# INTRODUCTION TO OPENACC

Profiling, Parallelizing, and Optimizing with OpenACC

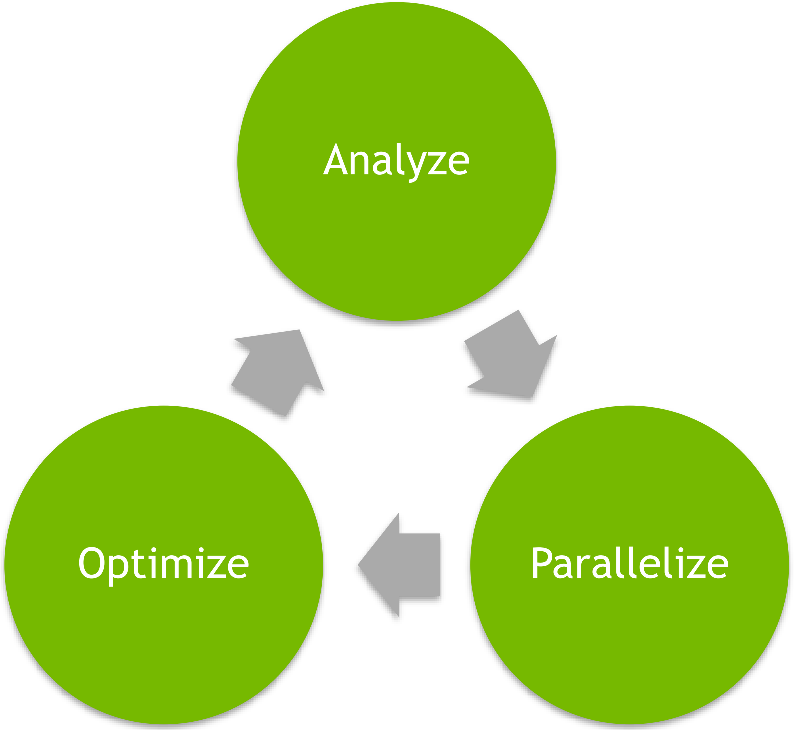
# Objectives

Understand what OpenACC is and why to use it

Learn how to obtain an application profile using PGProf

Learn how to add OpenACC directives to existing loops and build with OpenACC using PGI

Perform simple data and loop optimizations to improve performance



# Why OpenACC?

# OpenACC

Simple | Powerful | Portable

## Fueling the Next Wave of Scientific Discoveries in HPC

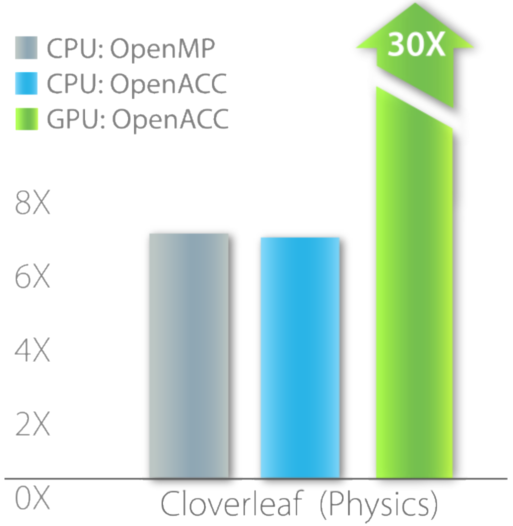
```
main()
{
  <serial code>
  #pragma acc kernels
  //automatically runs on GPU
  {
    <parallel code>
  }
}
```

University of Illinois
PowerGrid- MRI Reconstruction

70x Speed-Up
2 Days of Effort

RIKEN Japan
NICAM- Climate Modeling

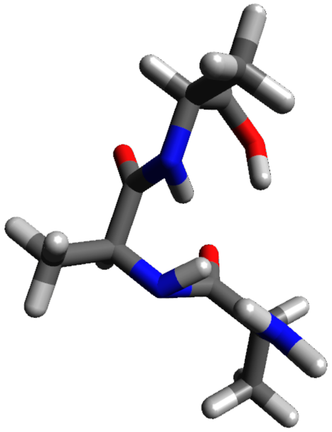
7-8x Speed-Up
5% of Code Modified




http://www.cray.com/sites/default/files/resources/OpenACC_213462.12_OpenACC_Cosmo_CS_FNL.pdf
http://www.hpcwire.com/off-the-wire/first-round-of-2015-hackathons-gets-underway
http://on-demand.gputechconf.com/gtc/2015/presentation/S5297-Hisashi-Yashiro.pdf
http://www.openacc.org/content/experiences-porting-molecular-dynamics-code-gpus-cray-xk7

# LS-DALTON

Large-scale application for calculating high-accuracy molecular energies



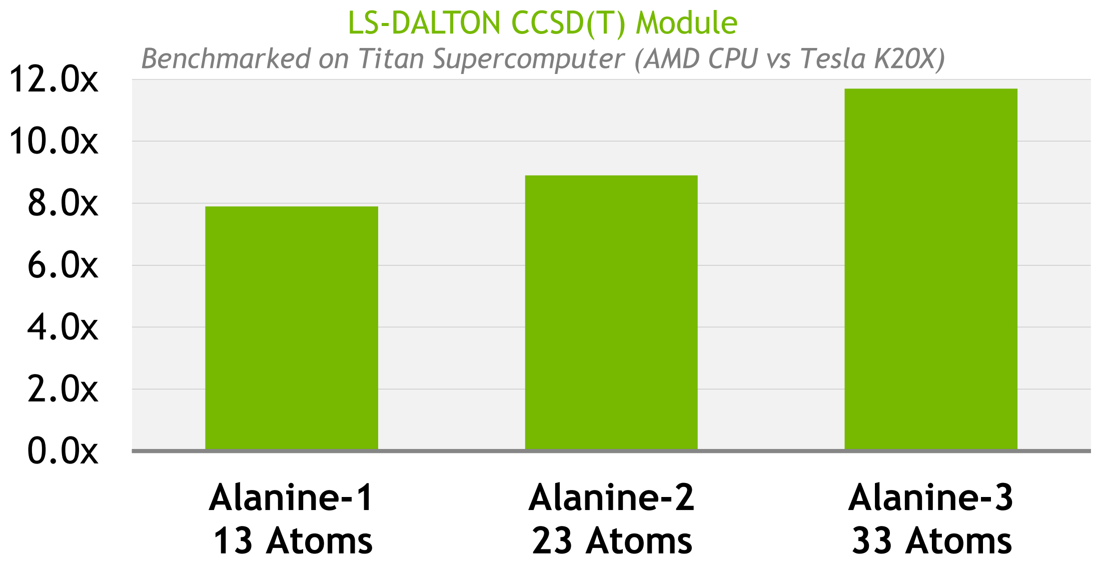
*“OpenACC makes GPU computing approachable for domain scientists. Initial OpenACC implementation required only minor effort, and more importantly, no modifications of our existing CPU implementation.”*

*Janus Juul Eriksen, PhD Fellow*
*qLEAP Center for Theoretical Chemistry, Aarhus University*

## Minimal Effort

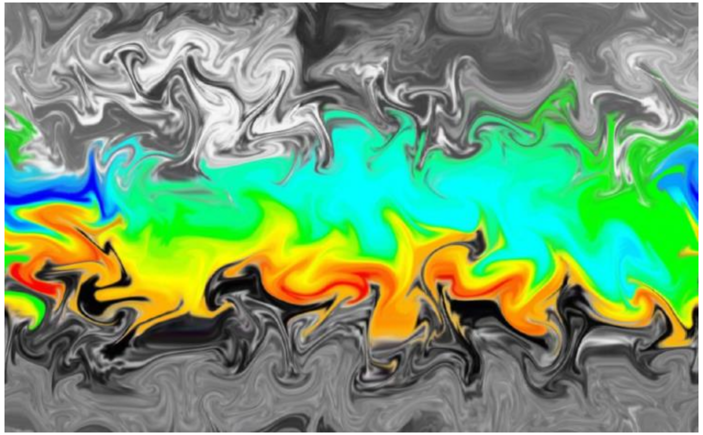
| Lines of Code Modified | # of Weeks Required | # of Codes to Maintain |
|---|---|---|
| **<100 Lines** | **1 Week** | **1 Source** |

## Big Performance



# OpenACC Performance Portability: CloverLeaf

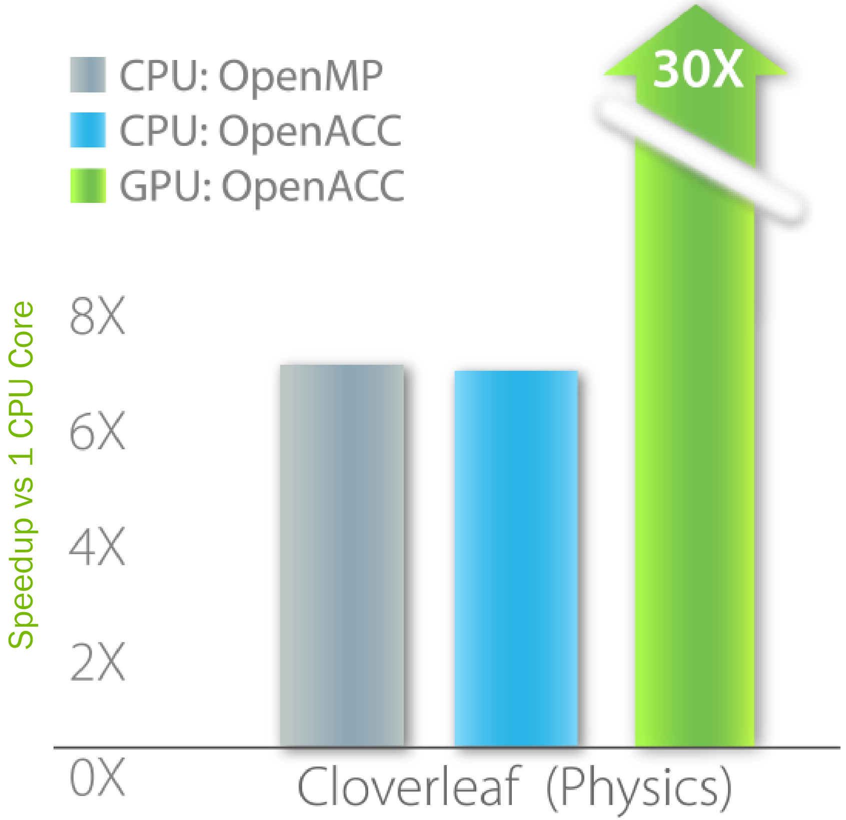
## Hydrodynamics Application



*"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."*

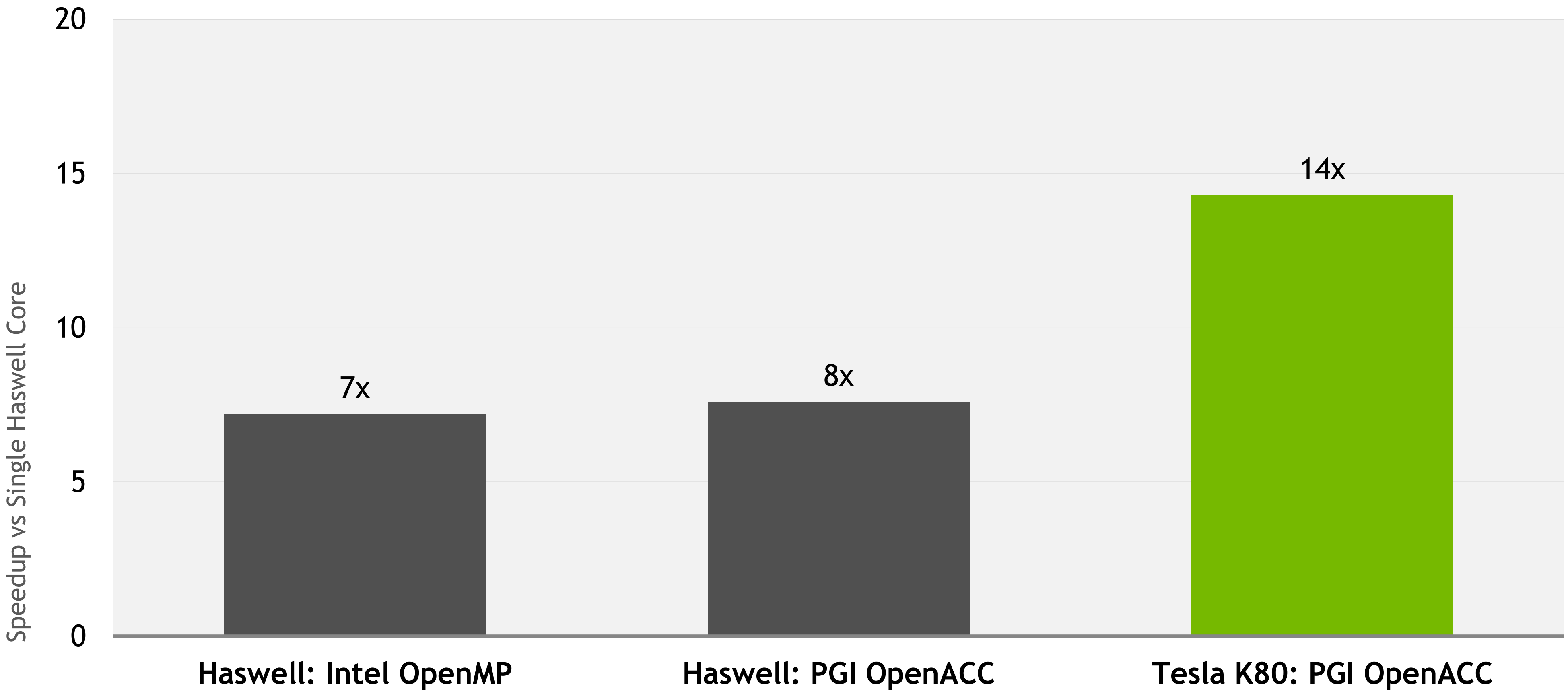
*Wayne Gaudin and Oliver Perks*
*Atomic Weapons Establishment, UK*

## OpenACC Performance Portability



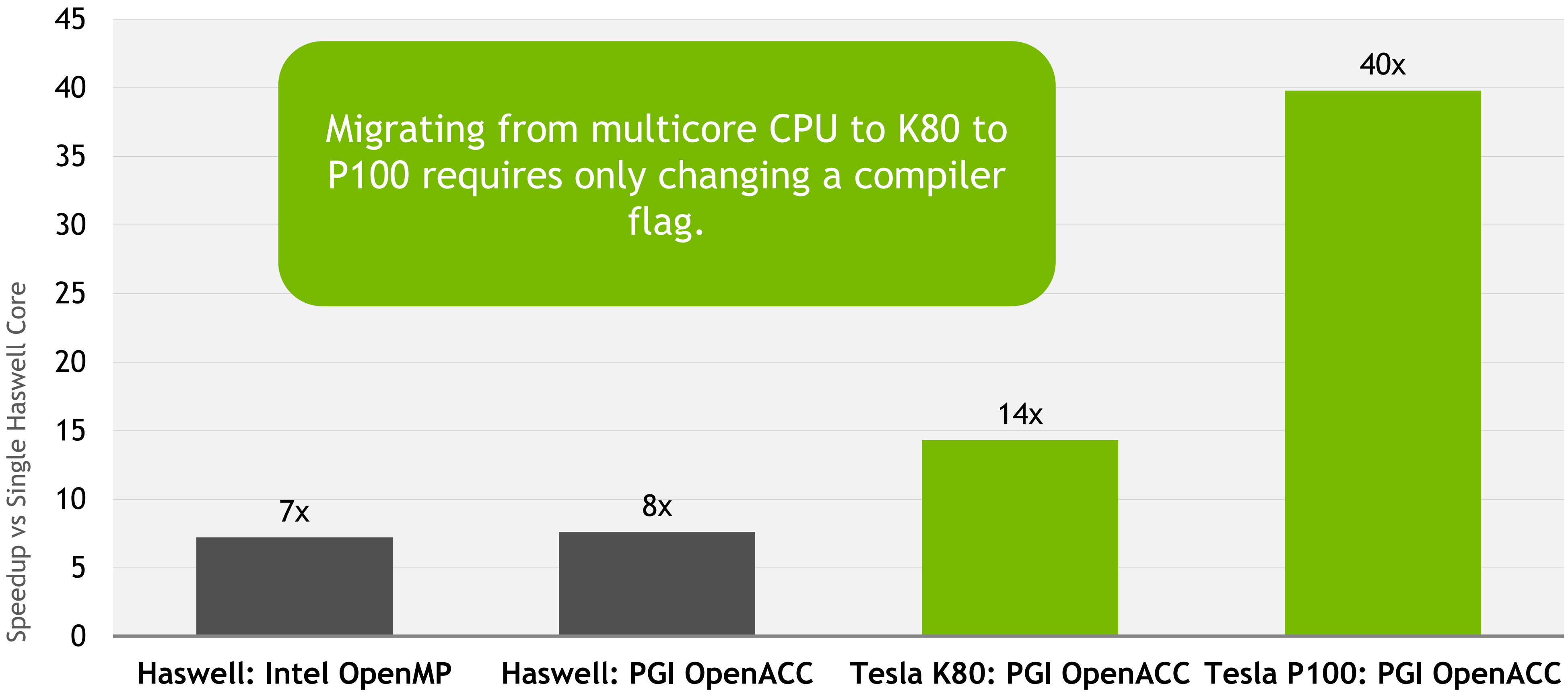

*Benchmarked Intel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz, Accelerator: Tesla K80*

# CloverLeaf on Dual Haswell vs Tesla K80



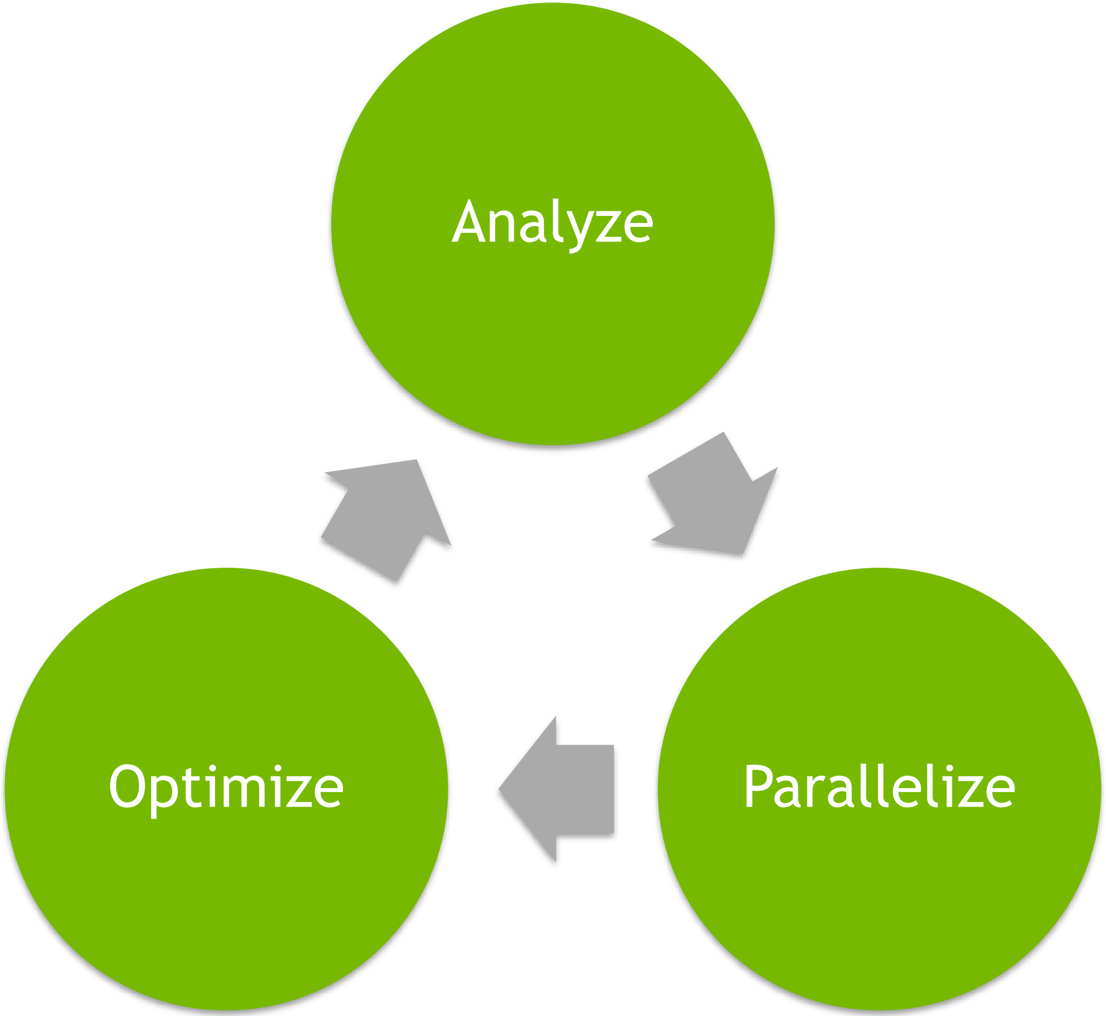

CPU: Intel Xeon E5-2698 v3, 2 sockets, 32 cores, 2.30 GHz, HT disabled
GPU: NVIDIA Tesla K80 (single GPU)
OS: CentOS 6.6, Compiler: PGI 16.5

# CloverLeaf on 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

# 3 Steps to Accelerate with OpenACC

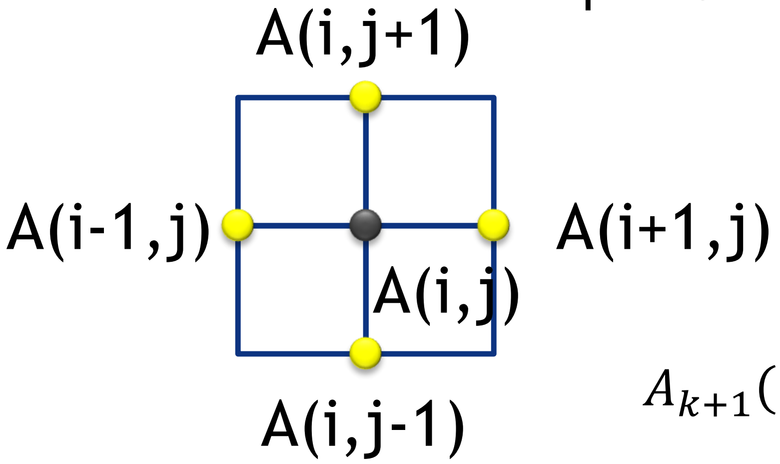


# Example: Jacobi Iteration

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




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

# 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

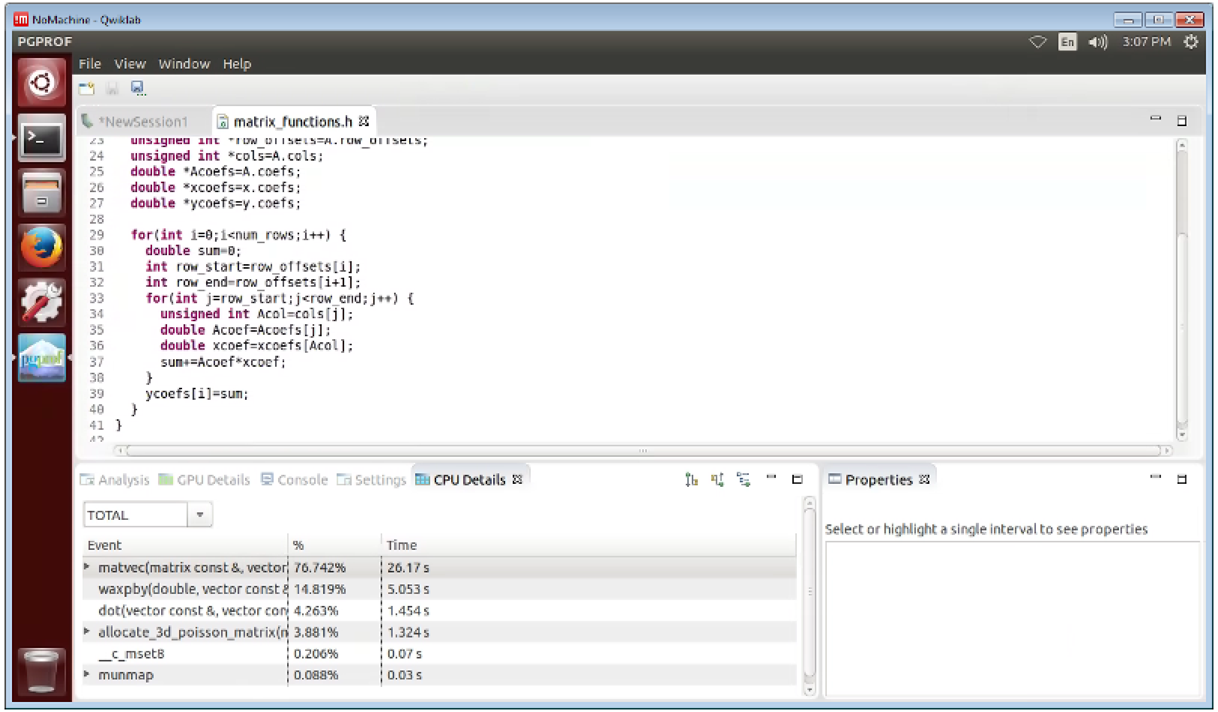
Compute max error for convergence

Swap input/output arrays

# Analyze

# Analyze

- Obtain a performance profile
- Read compiler feedback
- Understand the code



# Obtain a Profile

A application profile helps to understand where time is spent

What routines are *hotspots*?

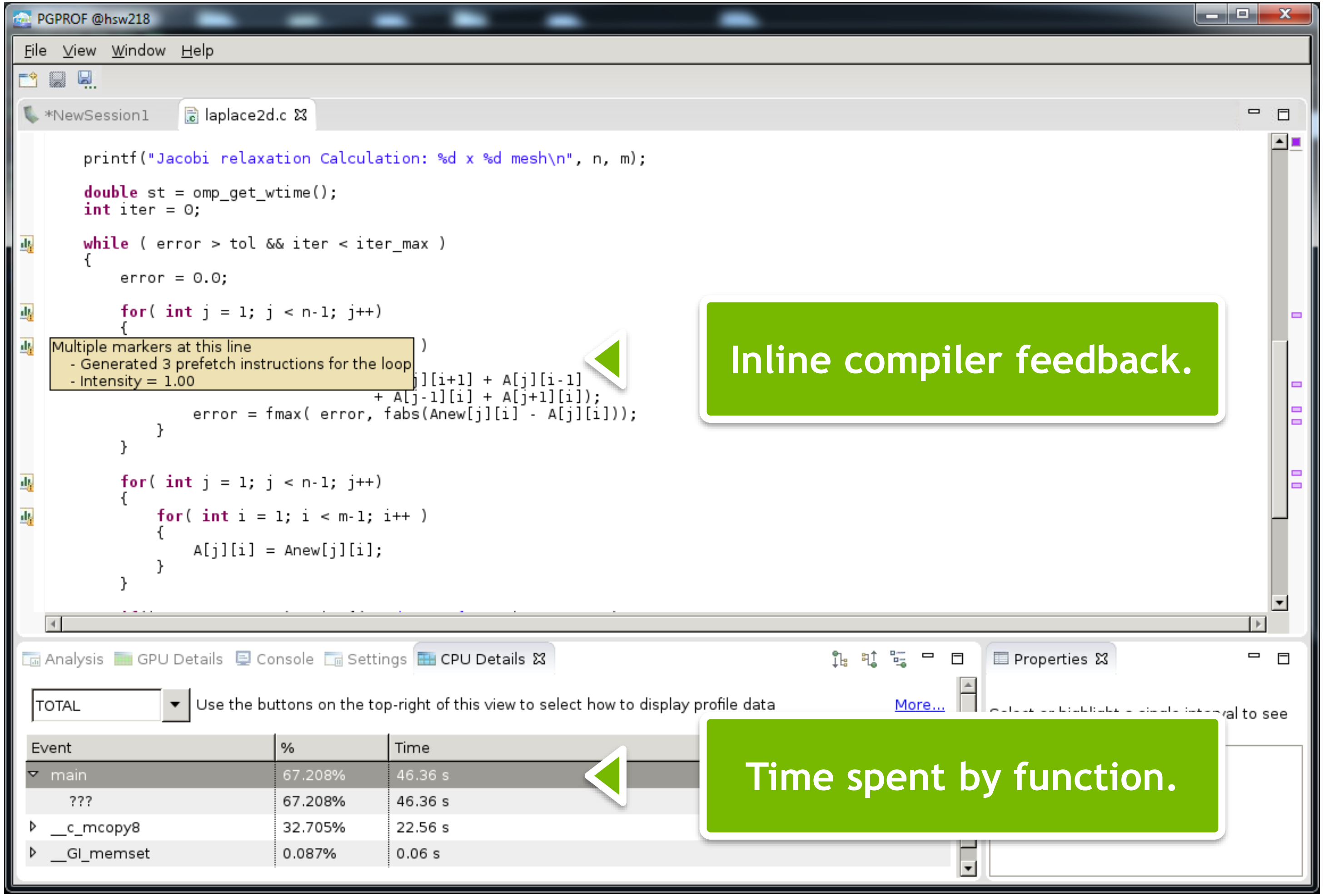
Focusing on the hotspots delivers the greatest performance impact

A variety of profiling tools are available: gprof, nvprof, CrayPAT, TAU, Vampir

We'll use PGProf, which comes with the PGI compiler

```
$ pgprof &
```

# PGPROF Profiler



# Compiler Feedback

- Before we can make changes to the code, we need to understand how the compiler is optimizing
- With PGI, this can be done with the -Minfo and -Mneginfo flags

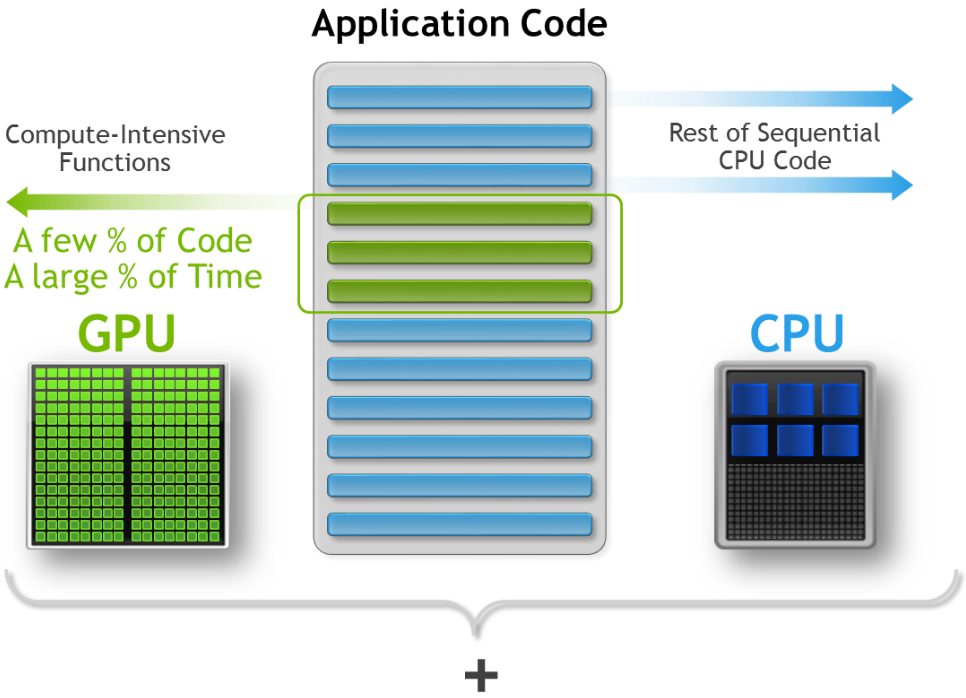
```
$ pgc++ -Minfo=all,ccff -Mneginfo
```

```
main:
     40, Loop not fused: function call before
adjacent loop
         Loop not vectorized: may not be
beneficial
         Unrolled inner loop 4 times
         Generated 3 prefetches in scalar loop
     57, Generated vector simd code for the loop
containing reductions
         Generated 3 prefetch instructions for
the loop
     67, Memory copy idiom, loop replaced by
call to __c_mcopy8
```

# Parallelize

# Parallelize

- Insert OpenACC directives around important loops
- Enable OpenACC in the compiler
- Run on a parallel platform



# OpenACC Directives

Manage Data Movement

Initiate Parallel Execution

Optimize Loop Mappings

```
#pragma acc data copyin(a,b) copyout(c)
{
  ...
  #pragma acc parallel
  {
  #pragma acc loop gang vector
      for (i = 0; i < n; ++i) {
          z[i] = x[i] + y[i];
          ...
      }
  }
  ...
}
```

OpenACC
Directives for Accelerators

- Incremental
- Single source
- Interoperable
- Performance portable
- CPU, GPU, MIC

# OpenACC Parallel Directive

## Generates parallelism

```
#pragma acc parallel
{
```

When encountering the *parallel* directive, the compiler will generate *1 or more parallel gangs*, which execute redundantly.

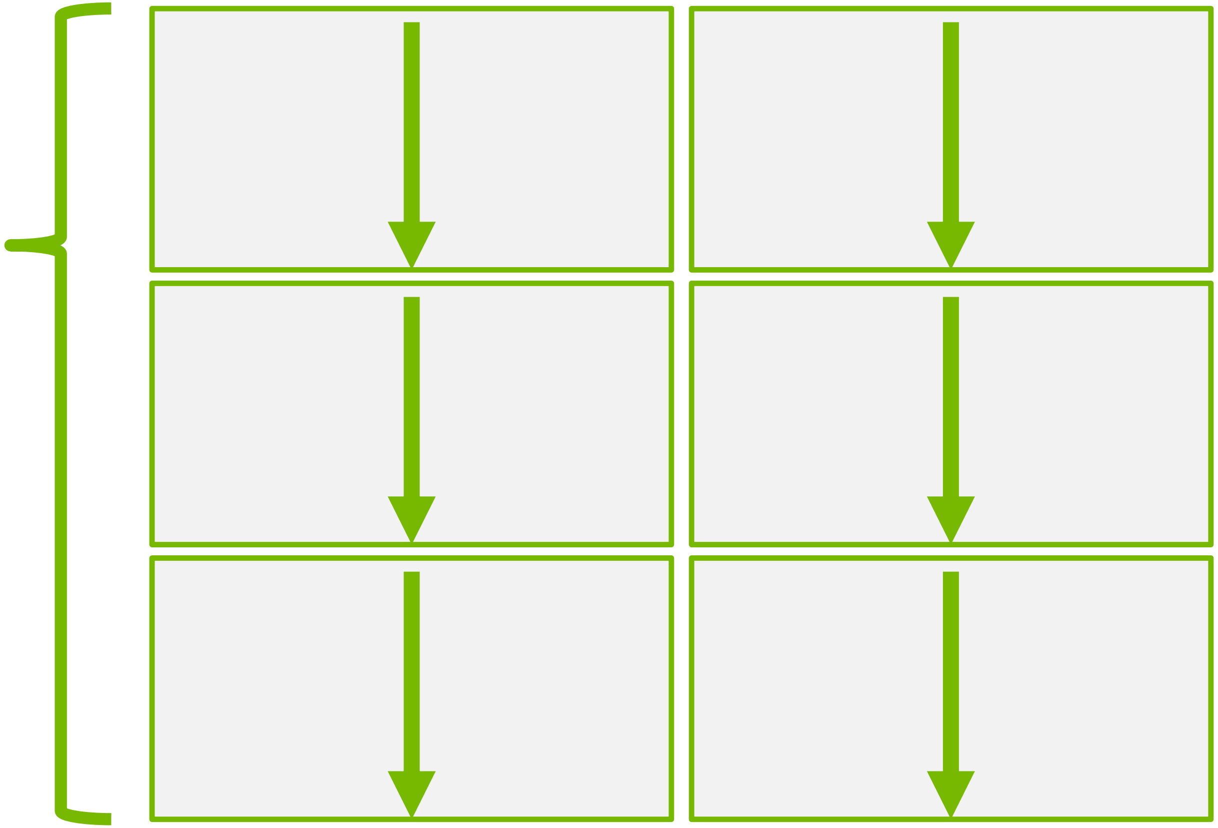
```
}
```

# OpenACC Parallel Directive

## Generates parallelism

```
#pragma acc parallel
{

}
```

When encountering the *parallel* directive, the compiler will generate *1 or more parallel gangs*, which execute redundantly.

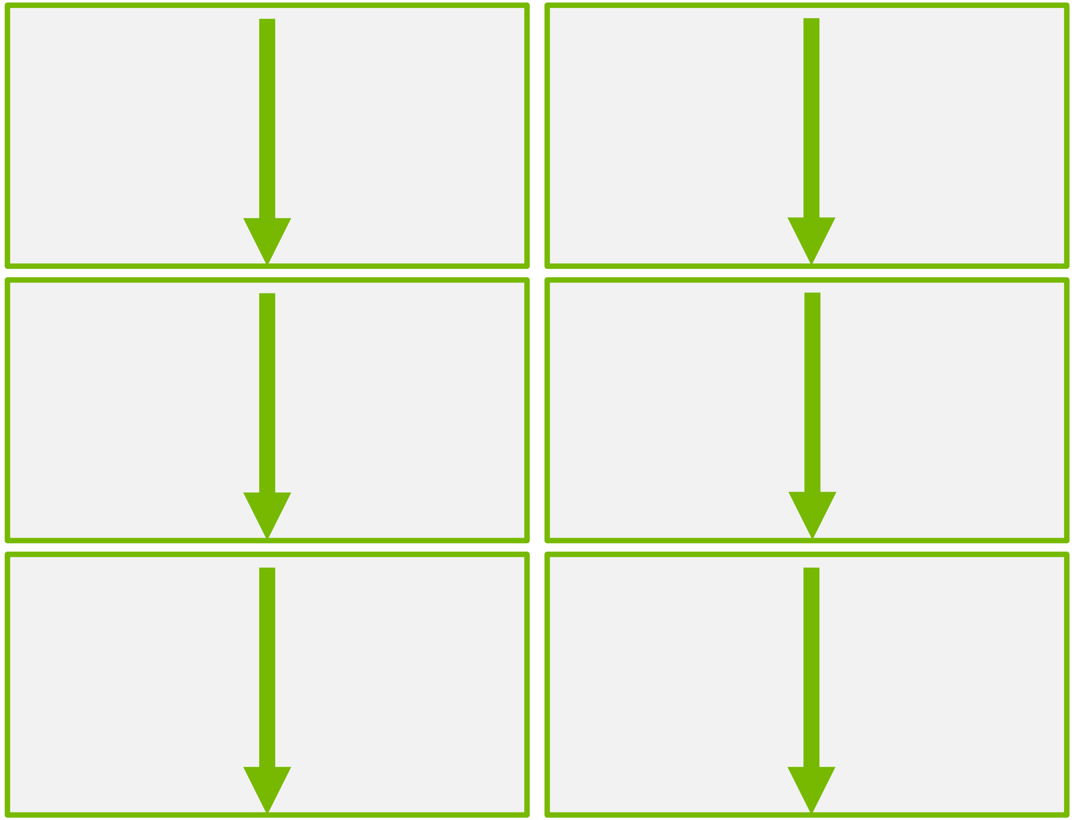


# OpenACC Loop Directive

## Identifies loops to run in parallel

```
#pragma acc parallel
{
  #pragma acc loop
  for (i=0;i<N;i++)
  {
  }
}
```

The *loop* directive informs the compiler which loops to parallelize.

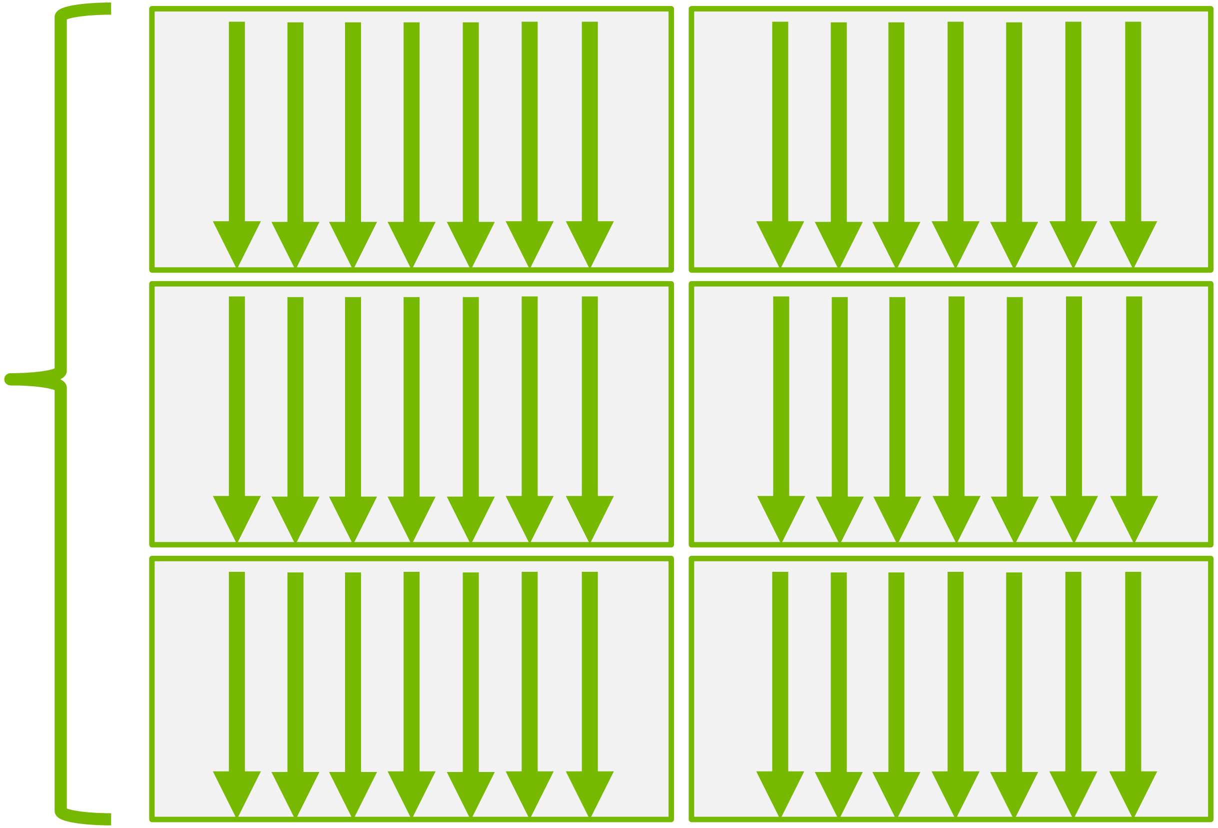


# OpenACC Loop Directive

## Identifies loops to run in parallel

```
#pragma acc parallel
{
  #pragma acc loop
  for (i=0;i<N;i++)
  {
  }
}
```

The *loop* directive informs the compiler which loops to parallelize.

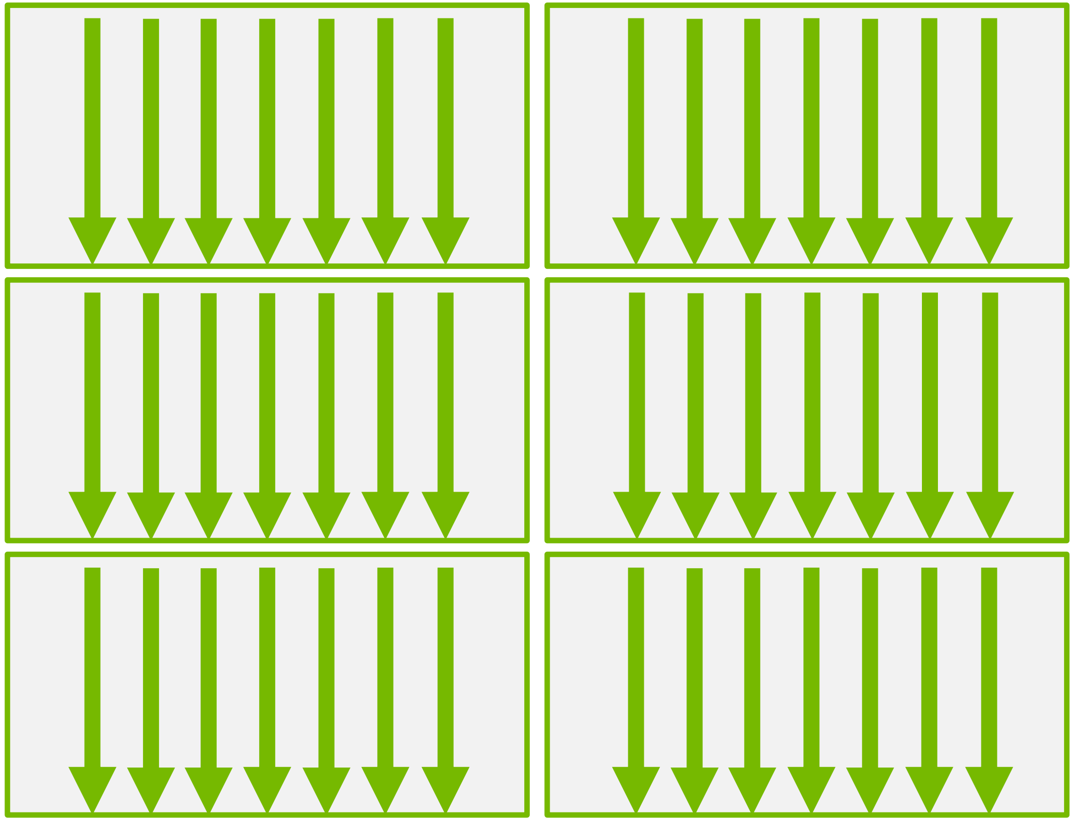


# OpenACC Parallel Loop Directive

## Generates parallelism and identifies loop in one directive

```
#pragma acc parallel loop
  for (i=0;i<N;i++)
  {
  }
```

The *parallel* and *loop* directives are frequently combined into one.



# Parallelize with OpenACC Parallel Loop

```
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 first loop nest, max *reduction* required.

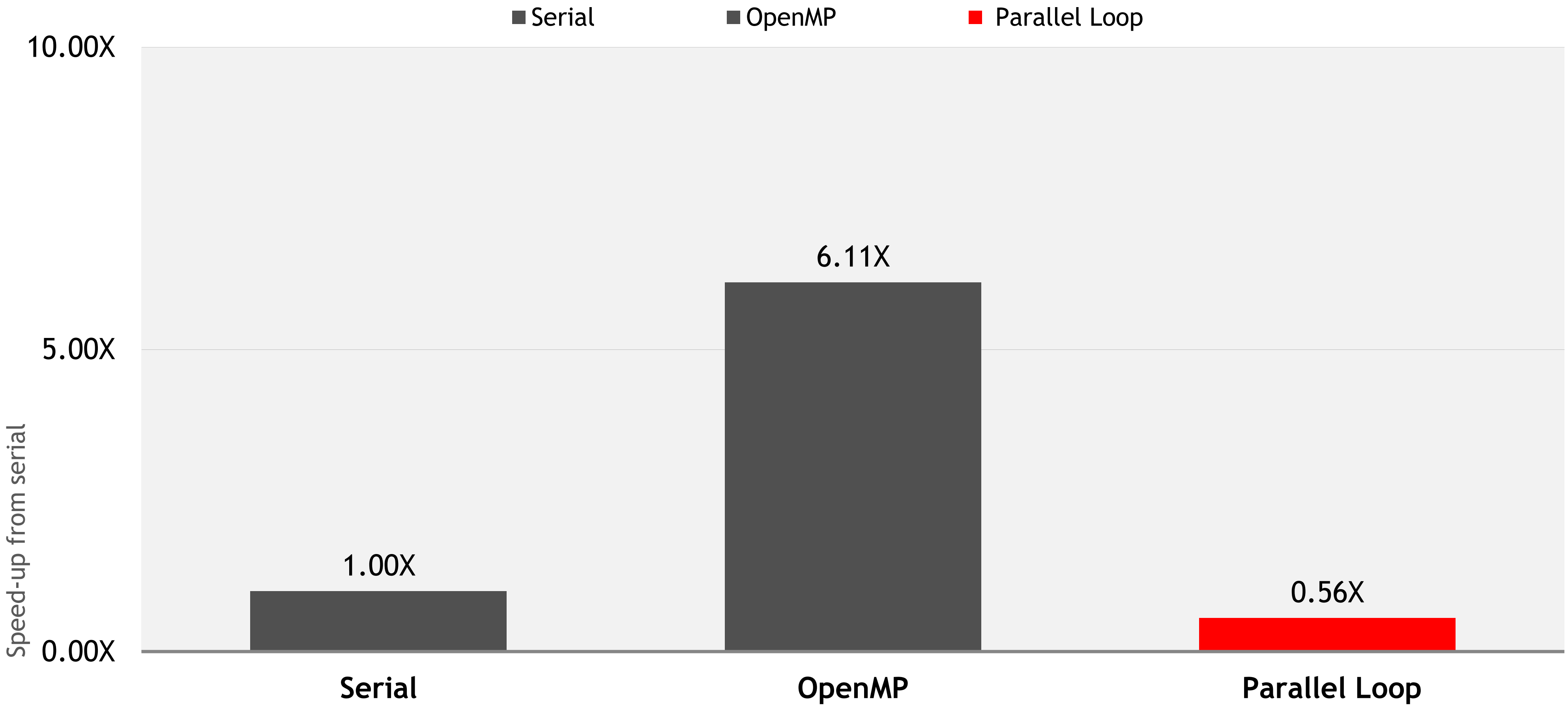
Parallelize second loop.

We didn't detail *how* to parallelize the loops, just *which* loops to parallelize.

# Building the code

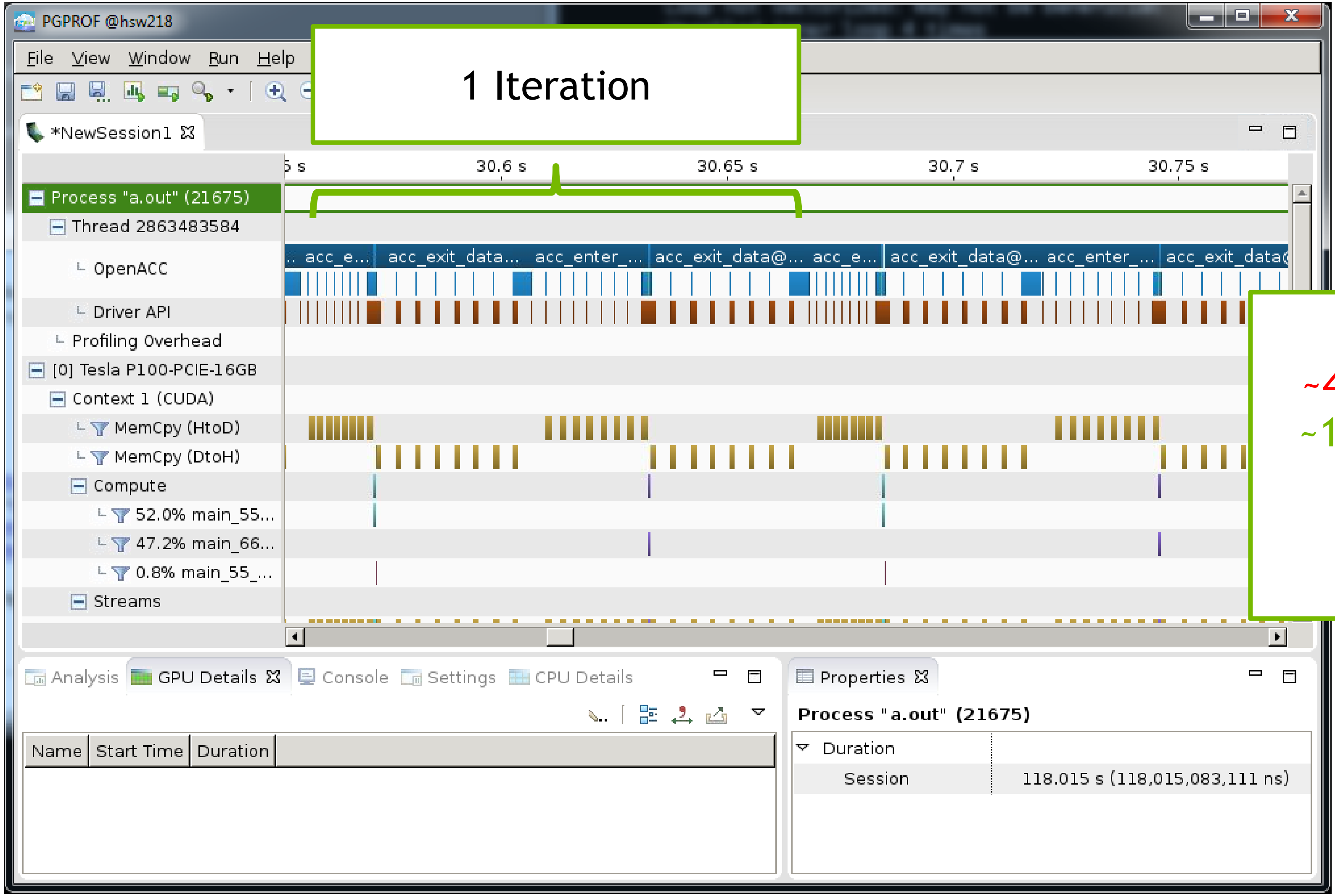
```
$ pgcc -fast -ta=tesla,cc60 -Minfo=all,ccff laplace2d.c
main:
     40, Loop not fused: function call before adjacent loop
         Loop not vectorized: may not be beneficial
         Unrolled inner loop 4 times
         Generated 3 prefetches in scalar loop
     51, Loop not vectorized/parallelized: potential early exits
     55, Accelerator kernel generated
         Generating Tesla code
         55, Generating reduction(max:error)
         56, #pragma acc loop gang /* blockIdx.x */
         58, #pragma acc loop vector(128) /* threadIdx.x */
     55, Generating implicit copyout(Anew[1:4094][1:4094])
         Generating implicit copyin(A[:][:])
     58, Loop is parallelizable
     66, Accelerator kernel generated
         Generating Tesla code
         67, #pragma acc loop gang /* blockIdx.x */
         69, #pragma acc loop vector(128) /* threadIdx.x */
     66, Generating implicit copyin(Anew[1:4094][1:4094])
         Generating implicit copyout(A[1:4094][1:4094])
     69, Loop is parallelizable
```

# OpenACC Performance (Step 1)




*Source: PGI 17.5, CPU: Intel Xeon CPU E5-2698 v3 @ 2.30GHz, GPU: NVIDIA Tesla P100*

# What went wrong?




1 Iteration

Per-iteration:
~450ms Copying Data
~1.1ms Computing on GPU

*Why?*

# Step 1 Compiler Feedback

```
51 while ( error > tol && iter < iter_max )
52 {
53   error = 0.0;
54
55 #pragma acc parallel loop
        reduction(max:error)
56   for( int j=1; j < n-1; j++)
57   {
58     for(int i=1; i < m-1; i++)
59     {
60-62      …
63     }
64   }
65-76 …
77   iter++;
78 }
```

```
main:
51, Loop not vectorized/parallelized:
potential early exits
     55, Accelerator kernel generated
         Generating Tesla code
         55, Generating reduction(max:error)
         56, #pragma acc loop gang /*
blockIdx.x */
         58, #pragma acc loop vector(128) /*
threadIdx.x */
     55, Generating implicit copyin(A[:][:])
         Generating implicit
copyout(Anew[1:4094][1:4094])
```

The compiler implicitly copies A and Anew to/from the GPU in case we need them, but do we?

# Optimizing Data Movement

```
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?

# Optimize

# Optimize

- Get new performance data from parallel execution
- Remove unnecessary data transfer to/from GPU
- Guide the compiler to better loop decomposition

# Structured Data Regions

The **`data`** directive 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

| | |
|---|---|
| `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. |
| `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. (Structured Only) |
| `create ( list )` | Allocates memory on GPU but does not copy. |
| `delete( list )` | Deallocate memory on the GPU without copying. (Unstructured Only) |
| `present ( list )` | Data is already present on GPU from another containing data region. |

(!) All of these will check if the data is already present first and reuse if found.

# Optimized Data Movement

```
#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.

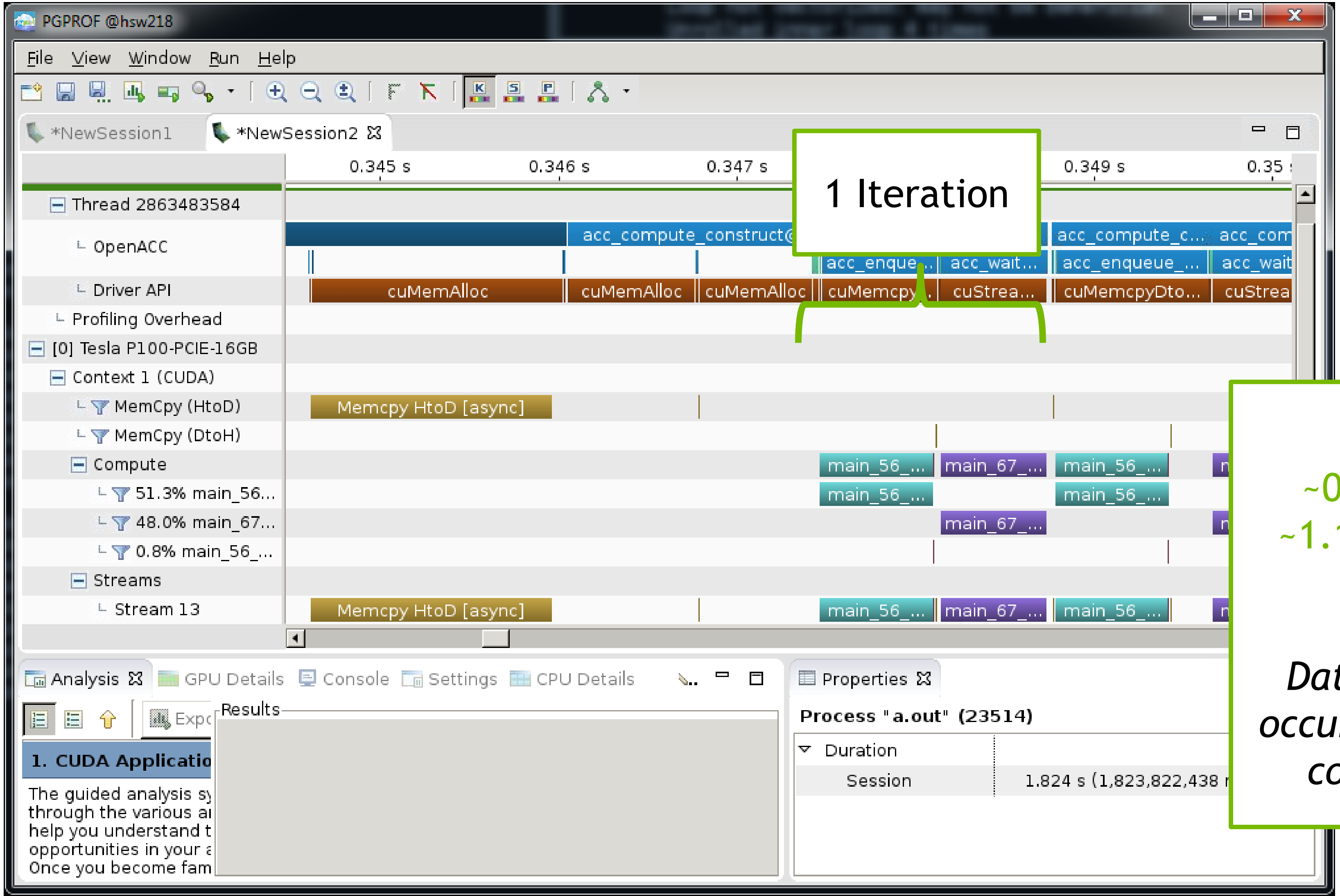
Create Anew as a device temporary.

# Rebuild the Code

```
main:
     51, Generating create(Anew[:][:])
         Generating copy(A[:][:])
     56, Accelerator kernel generated
         Generating Tesla code
         56, Generating reduction(max:error)
         57, #pragma acc loop gang /* blockIdx.x */
         59, #pragma acc loop vector(128) /* threadIdx.x */
     59, Loop is parallelizable
     67, Accelerator kernel generated
         Generating Tesla code
         68, #pragma acc loop gang /* blockIdx.x */
         70, #pragma acc loop vector(128) /* threadIdx.x */
     70, Loop is parallelizable
```

Now data movement only happens at our data region.

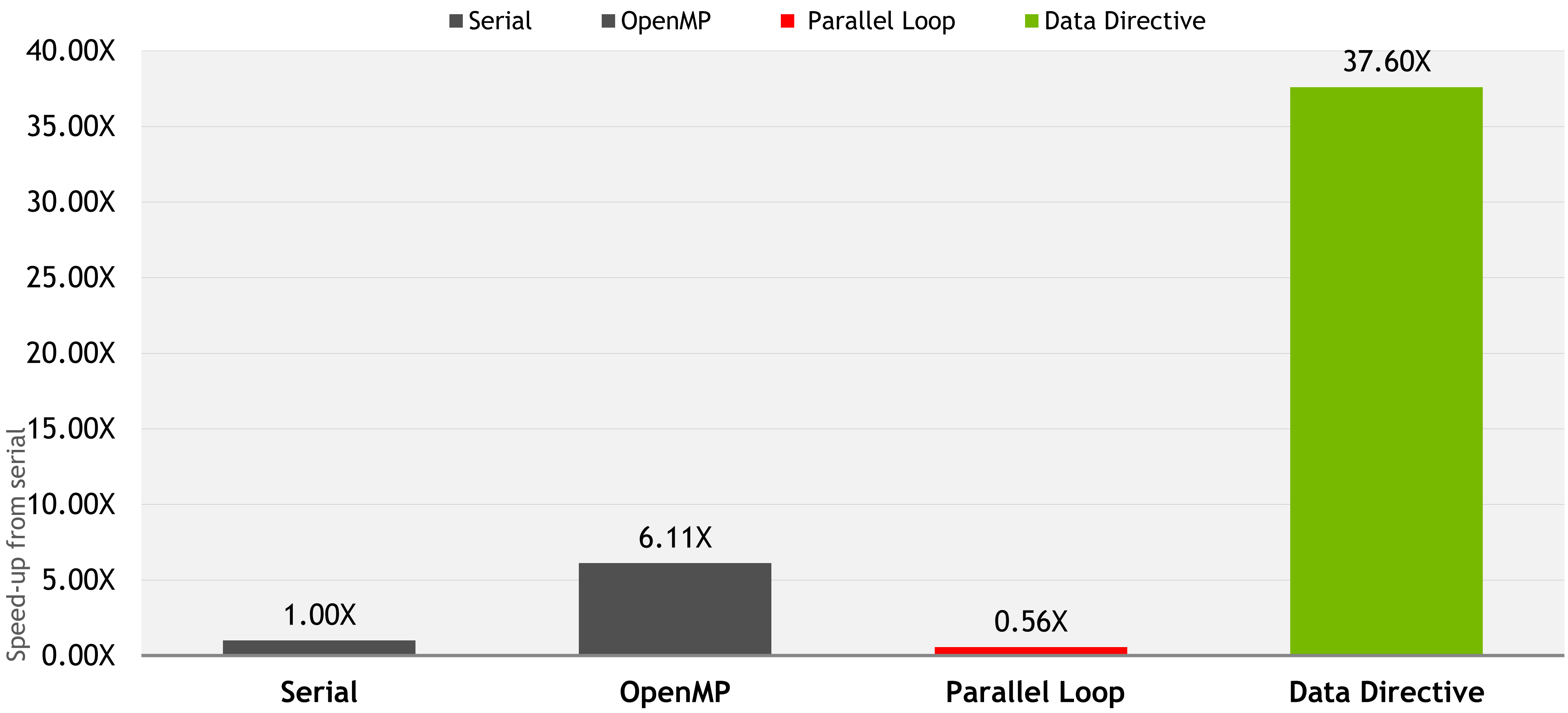
# PGProf Timeline Now




Per-iteration:
~0ms Copying Data
~1.1ms Computing on GPU

*Data movement now occurs before and after convergence loop.*

# OpenACC Performance (Step 2)




*Source: PGI 17.5, CPU: Intel Xeon CPU E5-2698 v3 @ 2.30GHz, GPU: NVIDIA Tesla P100*

# Array Shaping

Compiler sometimes cannot determine size of arrays

- Must specify explicitly using data clauses and array “shape”
- Partial arrays must be contiguous

C/C++

```
#pragma acc data copyin(a[0:N]) copyout(b[start:count])
```

Fortran

```
!$acc data copyin(a(1:N)) copyout(b(start:start+count))
```

# Unstructured Data: C++ Classes

- 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 (e.g. Fortran modules).

```
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;
};
```

# OpenACC Update Directive

Programmer specifies an array (or part of an array) that should be refreshed within a data region.

```
#pragma acc data create(a)
{
do_something_on_device()

#pragma acc update self(a)

do_something_on_host()

#pragma acc update device(a)

}
```

Copy "a" from GPU to CPU

Copy "a" from CPU to GPU

# Optimize Loops

Now let's look at how our iterations get mapped to hardware.

Compilers give their best guess about how to transform loops into parallel kernels, but sometimes they need more information.

This information could be our knowledge of the code or based on profiling.

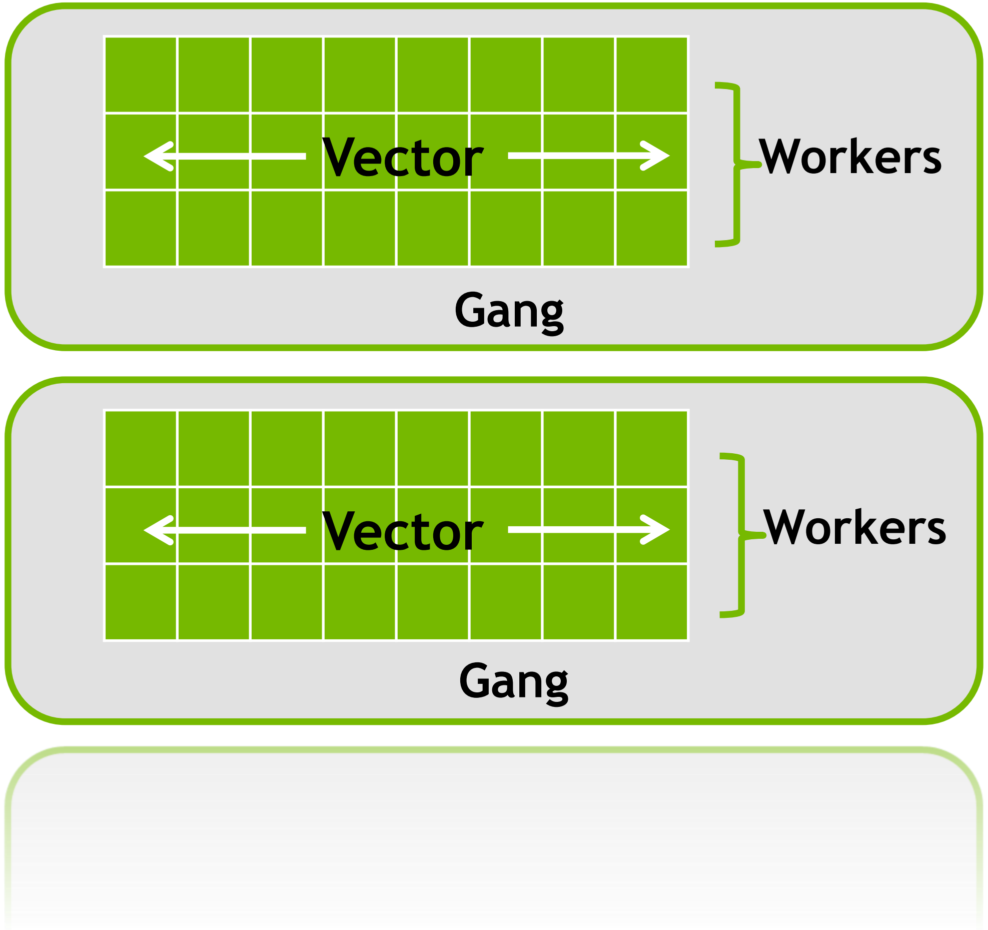
# Step 2 Compiler Feedback

```
51 #pragma acc data copy(A) create(Anew)
52 while ( error > tol && iter < iter_max )
53 {
54   error = 0.0;
55
56 #pragma acc parallel loop
            reduction(max:error)
57   for( int j=1; j < n-1; j++)
58   {
59     for(int i=1; i < m-1; i++)
60     {
61-63      …
64     }
65   }
66-77 …
78   ite
79 }
```

```
main:
     51, Generating create(Anew[:][:])
         Generating copy(A[:][:])
     56, Accelerator kernel generated
         Generating Tesla code
         56, Generating reduction(max:error)
         57, #pragma acc loop gang /*
blockIdx.x */
         59, #pragma acc loop vector(128) /*
threadIdx.x */
     59, Loop is parallelizable
     67, Accelerator kernel generated
         Generating Tesla code
         68, #pragma acc loop gang /*
blockIdx.x */
         70, #pragma acc loop vector(128) /*
threadIdx.x */
     70, Loop is parallelizable
```

The compiler is *vectorizing* the inner loops and breaking the outer loops across *gangs*.

# OpenACC: 3 Levels of Parallelism




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

# The loop Directive

The `loop` directive gives the compiler additional information about the *next* loop in the source code through several clauses.

- `independent` – all iterations of the loop are independent
- `auto` - instructs the compiler to analyze the loop
- `collapse(N)` – turn the next N loops into one, flattened loop
- `tile(N[,M,…])` - break the next 1 or more loops into *tiles* based on the provided dimensions.
- `gang`, `worker`, `vector`, `seq` – Describes how to parallelize the loop.

# 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

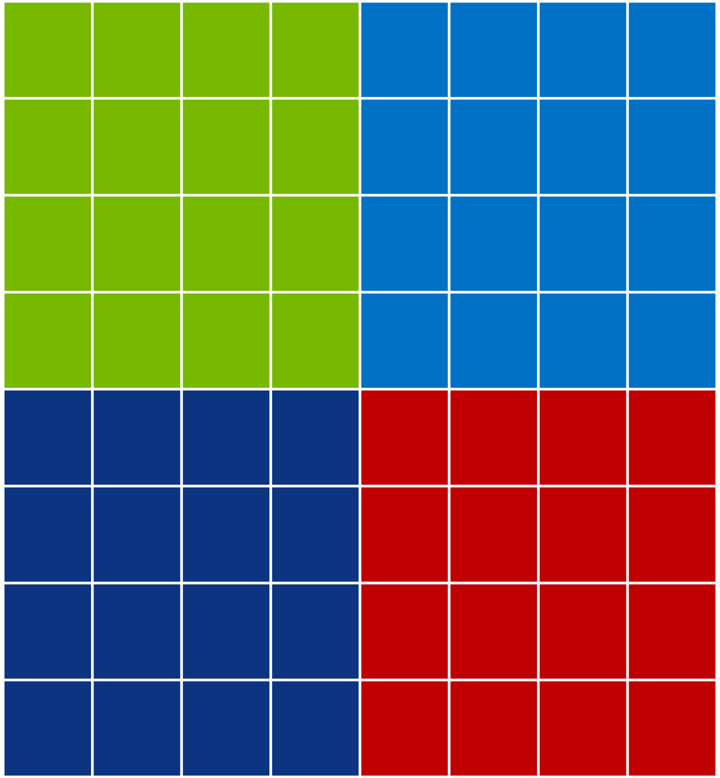
num_gangs(n), num_workers(n), vector_length(n)

```
#pragma acc parallel loop gang
for (int i = 0; i < n; ++i)
  #pragma acc loop vector
  for (int j = 0; j < n; ++j)
   ...
```

```
#pragma acc parallel vector_length(32)
#pragma acc loop gang worker
for (int i = 0; i < n; ++i)
  #pragma acc loop vector
  for (int j = 0; j < n; ++j)
    ...
```

# The tile Clause

Operate on smaller blocks of the operation to exploit data locality



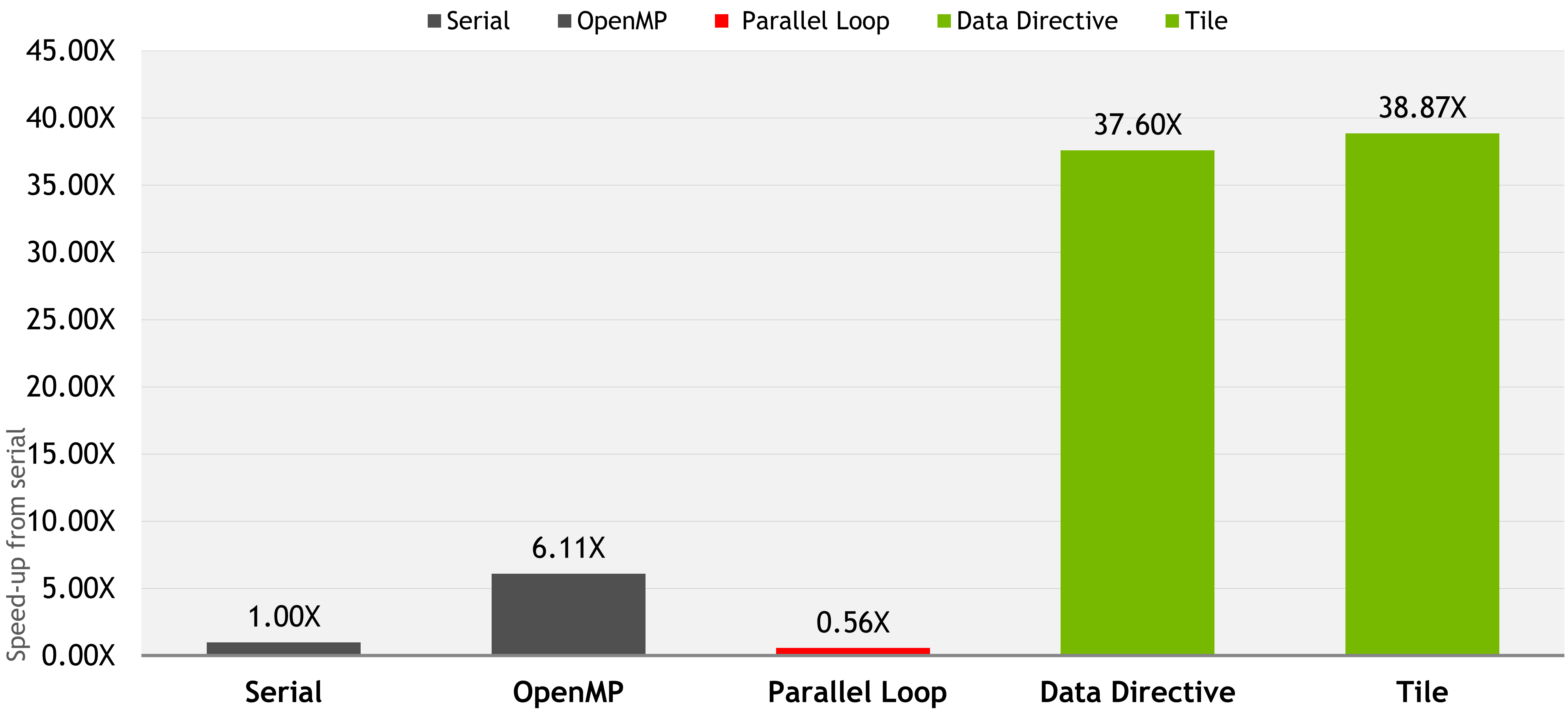
```
#pragma acc loop tile(4,4)
  for(i = 1; i <= ROWS; i++) {
    for(j = 1; j <= COLUMNS; j++) {
      Temp[i][j] = 0.25 *
       (Temp_last[i+1][j] +
        Temp_last[i-1][j] +
        Temp_last[i][j+1] +
        Temp_last[i][j-1]);
    }
  }
```

# Tile to Exploit Locality

```
#pragma acc data copy(A) create(Anew)
while ( err > tol && iter < iter_max ) {
  err=0.0;
#pragma acc parallel loop reduction(max:err) tile(32,16)
  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 tile(32,16)
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

Through experimentation I found that 32x16 tiles gave me the best performance.

# OpenACC Performance (Final)




*Source: PGI 17.5, CPU: Intel Xeon CPU E5-2698 v3 @ 2.30GHz, GPU: NVIDIA Tesla P100*

# OpenACC kernels Directive

## Identifies a region of code where I think the compiler can turn *loops* into *kernels*

```
#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.

# OpenACC parallel loop vs. kernels

**PARALLEL LOOP**

- Programmer's responsibility to ensure safe parallelism
- Will parallelize what a compiler may miss
- Straightforward path from OpenMP

**KERNELS**

- Compiler's responsibility to analyze the code and parallelize what is safe.
- Can cover larger area of code with single directive
- Gives compiler additional leeway to optimize.
- Compiler sometimes gets it wrong.

Both approaches are equally valid and can perform equally well.

# In Closing

# Where to find help

- OpenACC Course Recordings - https://developer.nvidia.com/openacc-courses
- PGI Website - http://www.pgroup.com/resources
- OpenACC on StackOverflow - http://stackoverflow.com/questions/tagged/openacc
- PGI Community Edition - http://www.pgroup.com/products/community.htm
- Parallel Forall Blog - http://devblogs.nvidia.com/parallelforall/
- GPU Technology Conference - http://www.gputechconf.com/
- OpenACC Website - http://openacc.org/

Questions? Email openacc@nvidia.com

# QWIKLABS – Hands on in the cloud

# QWIKLAB

http://nvlabs.qwiklab.com

Navigate and login to qwiklab

Start lab:

OpenACC - 2x in 4 Steps

Complete lab