OPENACC ONLINE COURSE

Module 3 – Loop Optimizations with OpenACC

Robert Searles

NVIDIA Corporation

More Science, Less Programming

ABOUT THIS COURSE

3 Part Introduction to OpenACC

- Module 1 – Introduction to OpenACC ✓
- Module 2 – Data Management with OpenACC ✓
- Module 3 – Loop Optimizations with OpenACC

Each module will have a corresponding lab

COURSE OBJECTIVE

Enable ***YOU*** to accelerate ***YOUR*** applications with OpenACC.

MODULE 3 OUTLINE

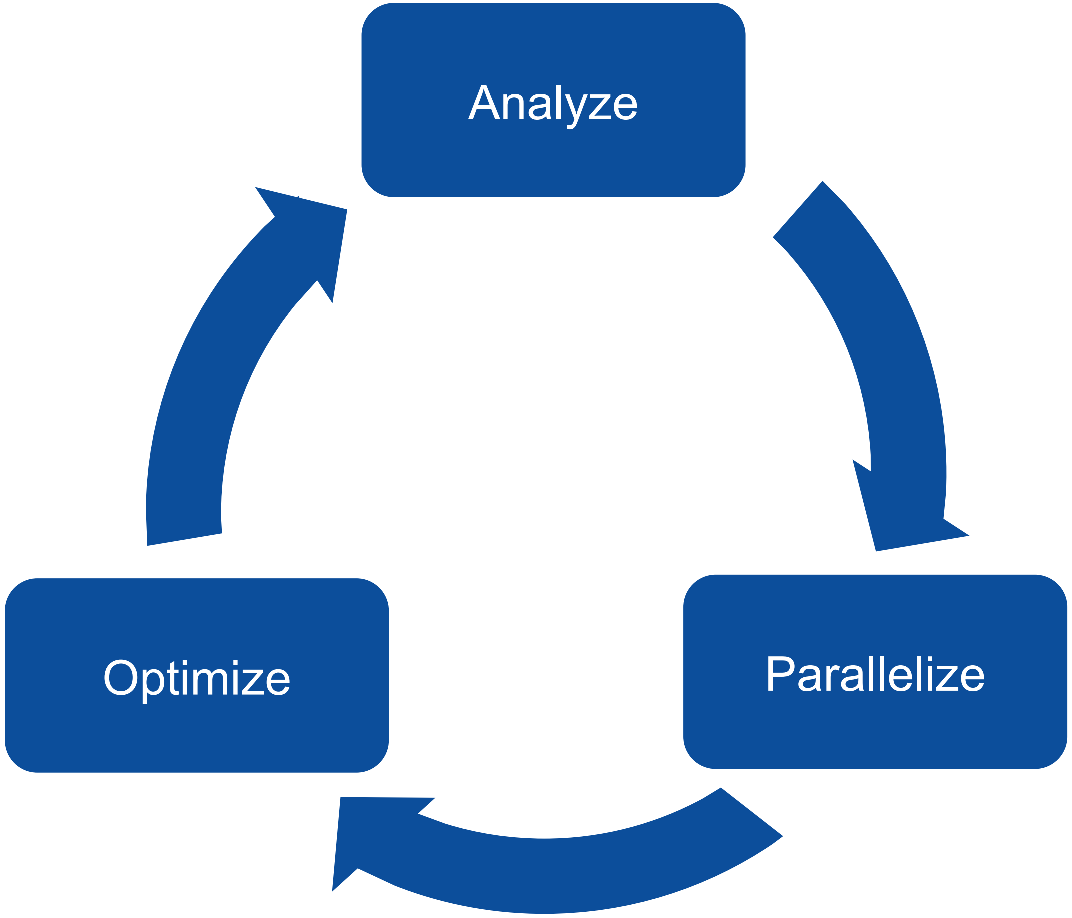
Topics to be covered

- Gangs, Workers, and Vectors Demystified
- GPU Profiles
- Loop Optimizations
- Module 3 Lab
- Where to Get Help

MODULES 1 & 2 REVIEW

OPENACC DEVELOPMENT CYCLE

- **Analyze** your code to determine most likely places needing parallelization or optimization.
- **Parallelize** your code by starting with the most time consuming parts and check for correctness.
- **Optimize** your code to improve observed speed-up from parallelization.



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
      for (i = 0; i < n; ++i) {
  #pragma acc loop vector
        for (j = 0; j < n; ++j) {
          c[i][j] = a[i][j] + b[i][j];
          ...
        }
      }
  }
  ...
}
```

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

OpenACC
Directives for Accelerators

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.

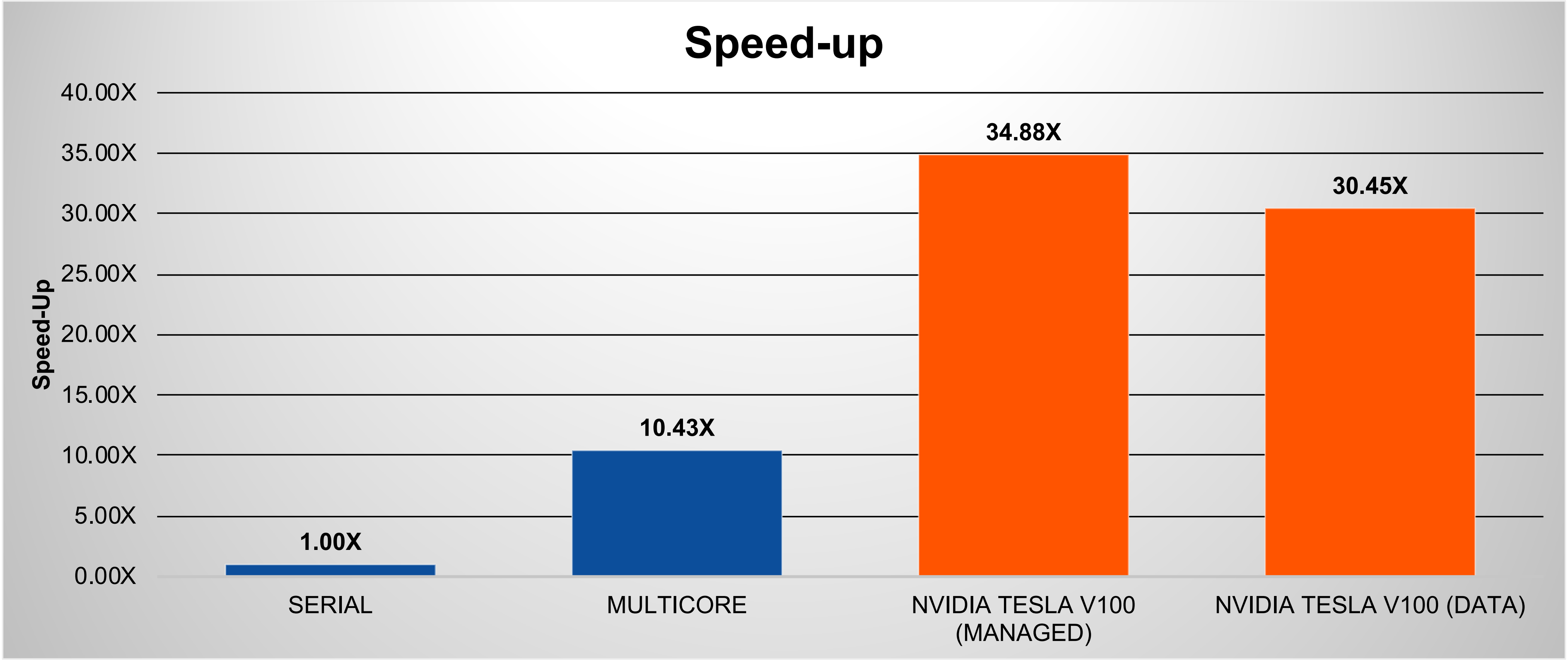
OPTIMIZED DATA MOVEMENT

```
#pragma acc data copy(A[:n*m]) copyin(Anew[:n*m])
while ( err > tol && iter < iter_max ) {
  err=0.0;

#pragma acc parallel loop reduction(max:err) copyin(A[0:n*m])
  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 copyin(Anew[0:n*m]) copyout(A[0:n*m])
  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.

Copy initial condition of Anew, but not final value

OPENACC SPEED-UP



OpenACC More Science, Less Programming NVIDIA

PGI 19.10, NVIDIA Tesla V100, IBM POWER9 22-core CPU @ 3.07GHz

GANGS, WORKERS, AND VECTORS DEMYSTIFIED

GANGS, WORKERS, AND VECTORS DEMYSTIFIED

GANGS, WORKERS, AND VECTORS DEMYSTIFIED

GANGS, WORKERS, AND VECTORS DEMYSTIFIED

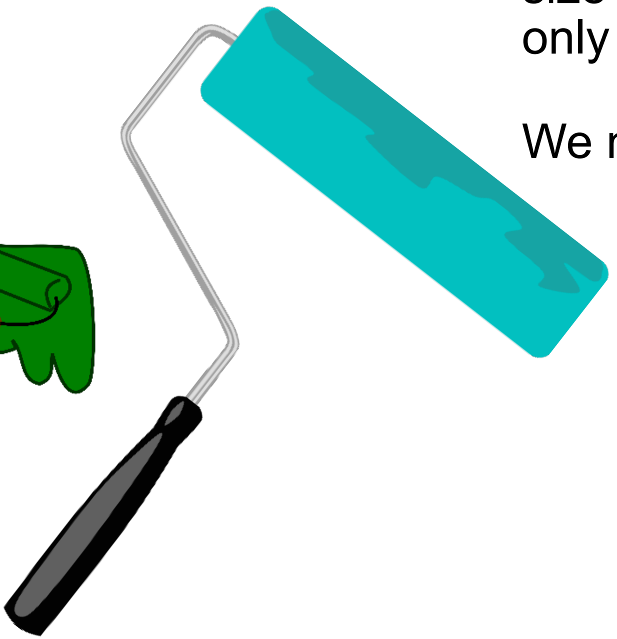
!

How much work 1 worker can do is limited by his speed.

A single worker can only move so fast.

GANGS, WORKERS, AND VECTORS DEMYSTIFIED

!



Even if we increase the size of his roller, he can only paint so fast.

We need more workers!

GANGS, WORKERS, AND VECTORS DEMYSTIFIED

Multiple workers can do more work and share resources, if organized properly.

GANGS, WORKERS, AND VECTORS DEMYSTIFIED

By organizing our workers into groups (gangs), they can effectively work together within a floor.

Groups (gangs) on different floors can operate independently.

Since gangs operate independently, we can use as many or few as we need.

GANGS, WORKERS, AND VECTORS DEMYSTIFIED

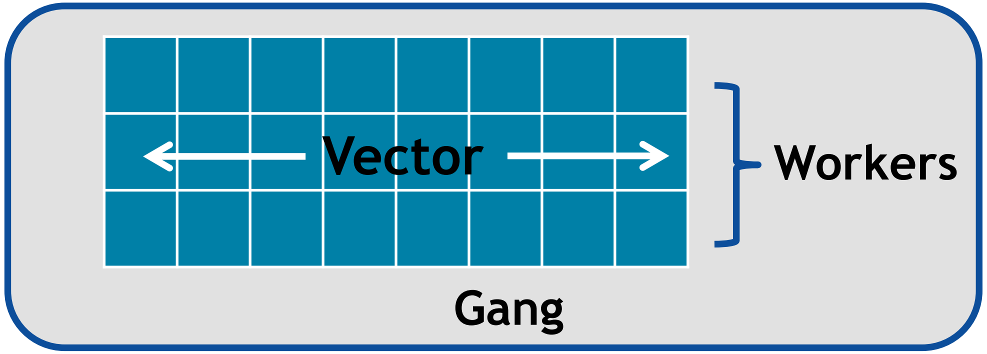
Even if there's not enough gangs for each floor, they can move to another floor when ready.

GANGS, WORKERS, AND VECTORS DEMYSTIFIED

Our painter is like an OpenACC ***worker***, he can only do so much.

His roller is like a ***vector***, he can move faster by covering more wall at once.

Eventually we need more workers, which can be organized into ***gangs*** to get more done.

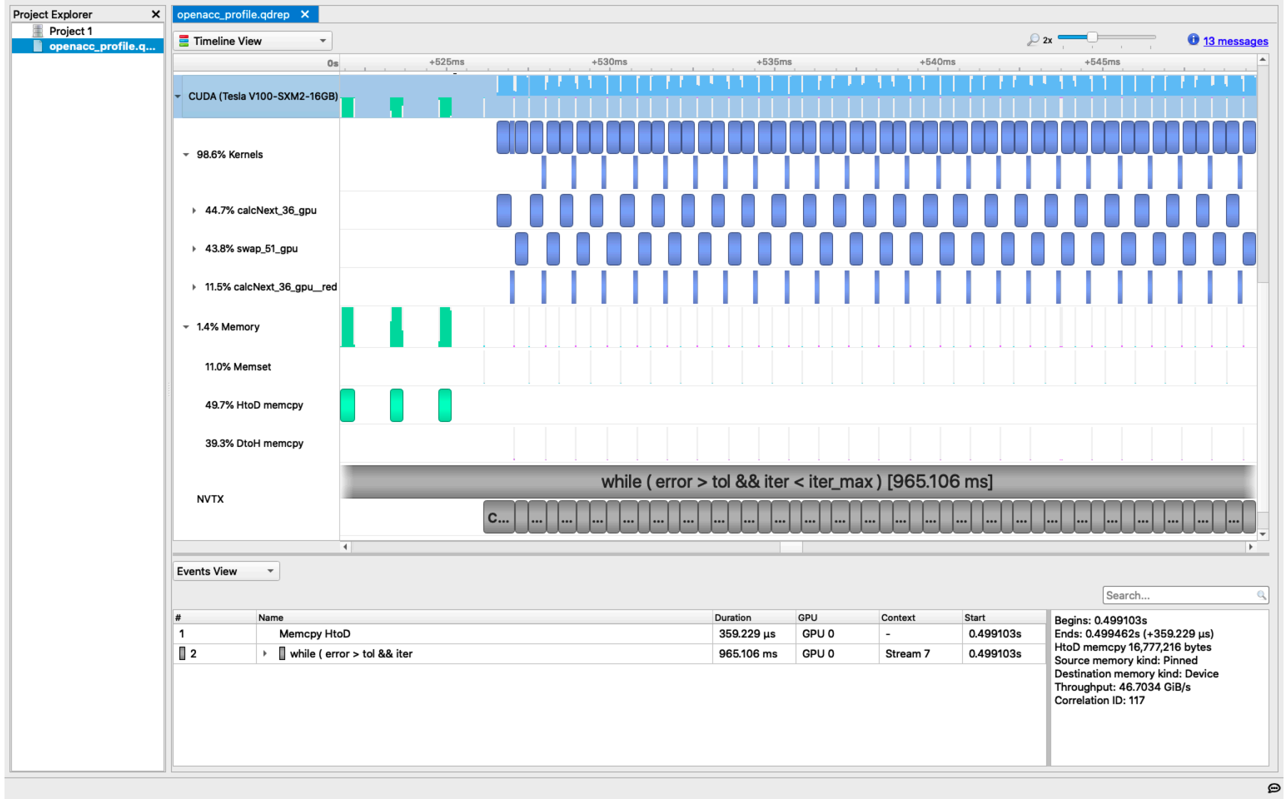


GPU PROFILES

PROFILING GPU CODE (NSIGHT-SYSTEMS)

Using Nsight-Systems to profile GPU code

- Nsight-Systems presents far more information when running on a GPU
- We can view Kernel Details, Memory Details, a Timeline, and even do Analysis of the performance



PROFILING GPU CODE (PGPROF)

Using PGPROF to profile GPU code

- **MemCpy(HtoD):** This includes data transfers from the Host to the Device (CPU to GPU)
- **MemCpy(DtoH):** These are data transfers from the Device to the Host (GPU to CPU)
- **Kernels:** These are our computational functions. We can see our calcNext and swap functions

PROFILING GPU CODE (PGPROF)

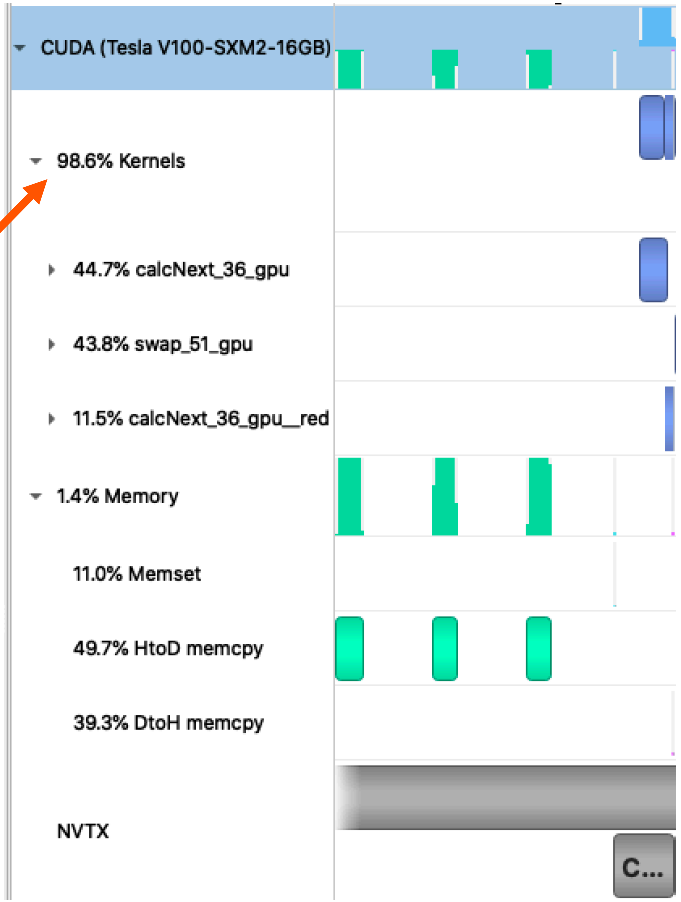
Using PGPROF to profile GPU code

- **MemCpy(HtoD):** This includes data transfers from the Host to the Device (CPU to GPU)
- **MemCpy(DtoH):** These are data transfers from the Device to the Host (GPU to CPU)
- **Kernels:** These are our computational functions. We can see our calcNext and swap functions

PROFILING GPU CODE (PGPROF)

Using PGPROF to profile GPU code

- **MemCpy(HtoD):** This includes data transfers from the Host to the Device (CPU to GPU)
- **MemCpy(DtoH):** These are data transfers from the Device to the Host (GPU to CPU)
- **Kernels:** These are our computational functions. We can see our calcNext and swap functions



LOOP OPTIMIZATIONS

OPENACC LOOP DIRECTIVE

Expressing parallelism

- Mark a single for loop for parallelization
- Allows the programmer to give additional information and/or optimizations about the loop
- Provides many different ways to describe the type of parallelism to apply to the loop
- Must be contained within an OpenACC compute region (either a kernels or a parallel region) to parallelize loops

C/C++

```
#pragma acc loop
for(int i = 0; i < N; i++)
  // Do something
```

Fortran

```
!$acc loop
do i = 1, N
  ! Do something
```

COLLAPSE CLAUSE

- `collapse( N )`
- Combine the next N tightly nested loops
- Can turn a multidimensional loop nest into a single-dimension loop
- This can be extremely useful for increasing memory locality, as well as creating larger loops to expose more parallelism

```
#pragma acc parallel loop collapse(2)
for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    double tmp = 0.0f;
    #pragma acc loop reduction(+:tmp)
    for( k = 0; k < size; k++ )
      tmp += a[i][k] * b[k][j];
    c[i][j] = tmp;
```

COLLAPSE CLAUSE

collapse(2)

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

```
#pragma acc parallel loop collapse(2)
for( i = 0; i < 4; i++ )
  for( j = 0; j < 4; j++ )
    array[i][j] = 0.0f;
```

COLLAPSE CLAUSE

When/Why to use it

- A single loop might not have enough iterations to warrant parallelization
- Collapsing outer loops gives more scalable (gangs) parallelism
- Collapsing inner loops gives more fine-grained (vector) parallelism
- Collapsing all loops gives the compiler total freedom, but may cost data locality

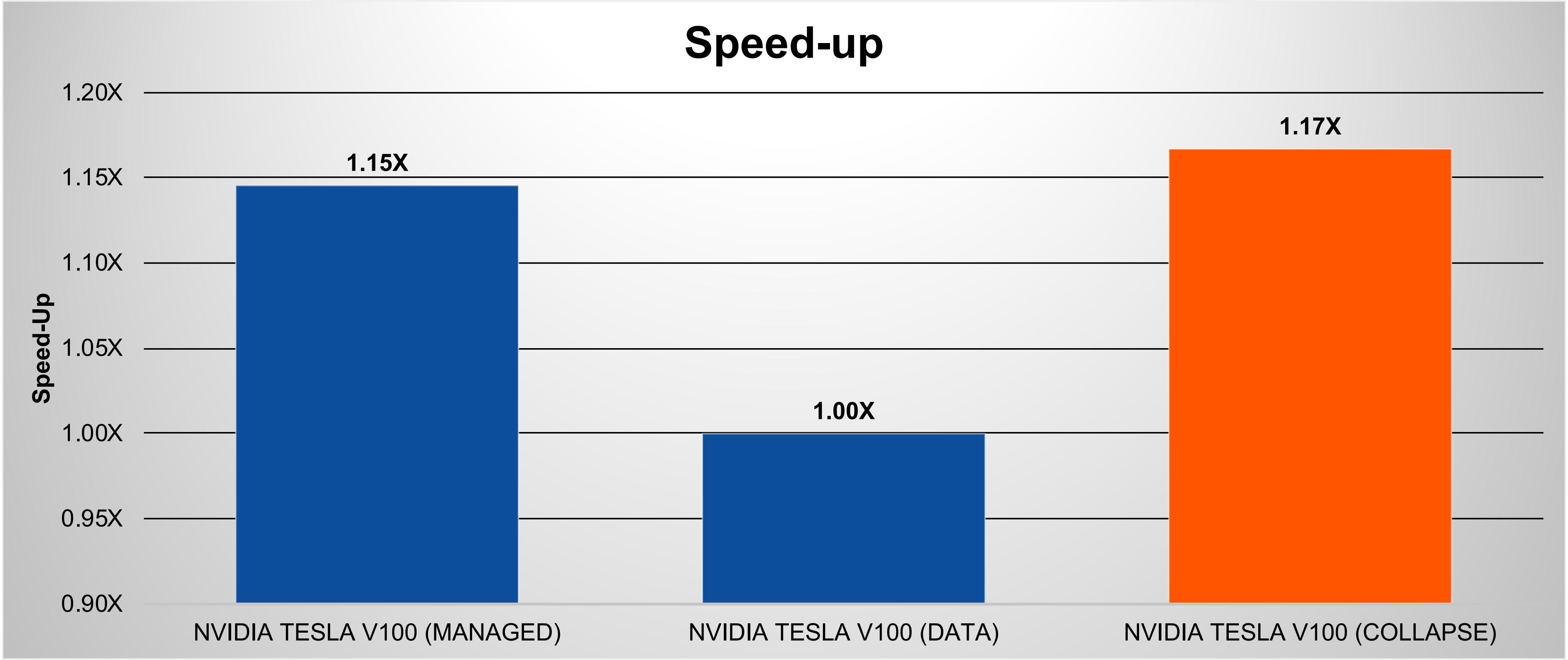
COLLAPSE CLAUSE

```
#pragma acc data copy(A[:n*m]) copyin(Anew[:n*m])
while ( err > tol && iter < iter_max ) {
  err=0.0;

#pragma acc parallel loop reduction(max:err) collapse(2) \
            copyin(A[0:n*m]) copy(Anew[0:n*m])
  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 collapse(2) \
            copyin(Anew[0:n*m]) copyout(A[0:n*m])
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

Collapse 2 loops into one for more flexibility in parallelizing.

OpenACC More Science, Less Programming NVIDIA

OPENACC SPEED-UP



PGI 19.10, NVIDIA Tesla V100, IBM POWER9 22-core CPU @ 3.07GHz

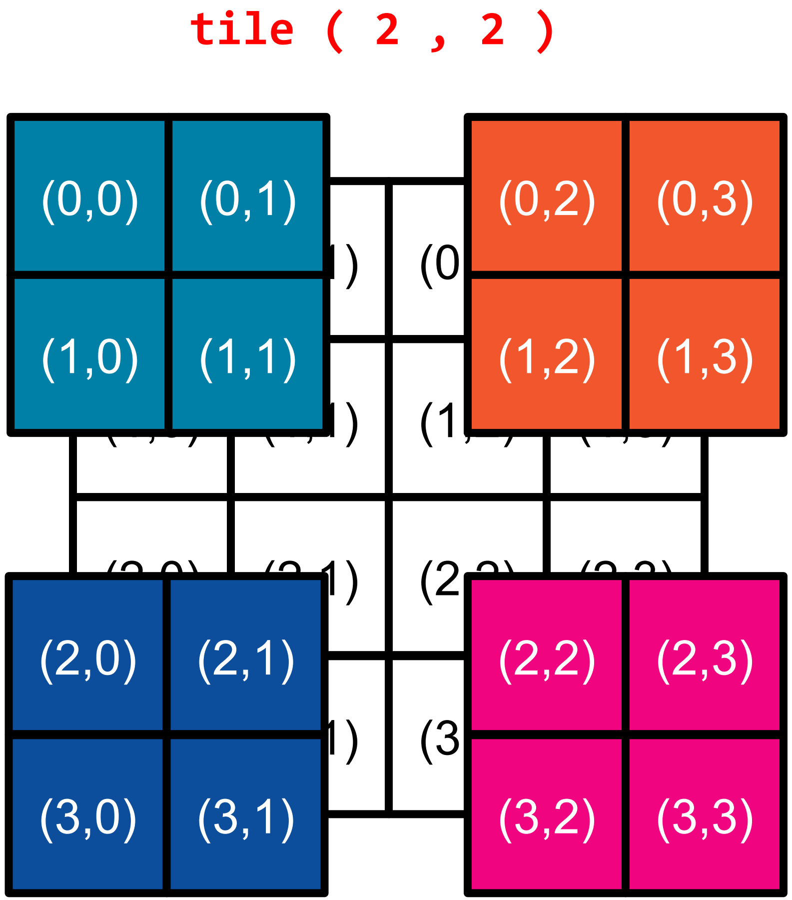
TILE CLAUSE

- **`tile ( x , y , z, ...)`**
- Breaks multidimensional loops into "tiles" or "blocks"
- Can increase data locality in some codes
- Will be able to execute multiple "tiles" simultaneously

```
#pragma acc kernels loop tile(32, 32)
for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    for( k = 0; k < size; k++ )
      c[i][j] += a[i][k] * b[k][j];
```

TILE CLAUSE

```
#pragma acc kernels loop tile(2,2)
for(int x = 0; x < 4; x++){
  for(int y = 0; y < 4; y++){
    array[x][y]++;
  }
}
```



TILE CLAUSE

```
#pragma acc data copy(A[:n*m]) copyin(Anew[:n*m])
while ( err > tol && iter < iter_max ) {
  err=0.0;

#pragma acc parallel loop reduction(max:err) tile(32,32) \
            copyin(A[0:n*m]) copy(Anew[0:n*m])
  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(32,32) \
            copyin(Anew[0:n*m]) copyout(A[0:n*m])
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

Create 32x32 tiles of the loops to better exploit data locality.

TILING RESULTS (V100)

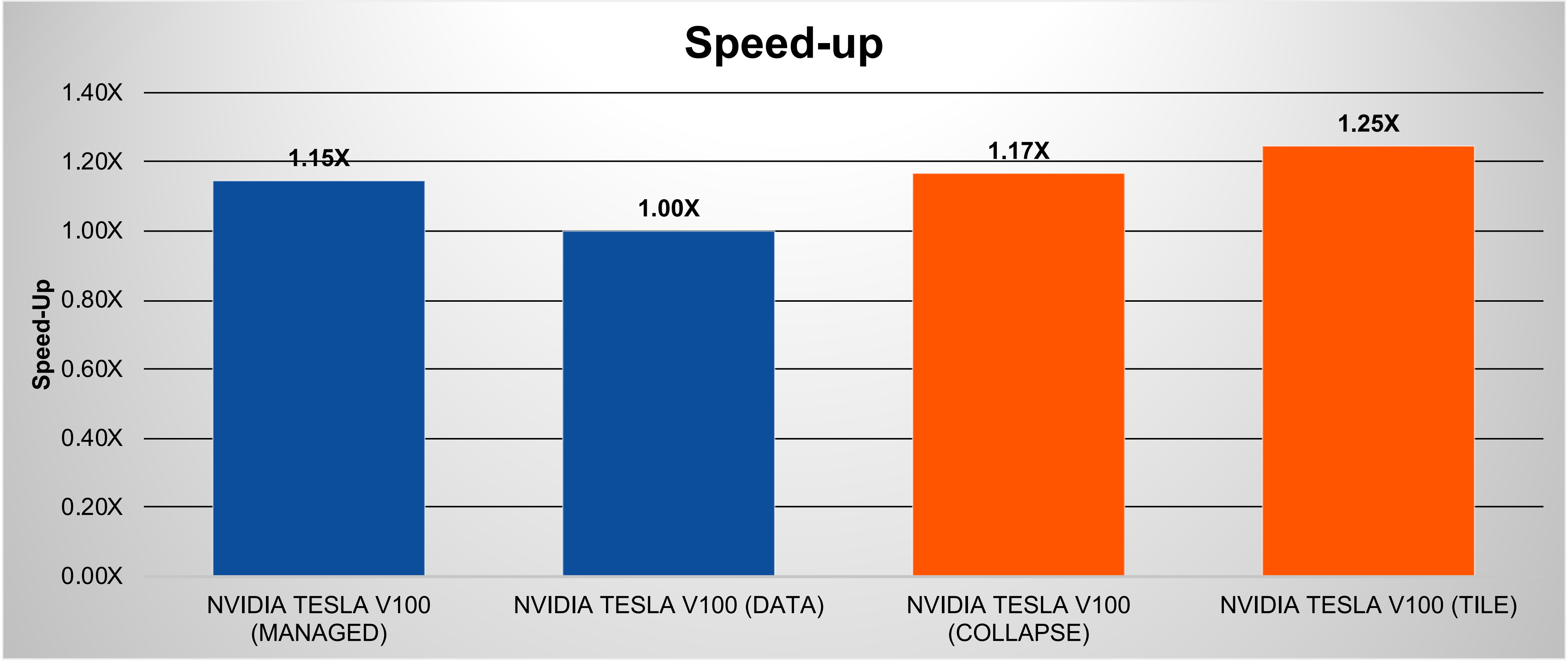
The collapse clause often requires an exhaustive search of options.

For our example code…

- CPU saw no benefit from tiling
- GPU saw anywhere from a 15% loss of performance to a 25% improvement

	CPU Improvement	GPU Improvement
Baseline	**1.00X**	**1.00X**
4x4	**1.00X**	**0.85X**
4x8	**1.00X**	**0.95X**
8x4	**1.00X**	**0.99X**
8x8	**1.00X**	**0.99X**
8x16	**1.00X**	**1.03X**
16x8	**1.00X**	**1.09X**
16x16	**1.00X**	**1.11X**
16x32	**1.00X**	**1.18X**
32x16	**1.00X**	**1.22X**
32x32	**1.00X**	**1.25X**

OPENACC SPEED-UP



PGI 19.10, NVIDIA Tesla V100, IBM POWER9 22-core CPU @ 3.07GHz

OpenACC More Science, Less Programming
NVIDIA

GANG, WORKER, AND VECTOR CLAUSES

- The developer can instruct the compiler which levels of parallelism to use on given loops by adding clauses:
- gang – Mark this loop for gang parallelism
- worker – Mark this loop for worker parallelism
- vector – Mark this loop for vector parallelism

These can be combined on the same loop.

```
#pragma acc parallel loop gang
for( i = 0; i < size; i++ )
  #pragma acc loop worker
  for( j = 0; j < size; j++ )
    #pragma acc loop vector
    for( k = 0; k < size; k++ )
      c[i][j] += a[i][k] * b[k][j];
```

```
#pragma acc parallel loop \
          collapse(3) gang vector
for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    for( k = 0; k < size; k++ )
      c[i][j] += a[i][k] * b[k][j];
```

SEQ CLAUSE

- The **seq** clause (short for sequential) will tell the compiler to run the loop sequentially
- In the sample code, the compiler will parallelize the outer loops across the parallel threads, but each thread will run the inner-most loop sequentially
- The compiler may automatically apply the seq clause to loops as well

```
#pragma acc parallel loop
for( i = 0; i < size; i++ )
  #pragma acc loop
  for( j = 0; j < size; j++ )
    #pragma acc loop seq
    for( k = 0; k < size; k++ )
      c[i][j] += a[i][k] * b[k][j];
```

ADJUSTING GANGS, WORKERS, AND VECTORS

The compiler will choose a number of gangs, workers, and a vector length for you, but you can change it with clauses.

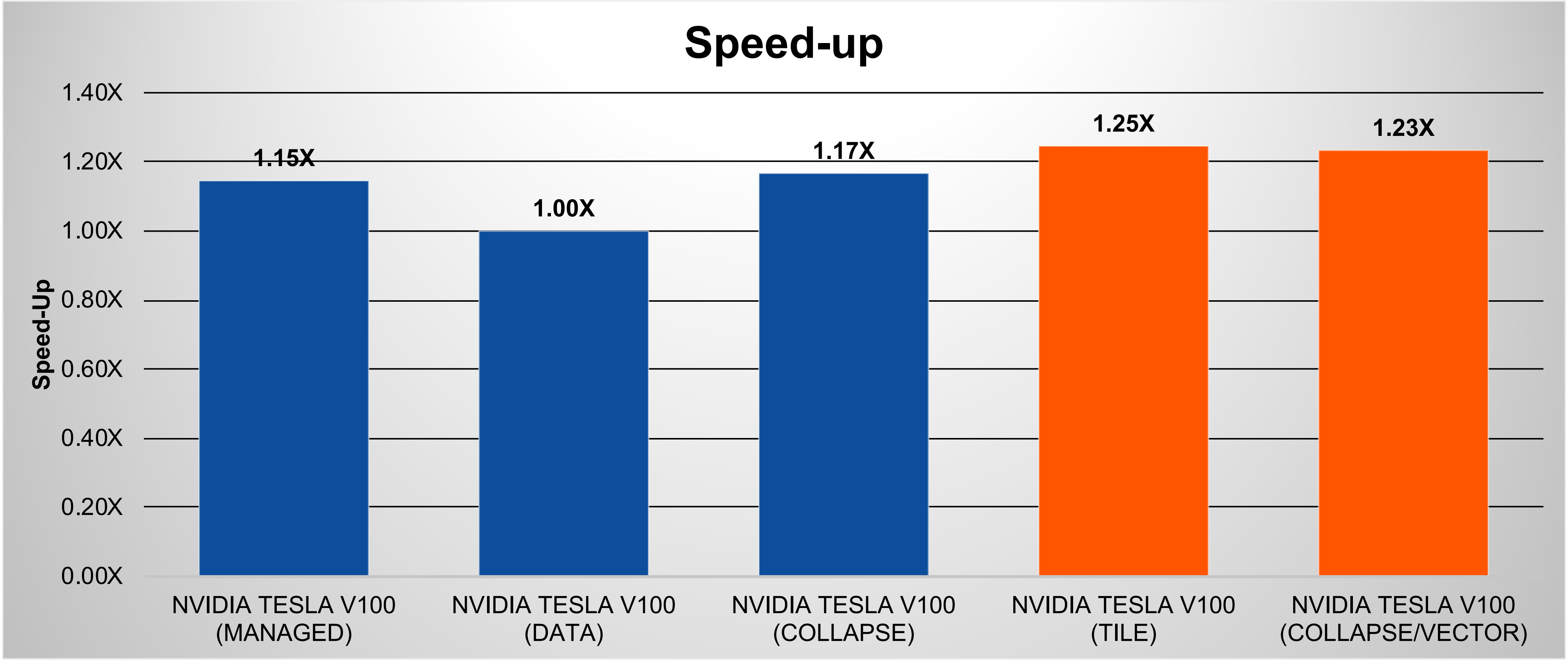
- **num_gangs(N)** – Generate N gangs for this parallel region
- **num_workers(M)** – Generate M workers for this parallel region
- **vector_length(Q)** – Use a vector length of Q for this parallel region

```
#pragma acc parallel num_gangs(2) \
  num_workers(2) vector_length(32)
{
  #pragma acc loop gang worker
  for(int x = 0; x < 4; x++){
    #pragma acc loop vector
    for(int y = 0; y < 32; y++){
      array[x][y]++;
    }
  }
}
```

COLLAPSE CLAUSE WITH VECTOR LENGTH

```
#pragma acc data copy(A[:n*m]) copyin(Anew[:n*m])
while ( err > tol && iter < iter_max ) {
  err=0.0;

#pragma acc parallel loop reduction(max:err) collapse(2) vector_length(1024) \
            copyin(A[0:n*m]) copy(Anew[0:n*m])
  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 collapse(2) vector_length(1024) \
            copyin(Anew[0:n*m]) copyout(A[0:n*m])
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

OPENACC SPEED-UP



PGI 19.10, NVIDIA Tesla V100, IBM POWER9 22-core CPU @ 3.07GHz

LOOP OPTIMIZATION RULES OF THUMB

- It is rarely a good idea to set the number of gangs in your code, let the compiler decide.
- Most of the time you can effectively tune a loop nest by adjusting only the vector length.
- It is rare to use a worker loop on NVIDIA GPUs. When the vector length is very short, a worker loop can increase the parallelism in your gang (thread block).
- When possible, the vector loop should step through your arrays consecutively (stride==1)
- Gangs should come from outer loops, vectors from inner

CLOSING REMARKS

KEY CONCEPTS

In this lab we discussed…

- Some details that are available to use from a GPU profile
- Gangs, Workers, and Vectors Demystified
- Collapse clause
- Tile clause
- Gang/Worker/Vector clauses

OPENACC RESOURCES

Guides ● Talks ● Tutorials ● Videos ● Books ● Spec ● Code Samples ● Teaching Materials ● Events ● Success Stories ● Courses ● Slack ● Stack Overflow

FREE Compilers

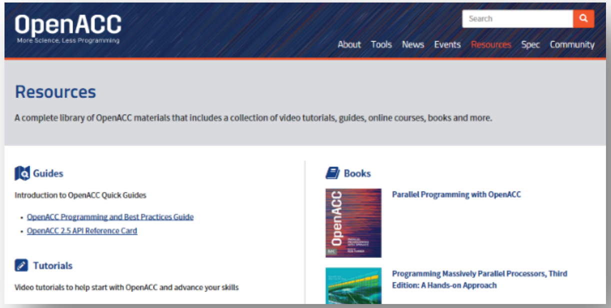
PGI Community EDITION

slack

https://www.openacc.org/community#slack

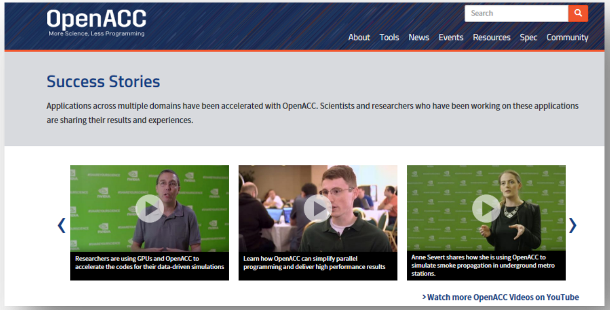
Resources

https://www.openacc.org/resources



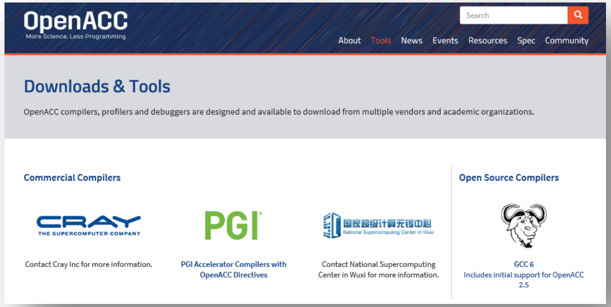
Success Stories

https://www.openacc.org/success-stories



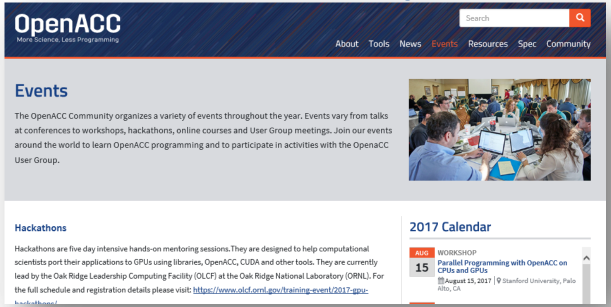
Compilers and Tools

https://www.openacc.org/tools



Events

https://www.openacc.org/events



OpenACC More Science, Less Programming

NVIDIA