OpenACC Fundamentals

Steve Abbott <sabbott@nvidia.com>, November 13, 2016

Who Am I?

2005 – B.S. Physics – Beloit College

2007 – M.S. Physics – University of Florida

2015 – Ph.D. Physics – University of New Hampshire

2015 – 2017 – Postdoctoral Associate – Oak Ridge National Lab

Ported and optimized fusion simulation code for Summit

August – Present – NVIDIA Corp.

Support deployment and use of Summit supercomputer

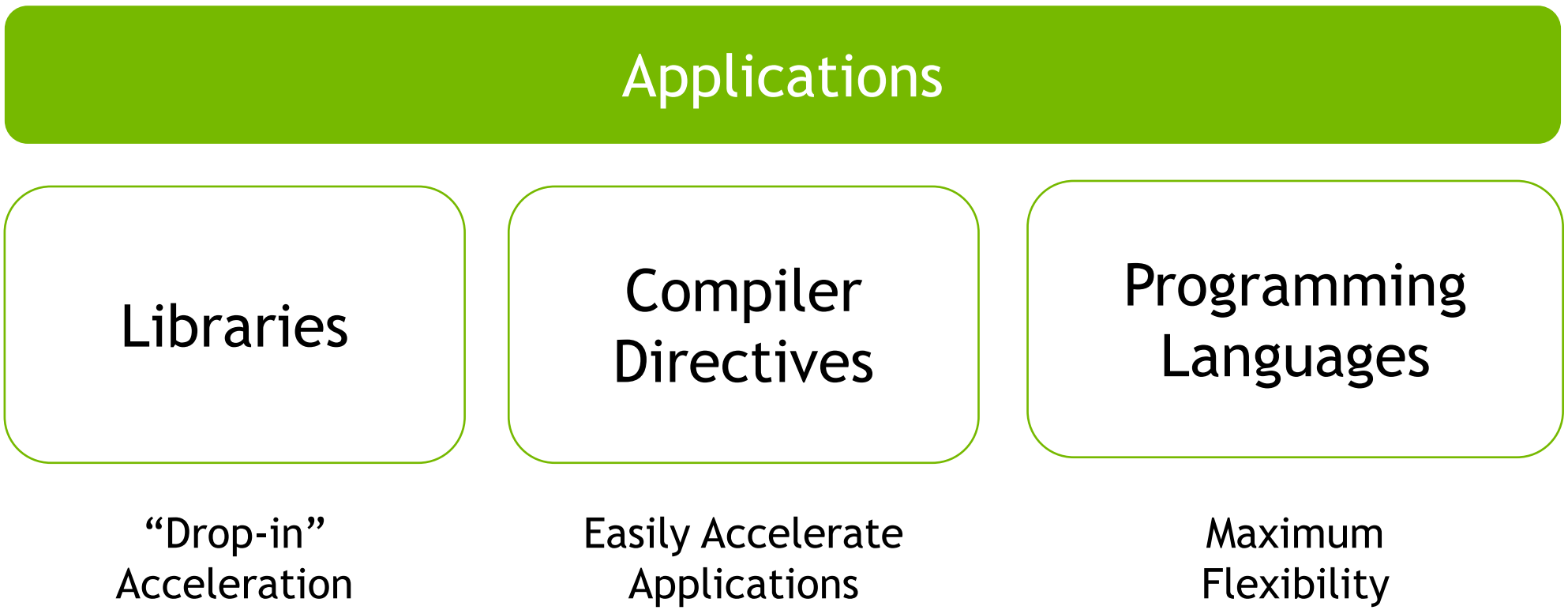
Problem solving, training, and optimizing science for GPUs

AGENDA

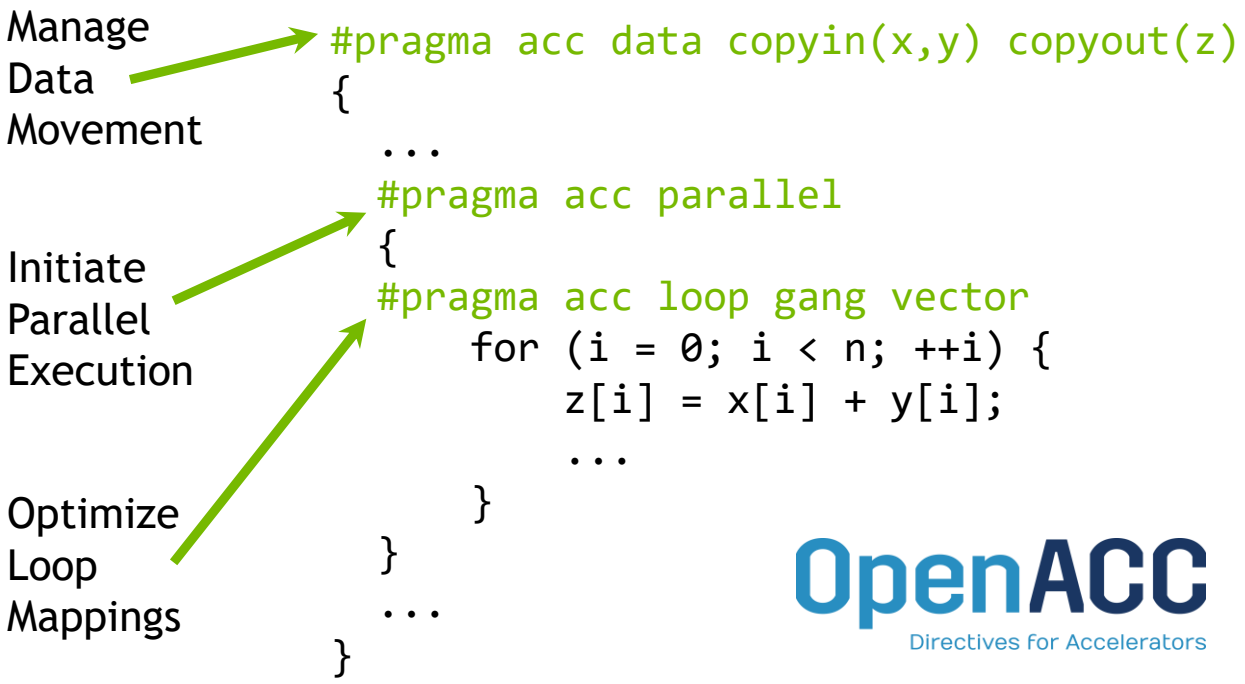
What is OpenACC?

OpenACC by Example

3 Ways to Program GPUs



OpenACC Directives



OpenACC
Directives for Accelerators

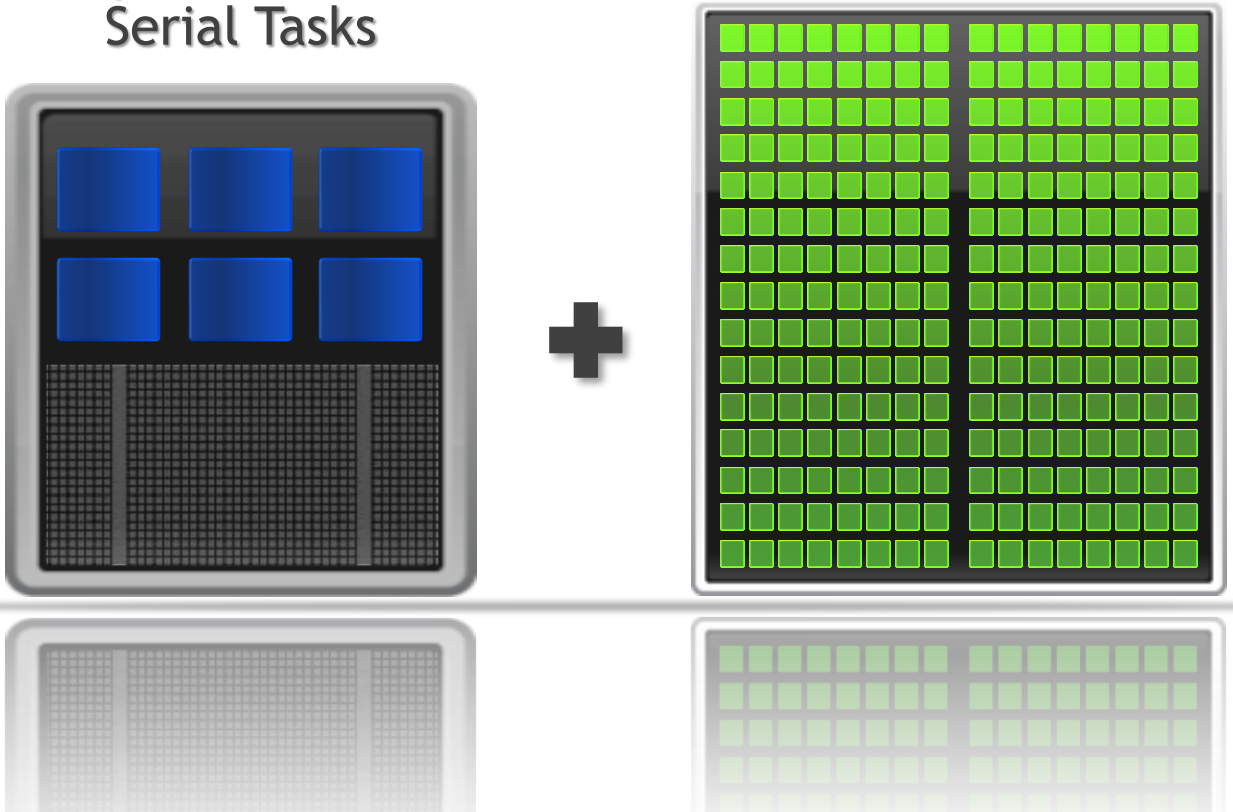
- Incremental
- Single source
- Interoperable
- Performance portable
- CPU, GPU, MIC

Accelerated Computing

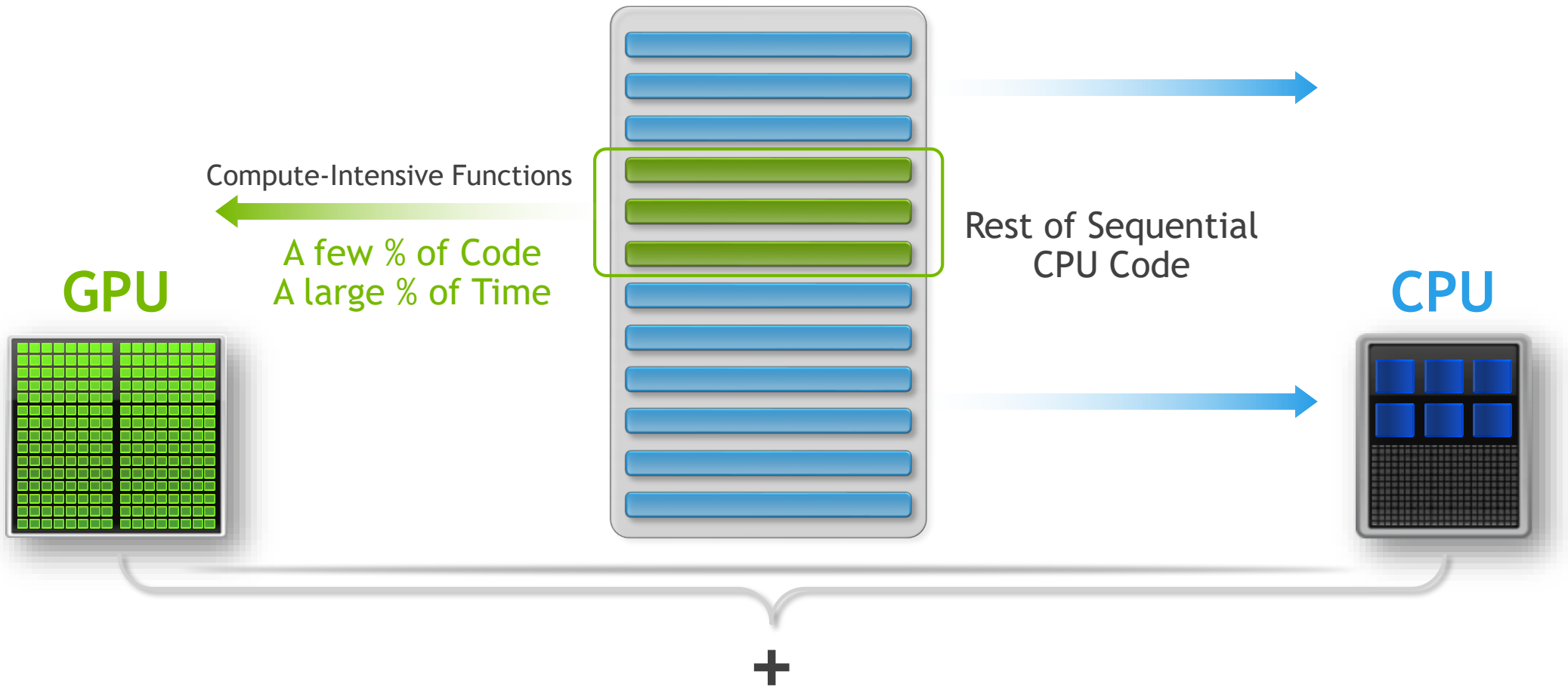
10x Performance & 5x Energy Efficiency for HPC

CPU
Optimized for
Serial Tasks

GPU Accelerator
Optimized for
Parallel Tasks

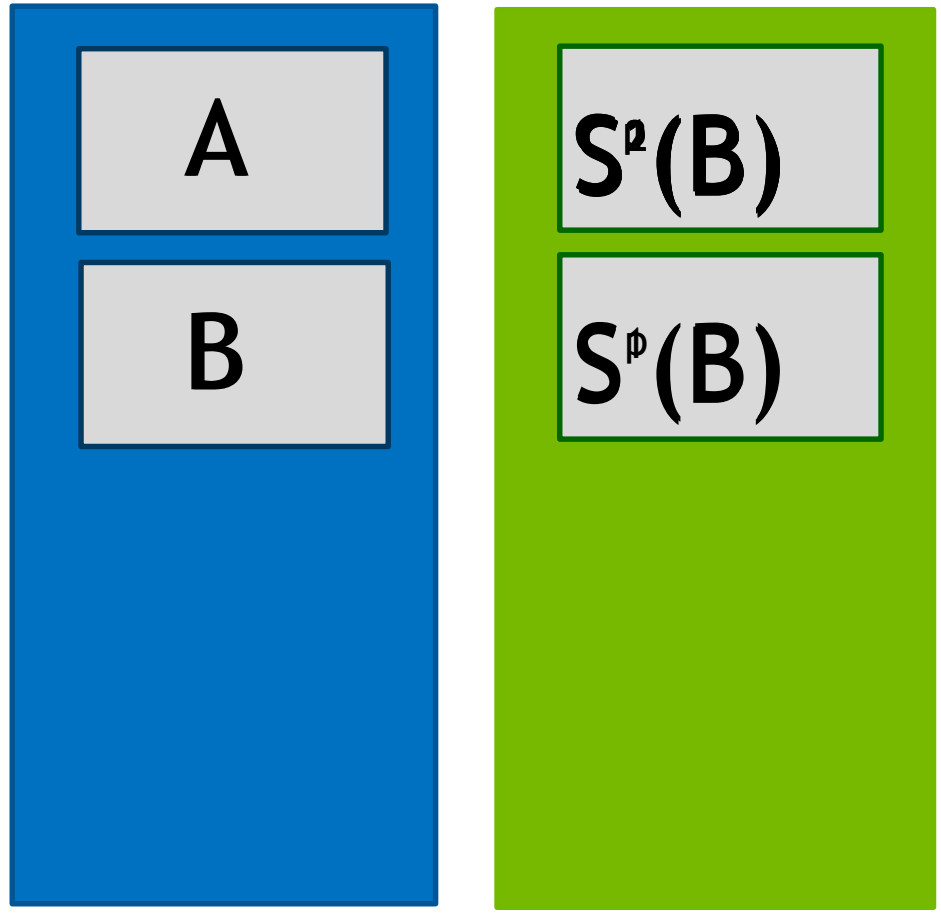


What is Accelerated Computing?



OpenACC Example

```
#pragma acc data \
        copy(b[0:n][0:m]) \
        create(a[0:n][0:m])
{
for (iter = 1; iter <= p; ++iter){
  #pragma acc kernels
  {
  for (i = 1; i < n-1; ++i){
    for (j = 1; j < m-1; ++j){
      a[i][j]=w0*b[i][j]+
              w1*(b[i-1][j]+b[i+1][j]+
                  b[i][j-1]+b[i][j+1])+
              w2*(b[i-1][j-1]+b[i-1][j+1]+
                  b[i+1][j-1]+b[i+1][j+1]);
  }
  for( i = 1; i < n-1; ++i )
    for( j = 1; j < m-1; ++j )
      b[i][j] = a[i][j];
  } }
}
}
```

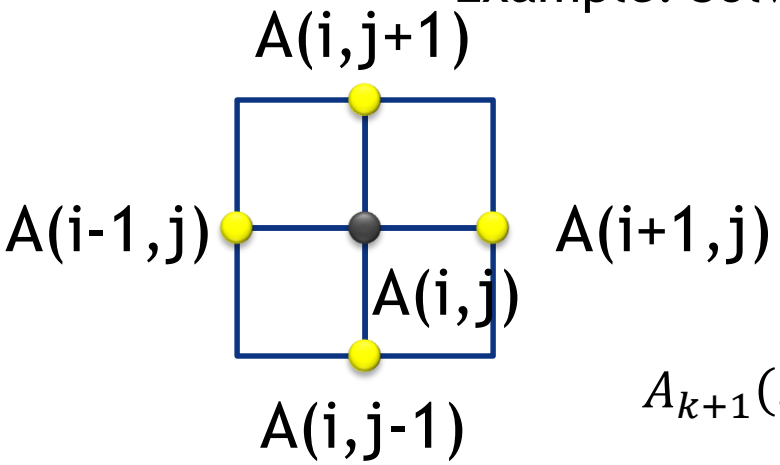


Example: Jacobi Iteration

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

Look For 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

Max Reduction required

Independent loop iterations

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 Directive

Generates parallelism

```
#pragma acc parallel
{
}
```

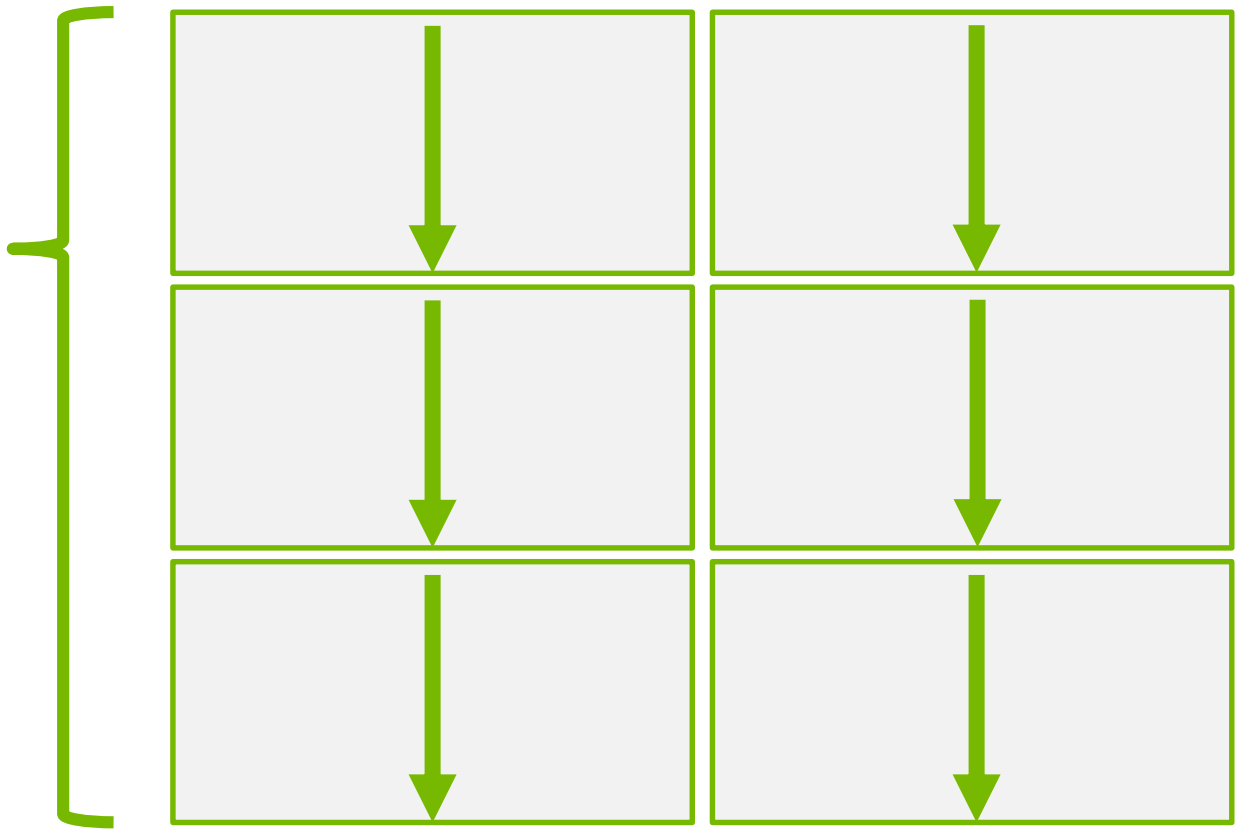
When encountering the *parallel* directive, the compiler will generate *1 or more parallel gangs*, which execute redundantly.

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.

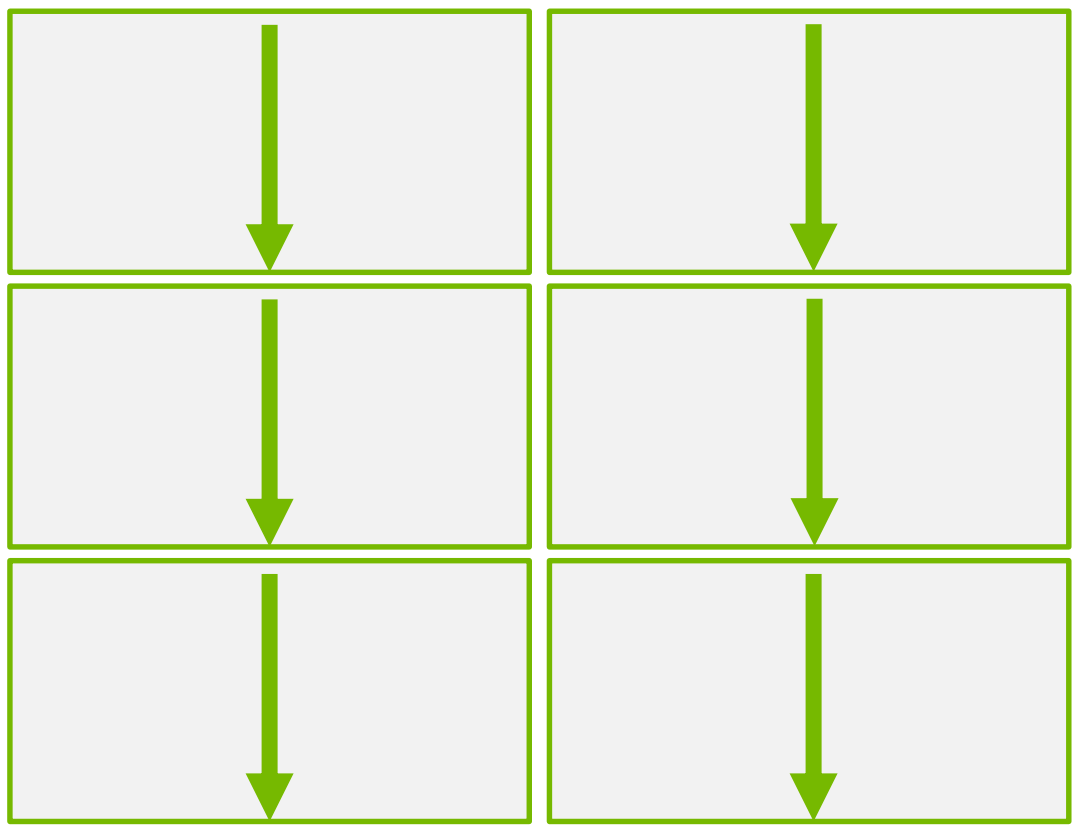


OpenACC Loop Directive

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.

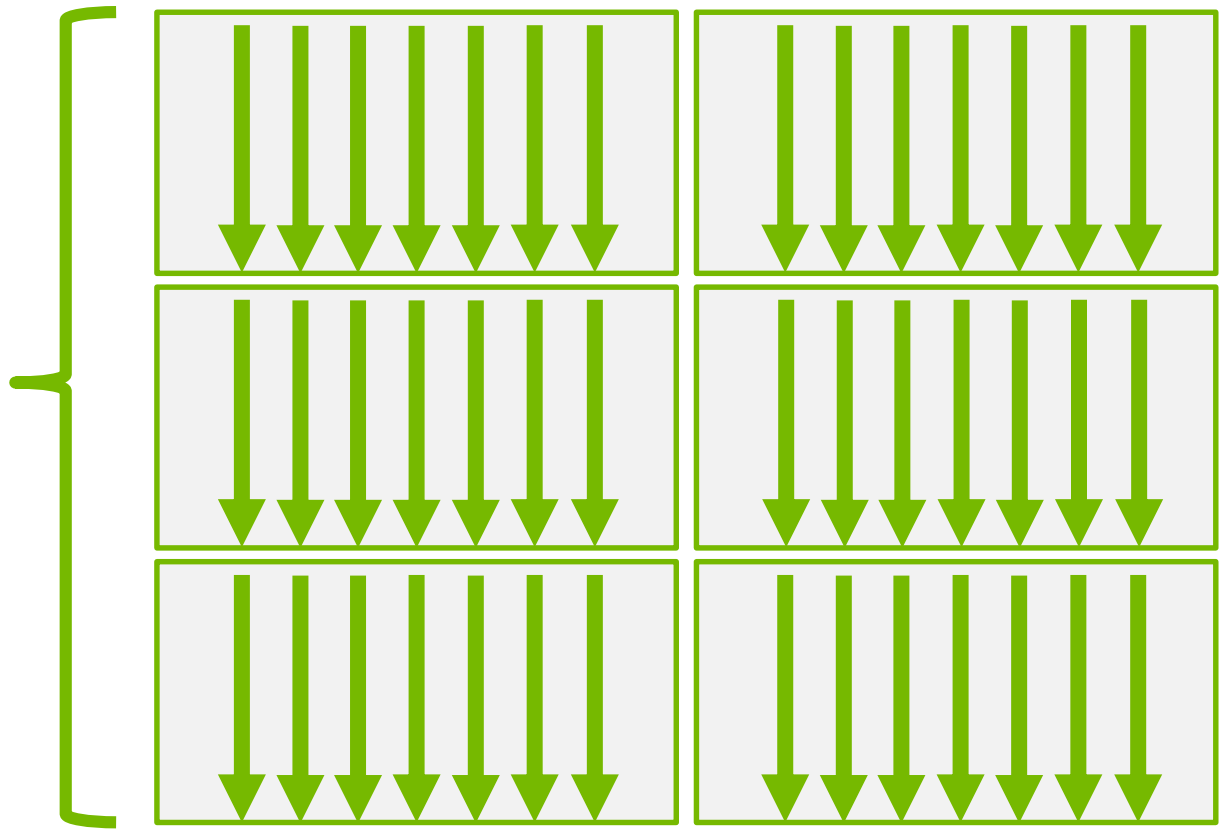


OpenACC Loop Directive

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.

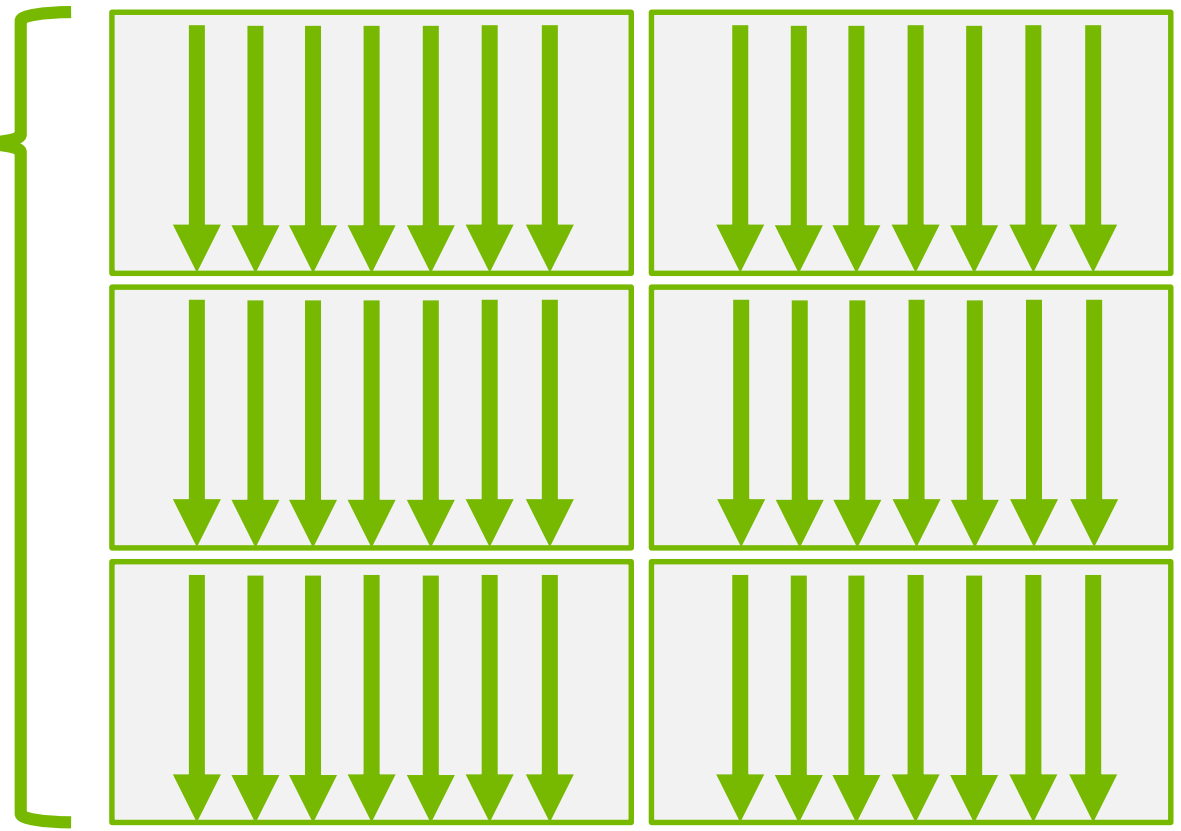


OpenACC Parallel Loop Directive

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

```
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
```

BUILDING THE CODE

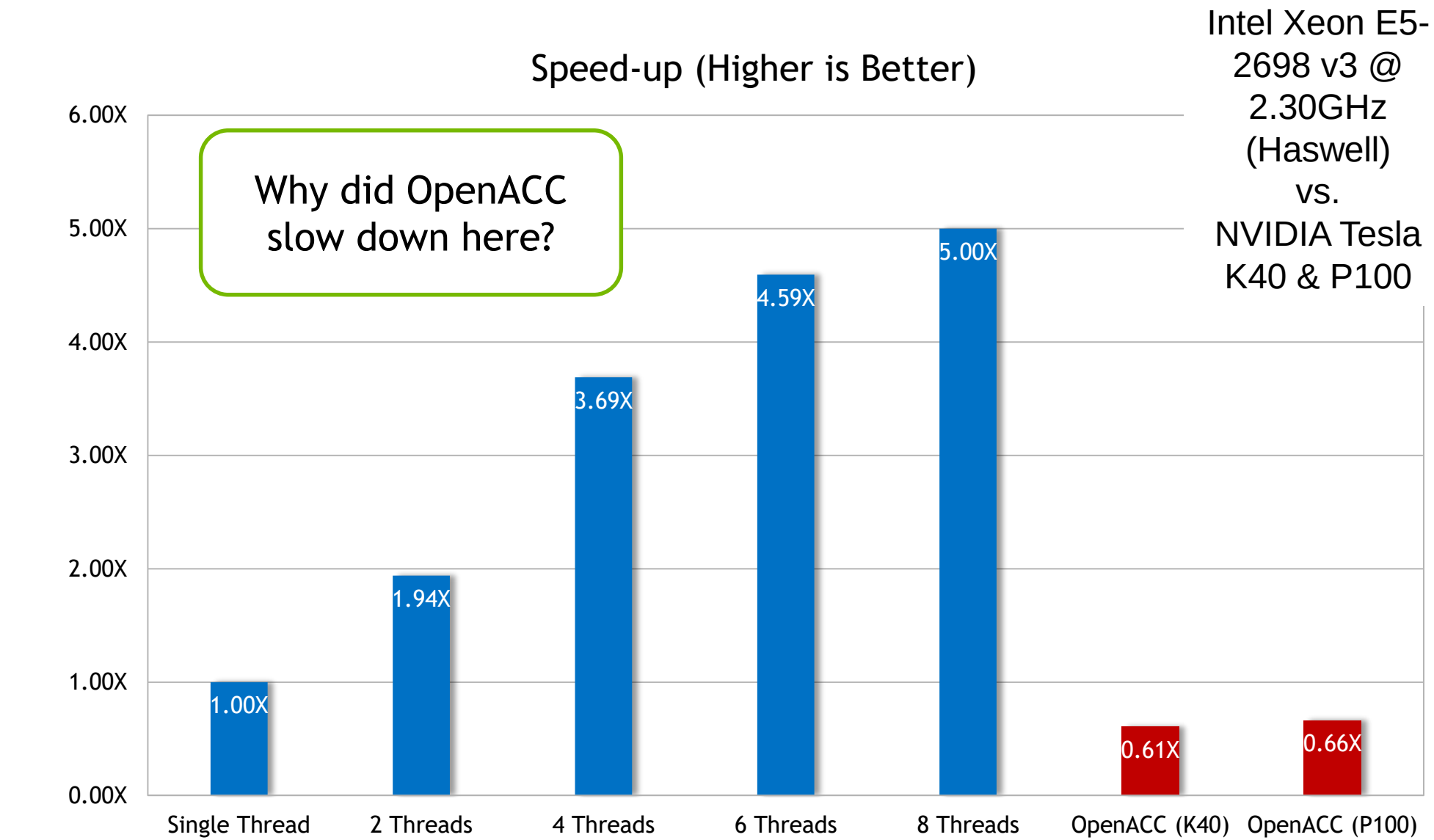
```
$ pgcc -fast -acc -ta=tesla -Minfo=all laplace2d.c
main:
     40, Loop not fused: function call before adjacent loop
         Generated vector sse code for the loop
     51, Loop not vectorized/parallelized: potential early exits
     55, Accelerator kernel generated
         55, Max reduction generated 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
```

BUILDING THE CODE

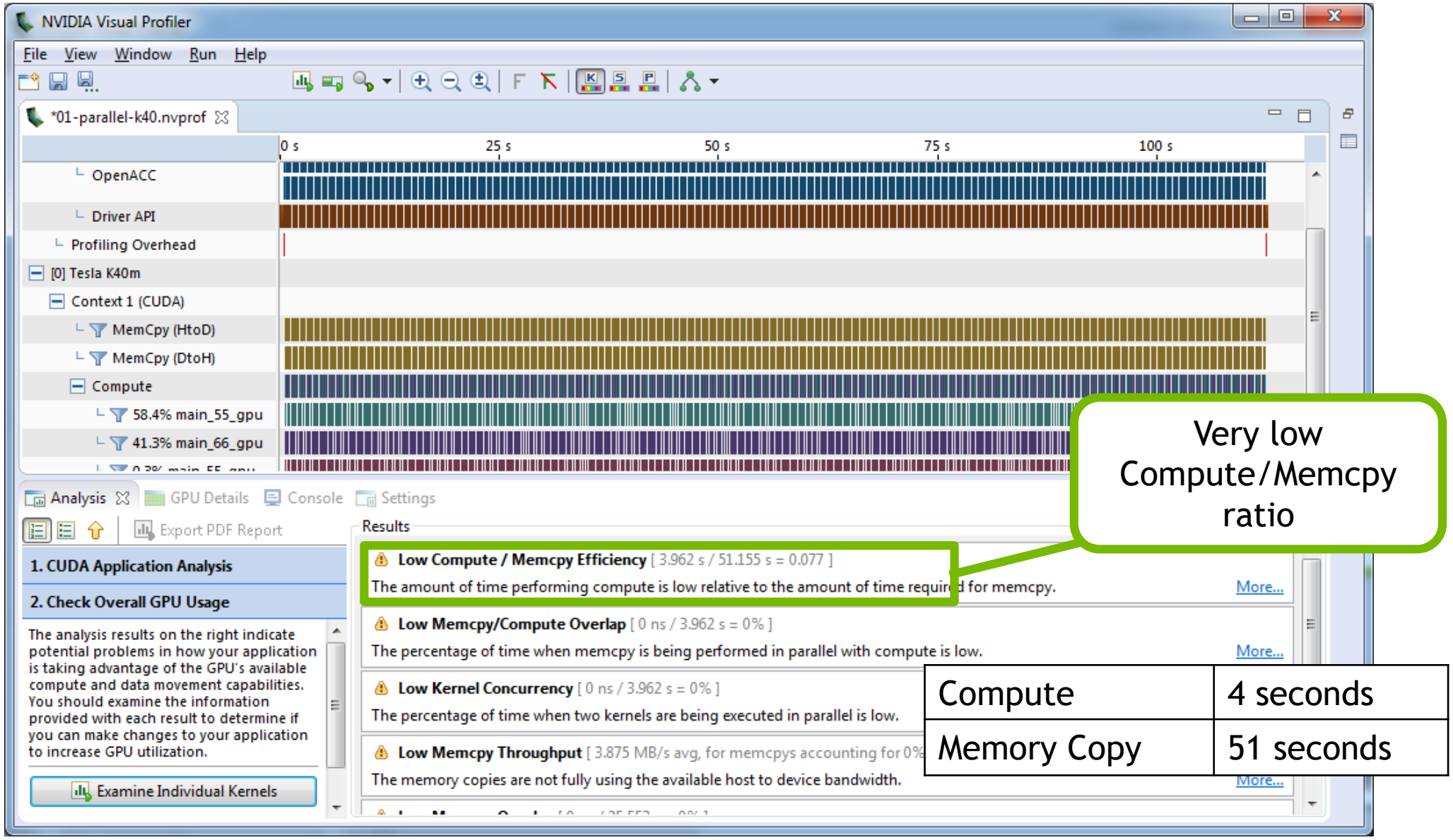
```
$ pgcc -fast -acc -ta=tesla -Minfo=all laplace2d.c
main:
     40, Loop not fused: function call before adjacent loop
         Generated vector sse code for the loop
     51, Loop not vectorized/parallelized: potential early exits
     55, Accelerator kernel generated
         55, Max reduction generated 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
```

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
```



Compiler: PGI 16.10


NVIDIA Visual Profiler
File View Window Run Help
*01-parallel-k40.nvprof
0 s
25 s
50 s
75 s
100 s
OpenACC
Driver API
Profiling Overhead
[0] Tesla K40m
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
58.4% main_55_gpu
41.3% main_66_gpu
Analysis
GPU Details
Console
Settings
Export PDF Report
1. CUDA Application Analysis
2. Check Overall GPU Usage
The analysis results on the right indicate potential problems in how your application is taking advantage of the GPU's available compute and data movement capabilities. You should examine the information provided with each result to determine if you can make changes to your application to increase GPU utilization.
Examine Individual Kernels
Results
Low Compute / Memcpy Efficiency [3.962 s / 51.155 s = 0.077]
The amount of time performing compute is low relative to the amount of time requir d for memcpy.
More...
Low Memcpy/Compute Overlap [0 ns / 3.962 s = 0%]
The percentage of time when memcpy is being performed in parallel with compute is low.
More...
Low Kernel Concurrency [0 ns / 3.962 s = 0%]
The percentage of time when two kernels are being executed in parallel is low.
Low Memcpy Throughput [3.875 MB/s avg, for memcpys accounting for 0%
The memory copies are not fully using the available host to device bandwidth.
More...
Very low Compute/Memcpy ratio
Compute 4 seconds
Memory Copy 51 seconds

Rest of the Week

Wednesday – Data directives

Friday – Advanced OpenACC and Tuning