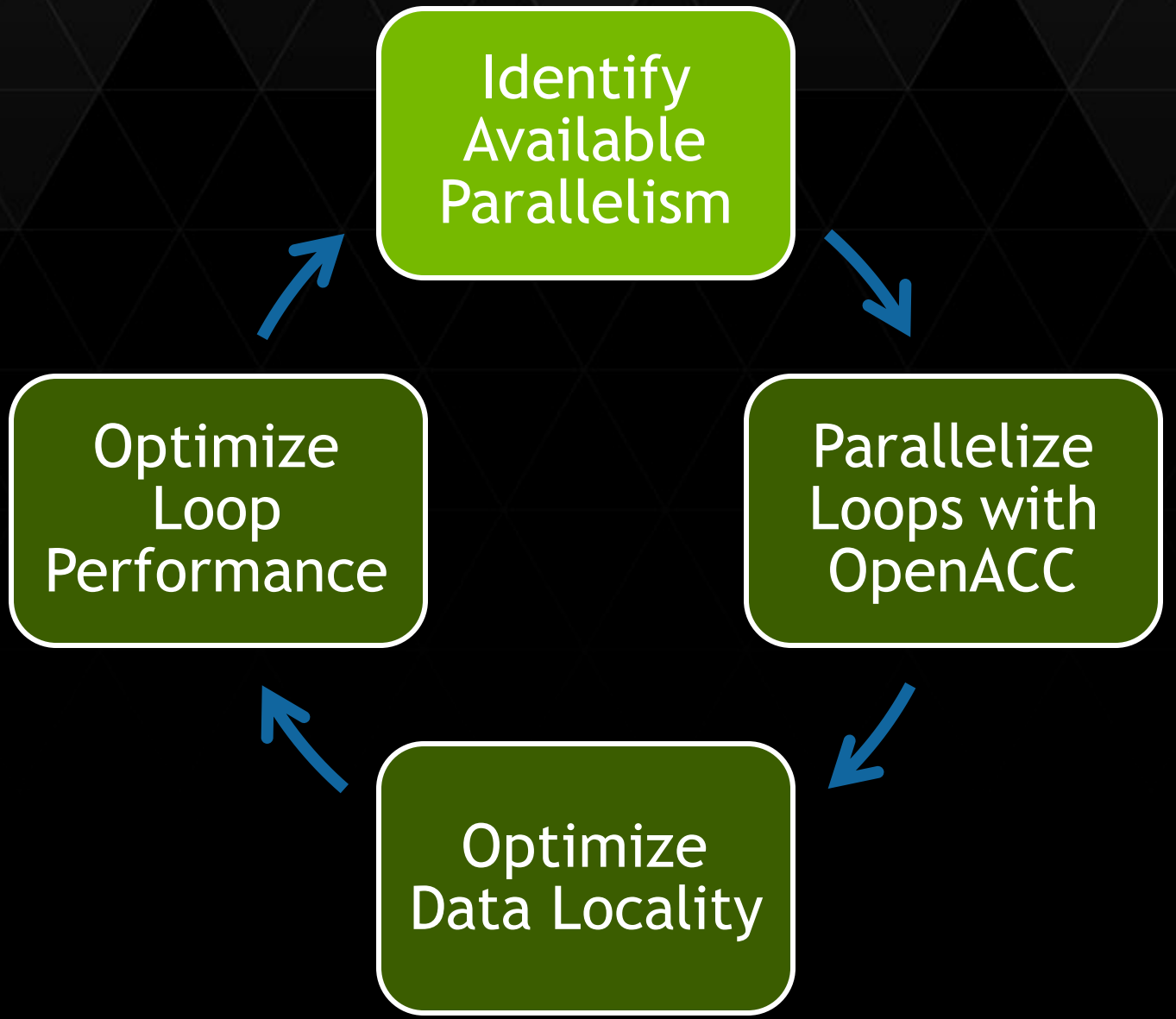


$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

JACOBI ITERATION: C CODE

```
while ( err > tol && iter < iter_max ) {
  err=0.0;

  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }


  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

- Iterate until converged
- Iterate across matrix elements
- Calculate new value from neighbors
- Compute max error for convergence
- Swap input/output arrays


Identify Available Parallelism
Parallelize Loops with OpenACC
Optimize Data Locality
Optimize Loop Performance

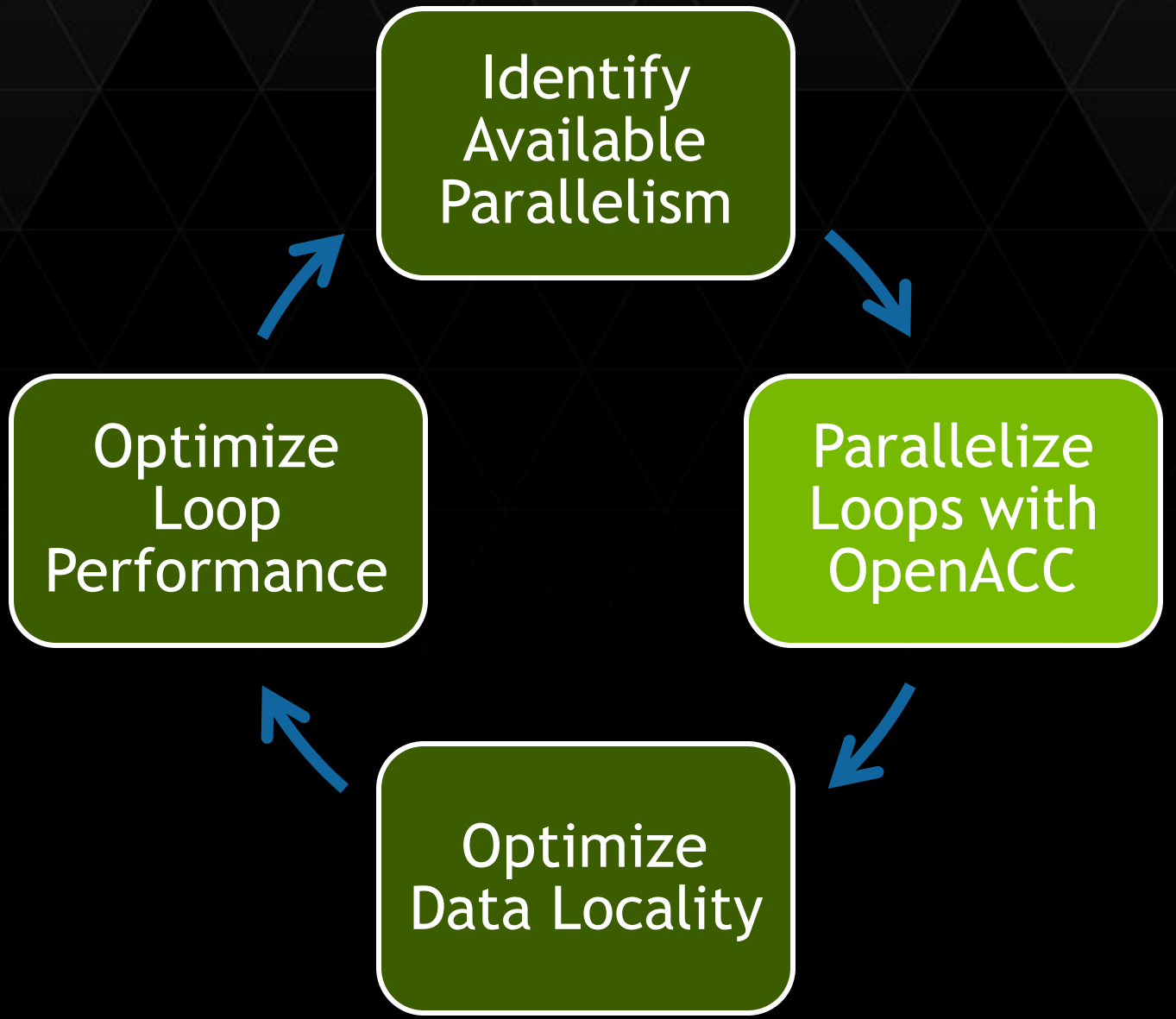
IDENTIFY PARALLELISM

```
while ( err > tol && iter < iter_max ) {
  err=0.0;

  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }


  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

Data dependency between iterations.

Independent loop iterations

Independent loop iterations


Identify Available Parallelism
Parallelize Loops with OpenACC
Optimize Data Locality
Optimize Loop Performance

OPENACC DIRECTIVE SYNTAX

- 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 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++)
{
  y[i] = a*x[i]+y[i];
}
```

Kernel:
A function that runs in parallel on the GPU

PARALLELIZE WITH OPENACC

```
while ( err > tol && iter < iter_max ) {
  err=0.0;

#pragma acc parallel loop reduction(max:err)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }

#pragma acc parallel loop
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

Parallelize loop on accelerator

Parallelize loop on accelerator

* A *reduction* means that all of the N*M values for err will be reduced to just one, the max.

BUILDING THE CODE

```
$ pgcc -fast -acc -ta=tesla -Minfo=all laplace2d.c
main:
     40, Loop not fused: function call before adjacent loop
         Generated vector sse code for the loop
     51, Loop not vectorized/parallelized: potential early exits
     55, Accelerator kernel generated
         55, Max reduction generated for error
         56, #pragma acc loop gang /* blockIdx.x */
         58, #pragma acc loop vector(256) /* threadIdx.x */
     55, Generating copyout(Anew[1:4094][1:4094])
         Generating copyin(A[:][:])
         Generating Tesla code
     58, Loop is parallelizable
     66, Accelerator kernel generated
         67, #pragma acc loop gang /* blockIdx.x */
         69, #pragma acc loop vector(256) /* threadIdx.x */
     66, Generating copyin(Anew[1:4094][1:4094])
         Generating copyout(A[1:4094][1:4094])
         Generating Tesla code
     69, Loop is parallelizable
```

OPENACC KERNELS DIRECTIVE

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++)
{
  x[i] = 1.0;
  y[i] = 2.0;
}

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

kernel 1

kernel 2

The compiler identifies 2 parallel loops and generates 2 kernels.

PARALLELIZE WITH OPENACC KERNELS

```
while ( err > tol && iter < iter_max ) {
  err=0.0;

#pragma acc kernels
{
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }

  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
}
  iter++;
}
```

BUILDING THE CODE

```
$ pgcc -fast -ta=tesla -Minfo=all laplace2d.c
main:
     40, Loop not fused: function call before adjacent loop
         Generated vector sse code for the loop
     51, Loop not vectorized/parallelized: potential early exits
     55, Generating copyout(Anew[1:4094][1:4094])
         Generating copyin(A[:][:])
         Generating copyout(A[1:4094][1:4094])
         Generating Tesla code
     57, Loop is parallelizable
     59, Loop is parallelizable
         Accelerator kernel generated
         57, #pragma acc loop gang /* blockIdx.y */
         59, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         63, Max reduction generated for error
     67, Loop is parallelizable
     69, Loop is parallelizable
         Accelerator kernel generated
         67, #pragma acc loop gang /* blockIdx.y */
         69, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

OPENACC PARALLEL LOOP VS. KERNELS

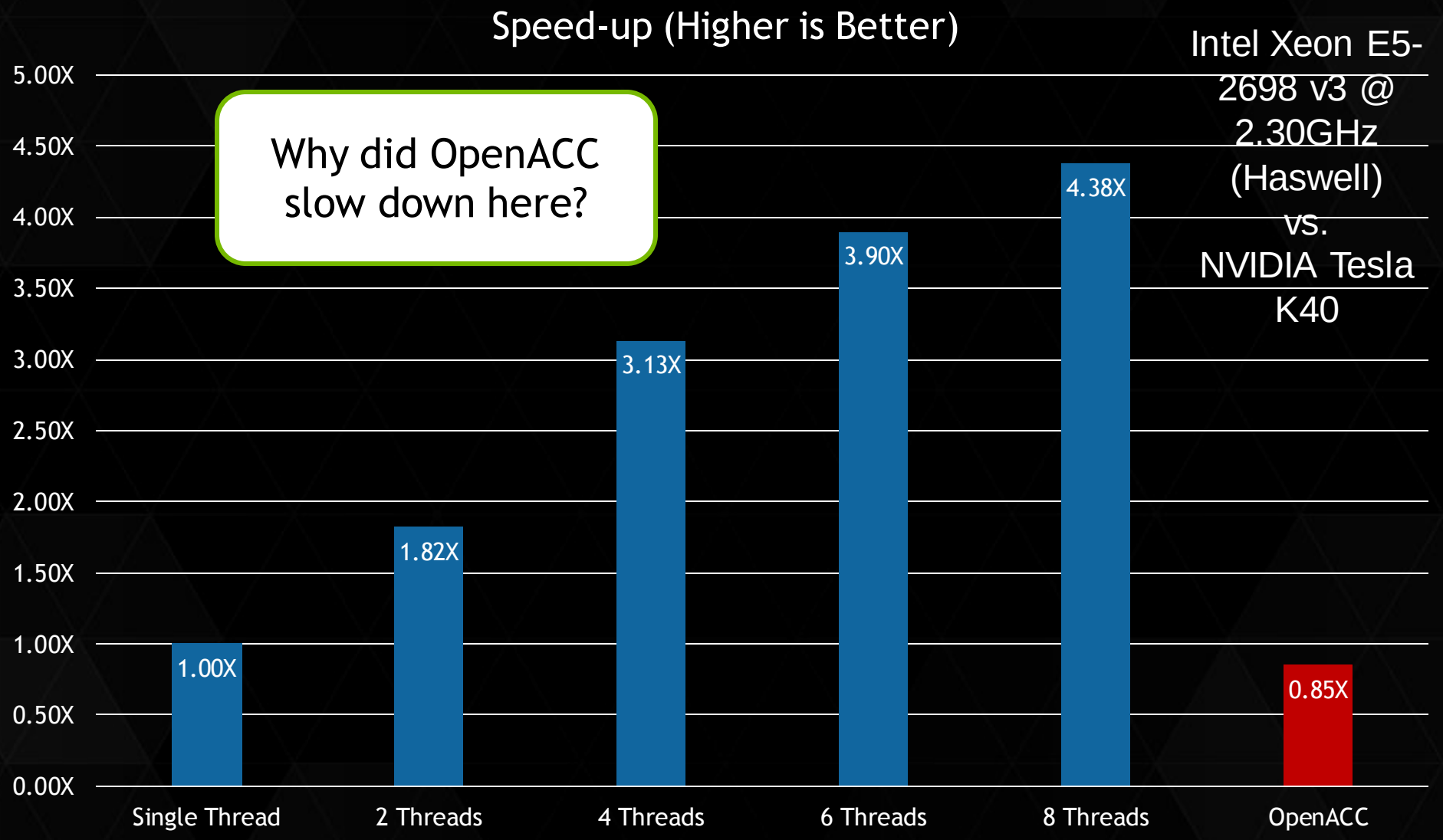
PARALLEL LOOP

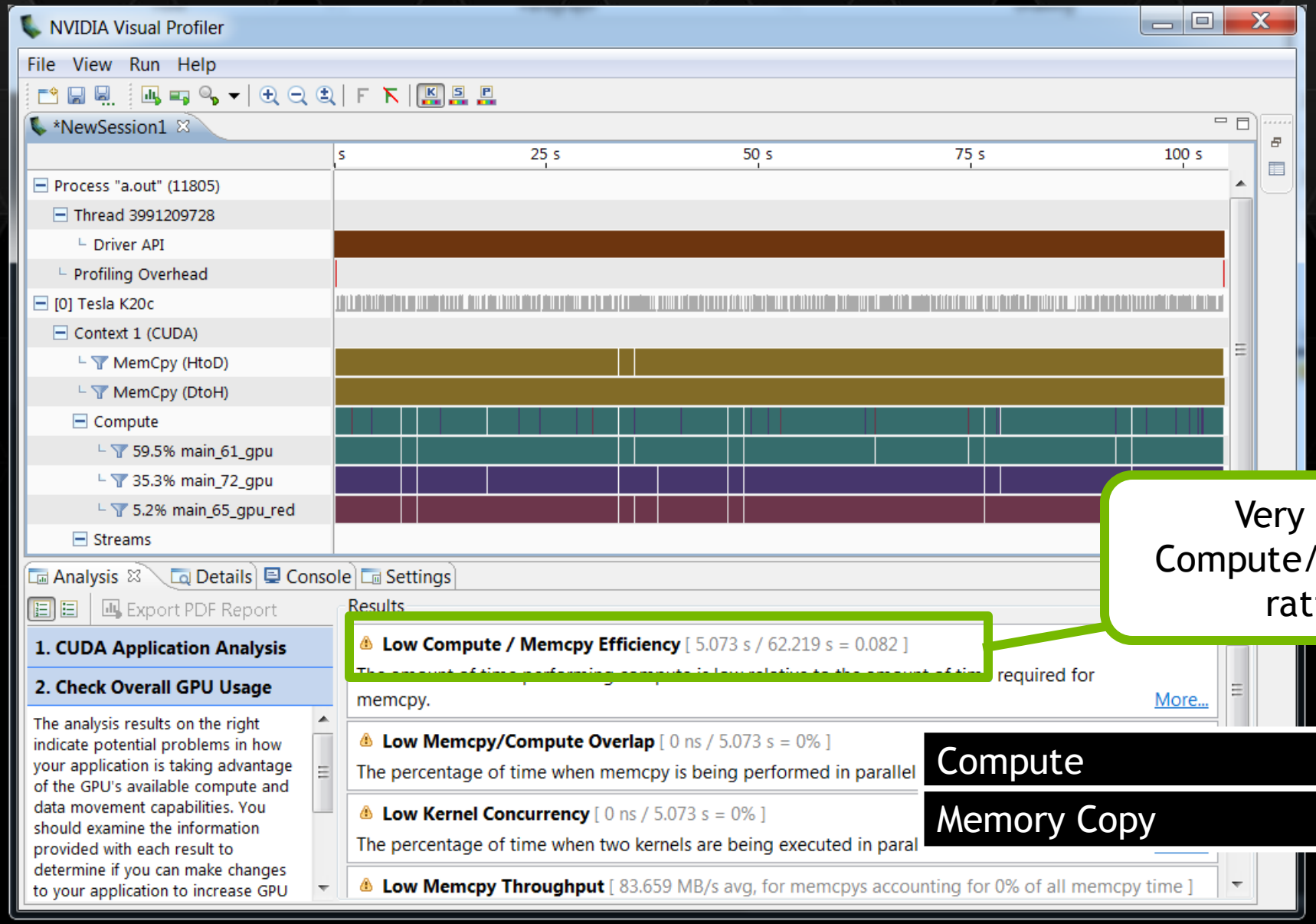
- Requires analysis by programmer to ensure safe parallelism
- Will parallelize what a compiler may miss
- Straightforward path from OpenMP

KERNELS

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

Both approaches are equally valid and can perform equally well.


Speed-up (Higher is Better)
Intel Xeon E5-2698 v3 @ 2.30GHz (Haswell) vs. NVIDIA Tesla K40
Why did OpenACC slow down here?
5.00X
4.50X
4.00X
3.50X
3.00X
2.50X
2.00X
1.50X
1.00X
0.50X
0.00X
1.00X
1.82X
3.13X
3.90X
4.38X
0.85X
Single Thread
2 Threads
4 Threads
6 Threads
8 Threads
OpenACC



Very low Compute/Memcpy ratio

Compute	5.0s
Memory Copy	62.2s

EXCESSIVE DATA TRANSFERS

```
while ( err > tol && iter < iter_max )
{
  err=0.0;
```

A, Anew resident on host

```
#pragma acc parallel loop
```

A, Anew resident on accelerator

```
for( int j = 1; j < n-1; j++) {
  for(int i = 1; i < m-1; i++) {
    Anew[j][i] = 0.25 * (A[j][i+1] +
                 A[j][i-1] + A[j-1][i] +
                 A[j+1][i]);
    err = max(err, abs(Anew[j][i] -
                       A[j][i]);
  }
}
...
```

These copies happen every iteration of the outer while loop!

A, Anew resident on host

A, Anew resident on accelerator

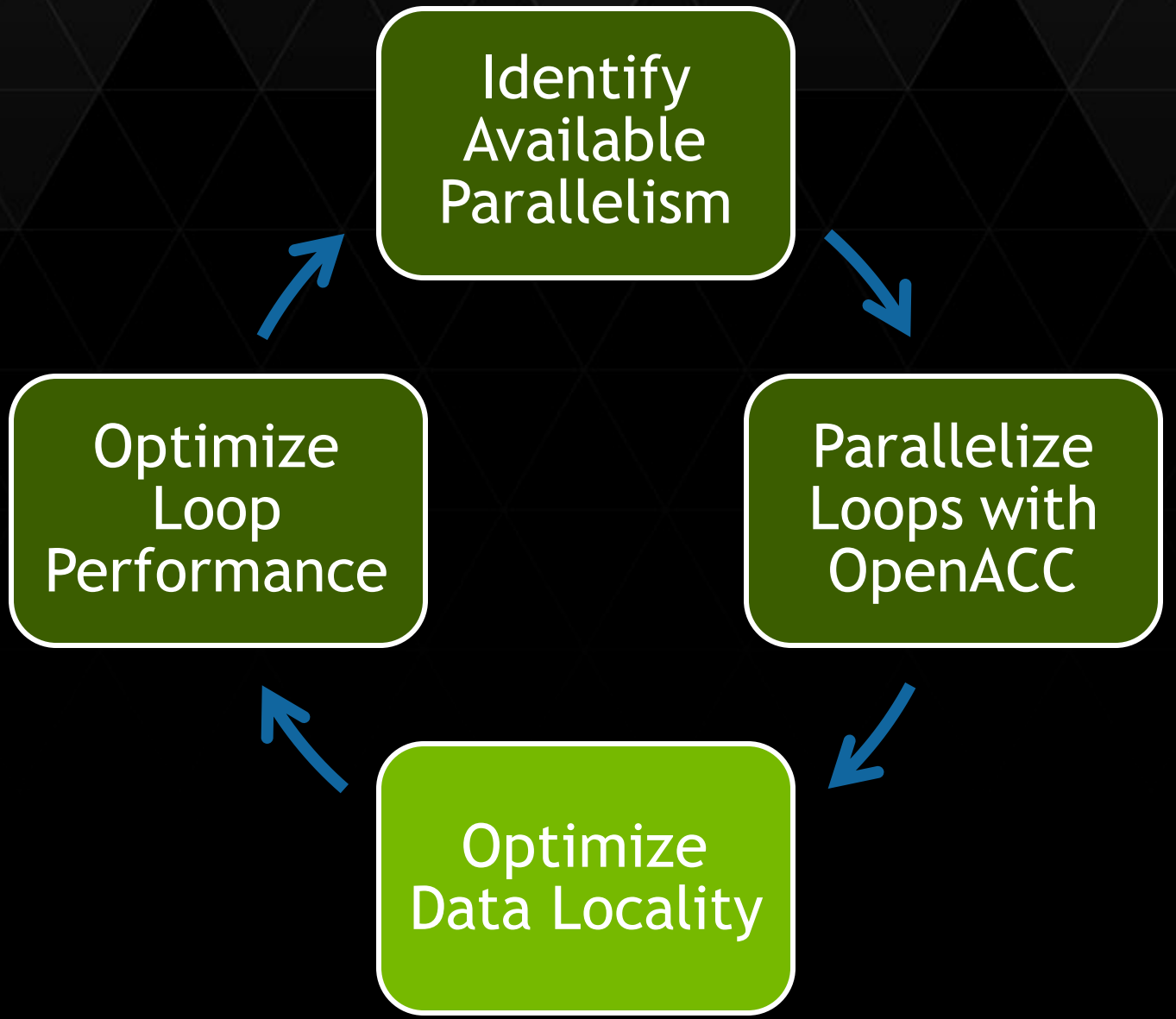
```
  ...
}
```

IDENTIFYING DATA LOCALITY

```
while ( err > tol && iter < iter_max ) {
  err=0.0;

#pragma acc parallel loop
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }

#pragma acc parallel loop
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

Does the CPU need the data between these loop nests?

Does the CPU need the data between iterations of the convergence loop?


Identify Available Parallelism
Parallelize Loops with OpenACC
Optimize Data Locality
Optimize Loop Performance

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

`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.

and `present_or_copy[in|out]`, `present_or_create`, `deviceptr`.

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: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`

OPTIMIZE DATA LOCALITY

```
#pragma acc data copy(A) create(Anew)
while ( err > tol && iter < iter_max ) {
  err=0.0;
#pragma acc parallel loop
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }

#pragma acc parallel loop
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
}
  iter++;
}
```

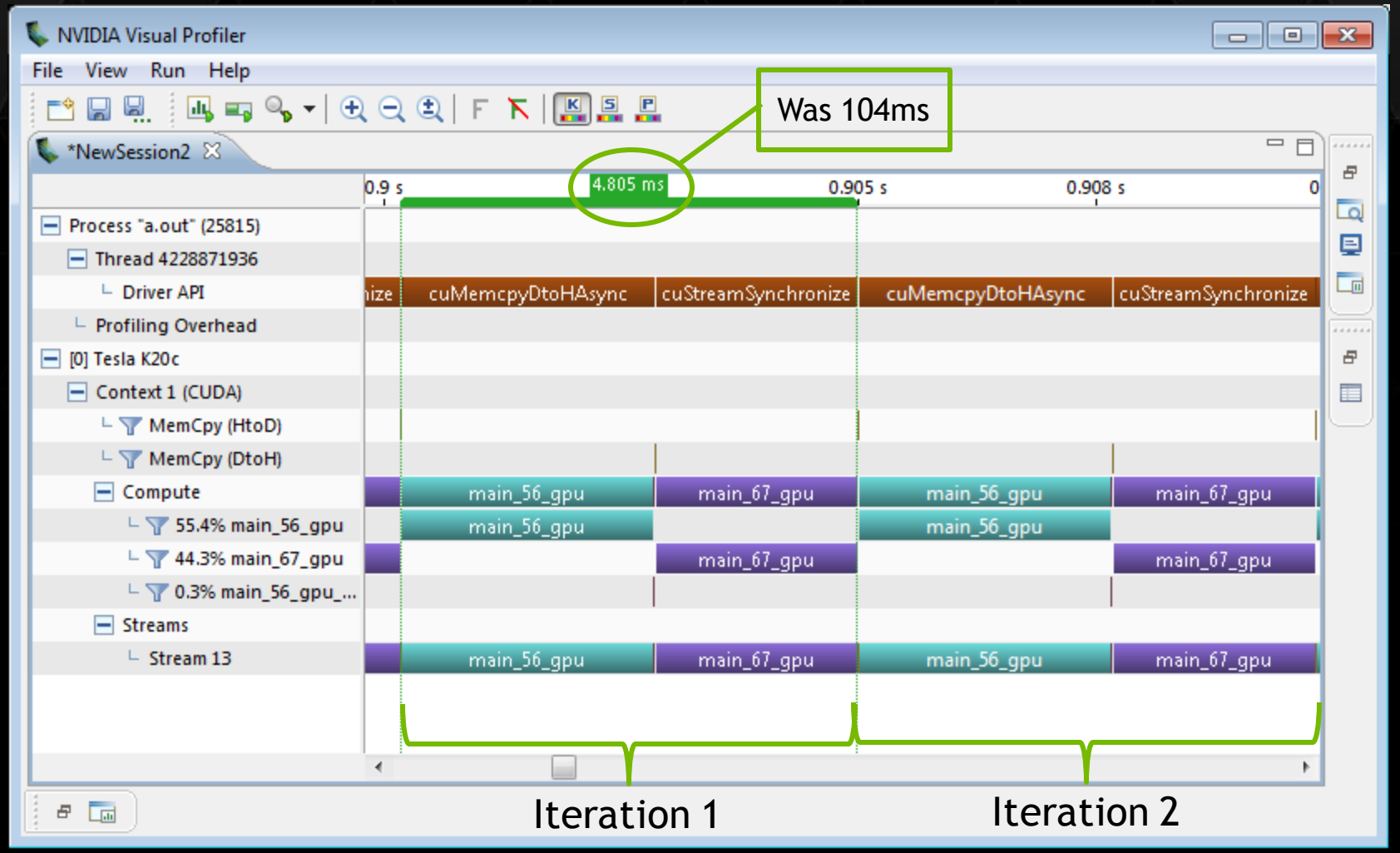
Copy A to/from the accelerator only when needed.

Create Anew as a device temporary.

REBUILDING THE CODE

```
$ pgcc -fast -acc -ta=tesla -Minfo=all laplace2d.c
main:
     40, Loop not fused: function call before adjacent loop
         Generated vector sse code for the loop
     51, Generating copy(A[:][:])
         Generating create(Anew[:][:])
         Loop not vectorized/parallelized: potential early exits
     56, Accelerator kernel generated
         56, Max reduction generated for error
         57, #pragma acc loop gang /* blockIdx.x */
         59, #pragma acc loop vector(256) /* threadIdx.x */
     56, Generating Tesla code
     59, Loop is parallelizable
     67, Accelerator kernel generated
         68, #pragma acc loop gang /* blockIdx.x */
         70, #pragma acc loop vector(256) /* threadIdx.x */
     67, Generating Tesla code
     70, Loop is parallelizable
```

VISUAL PROFILER: DATA REGION




Speed-Up (Higher is Better)
30.00X
25.00X
20.00X
15.00X
10.00X
5.00X
0.00X
Intel Xeon E5-2698 v3 @ 2.30GHz (Haswell)
vs.
NVIDIA Tesla K40
1.00X
1.82X
3.13X
3.90X
4.38X
27.30X
Socket/Socket:
6.24X
Single Thread
2 Threads
4 Threads
6 Threads
8 Threads
OpenACC

OPENACC PRESENT CLAUSE

It's sometimes necessary for a data region to be in a different scope than the compute region.

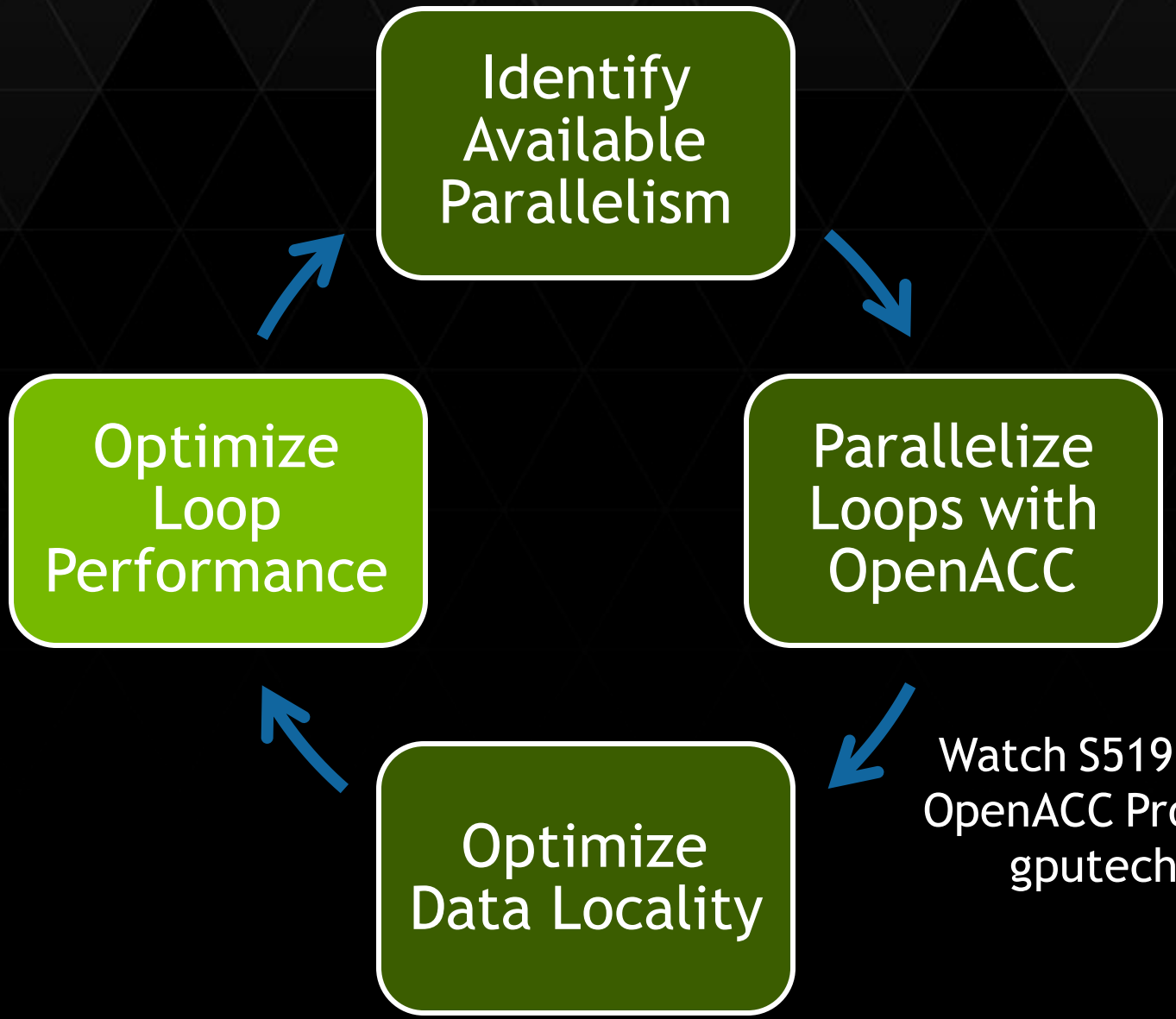
When this occurs, the **present** clause can be used to tell the compiler data is already on the device.

Since the declaration of A is now in a higher scope, it's necessary to shape A in the present clause.

High-level data regions and the present clause are often critical to good performance.

```
function main(int argc, char **argv)
{

  #pragma acc data copy(A)
  {
    laplace2D(A,n,m);
  }
}
```

```
function laplace2D(double[N][M] A,n,m)
{

  #pragma acc data present(A[n][m]) create(Anew)
  while ( err > tol && iter < iter_max ) {
    err=0.0;
    ...
  }
}
```



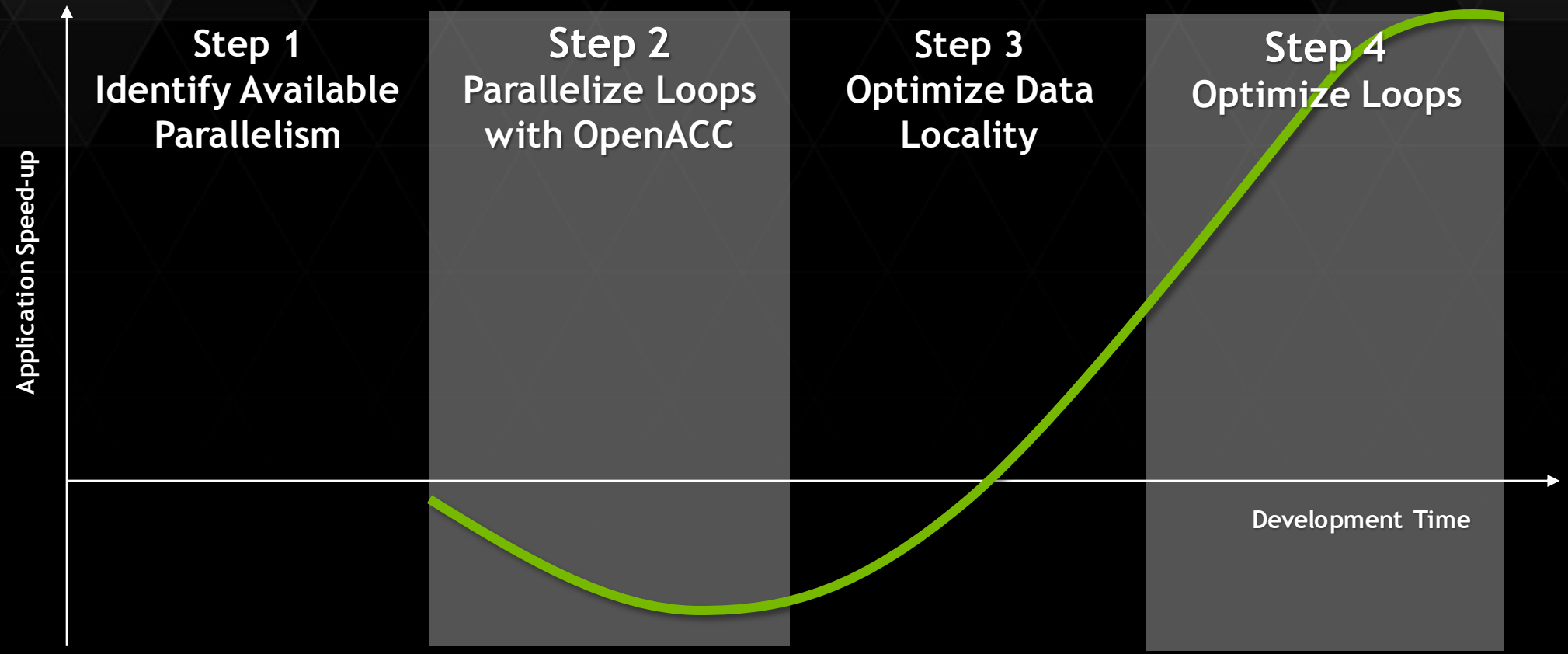
Watch S5195 - Advanced OpenACC Programming on gputechconf.com

NEXT STEPS

ACCELERATING WITH OPENACC

1. Identify Available Parallelism
   - What important parts of the code have available parallelism?
2. Parallelize Loops
   - Express as much parallelism as possible and ensure you still get correct results.
   - Because the compiler *must* be cautious about data movement, the code will generally slow down.
3. Optimize Data Locality
   - The programmer will *always* know better than the compiler what data movement is unnecessary.
4. Optimize Loop Performance
   - Don't try to optimize a kernel that runs in a few *us* or *ms* until you've eliminated the excess data motion that is taking *many seconds*.

TYPICAL PORTING EXPERIENCE WITH OPENACC DIRECTIVES



FOR MORE INFORMATION

- Check out http://openacc.org/
- Watch tutorials at http://www.gputechconf.com/
- Share your successes at WACCPD at SC15. http://www.openacc.org/content/Events/waccpd_2015
- Take a self-paced lab at https://nvidia.qwiklab.com/