INTRODUCTION TO COMPILER DIRECTIVES WITH OPENACC

DR. CHRISTOPH ANGERER, NVIDIA

*) THANKS TO JEFF LARKIN, NVIDIA, FOR THE SLIDES

3 APPROACHES TO GPU PROGRAMMING

Applications

Libraries	Compiler Directives	Programming Languages
Easy to use Most Performance	Easy to use Portable code	Most Performance Most Flexibility

AGENDA

- What are Compiler Directives?
- Accelerating Applications with OpenACC
  - Identifying Available Parallelism
  - Exposing Parallelism
  - Optimizing Data Locality
- Coffee(?)

WHAT ARE COMPILER DIRECTIVES?

WHAT ARE COMPILER DIRECTIVES?

```
int main() {

  do_serial_stuff()


  for(int i=0; i < BIGN; i++)
  {
    …compute intensive work
  }

  do_more_serial_stuff();

}
```

Programmer inserts compiler hints.

Execution Begins on the CPU.

Compiler Generates GPU Code

Data and Execution moves to the GPU.

Data and Execution returns to the CPU.

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
- **Portable:** GPU Directives represent parallelism at a high level, allowing portability to a wide range of architectures with the same code.

OpenACC® DIRECTIVES FOR ACCELERATORS

OPENACC MEMBERS AND PARTNERS

TOTAL

ROGUE WAVE SOFTWARE

OAK RIDGE National Laboratory

PGI

TECHNISCHE UNIVERSITÄT DRESDEN

CSCS

INDIANA

Sandia National Laboratories

LSU LOUISIANA STATE UNIVERSITY

TOKYO TECH Pursuing Excellence

NOAA

CRAY THE SUPERCOMPUTER COMPANY

NVIDIA

mentor embedded

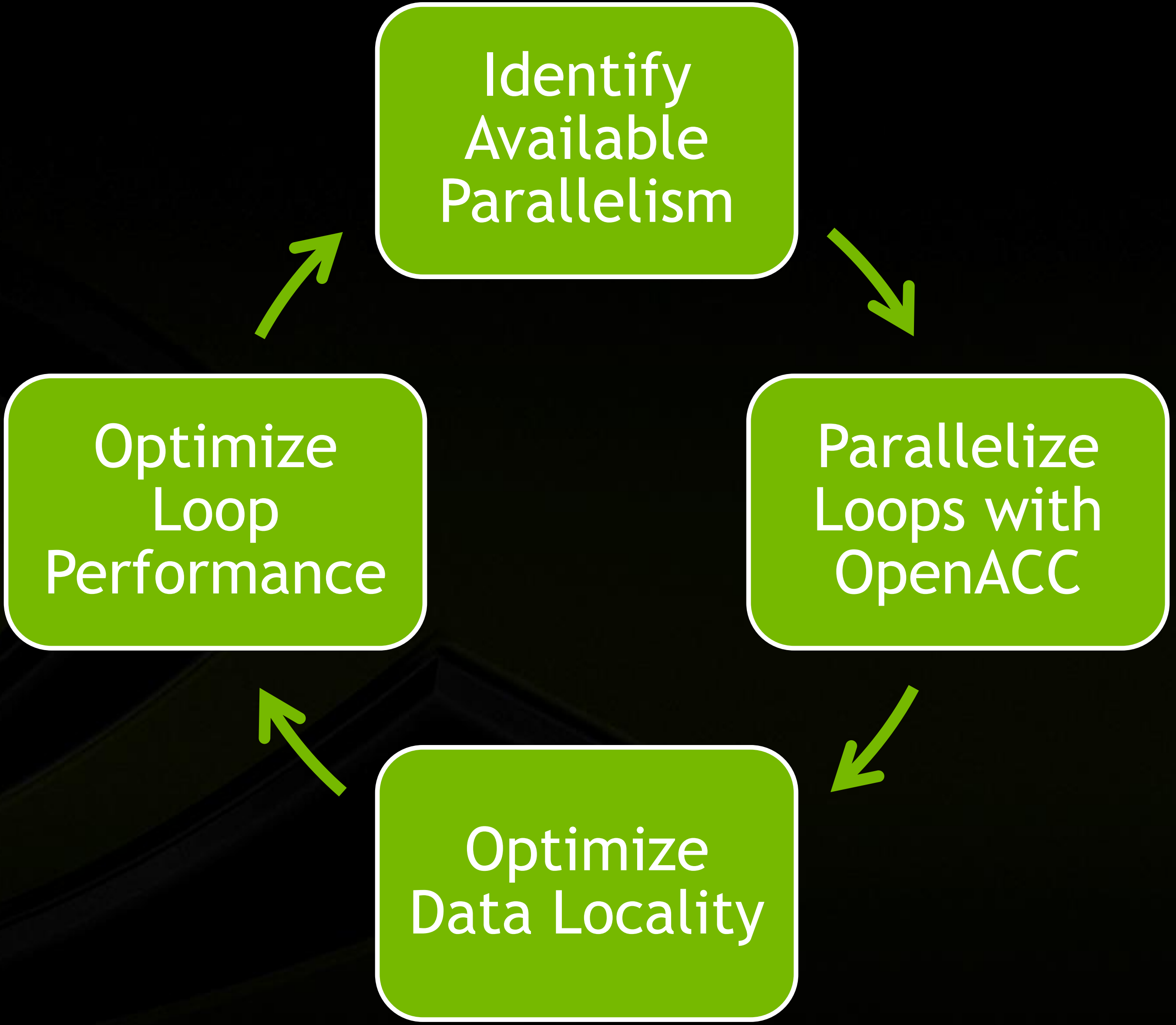
allinea

PathScale

AMD

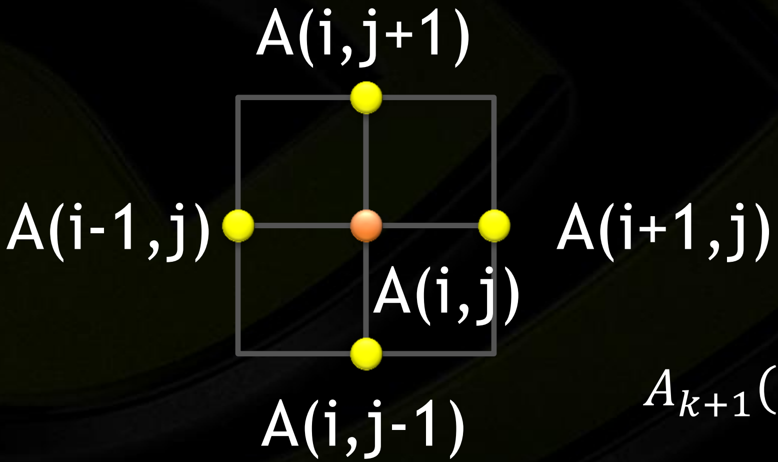
epcc

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: $\nabla^2 \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}) = \mathbf{0}$



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

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++;
}
```

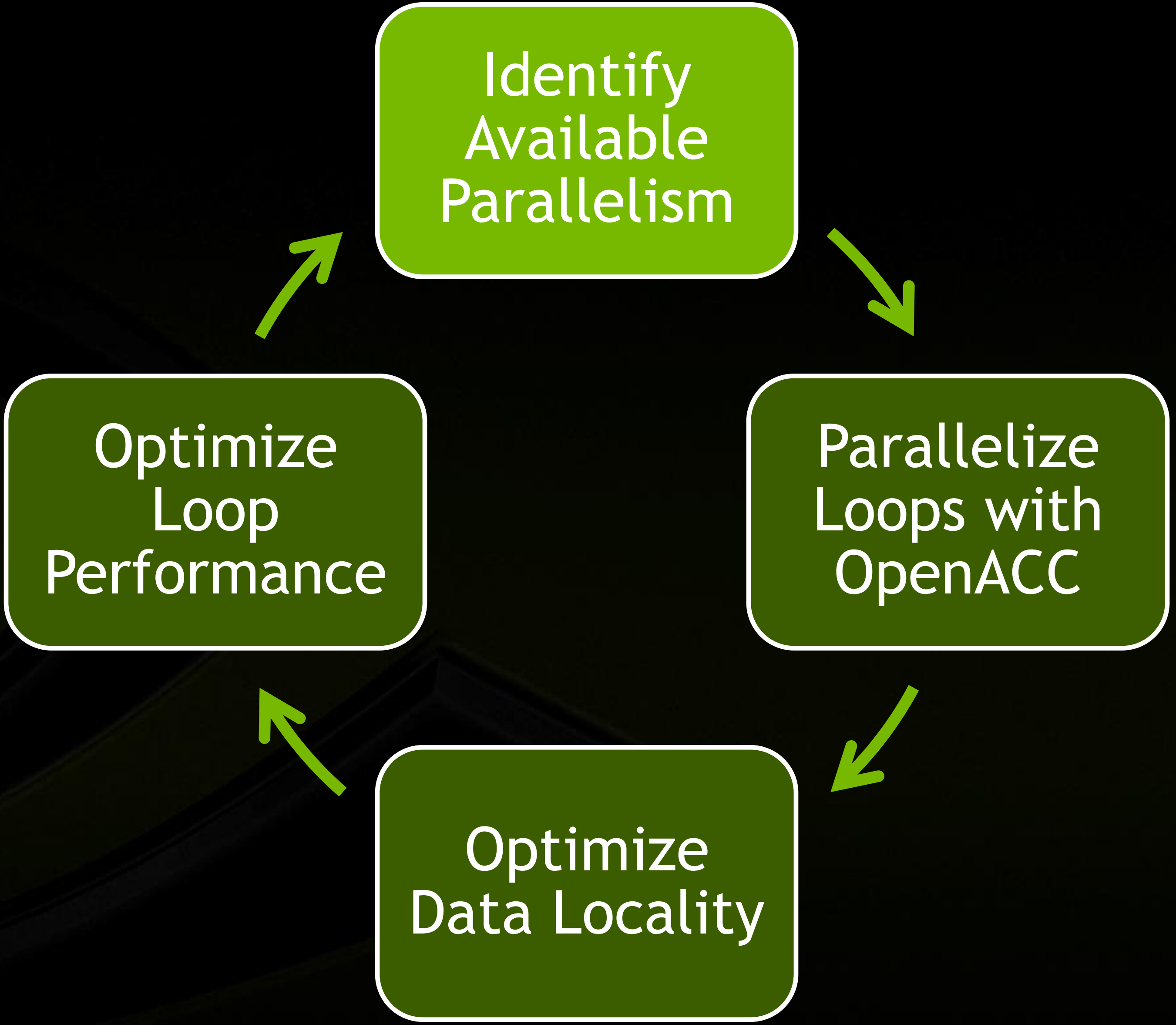
Iterate until converged

Iterate across matrix elements

Calculate new value from neighbors

Compute max error for convergence

Swap input/output arrays

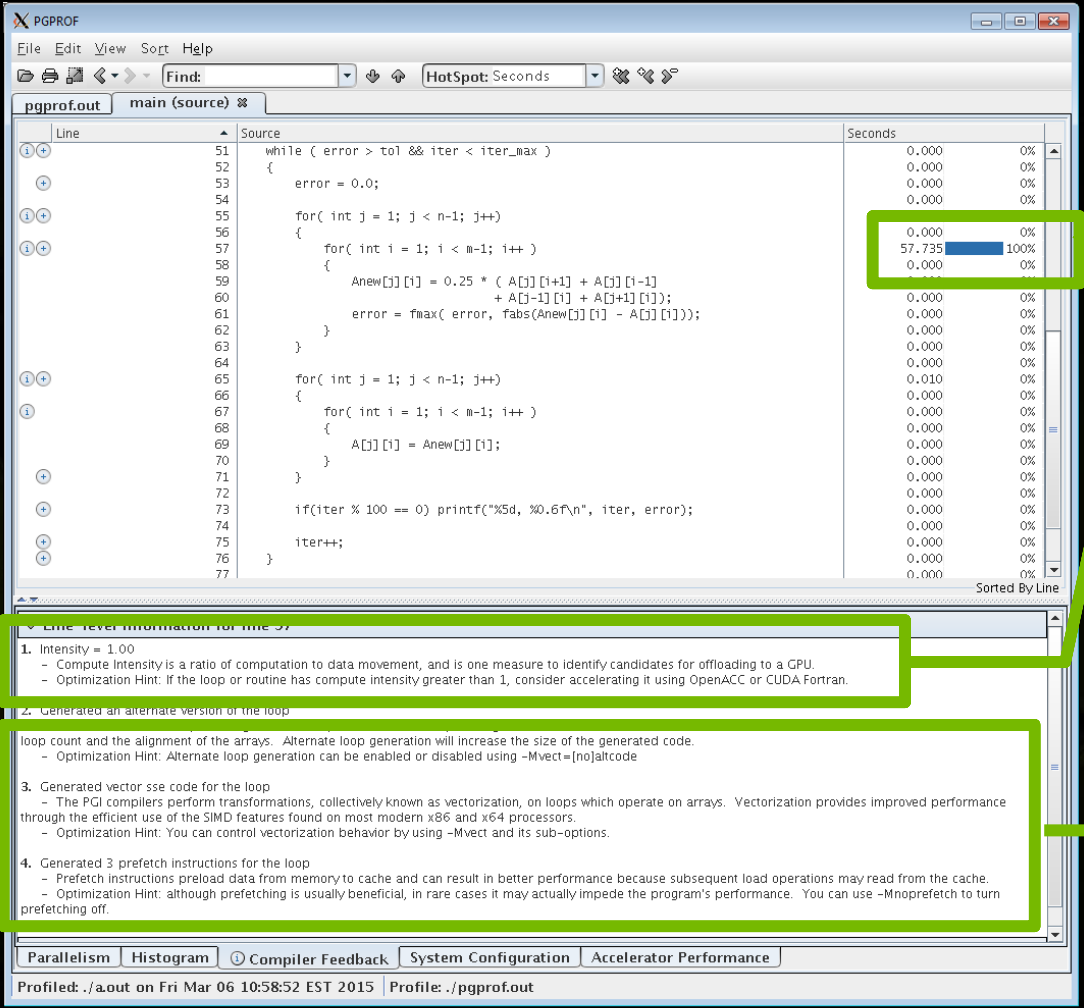

Identify Available Parallelism
Parallelize Loops with OpenACC
Optimize Data Locality
Optimize Loop Performance

IDENTIFY AVAILABLE PARALLELISM

- A variety of profiling tools are available:
  - gprof, pgprof, Vampir, Score-p, HPCToolkit, CrayPAT, …
- Using the tool of your choice, obtain an application profile to identify hotspots
- Since we're using PGI, I'll use pgprof

```
$  pgcc -fast -Minfo=all -Mprof=ccff laplace2d.c
main:
     40, Loop not fused: function call before adjacent loop
         Generated vector sse code for the loop
     57, Generated an alternate version of the loop
         Generated vector sse code for the loop
         Generated 3 prefetch instructions for the loop
     67, Memory copy idiom, loop replaced by call to __c_mcopy8
$ pgcollect ./a.out
$ pgprof -exe ./a.out
```

IDENTIFY PARALLELISM WITH PGPROF



PGPROF informs us:

1. A significant amount of time is spent in the loops at line 56/57.
2. The computational intensity (Calculations/Loads&Stores) is high enough to warrant OpenACC or CUDA.
3. How the code is currently optimized.

NOTE: the compiler recognized the swapping loop as data movement and replaced it with a memcpy, but we know it's expensive too.

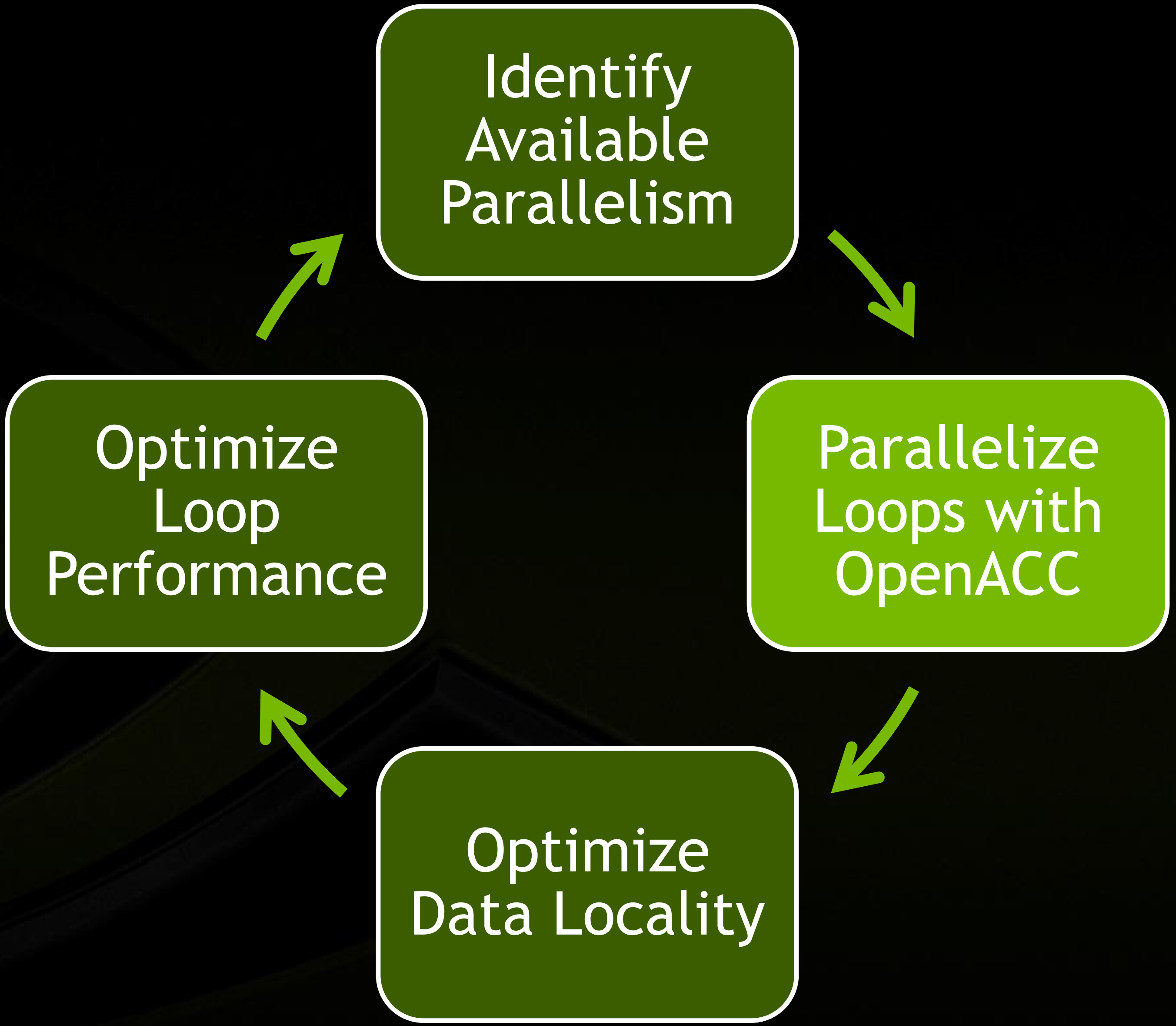
IDENTIFY PARALLELISM

```
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
```

Don’t forget `acc`

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];
}
```

Parallel kernel

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.

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

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++;
}
```

Look for parallelism within this region.

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, 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

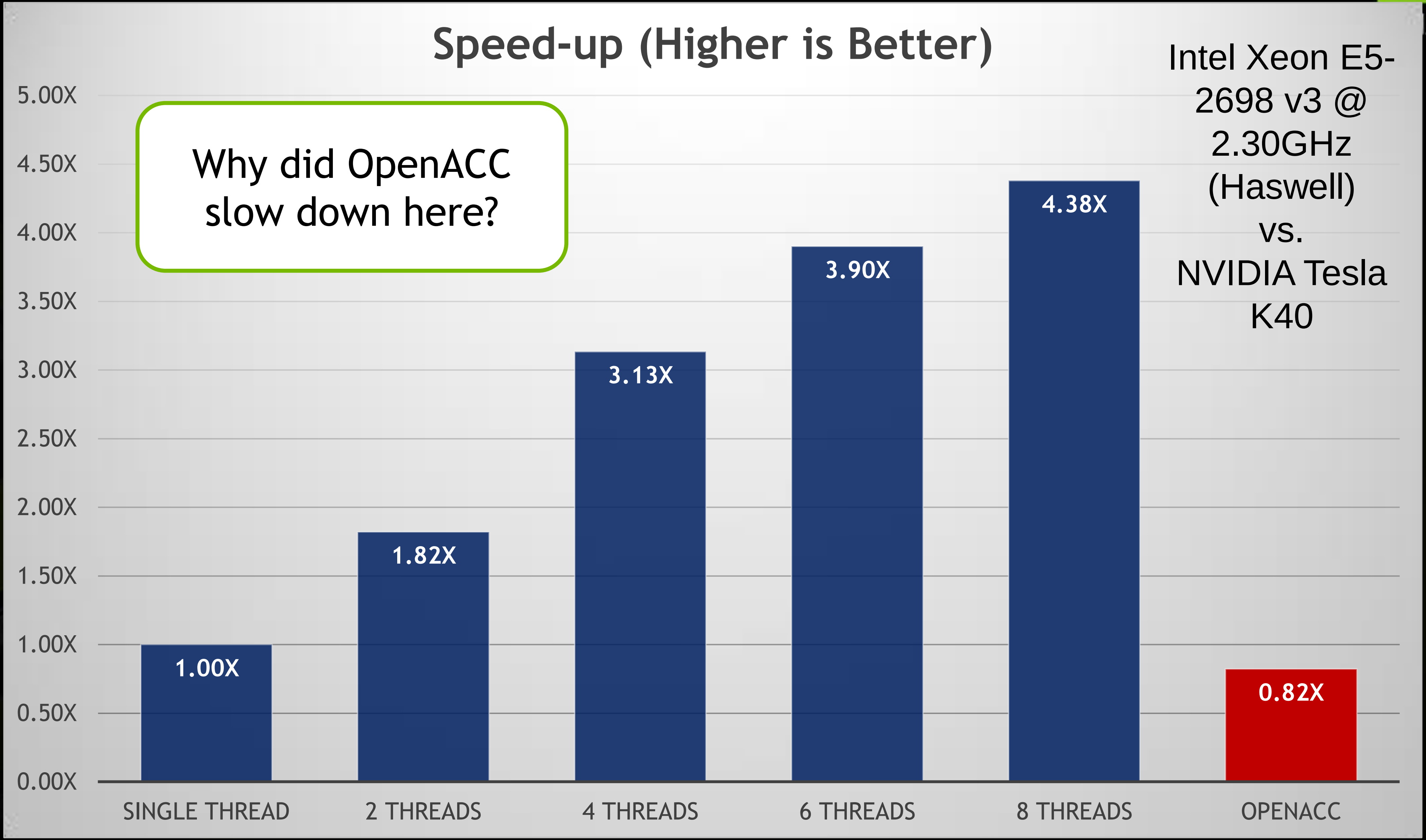
PARALLEL LOOP

- 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.

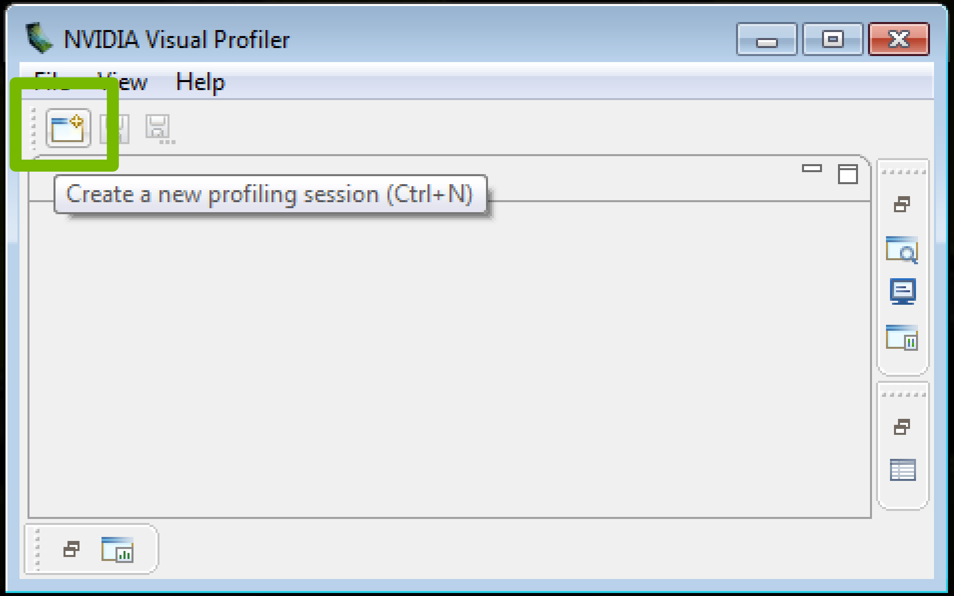
Both approaches are equally valid and can perform equally well.

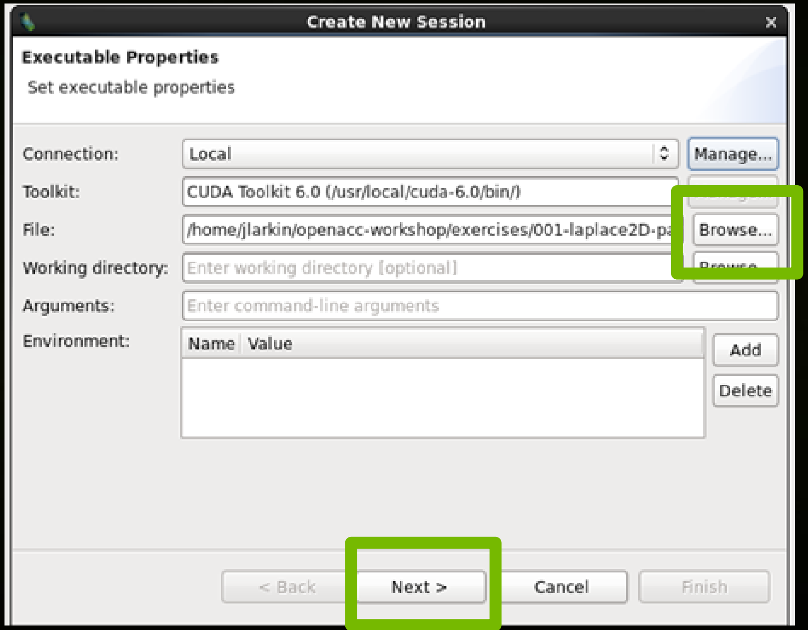


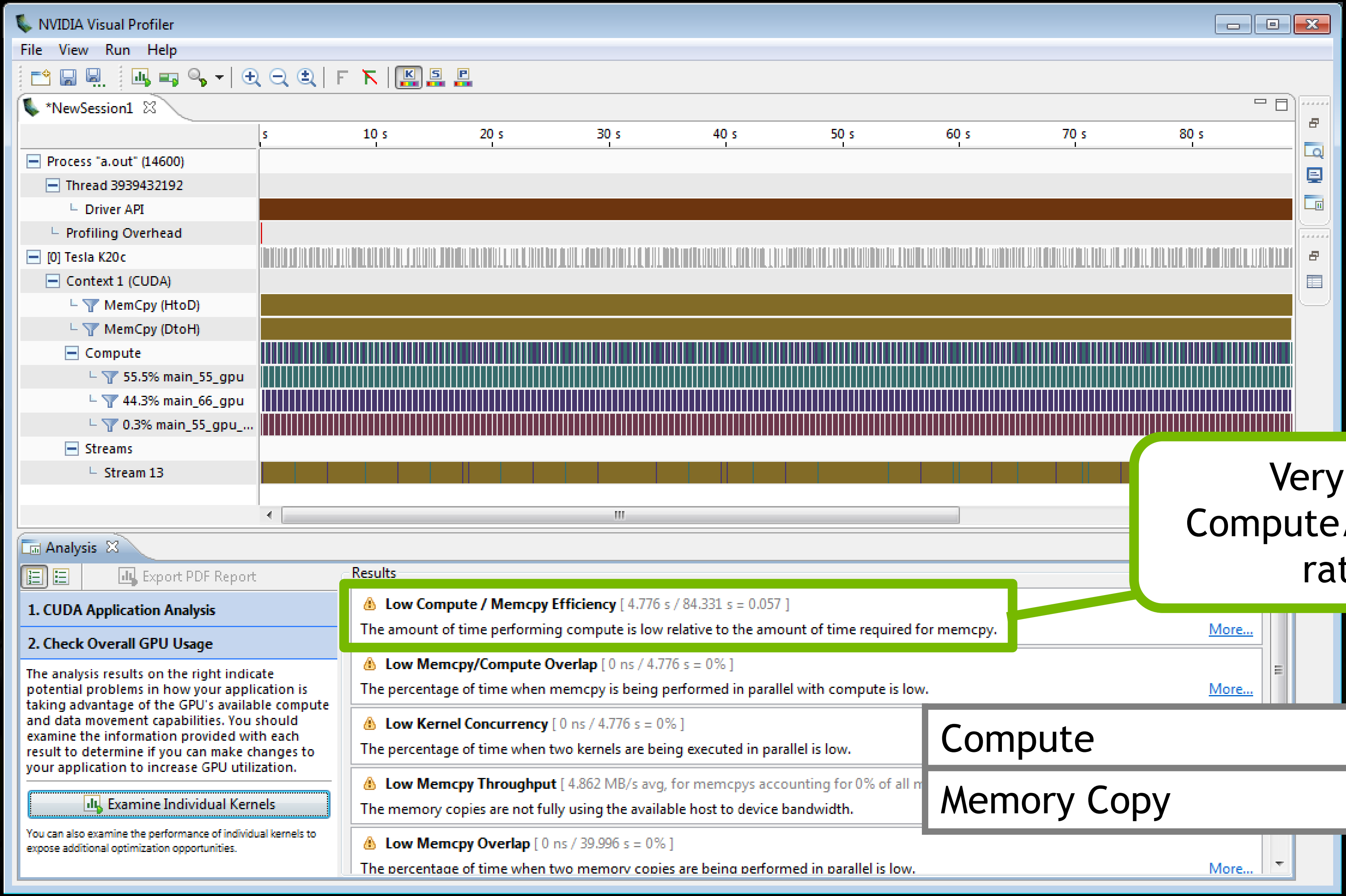
Speed-up (Higher is Better)
Intel Xeon E5-2698 v3 @ 2.30GHz (Haswell) vs. NVIDIA Tesla K40
Why did OpenACC slow down here?
5.00X
4.50X
4.00X
3.50X
3.00X
2.50X
2.00X
1.50X
1.00X
0.50X
0.00X
1.00X
1.82X
3.13X
3.90X
4.38X
0.82X
SINGLE THREAD
2 THREADS
4 THREADS
6 THREADS
8 THREADS
OPENACC

ANALYZING OPENACC PERFORMANCE

- Any tool that supports CUDA can likewise obtain performance information about OpenACC.
- Nvidia Visual Profiler (nvvp) comes with the CUDA Toolkit, so it will be available on any machine with CUDA installed



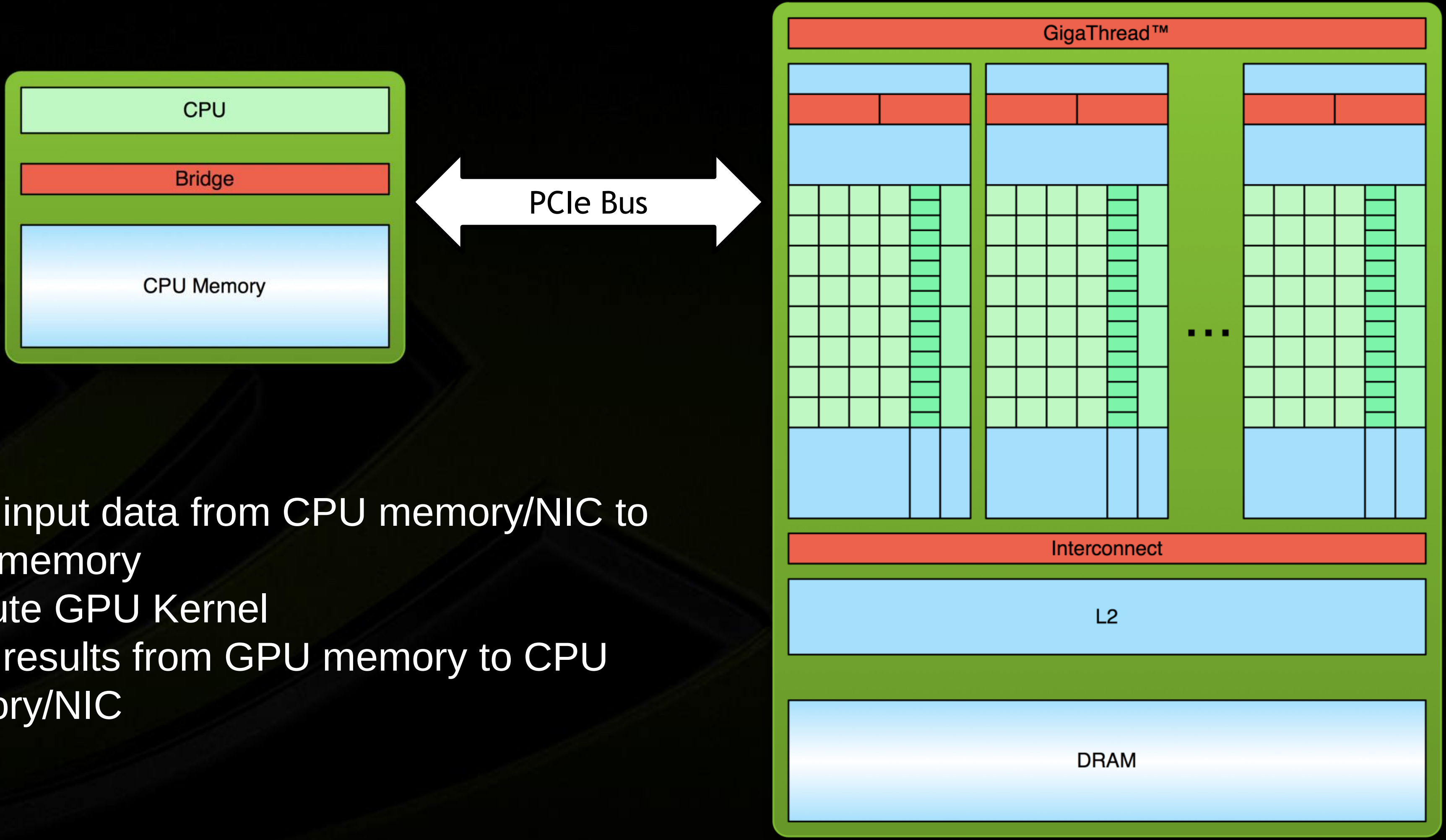




Very low Compute/Memcpy ratio

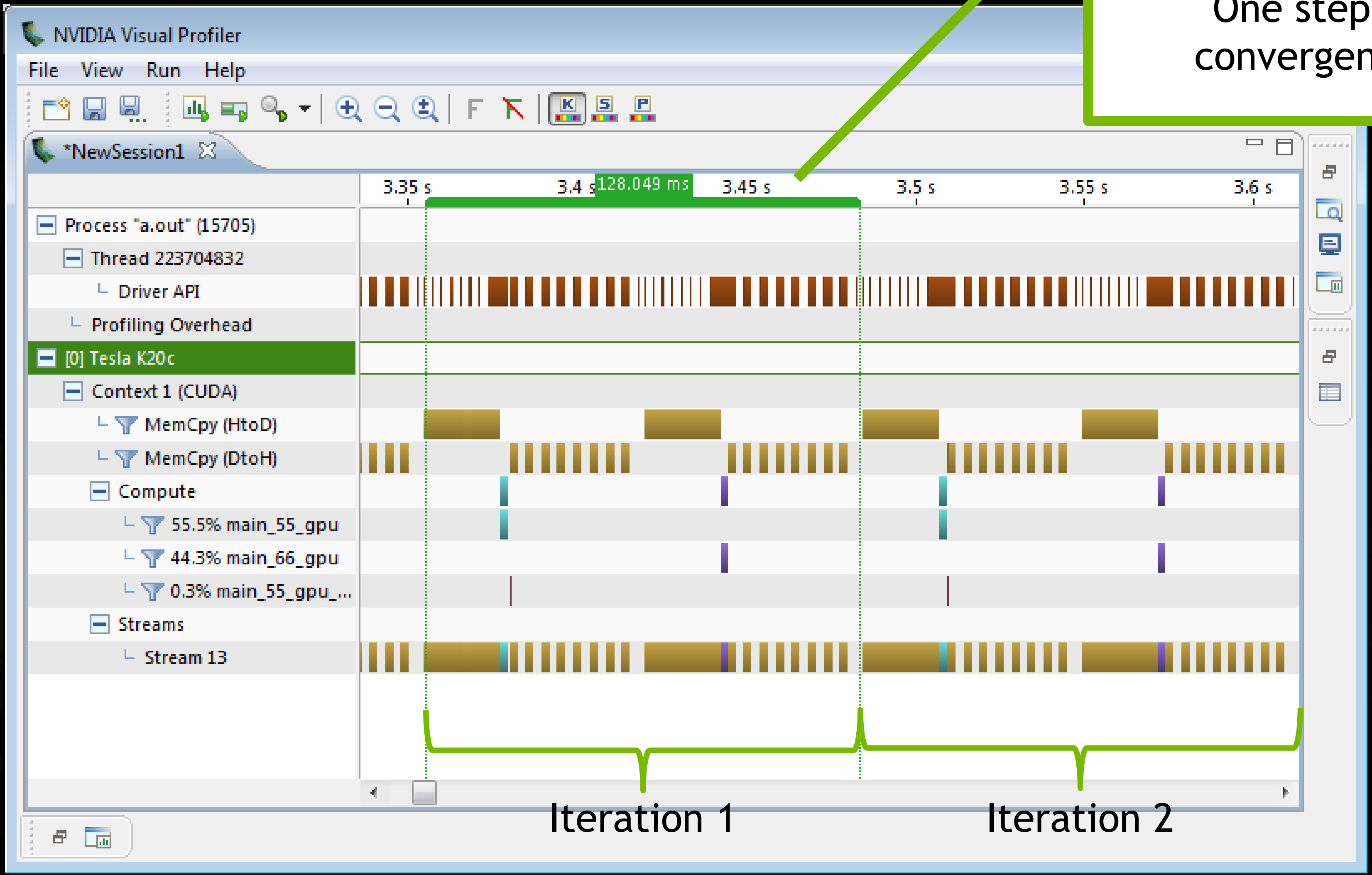
Compute	4.7s
Memory Copy	84.3s

PROCESSING FLOW



1. Copy input data from CPU memory/NIC to GPU memory
2. Execute GPU Kernel
3. Copy results from GPU memory to CPU memory/NIC

One step of the convergence loop



EXCESSIVE DATA TRANSFERS

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

A, Anew resident on host

Copy

These copies happen every iteration of the outer while loop!*

```
#pragma acc parallel loop reduction(max:err)
```

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]);
  }
}
```

A, Anew resident on accelerator

Copy

A, Anew resident on host

```
  ...
}
```

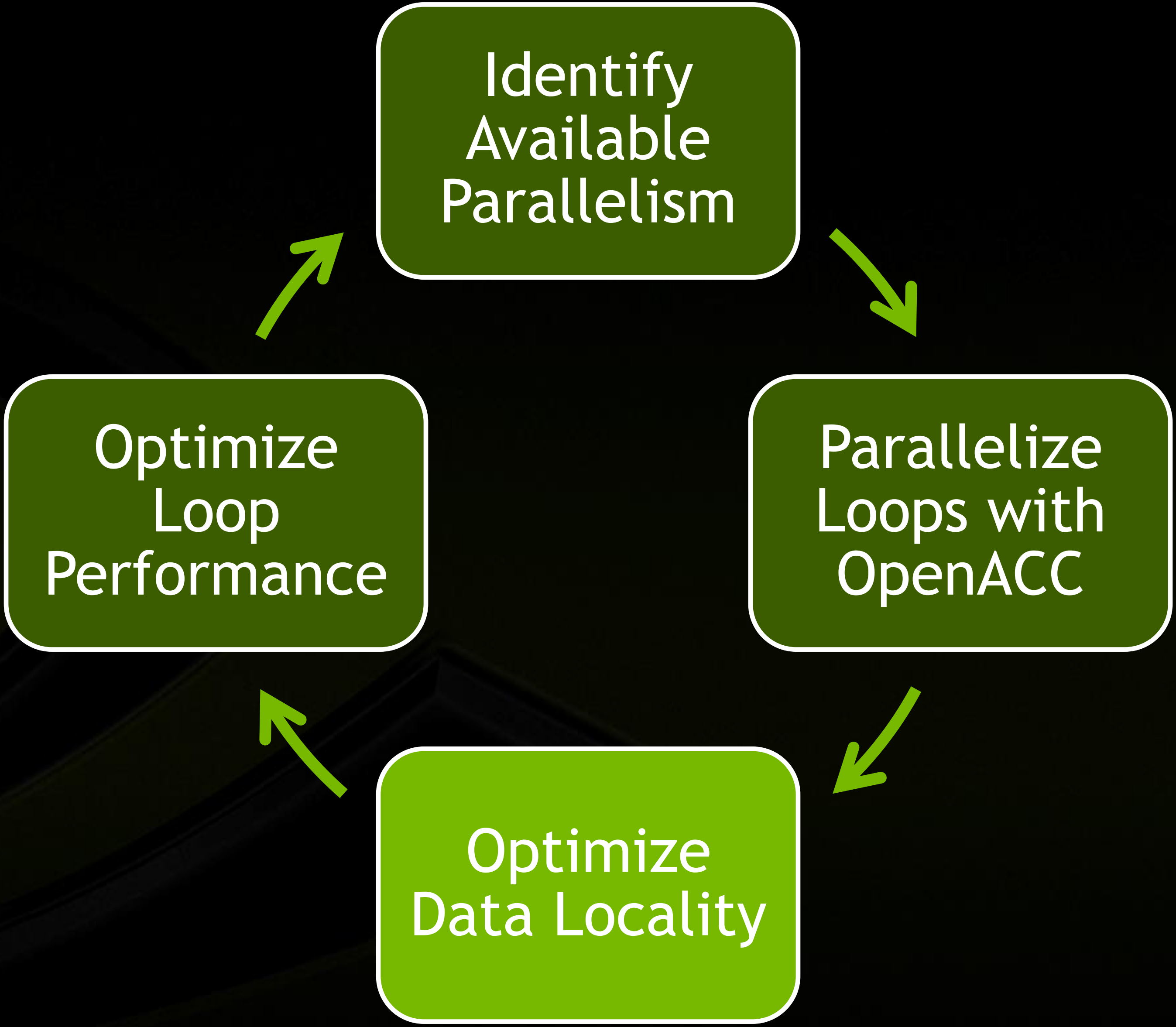
And note that there are two #pragma acc parallel, so there are 4 copies per while loop iteration!

IDENTIFYING DATA LOCALITY

```
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++;
}
```

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`.

The next OpenACC makes present_or_* the default behavior.

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 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++;
}
```

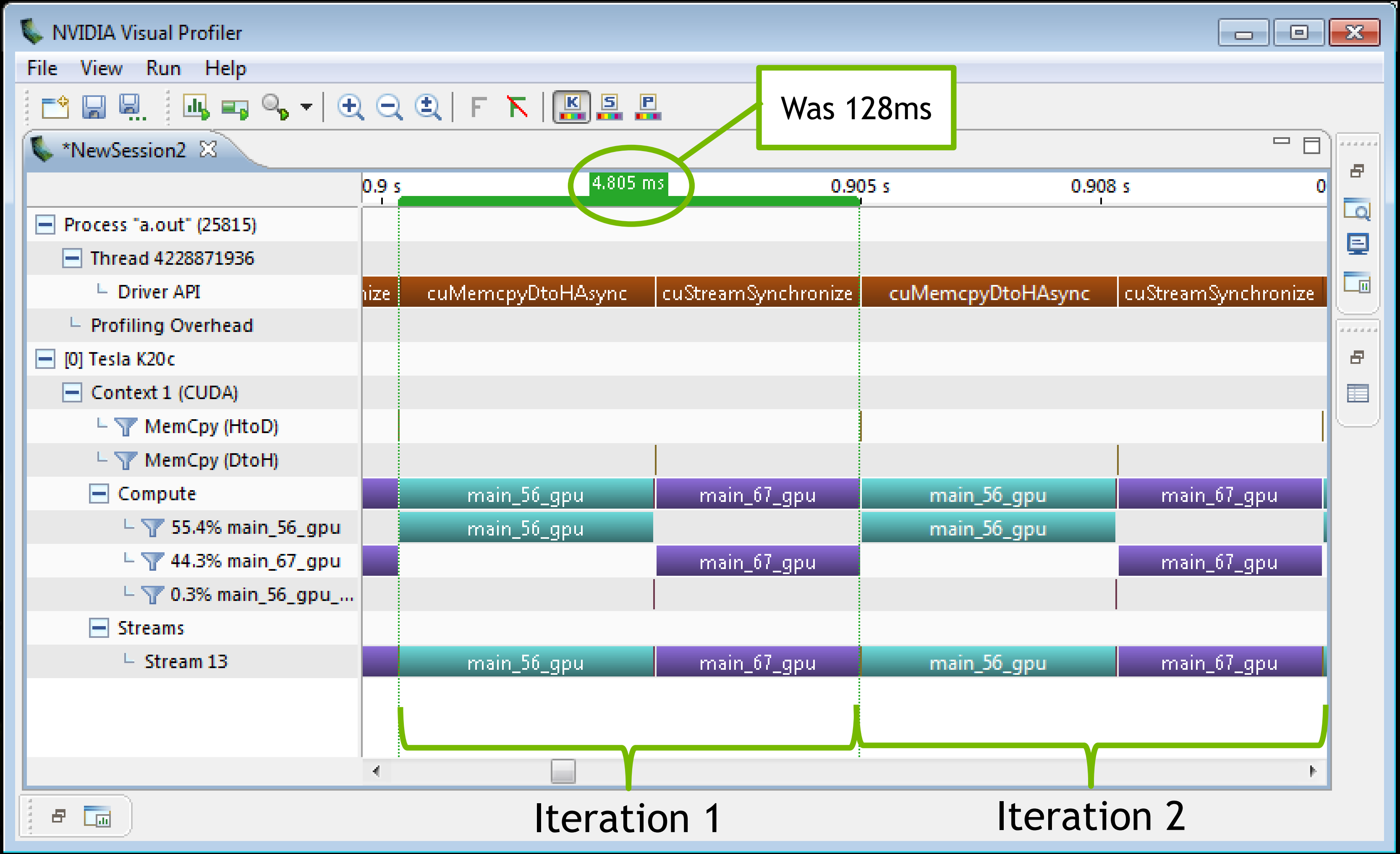
Copy A to/from the accelerator only when needed.

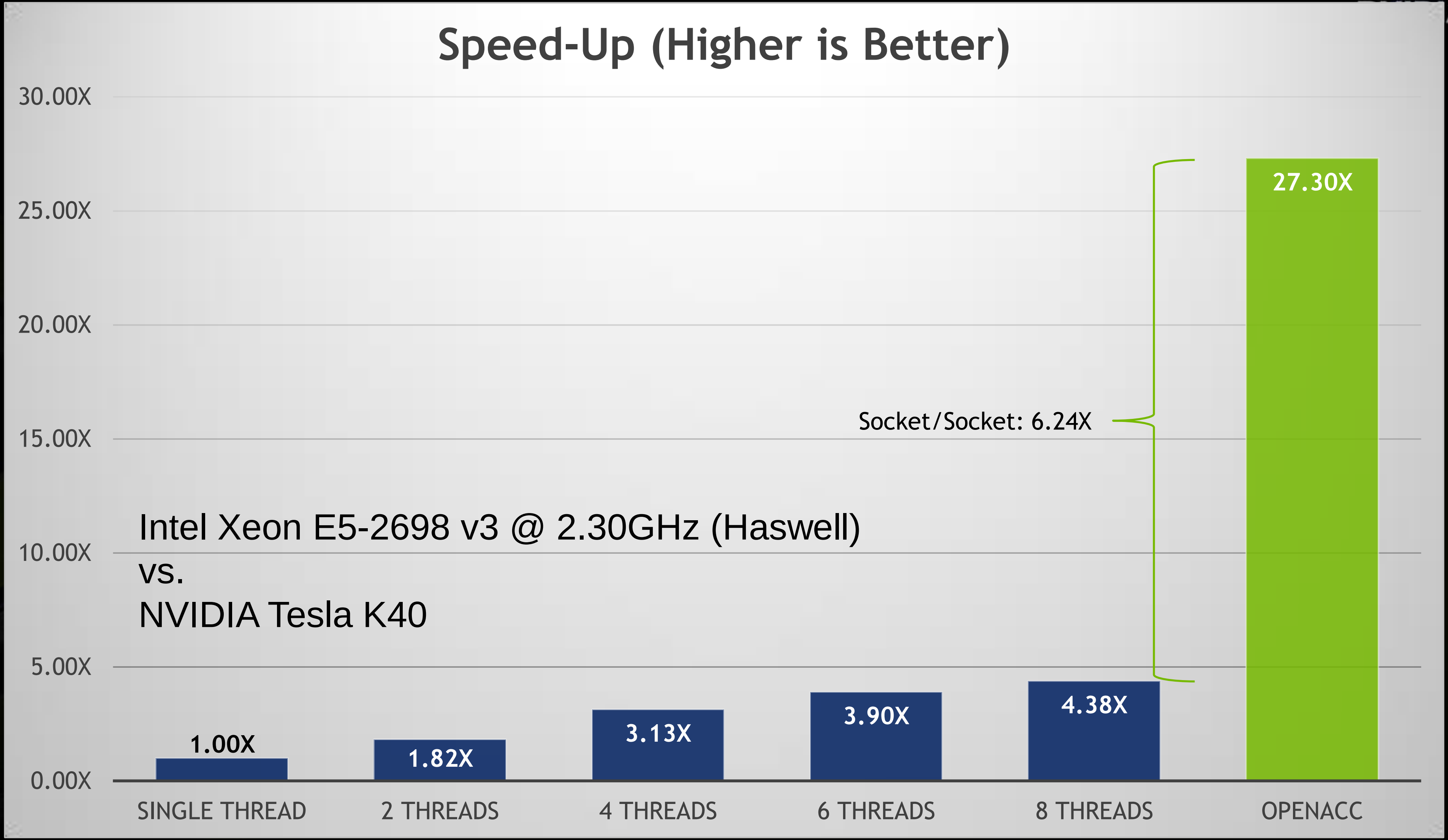
Create Anew as a device temporary.

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
27.30X
Socket/Socket: 6.24X
Intel Xeon E5-2698 v3 @ 2.30GHz (Haswell)
vs.
NVIDIA Tesla K40
1.00X
1.82X
3.13X
3.90X
4.38X
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;
    ...
  }
}
```

UNSTRUCTURED DATA DIRECTIVES

Used to define data regions when scoping doesn't allow the use of normal data regions (e.g. The constructor/destructor of a class).

`enter data` Defines the start of an unstructured data lifetime

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

`exit data` Defines the end of an unstructured data lifetime

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

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

UNSTRUCTURED DATA REGIONS: C++ CLASSES

```
class Matrix {
  Matrix(int n) {
    len = n;
    v = new double[len];
    #pragma acc enter data create(v[0:len])
  }
  ~Matrix() {
    #pragma acc exit data delete(v[0:len])
    delete[] v;
  }

  private:
    double* v;
    int len;
};
```

- Unstructured Data Regions enable OpenACC to be used in C++ classes
- Unstructured data regions can be used whenever data is allocated and initialized in a different scope than where it is freed.

Identify Available Parallelism

Parallelize Loops with OpenACC

Optimize Data Locality

Optimize Loop Performance

Come to S5195 - Advanced OpenACC Programming on Friday to see this step and more.

COFFEE(?)