GPU TECHNOLOGY CONFERENCE

NVIDIA

INTRODUCTION TO ACCELERATED COMPUTING USING COMPILER DIRECTIVES

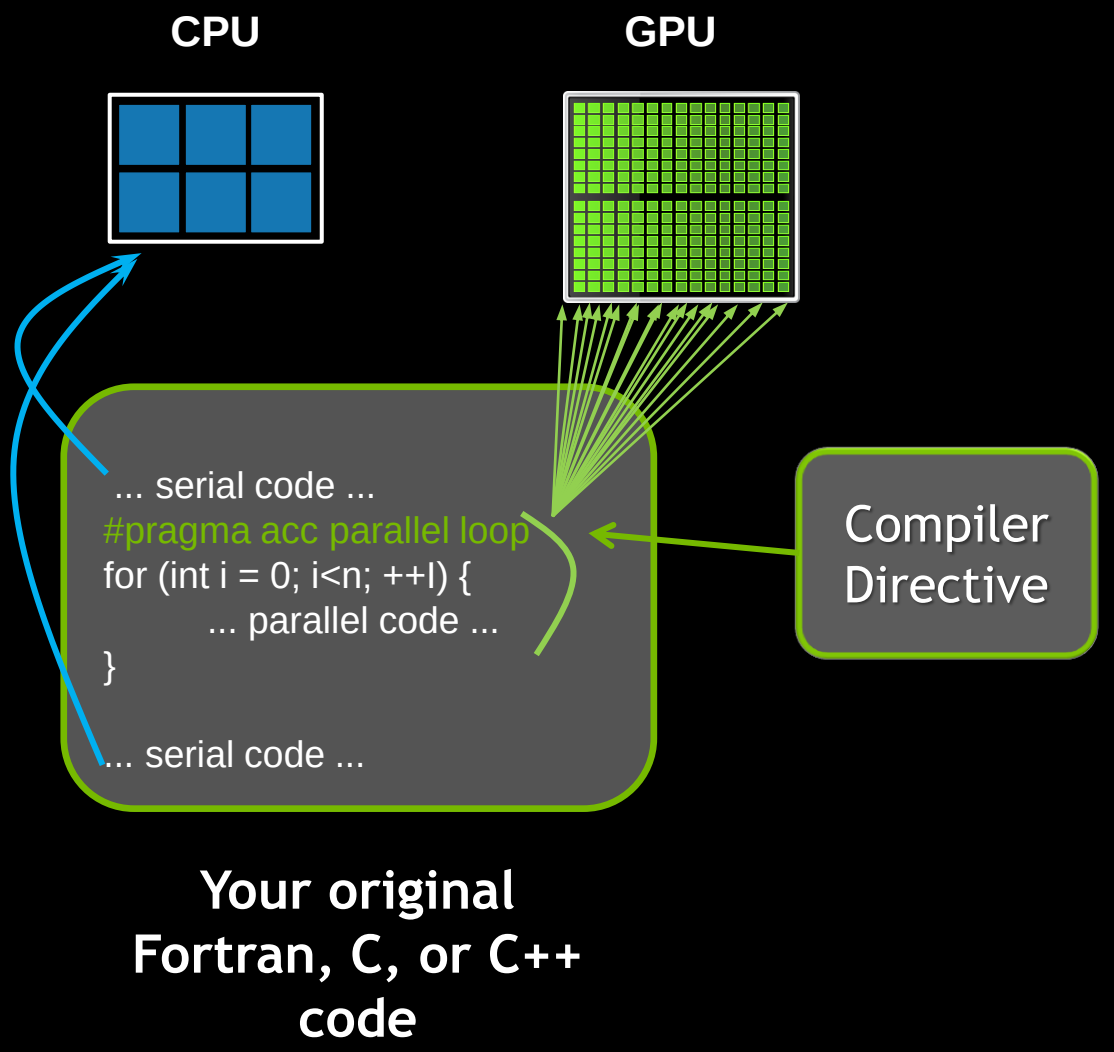
Jeff Larkin

OUTLINE

- What are compiler directives?
- OpenACC 2.0 and OpenMP 4.0
- Compiler directives by example
  - OpenACC `parallel` and `kernels`
  - Profiling an application
  - OpenACC data optimization
- What's next

WHAT ARE COMPILER DIRECTIVES?

WHAT ARE COMPILER DIRECTIVES?



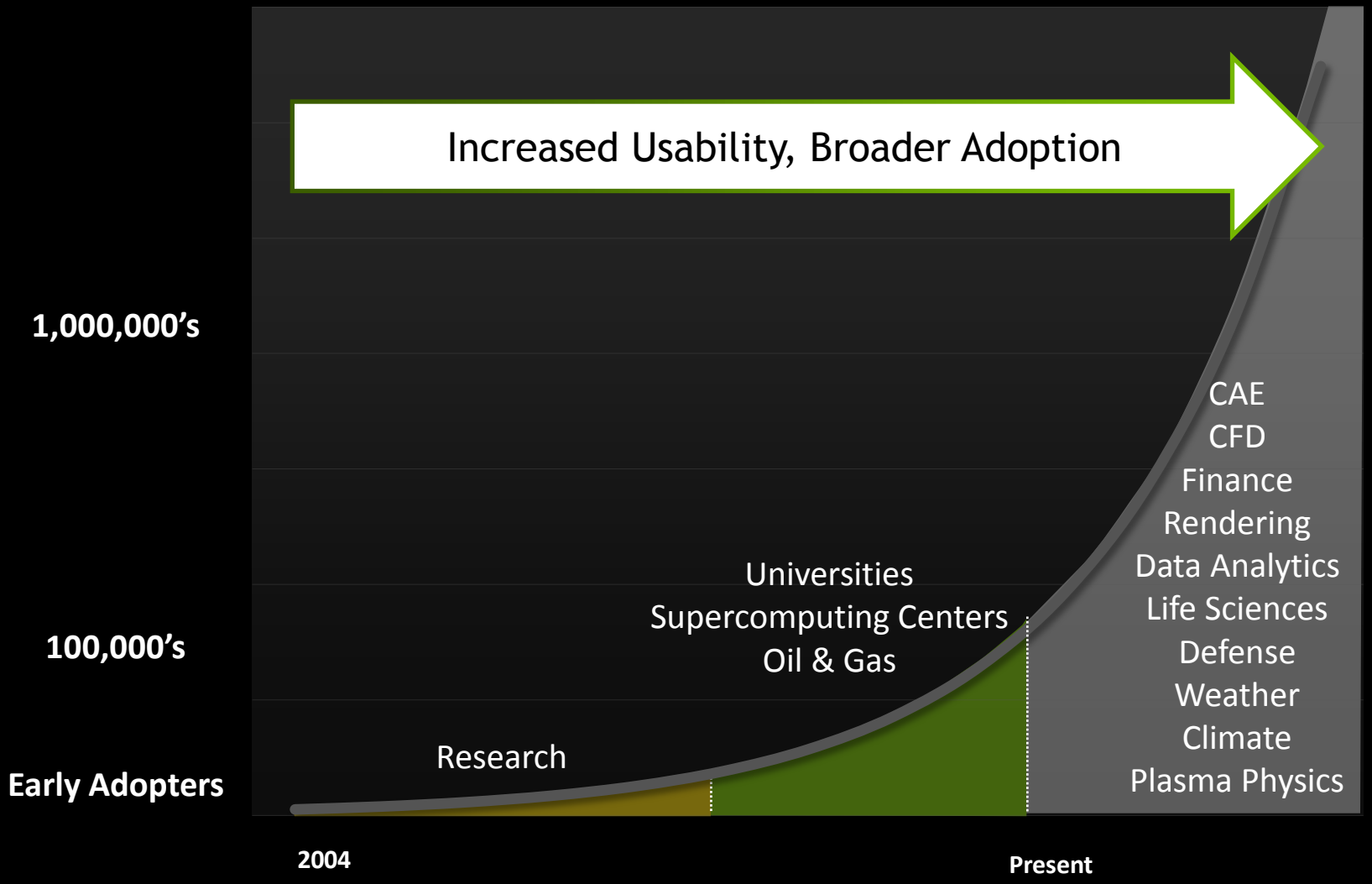
When a compiler directive is encountered the compiler/runtime will…

1. Generate parallel code for GPU
2. Allocate GPU memory and copy input data
3. Execute parallel code on GPU
4. Copy output data to CPU and deallocate GPU memory

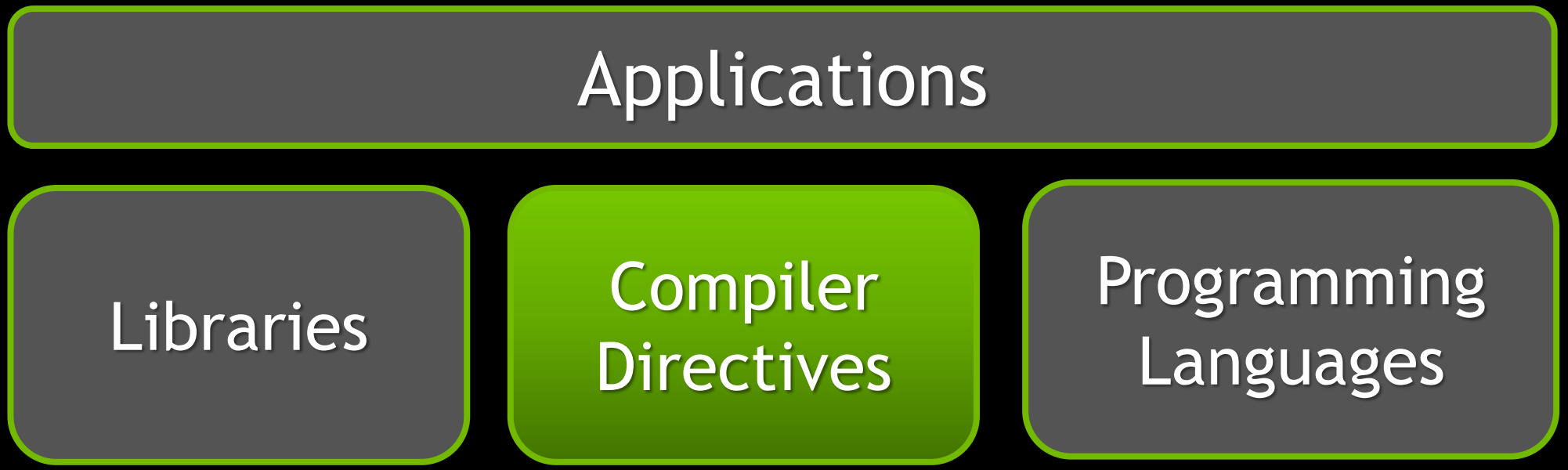
WHY USE COMPILER DIRECTIVES?

- Single Source Code
  - No need to maintain multiple code paths
- High Level
  - Abstract away device details, focus on expressing the parallelism and data locality
- Low Learning Curve
  - Programmer remains in same language and adds directives to existing code
- Rapid Development
  - Fewer code changes means faster development

WHY SUPPORT COMPILER DIRECTIVES?



3 WAYS TO ACCELERATE APPLICATIONS



"Drop-in" Acceleration

Easily Accelerate Applications

Maximum Flexibility

OPENACC 2.0 AND OPENMP 4.0

OPENACC 2.0

- OpenACC is a specification for high-level, compiler directives for expressing parallelism for accelerators.
  - Aims to be performance portable to a wide range of accelerators.
  - Multiple Vendors, Multiple Devices, One Specification
- The OpenACC specification was first released in November 2011.
  - Original members: CAPS, Cray, NVIDIA, Portland Group
- OpenACC 2.0 was released in June 2013, expanding functionality and improving portability
- At the end of 2013, OpenACC had more than 10 member organizations

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 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)
...
```

OPENMP 4.0

- OpenMP has existed since 1997 as a specification for compiler directives for shared memory parallelism.
- In 2013, OpenMP 4.0 was released, expanding the focus beyond shared memory parallel computers, including accelerators.
- The OpenMP 4.0 **`target`** construct provides the means to offload data and computation to accelerators.
- Additional directives were added to support multiple thread teams and simd parallelism.
- OpenMP continues to improve upon its support for offloading to accelerators.

OPENMP DIRECTIVE SYNTAX

- C/C++

```
#pragma omp target directive [clause [,] clause]…]
```

…often followed by a structured code block

- Fortran

```
!$omp target directive [clause [,] clause] …]
```

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

```
!$omp end target directive
```

OPENMP TARGET EXAMPLE: SAXPY

SAXPY in C

```
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma omp target teams \
        distribute parallel for
  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);
...
```

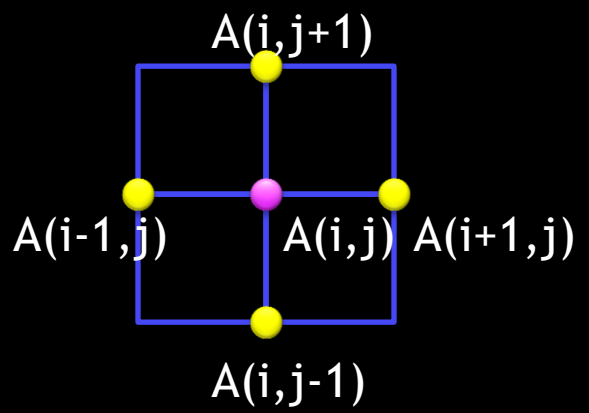
SAXPY in Fortran

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

  !$omp  target teams &
  !$omp& distribute parallel do
  do i=1,n
    y(i) = a*x(i)+y(i)
  enddo
  !$omp end target teams &
  !$omp& distribute parallel do
end subroutine saxpy
 ...
! Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x, y)
...
```

COMPILER DIRECTIVES BY EXAMPLE

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++;
}
```

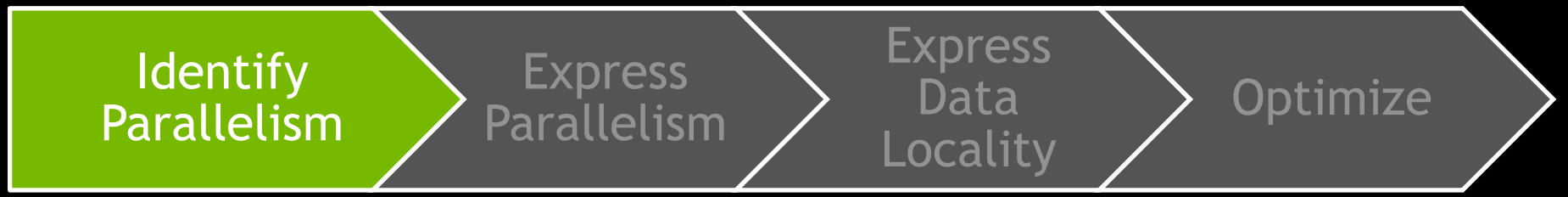
Iterate until converged

Iterate across matrix elements

Calculate new value from neighbors

Compute max error for convergence

Swap input/output arrays



IDENTIFY PARALLELISM

JACOBI ITERATION: C CODE

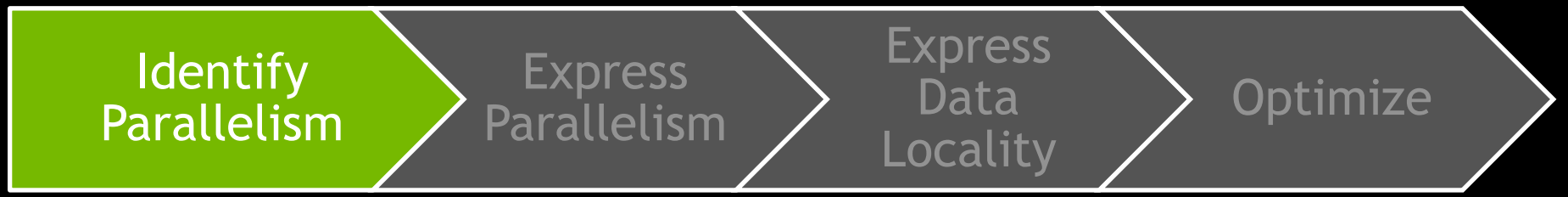
```
while ( err > tol && iter < iter_max ) {
  err=0.0;

  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++;
}
```

Data dependency between iterations.

Independent loop iterations

Independent loop iterations

Identify Parallelism | Express Parallelism | Express Data Locality | Optimize



EXPRESS PARALLELISM

JACOBI ITERATION: OPENMP C CODE

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

#pragma omp parallel for shared(m, n, Anew, A) 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 omp parallel for shared(m, n, Anew, A)
  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 across CPU threads

Parallelize loop across CPU threads

Identify Parallelism | Express Parallelism | Express Data Locality | Optimize

OPENACC PARALLEL LOOP DIRECTIVE

Programmer identifies a block of code as having parallelism, compiler generates a parallel kernel for that loop.

```
#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

JACOBI ITERATION: OPENACC C CODE

```
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

Identify Parallelism | Express Parallelism | Express Data Locality | Optimize

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.

JACOBI ITERATION: OPENACC C CODE (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++;
}
```

Look for parallelism within this region.

OPENACC `PARALLEL LOOP` VS. `KERNELS`

`PARALLEL LOOP`

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

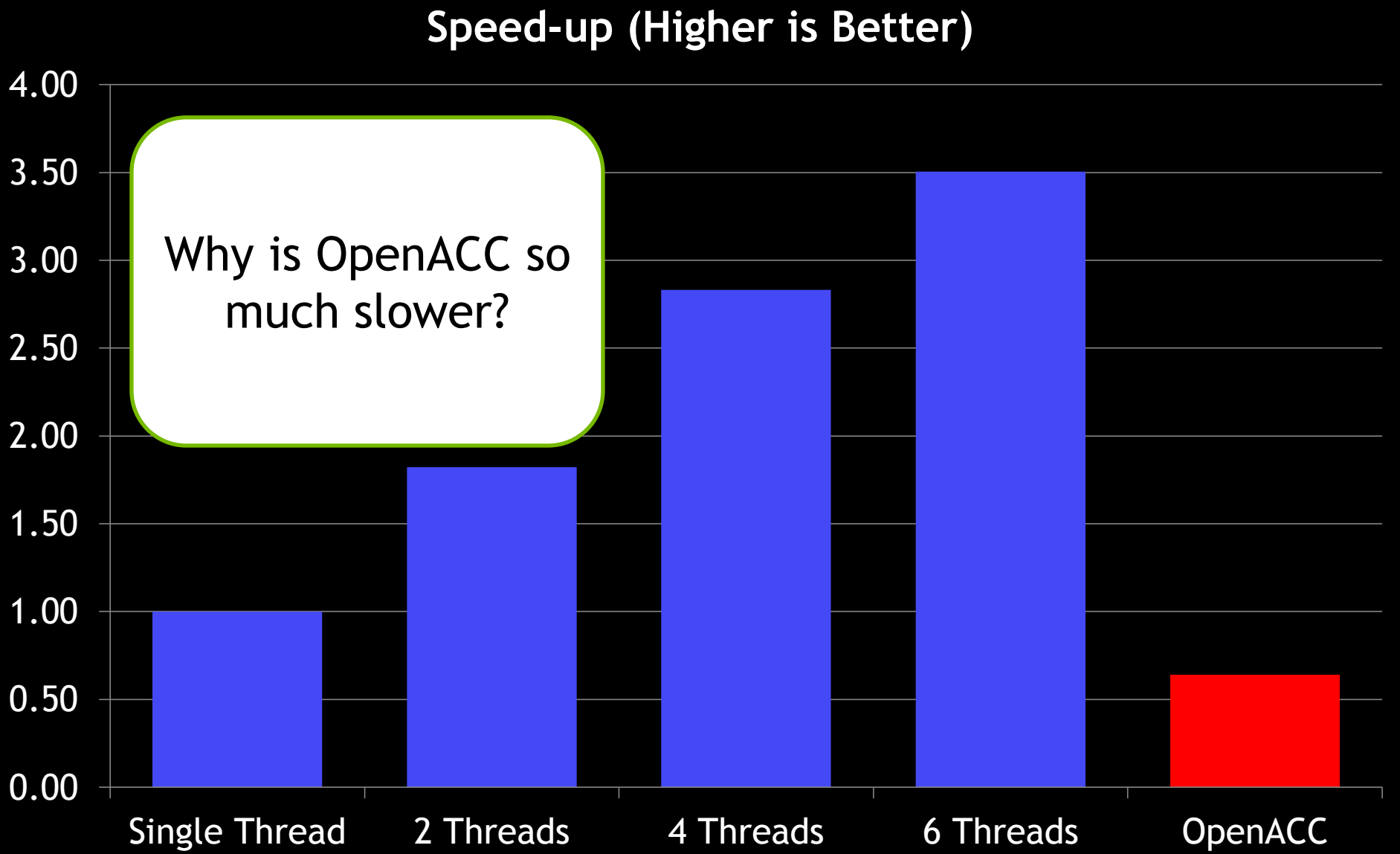
`KERNELS`

- 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.

BUILDING THE CODE

```
$ pgcc -acc -ta=nvidia:5.5,kepler -Minfo=accel -o laplace2d_acc laplace2d.c
main:
     56, Accelerator kernel generated
         57, #pragma acc loop gang /* blockIdx.x */
         59, #pragma acc loop vector(256) /* threadIdx.x */
     56, Generating present_or_copyout(Anew[1:4094][1:4094])
         Generating present_or_copyin(A[0:][0:])
         Generating NVIDIA code
         Generating compute capability 3.0 binary
     59, Loop is parallelizable
     63, Max reduction generated for error
     68, Accelerator kernel generated
         69, #pragma acc loop gang /* blockIdx.x */
         71, #pragma acc loop vector(256) /* threadIdx.x */
     68, Generating present_or_copyin(Anew[1:4094][1:4094])
         Generating present_or_copyout(A[1:4094][1:4094])
         Generating NVIDIA code
         Generating compute capability 3.0 binary
     71, Loop is parallelizable
```



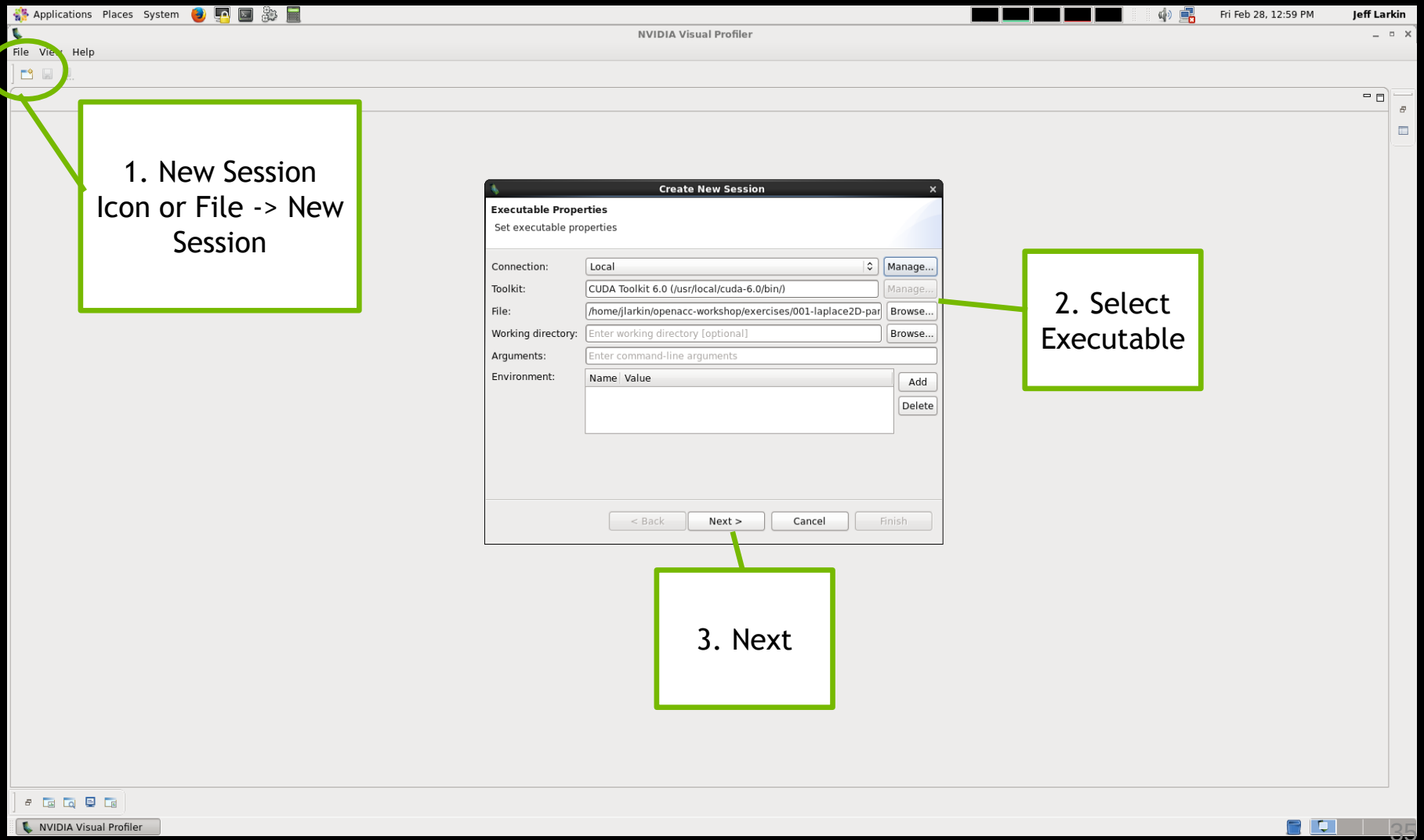
PROFILING AN OPENACC APPLICATION

```
$ nvprof ./laplace2d_acc
Jacobi relaxation Calculation: 4096 x 4096 mesh
==10619== NVPROF is profiling process 10619, command: ./laplace2d_acc
    0, 0.250000
  100, 0.002397
  200, 0.001204
  300, 0.000804
  400, 0.000603
  500, 0.000483
  600, 0.000403
  700, 0.000345
  800, 0.000302
  900, 0.000269
 total: 134.259326 s
==10619== Profiling application: ./laplace2d_acc
==10619== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 49.59%  44.0095s     17000  2.5888ms     864ns  2.9822ms  [CUDA memcpy HtoD]
 45.06%  39.9921s     17000  2.3525ms  2.4960us  2.7687ms  [CUDA memcpy DtoH]
  2.95%  2.61622s      1000  2.6162ms  2.6044ms  2.6319ms  main_56_gpu
  2.39%  2.11884s      1000  2.1188ms  2.1023ms  2.1374ms  main_68_gpu
  0.01%  12.431ms      1000  12.430us  12.192us  12.736us  main_63_gpu_red
```

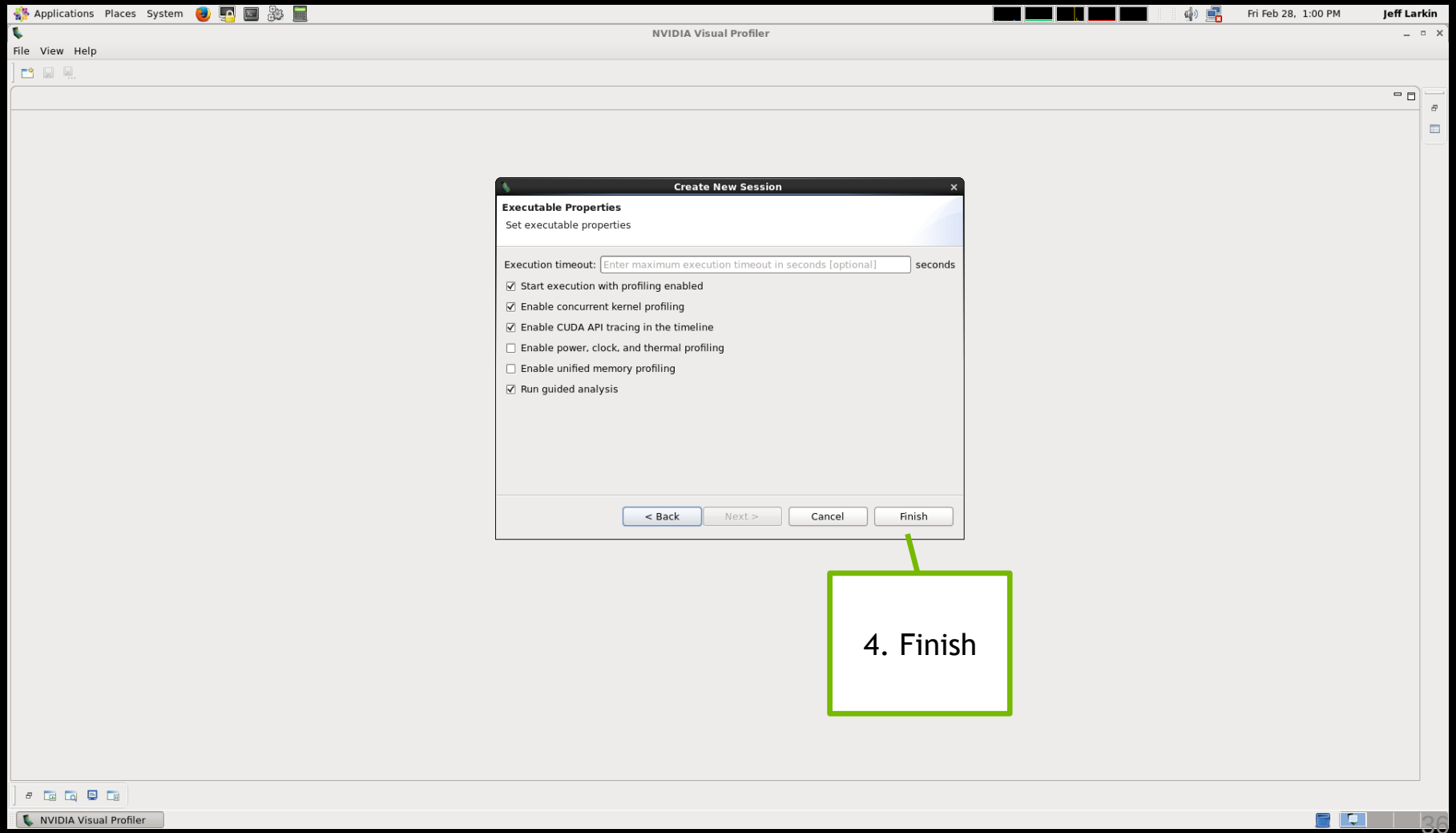
PROFILING AN OPENACC APPLICATION

```
$ PGI_ACC_TIME=1 ./laplace2d_acc
Accelerator Kernel Timing data
/home/jlarkin/openacc-workshop/exercises/001-laplace2D-parallel/laplace2d.c
  main  NVIDIA  devicenum=0
    time(us): 89,242,926
    56: compute region reached 1000 times
        56: data copyin reached 8000 times
             device time(us): total=22,334,806 max=3,022 min=2,747 avg=2,791
        56: kernel launched 1000 times
            grid: [4094]  block: [256]
             device time(us): total=2,643,298 max=2,841 min=2,629 avg=2,643
            elapsed time(us): total=2,654,729 max=2,855 min=2,640 avg=2,654
        56: reduction kernel launched 1000 times
            grid: [1]  block: [256]
             device time(us): total=19,182 max=75 min=17 avg=19
            elapsed time(us): total=29,669 max=87 min=28 avg=29
        68: data copyout reached 8000 times
             device time(us): total=20,100,500 max=2,797 min=2,494 avg=2,512
    68: compute region reached 1000 times
        68: data copyin reached 8000 times
             device time(us): total=21,902,474 max=2,939 min=2,724 avg=2,737
        68: kernel launched 1000 times
            grid: [4094]  block: [256]
             device time(us): total=2,144,960 max=2,202 min=2,130 avg=2,144
            elapsed time(us): total=2,157,152 max=2,215 min=2,143 avg=2,157
        77: data copyout reached 8000 times
             device time(us): total=20,097,706 max=2,809 min=2,494 avg=2,512
```

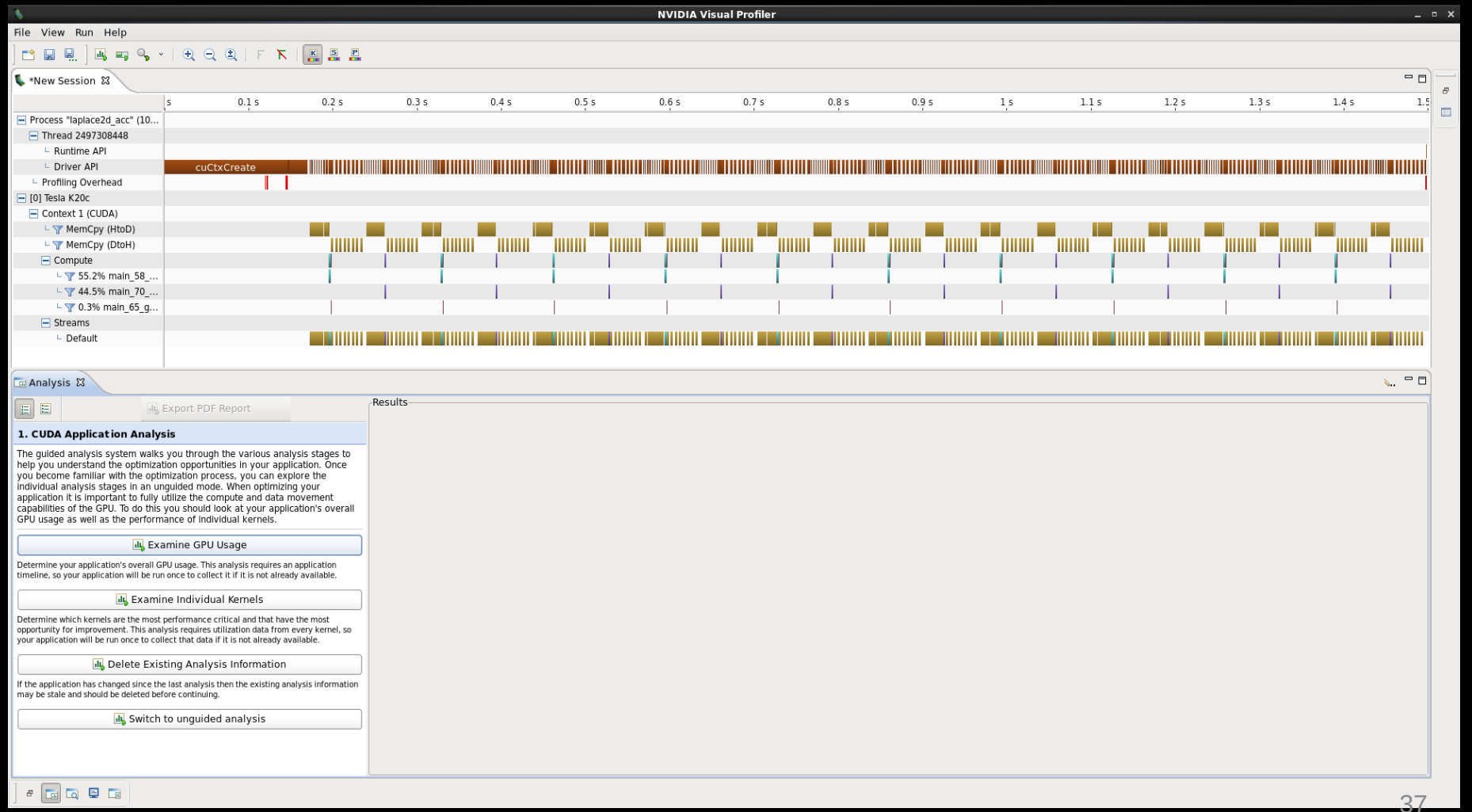
NVIDIA VISUAL PROFILER: NEW SESSION



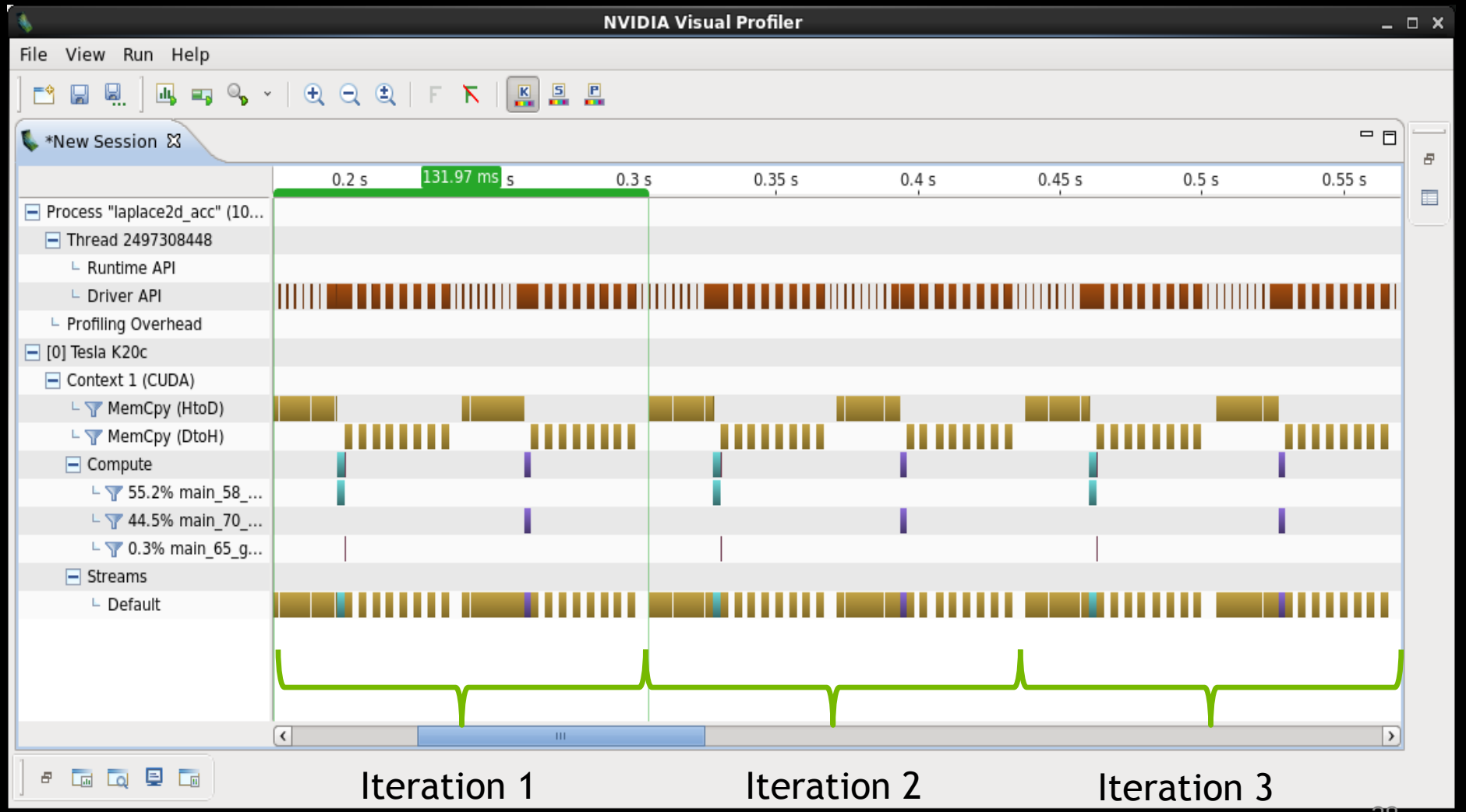
NVIDIA VISUAL PROFILER: NEW SESSION (CONT.)



NVIDIA VISUAL PROFILER: GUIDED ANALYSIS

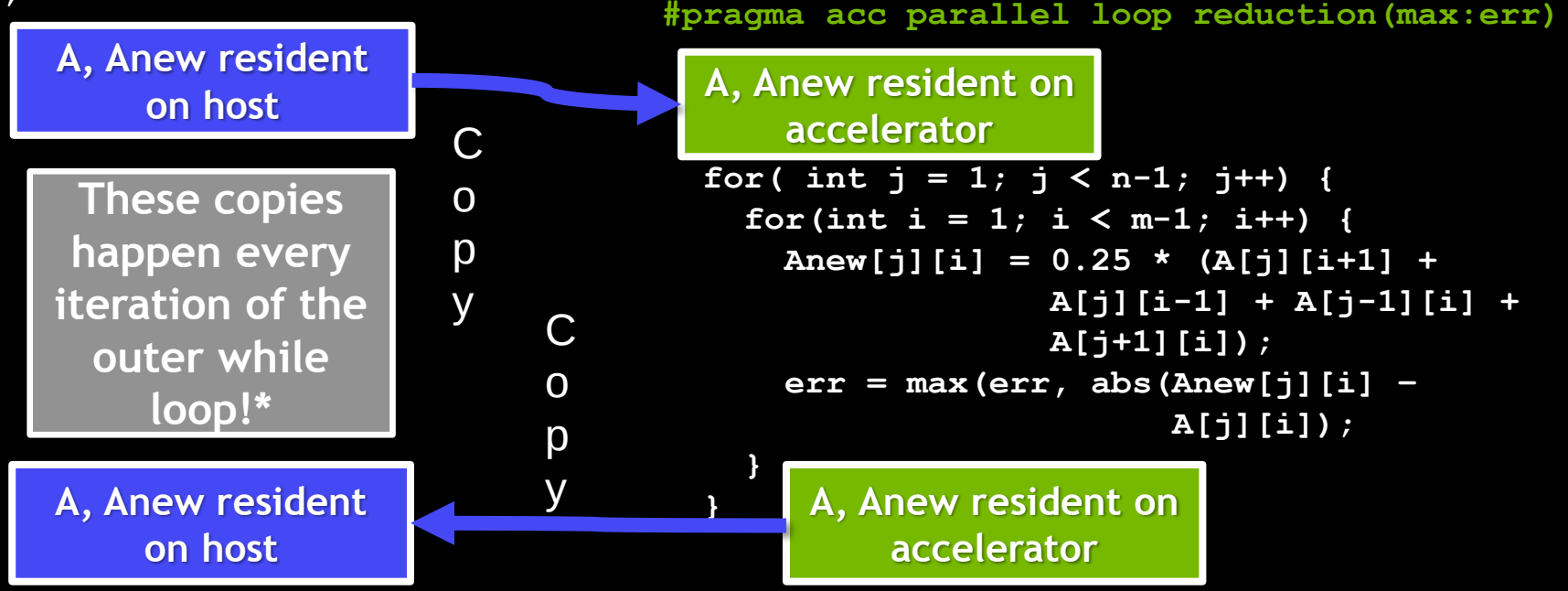


VISUAL PROFILER: EXCESS DATA MOVEMENT



EXCESSIVE DATA TRANSFERS

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

    ...
}
```



And note that there are two #pragma acc parallel, so there are 4 copies per while loop iteration!

EXPRESS DATA LOCALITY

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

```
#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))
```

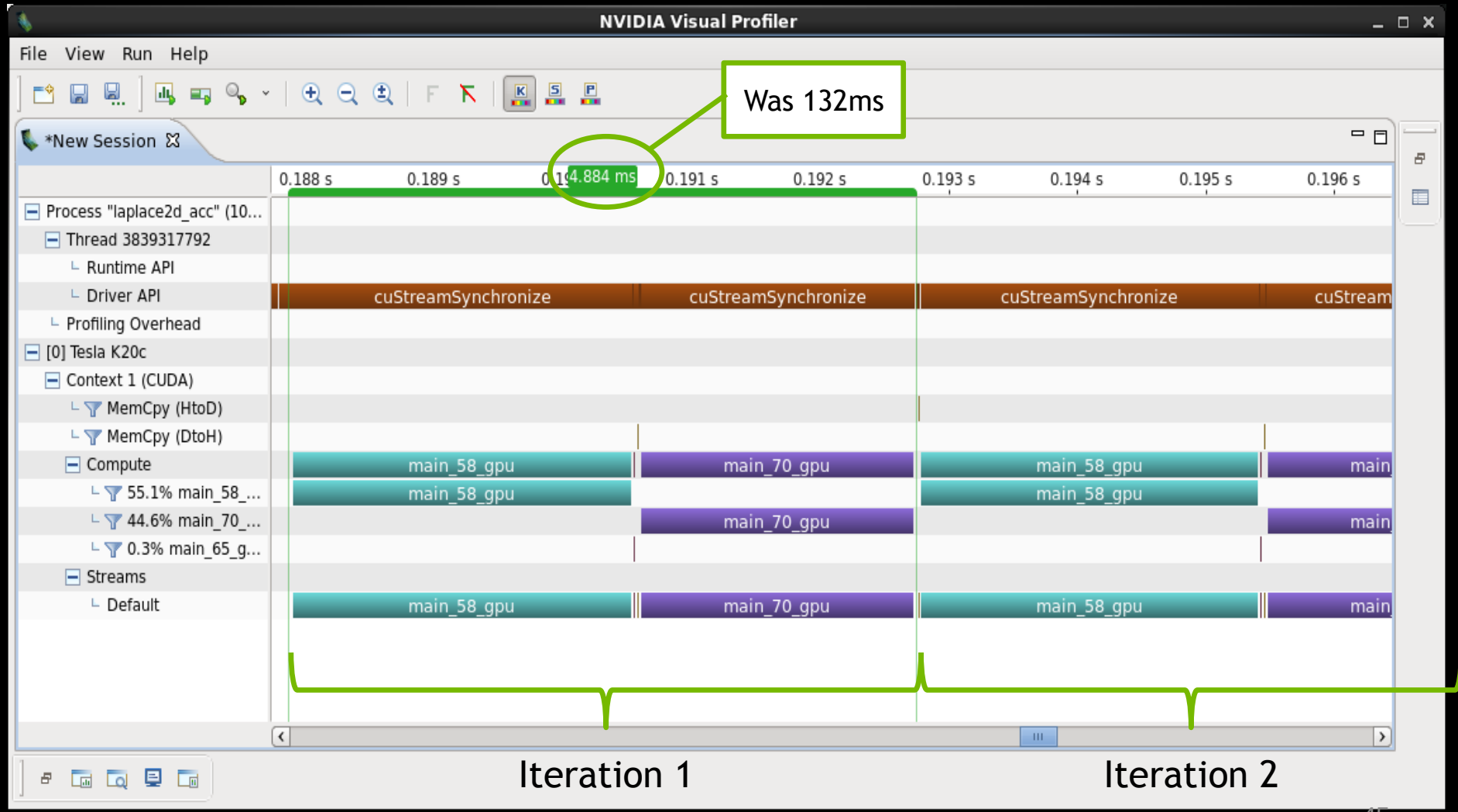
- Note: data clauses can be used on **`data`**, **`parallel`**, or **`kernels`**

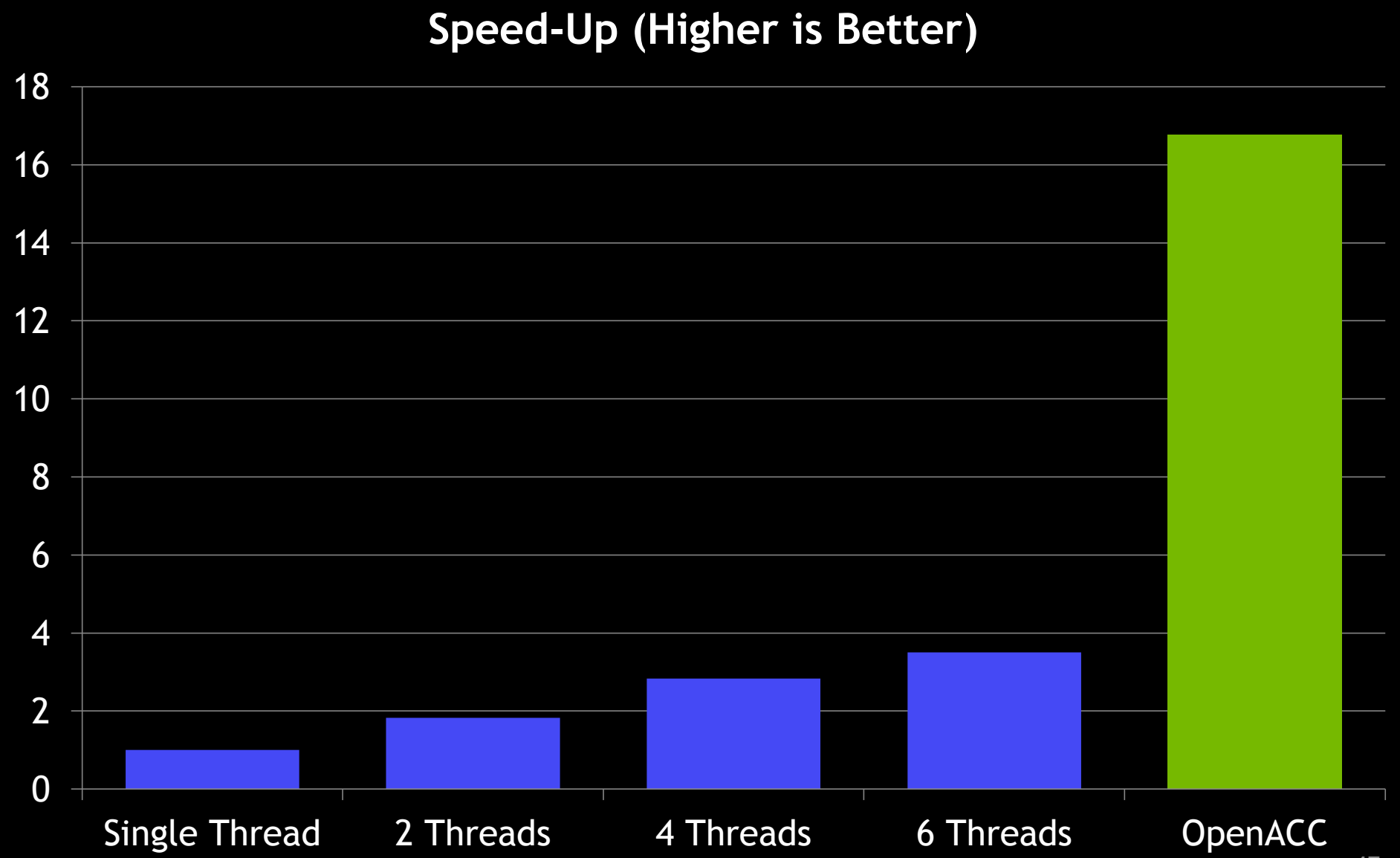
JACOBI ITERATION: OPENACC C CODE

```
#pragma acc data copy(A) create(Anew)
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++;
}
```

Copy A to/from the accelerator only when needed.
Create Anew as a device temporary.

VISUAL PROFILER: REDUCED DATA MOVEMENT





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.

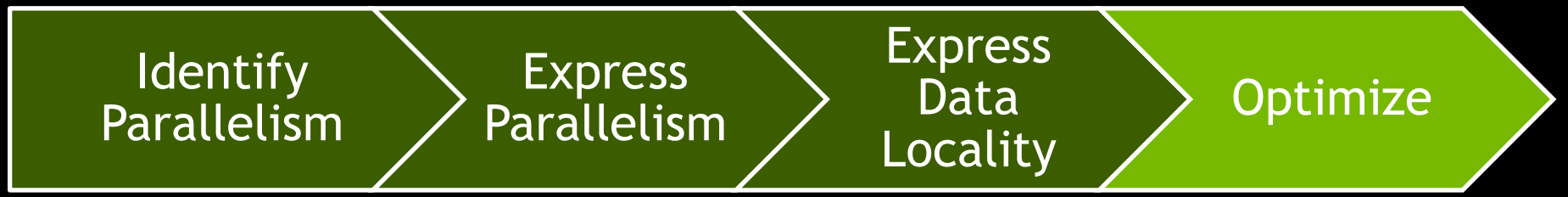
Since the declaration of A is now in a higher scope, it's necessary to shape A in the present clause.

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;
    ...
  }
}
```

Identify Parallelism > Express Parallelism > Express Data Locality > Optimize



OPTIMIZE

OPENACC `loop` DIRECTIVE

- The `loop` directive provides the compiler with additional information for the next loop.

Notable Clauses:

- `private & reduction`
- `gang/worker/vector/seq`
- `collapse`
- `tile`

OPENACC `LOOP` DIRECTIVE: PRIVATE & REDUCTION

- The private and reduction clauses are not optimization clauses, they may be required for correctness.

- `private` – A copy of the variable is made for each loop iteration
- `reduction` – A reduction is performed on the listed variables.
  - Supports +, *, max, min, and various logical operations

OPENACC `LOOP` DIRECTIVE: LOOP PARALLELISM

- OpenACC supports 3 levels of parallelism (gang, worker, vector).

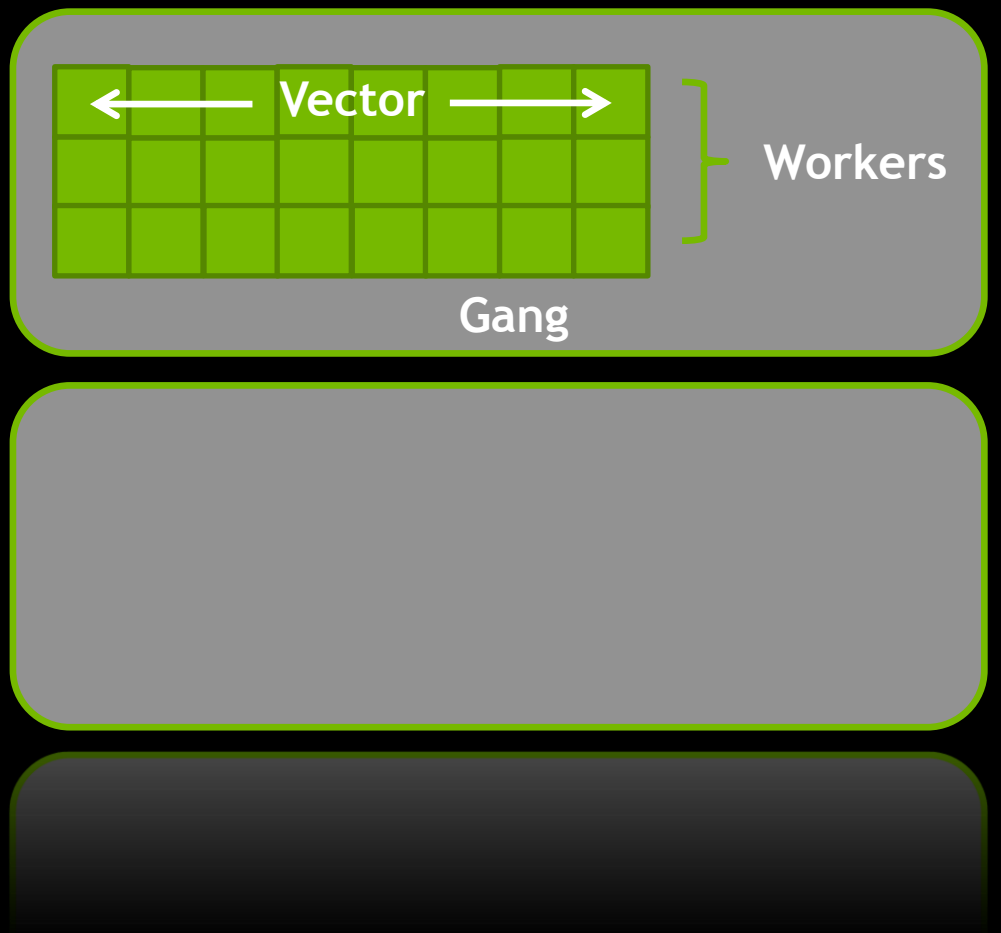
Parallel Loop

```
#pragma acc parallel loop num_gangs(X)
num_workers(Y) vector_length(Z)
  #pragma acc loop seq
```

Kernels

```
#pragma acc kernels
  #pragma acc loop gang(X)
  #pragma acc loop worker(Y)
  #pragma acc loop vector(Z)
  #pragma acc loop seq
```

OPENACC: 3 LEVELS OF PARALLELISM



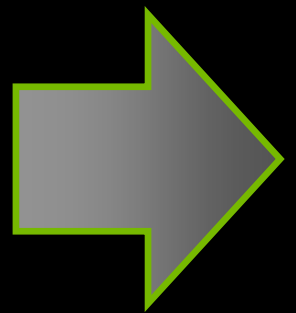
- *Vector* threads work in lockstep (SIMD/SIMT parallelism)
- *Workers* have 1 or more vectors.
- *Gangs* have 1 or more workers and share resources (such as cache, the streaming multiprocessor, etc.)
- Multiple gangs work independently of each other

On NVIDIA GPUs, vector lengths should be a multiple of 32 and the worker dimension can usually be ignored.

OPENACC LOOP DIRECTIVE: `COLLAPSE(N)`

- The **`collapse`** directive instructs the compiler to convert the next *N* loops into one flattened loop.
  - This is especially useful when some loops lack enough iterations to make effective use of the GPU

```
#pragma acc parallel loop collapse(2)
for(int i=0; i<N; i++)
{
  for(int j=0; j<4; j++)
    …
}
```



```
#pragma acc parallel for
for(int ij=0; ij<4*N; ij++)
{
  …
}
```

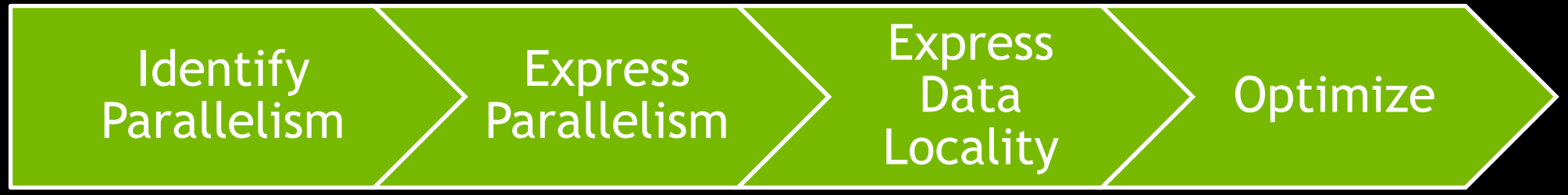
Identify Parallelism → Express Parallelism → Express Data Locality → Optimize

OPENACC LOOP DIRECTIVE: `TILE(N[,M])`

- The `tile` directive instructs the compiler to block the following loops to better exploit locality and data reuse.
  - The compiler will generate additional tile loops according to the tile dimensions provided.

```
#pragma acc parallel loop
for(int i=0; i < n; i++)
  #pragma acc loop  tile(8,8)
  for(int j=0; j < n; j++)
     for(int k=0; k < n; k++)
       c[i][j] = c[i][j] + a[i][j] * b[k][j];
```

Identify Parallelism > Express Parallelism > Express Data Locality > Optimize



1. Identify Parallelism
   - What important parts of the code have available parallelism?
2. Express Parallelism
   - 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. Express Locality
   - The programmer will *always* know better than the compiler what data movement is unnecessary.
4. Optimize
   - 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*.

NEXT STEPS

NEXT STEPS: TRY OPENACC

- GTC14 has multiple hands-on labs for trying OpenACC
  - S4803 - Getting Started with OpenACC – Wednesday 5PM
  - S4796 - Parallel Programming: OpenACC Profiling – Thursday 2PM
- Get a PGI Trial license for your machine
- NVIDIA Developer Blogs
  - Parallel Forall Blog
  - CUDACasts Screencast Series

MORE AT GTC14

- S4200 - Advanced Accelerated Computing Using Directives
  - 1PM in this room
  - Continues where this talk has left off
- S4514 - Panel on Compiler Directives for Accelerated Computing
  - Wednesday @ 4PM
  - Wednesday afternoon in LL20C has all compiler-directives talks
- Check the sessions agenda for more talks on compiler directives

Please remember to fill out your surveys.