

# INTRODUCTION TO OPENACC

Lecture 2: OpenACC Optimizations, November 2, 2016





Course Objective:

Enable *you* to accelerate your applications  
with OpenACC.

# Course Syllabus

Oct 26: Analyzing and Parallelizing with OpenACC

Nov 2: OpenACC Optimizations

Nov 9: Advanced OpenACC

Recordings:

<https://developer.nvidia.com/intro-to-openacc-course-2016>

# OPENACC OPTIMIZATIONS

Lecture 2: Jeff Larkin, NVIDIA



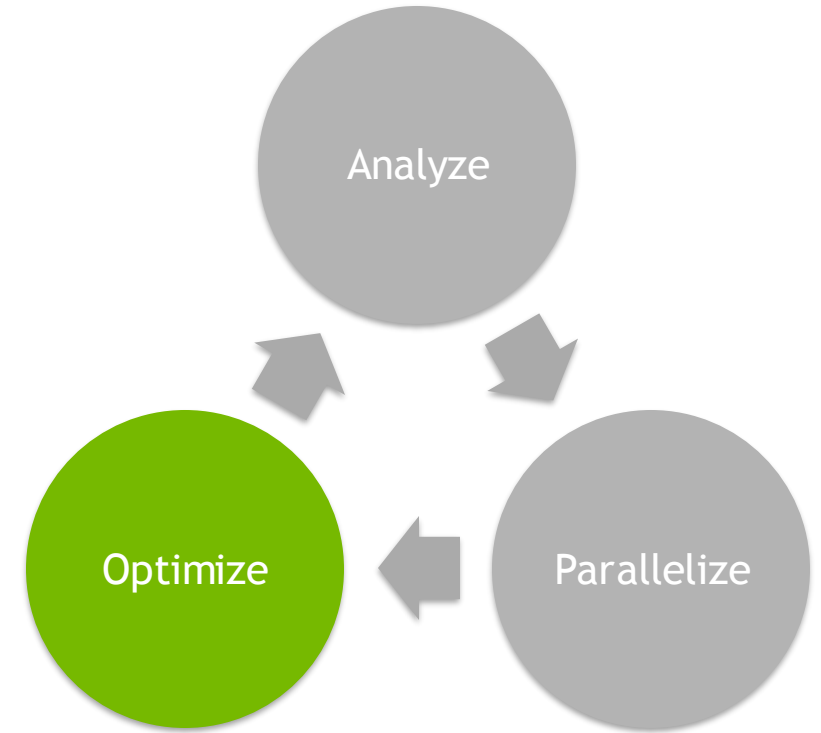
# Today's Objectives

Understand OpenACC data directives

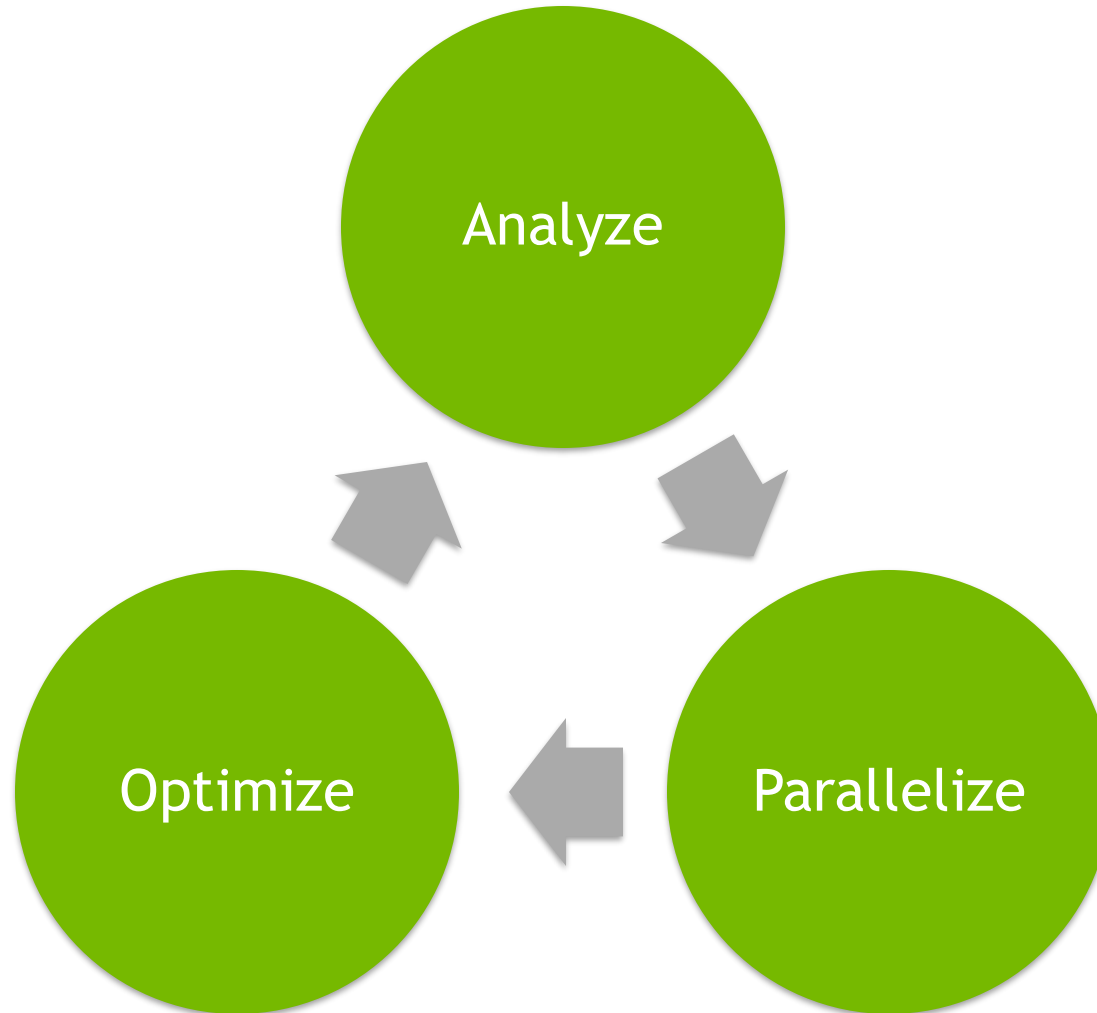
Understand the 3 levels of OpenACC parallelism

Understand how to optimize loop decomposition

Understand other common optimizations to OpenACC codes



# 3 Steps to Accelerate with OpenACC



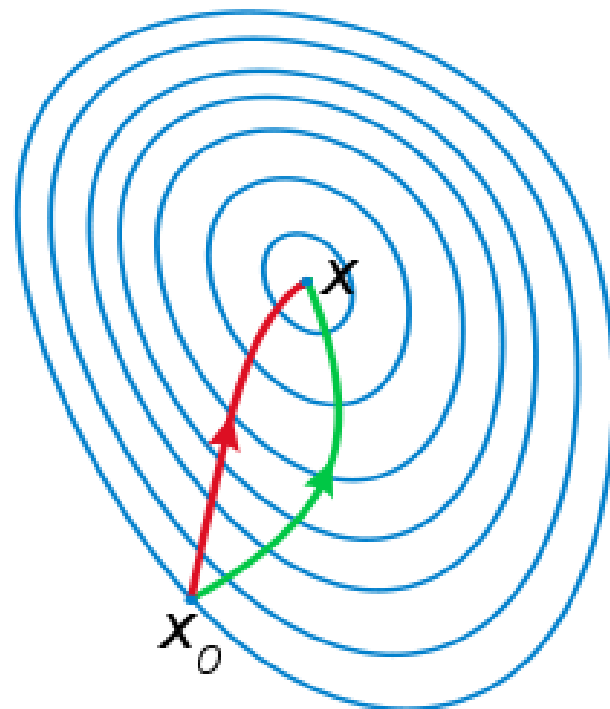
# Case Study: Conjugate Gradient

A sample code implementing the conjugate gradient method has been provided in C/C++ and Fortran.

- *To save space, only the C will be shown in slides.*

You do not need to understand the algorithm to proceed, but should be able to understand C, C++, or Fortran.

For more information on the CG method, see [https://en.wikipedia.org/wiki/Conjugate\\_gradient\\_method](https://en.wikipedia.org/wiki/Conjugate_gradient_method)





Analyze



# Analyze

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

The screenshot shows the PGPROF application interface. The top pane displays C code from `matrix_functions.h`. The bottom pane shows a performance profile with a table of events, their percentages, and execution times. The 'TOTAL' filter is selected in the dropdown.

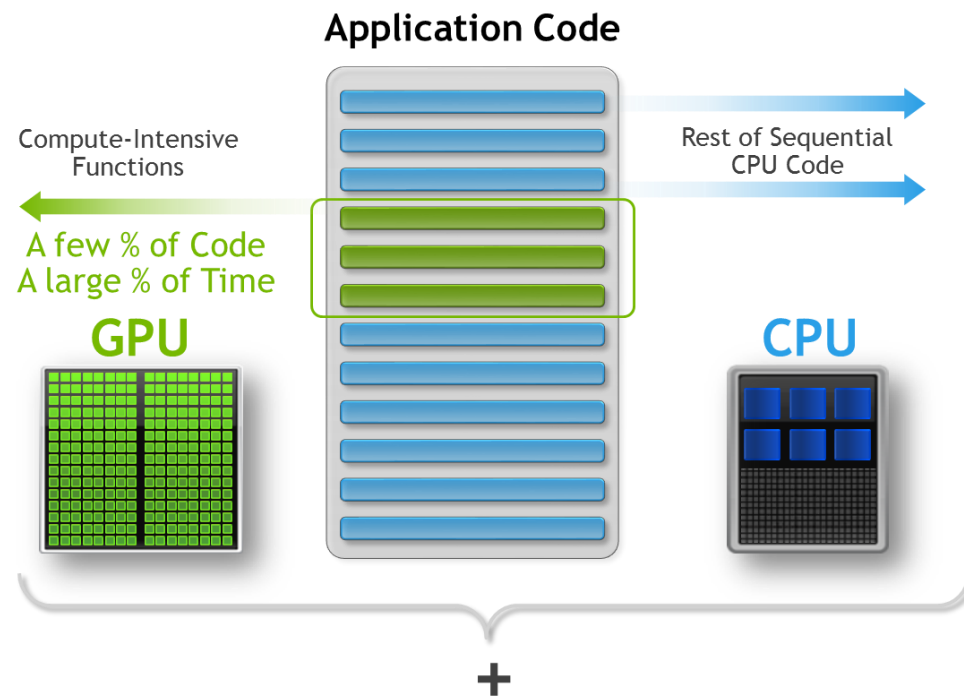
```
23 unsigned int *row_offsets=A.row_offsets;
24 unsigned int *cols=A.cols;
25 double *Acoefs=A.coefs;
26 double *xcoefs=x.coefs;
27 double *ycoefs=y.coefs;
28
29 for(int i=0;i<num_rows;i++) {
30     double sum=0;
31     int row_start=row_offsets[i];
32     int row_end=row_offsets[i+1];
33     for(int j=row_start;j<row_end;j++) {
34         unsigned int Acol=cols[j];
35         double Acoef=Acoefs[j];
36         double xcoef=xcoefs[Acol];
37         sum+=Acoef*xcoef;
38     }
39     ycoefs[i]=sum;
40 }
41 }
```

Event	%	Time
matvec(matrix const &, vector const &)	76.742%	26.17 s
waxpby(double, vector const &, vector const &)	14.819%	5.053 s
dot(vector const &, vector const &)	4.263%	1.454 s
allocate_3d_poisson_matrix(n)	3.881%	1.324 s
_c_mset8	0.206%	0.07 s
munmap	0.088%	0.03 s

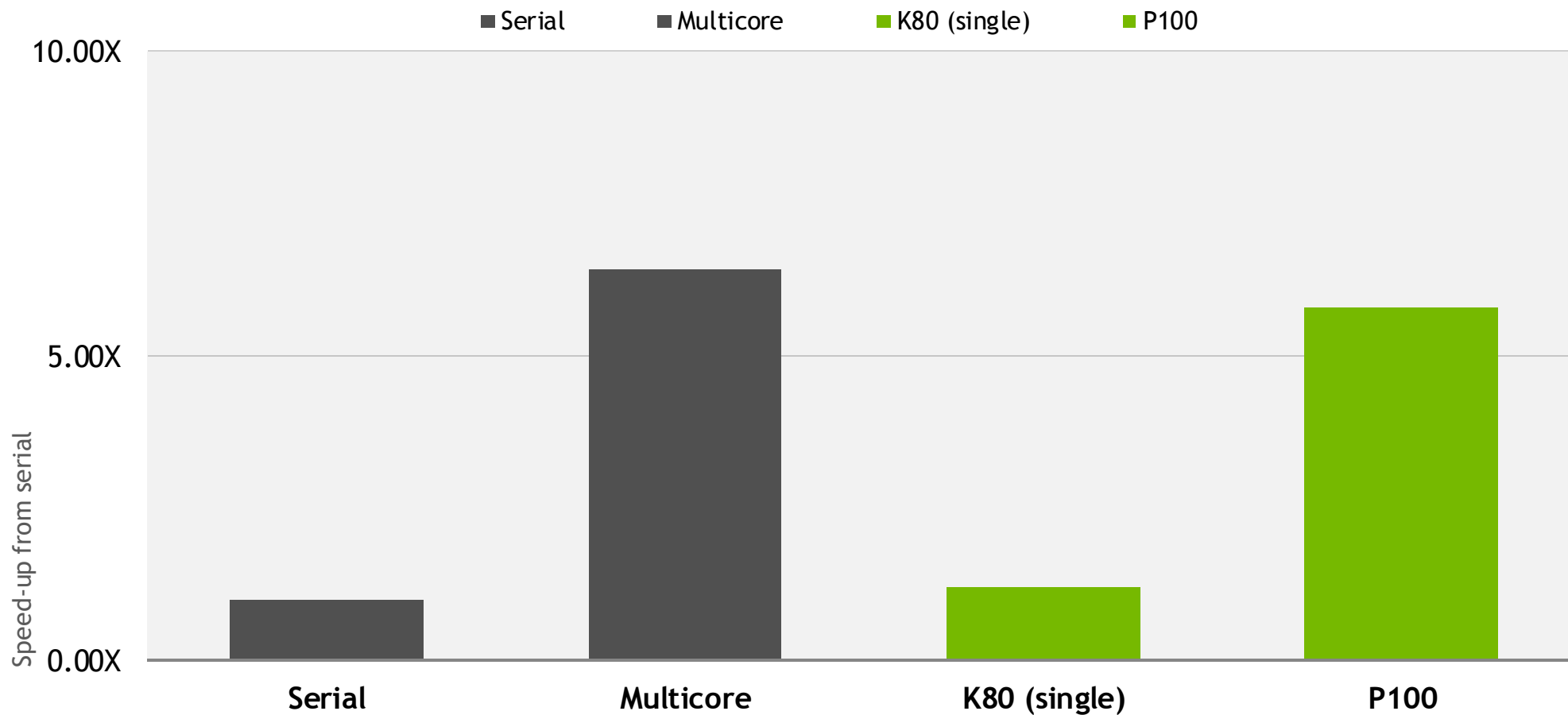
Parallelize

# Parallelize

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



# Performance after step 2...



Optimize

# Optimize

- ▶ Get new performance data from parallel execution
- ▶ Remove unnecessary data transfer to/from GPU
- ▶ Guide the compiler to better loop decomposition
- ▶ Refactor the code to make it more parallel



# Optimize Data Movement

Last week we relied on Unified Virtual Memory to expose our data to both the CPU and GPU.

To make our code more portable and give the compiler more information, we will replace UVM with OpenACC data directives



# Case Study: Remove Managed Memory

- ▶ Remove the “managed” suboption to the -ta compiler flag
- ▶ Now the compiler aborts because it doesn’t know the sizes of the arrays used in matvec function

```
PGCC-S-0155-Compiler failed to translate accelerator
region (see -Minfo messages): Could not find
allocated-variable index for symbol (main.cpp: 12)
matvec(const matrix &, const vector &, const vector
&):
    8, include "matrix_functions.h"
    12, Accelerator kernel generated
        Generating Tesla code
        15, #pragma acc loop gang /* blockIdx.x */
        20, #pragma acc loop vector(128) /*
threadIdx.x */
            Generating reduction(+:sum)
    20, Accelerator restriction: size of the GPU
copy of Acoefs,cols,xcoefs is unknown
        Loop is parallelizable
PGCC/x86 Linux 16.9-0: compilation completed with
severe errors
```

# 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.

# 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:nelem]) copyout(b[s/4:3*s/4])
```

Fortran

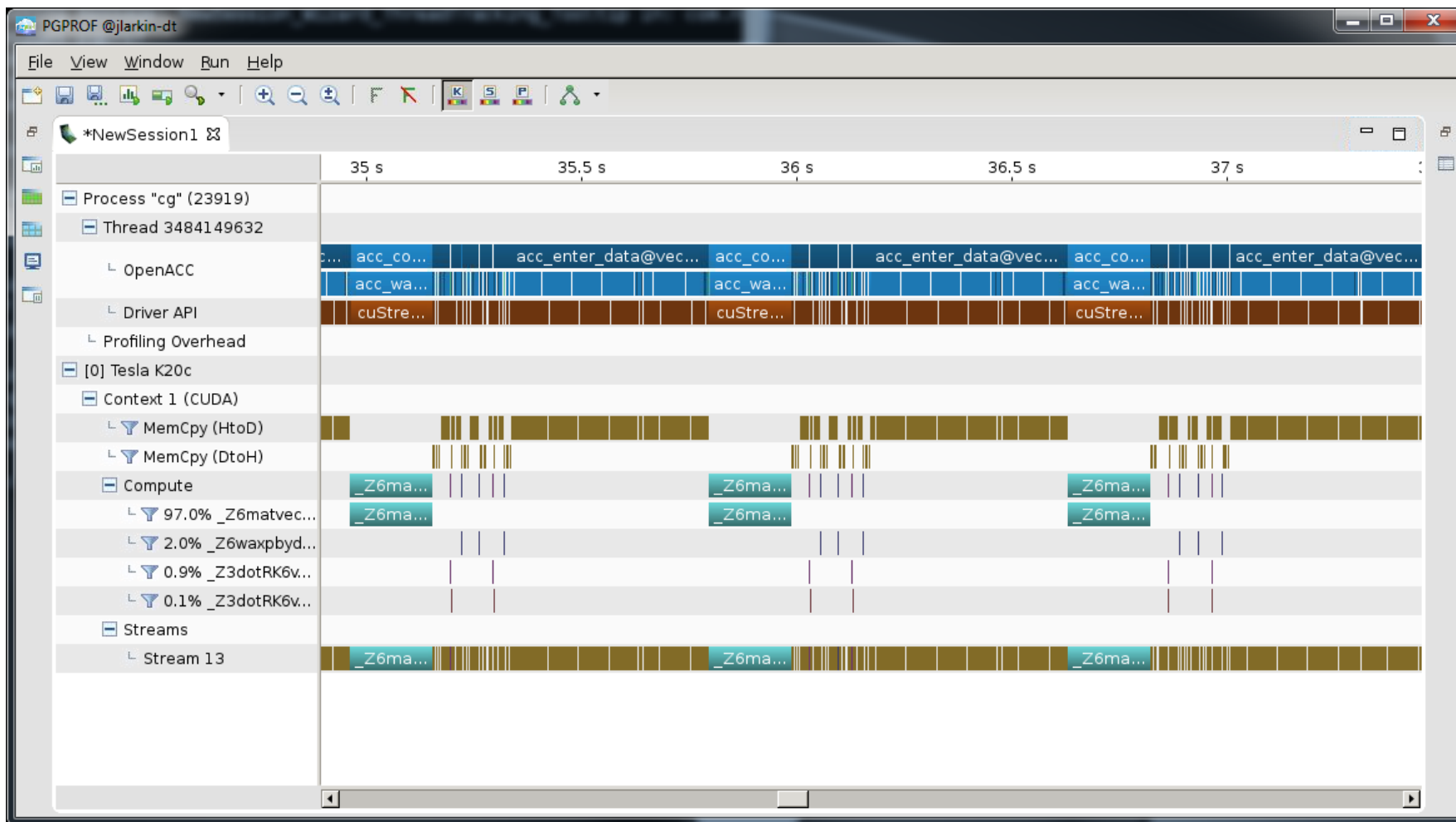
```
!$acc data copyin(a(1:end)) copyout(b(s/4:3*s/4))
```

# Matvec Data Clauses

```
#pragma acc parallel loop\  
    copyout(ycoefs[:num_rows])  
    copyin(Acoefs[:A.nnz],  
           xcoefs[:num_rows],  
           cols[:A.nnz])  
    for(int i=0;i<num_rows;i++) {  
        ...  
#pragma acc loop reduction(+:sum)  
        for(int j=row_start; j<row_end;  
j++) {  
            ...;  
        }  
        ycoefs[i]=sum;  
    }
```

- ▶ The compiler needs additional information about several arrays used in matvec.
- ▶ Compiler cannot determine the bounds of “j” loop to determine the bounds of these arrays.
- ▶ Data clauses aren’t strictly needed in dot and waxpby because the compiler can determine the array shape from the loop bounds.

# PGPROF: Data Movement



# Manage Data Higher in the Program

Currently data is moved at the beginning and end of each function, in case the data is needed on the CPU

We know that the data is only needed on the CPU after convergence

We should inform the compiler when data movement is really needed to improved performance

# 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.



# 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.

```
!$acc data  
!$acc parallel loop  
...  
  
!$acc parallel loop  
...  
!$acc end data
```

} Data Region

Arrays used within the data region will remain on the GPU until the end of the data region.

# 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)** , **finalize**

```
#pragma acc enter data copyin(a)
```

```
...
```

```
#pragma acc exit data delete(a)
```

# 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;  
};
```

# Explicit Data Movement: Copy In Matrix

```
void allocate_3d_poission_matrix(matrix &A, int N) {
    int num_rows=(N+1)*(N+1)*(N+1);
    int nnz=27*num_rows;
    A.num_rows=num_rows;
    A.row_offsets = (unsigned int*) \
        malloc((num_rows+1)*sizeof(unsigned int));
    A.cols = (unsigned int*)malloc(nnz*sizeof(unsigned int));
    A.coefs = (double*)malloc(nnz*sizeof(double));

    // Initialize Matrix

    A.row_offsets[num_rows]=nnz;
    A.nnz=nnz;
    #pragma acc enter data copyin(A)
    #pragma acc enter data \
    copyin(A.row_offsets[:num_rows+1],A.cols[:nnz],A.coefs[:nnz])
}
```

- ▶ After allocating and initializing our matrix, copy it to the device.
- ▶ Copy the structure first and its members second.

# Explicit Data Movement: Delete Matrix

```
void free_matrix(matrix &A) {  
    unsigned int *row_offsets=A.row_offsets;  
    unsigned int * cols=A.cols;  
    double * coefs=A.coefs;  
  
    #pragma acc exit data delete(A.row_offsets,A.cols,A.coefs)  
    #pragma acc exit data delete(A)  
        free(row_offsets);  
        free(cols);  
        free(coefs);  
}
```

- ▶ *Before* freeing the matrix, remove it from the device.
- ▶ Delete the members first, then the structure.
- ▶ We must do the same in vector.h.

# Running With Explicit Memory Management

- Rebuild the code without managed memory. Change `-ta=tesla:managed` to just `-ta=tesla`

## Expected:

```
Rows: 8120601, nnz: 218535025
Iteration: 0, Tolerance: 4.0067e+08
Iteration: 10, Tolerance: 1.8772e+07
Iteration: 20, Tolerance: 6.4359e+05
Iteration: 30, Tolerance: 2.3202e+04
Iteration: 40, Tolerance: 8.3565e+02
Iteration: 50, Tolerance: 3.0039e+01
Iteration: 60, Tolerance: 1.0764e+00
Iteration: 70, Tolerance: 3.8360e-02
Iteration: 80, Tolerance: 1.3515e-03
Iteration: 90, Tolerance: 4.6209e-05
Total Iterations: 100 Total Time:
8.458965s
```

## Actual:

```
Rows: 8120601, nnz: 218535025
Iteration: 0, Tolerance: 1.9497e+05
Iteration: 10, Tolerance: 1.6919e+02
Iteration: 20, Tolerance: 6.2901e+00
Iteration: 30, Tolerance: 2.0165e-01
Iteration: 40, Tolerance: 7.4122e-03
Iteration: 50, Tolerance: 2.5316e-04
Iteration: 60, Tolerance: 9.9229e-06
Iteration: 70, Tolerance: 3.4854e-07
Iteration: 80, Tolerance: 1.2859e-08
Iteration: 90, Tolerance: 5.3950e-10
Total Iterations: 100 Total Time:
8.454335s
```

# OpenACC Update Directive

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

```
do_something_on_device()
```

```
!$acc update host(a)
```



Copy “a” from GPU to CPU

```
do_something_on_host()
```

```
!$acc update device(a)
```



Copy “a” from CPU to GPU

Note: Update “host” has been deprecated and renamed “self”

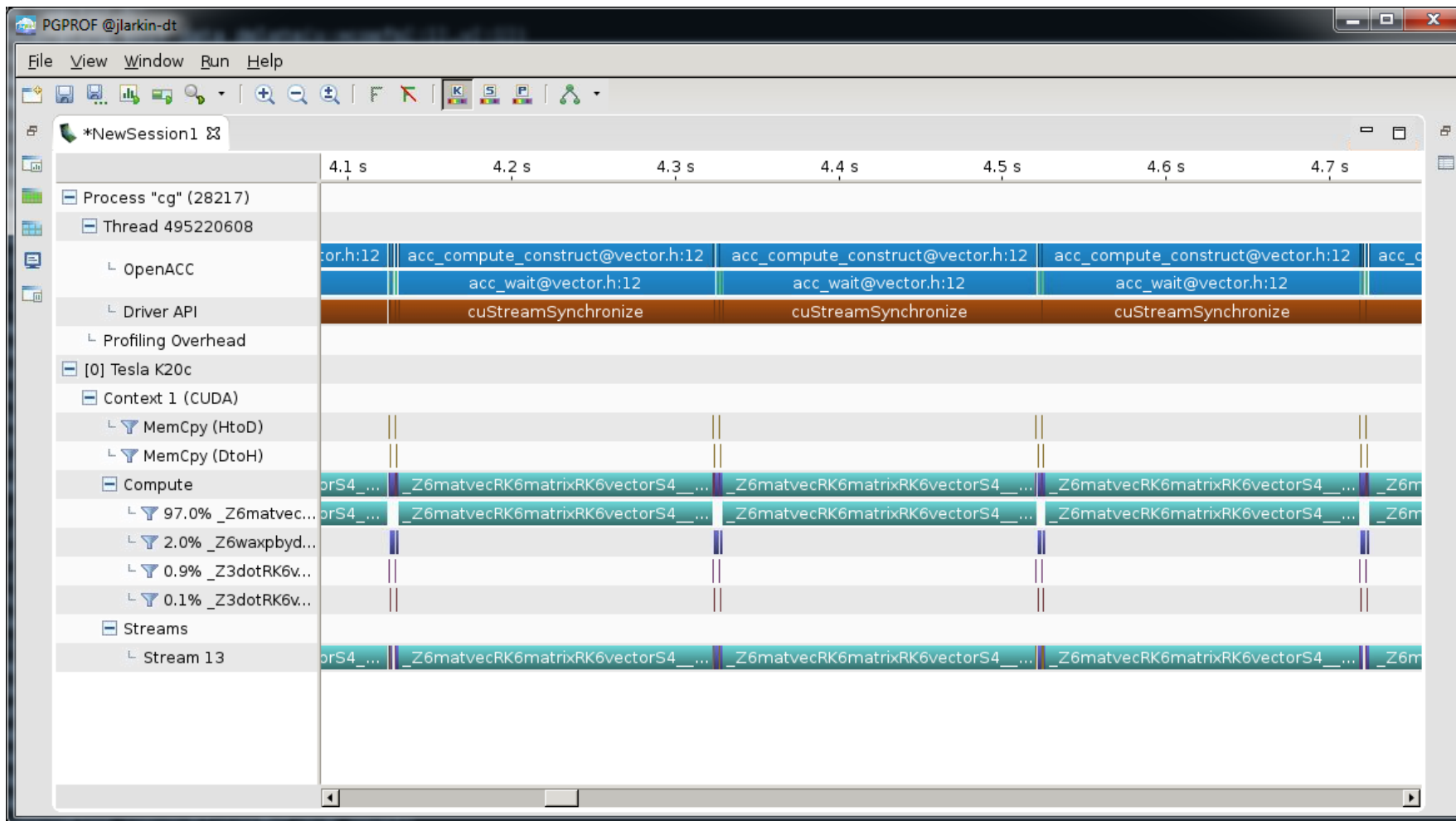


# Explicit Data Movement: Update Vector

```
void initialize_vector(vector &v, double val)
{
    for(int i=0; i<v.n; i++)
        v.coefs[i]=val;
    #pragma acc update device(v.coefs[:v.n])
}
```

- ▶ After we change vector on the CPU, we need to *update* it on the GPU.
- ▶ Update device : CPU -> GPU
- ▶ Update self/host: GPU -> CPU

# PGPROF: Data Movement Now



# 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.

# Optimizing Matvec Loops

```
matvec(const matrix &, const vector &,
const vector &):
    8, include "matrix_functions.h"
    12, Generating copyin(Acoefs[:A->nnz],
cols[:A->nnz])
    Generating implicit
copyin(row_offsets[:num_rows+1])
    Generating copyin(xcoefs[:num_rows])
    Generating copyout(ycoefs[:num_rows])
    Accelerator kernel generated
    Generating Tesla code
    16, #pragma acc loop gang /*
blockIdx.x */
    21, #pragma acc loop vector(128) /*
threadIdx.x */
    Generating reduction(+:sum)
    21, Loop is parallelizable
```

```
14 #pragma acc parallel loop \
15   copyout(ycoefs[:num_rows])
copyin(Acoefs[:A.nnz],xcoefs[:num_rows],cols[:A.n
nz])
16   for(int i=0;i<num_rows;i++) {
17       double sum=0;
18       int row_start=row_offsets[i];
19       int row_end=row_offsets[i+1];
20 #pragma acc loop reduction(+:sum)
21   for(int j=row_start;j<row_end;j++) {
22       unsigned int Acol=cols[j];
23       double Acoef=Acoefs[j];
24       double xcoef=xcoefs[Acol];
25       sum+=Acoef*xcoef;
26   }
27   ycoefs[i]=sum;
28 }
29 }
```

# Optimizing Matvec Loops

```
matvec(const matrix &, const vector &,  
const vector &):  
    8, include "matrix_functions.h"  
    12, Generating copyin(Acoefs[:A->nnz],  
cols[:A->nnz])  
        Generating implicit  
copyin(row_offsets[:num_rows+1])  
        Generating copyin(xcoefs[:num_rows])  
        Generating copyout(ycoefs[:num_rows])  
        Accelerator kernel generated  
        Generating Tesla code  
    16, #pragma acc loop gang /*  
blockIdx.x */  
    21, #pragma acc loop vector(128) /*  
threadIdx.x */  
        Generating reduction(+:sum)  
    21, Loop is parallelizable
```

```
14 #pragma acc parallel loop \  
15   copyout(ycoefs[:num_rows])  
copyin(Acoefs[:A.nnz],xcoefs[:num_rows],cols[:A.n  
nz])  
16   for(int i=0;i<num_rows;i++) {  
17     double sum=0;  
18     int row_start=row_offsets[i];  
19     int row_end=row_offsets[i+1];  
20 #pragma acc loop reduction(+:sum)  
21   for(int j=row_start;j<row_end;j++) {  
22     unsigned int Acol=cols[j];  
23     double Acoef=Acoefs[j];  
24     double xcoef=xcoefs[Acol];  
25     sum+=Acoef*xcoef;  
26   }
```

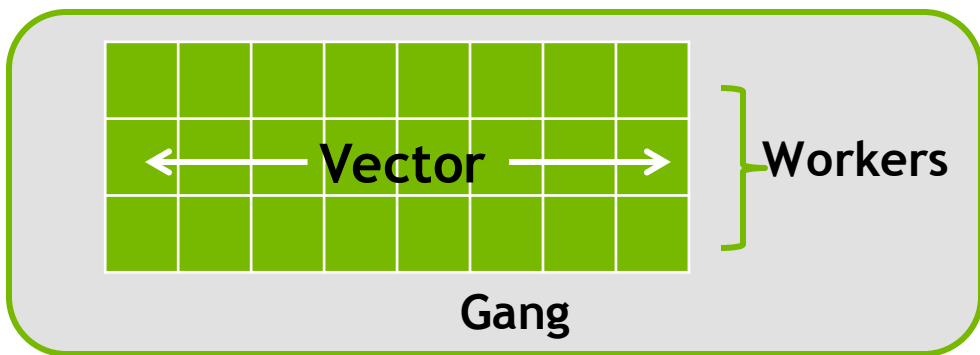
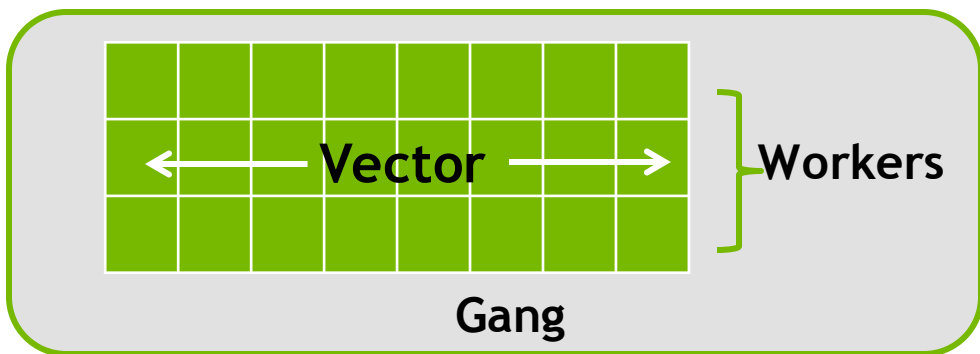
The compiler is vectorizing 128 iterations of this loop. How many iterations does it really do?

# Optimizing Matvec Loops (cont.)

- ▶ The compiler does not know how many iterations the inner loop will do, so it chooses a default value of 128.
- ▶ We can see in the initialization routine that it will only iterate 27 times (number of non-zeros per row).
- ▶ Reducing the vector length should improve hardware utilization

```
14 void allocate_3d_poisson_matrix(matrix
&A, int N) {
15     int num_rows=(N+1)*(N+1)*(N+1);
16     int nnz=27*num_rows;
17     A.num_rows=num_rows;
18     A.row_offsets=(unsigned
int*)malloc((num_rows+1)*sizeof(unsigned
int));
19     A.cols=(unsigned
int*)malloc(nnz*sizeof(unsigned int));
20     A.coefs=(double*)malloc(nnz*sizeof(double)
);
```

# 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



# 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)
        ...
```

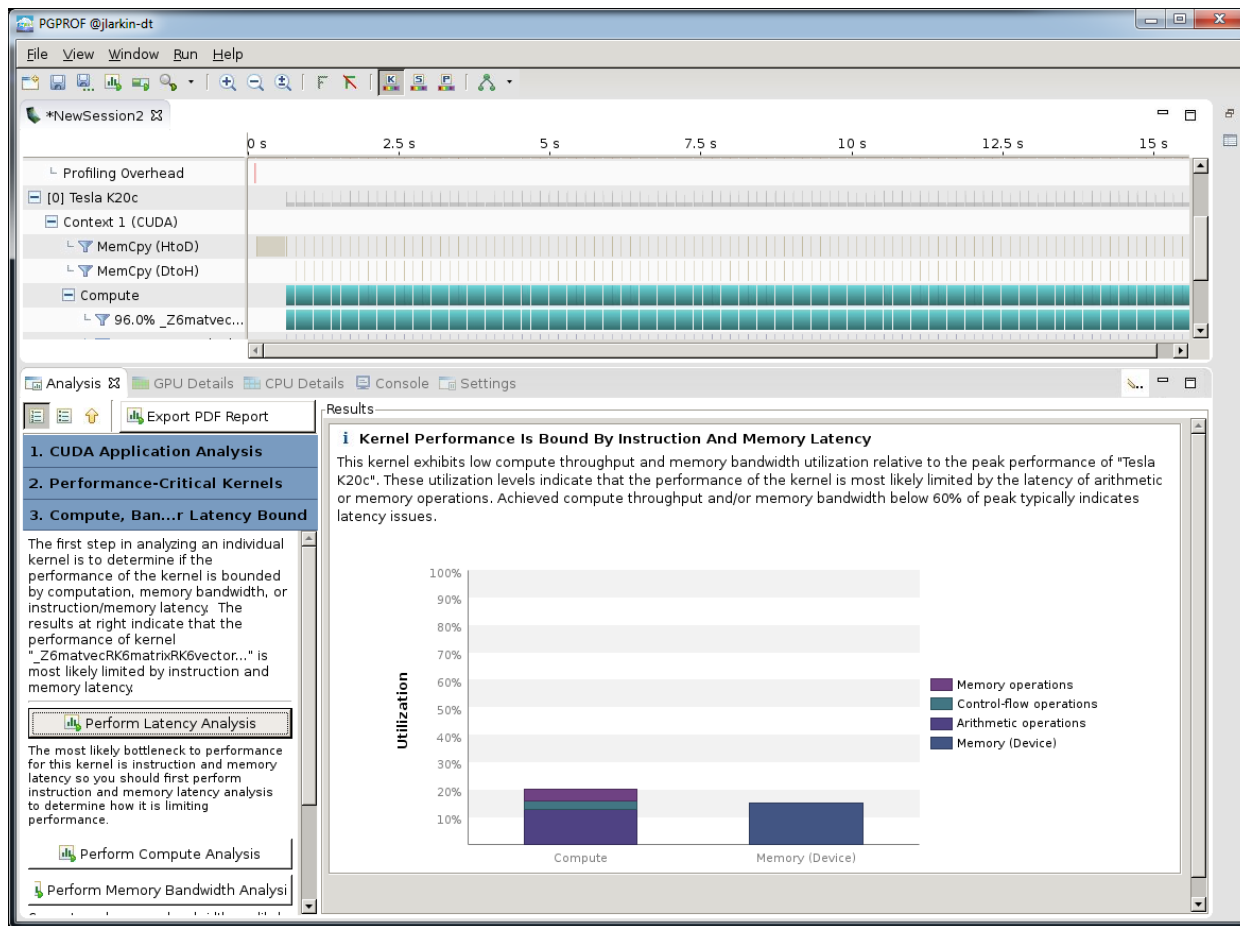
```
#pragma acc parallel vector_length(32)
#pragma acc loop gang worker
for (int i = 0; i < n; ++i)
    #pragma acc loop vector
    for (int j = 0; j < n; ++j)
        ...
```

# Optimizing Matvec Loops: Vector Length

- ▶ Use the OpenACC loop directive to force the compiler to vectorize the inner loop.
- ▶ Use `vector_length` to reduce the vector length closer to actual loop iterations
- ▶ Note: NVIDIA GPUs need vector lengths that are multiples of 32 (warp size)

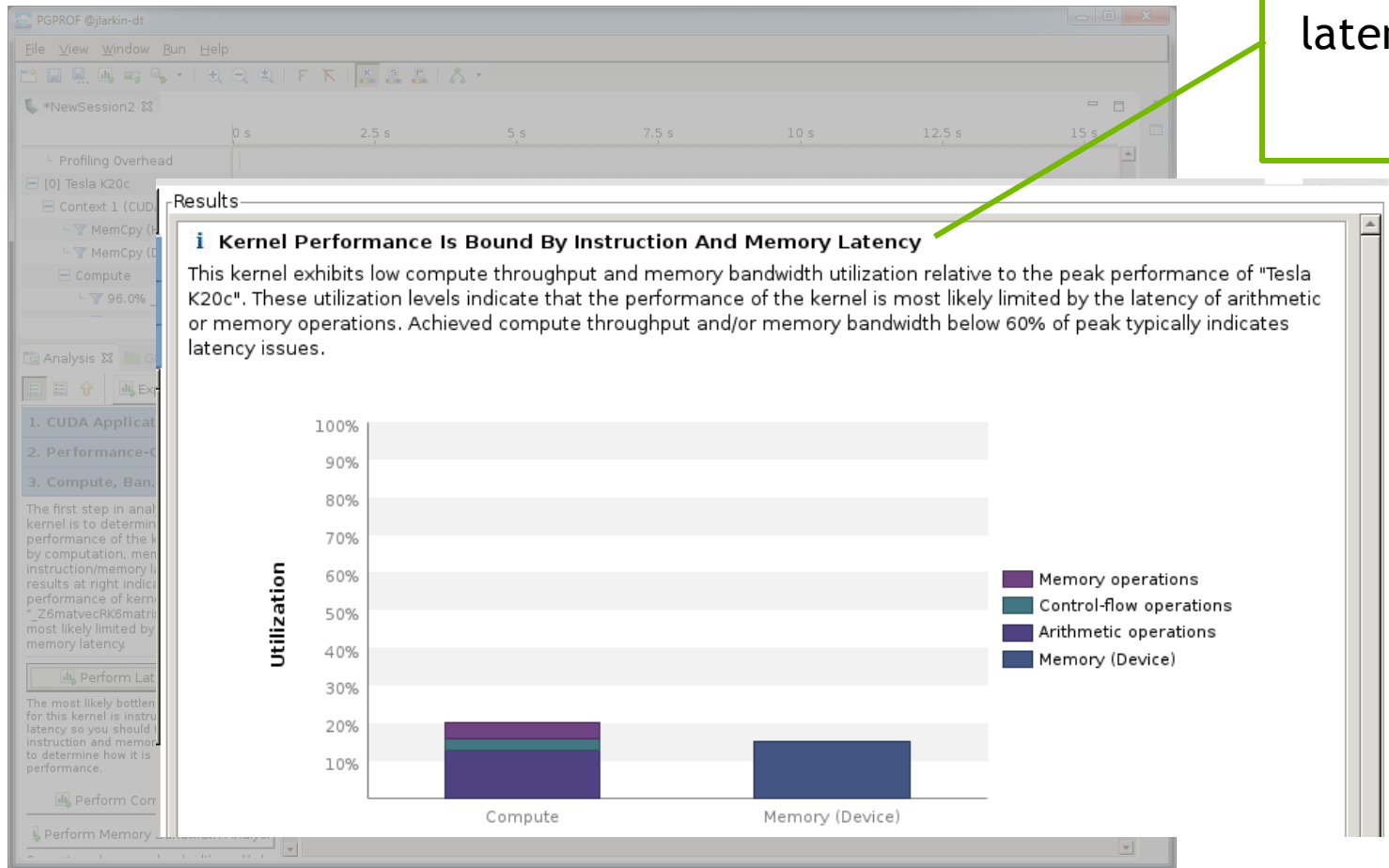
```
14 #pragma acc parallel loop
    vector_length(32) \
15     copyout(ycoefs[:num_rows])
    copyin(Acoefs[:A.nnz],xcoefs[:num_rows],cols
           [:A.nnz])
16     for(int i=0;i<num_rows;i++) {
17         double sum=0;
18         int row_start=row_offsets[i];
19         int row_end=row_offsets[i+1];
20 #pragma acc loop vector reduction(+:sum)
21         for(int j=row_start;j<row_end;j++) {
22             unsigned int Acol=cols[j];
23             double Acoef=Acoefs[j];
24             double xcoef=xcoefs[Acol];
25             sum+=Acoef*xcoef;
26         }
27         ycoefs[i]=sum;
28     }
```

# Matvec Performance Limiter



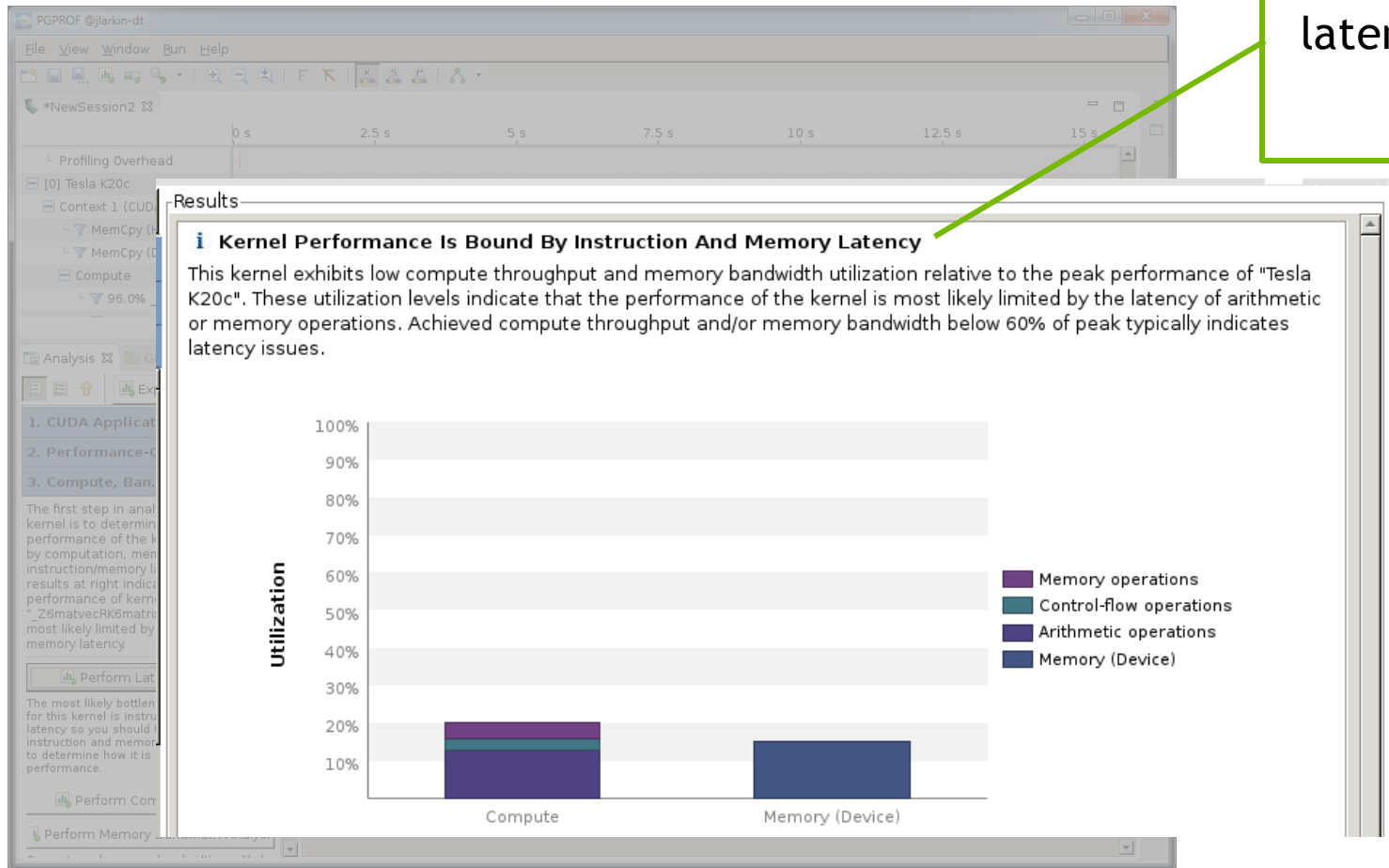
# Matvec Performance Limiter

Instruction and Memory latency are limiting kernel performance.



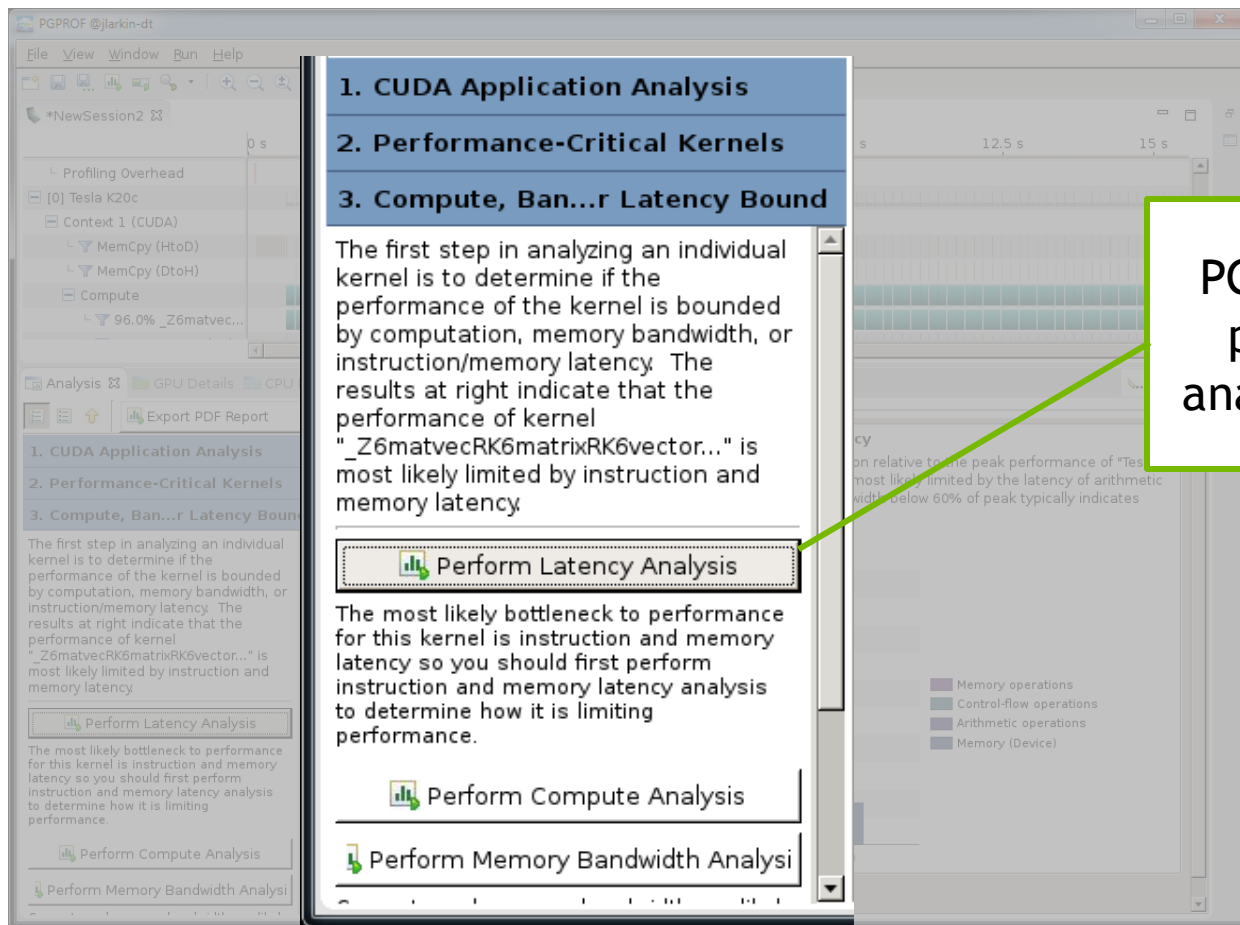
# Matvec Performance Limiter

Instruction and Memory latency are limiting kernel performance.



Recall: GPU's tolerate latency by having *enough* parallelism

# Matvec Performance Limiter



The screenshot displays the PGPROF application window. On the left, a sidebar shows a tree view of the profiling session, including 'Profiling Overhead', '[0] Tesla K20c', 'Context 1 (CUDA)', 'MemCpy (HtoD)', 'MemCpy (DtoH)', 'Compute', and a sub-entry for '96.0% \_Z6matvec...'. The main content area is divided into three sections: '1. CUDA Application Analysis', '2. Performance-Critical Kernels', and '3. Compute, Ban...r Latency Bound'. Section 3 contains a detailed analysis of the kernel '\_Z6matvecRK6matrixRK6vector...' and a list of analysis options: 'Perform Latency Analysis', 'Perform Compute Analysis', and 'Perform Memory Bandwidth Analysis'. A green box highlights the 'Perform Latency Analysis' option, with a green arrow pointing to a callout box on the right.

**1. CUDA Application Analysis**

**2. Performance-Critical Kernels**

**3. Compute, Ban...r Latency Bound**

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "\_Z6matvecRK6matrixRK6vector..." is most likely limited by instruction and memory latency.

**Perform Latency Analysis**

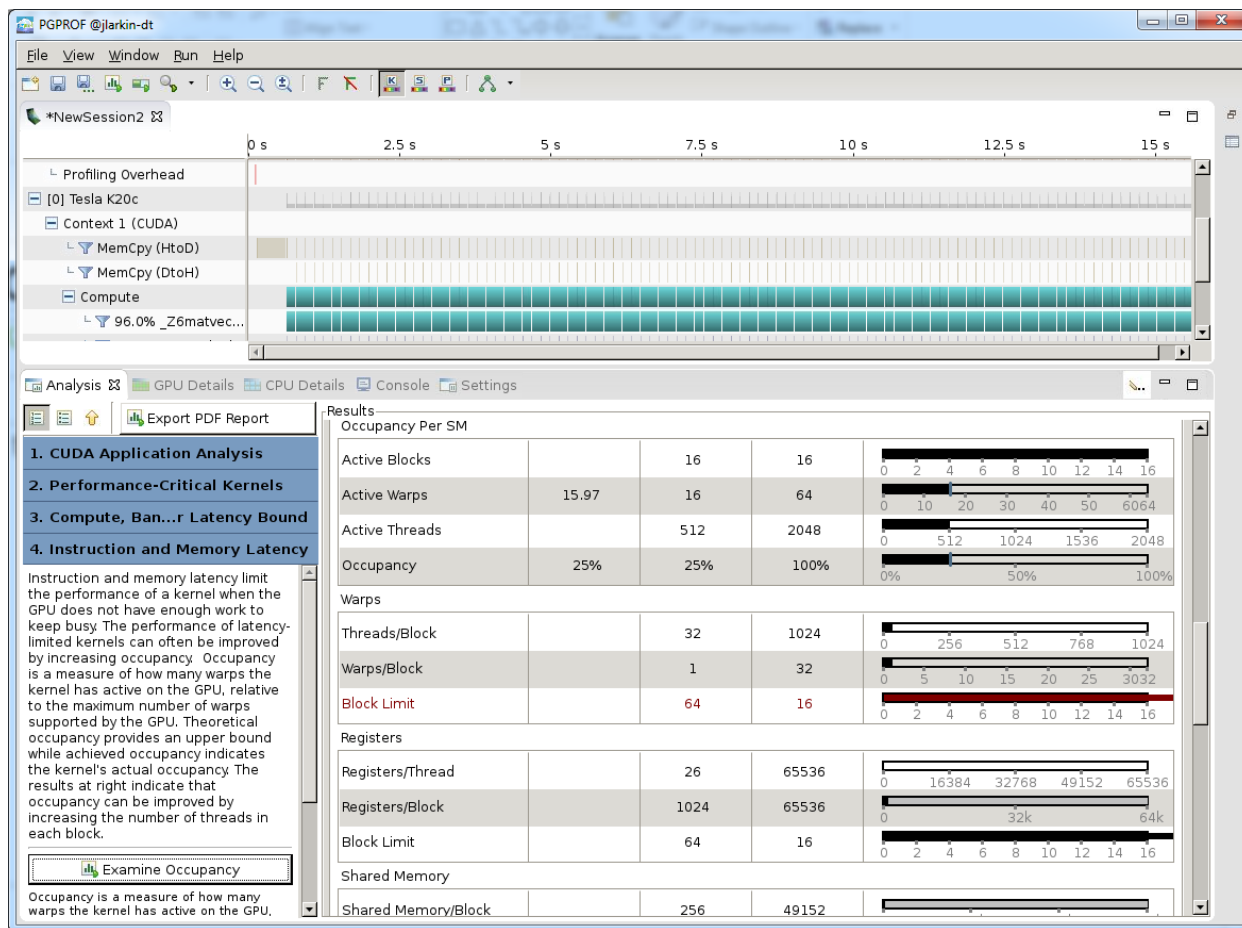
The most likely bottleneck to performance for this kernel is instruction and memory latency so you should first perform instruction and memory latency analysis to determine how it is limiting performance.

**Perform Compute Analysis**

**Perform Memory Bandwidth Analysis**

PGPROF will guide you to perform a latency analysis to understand why

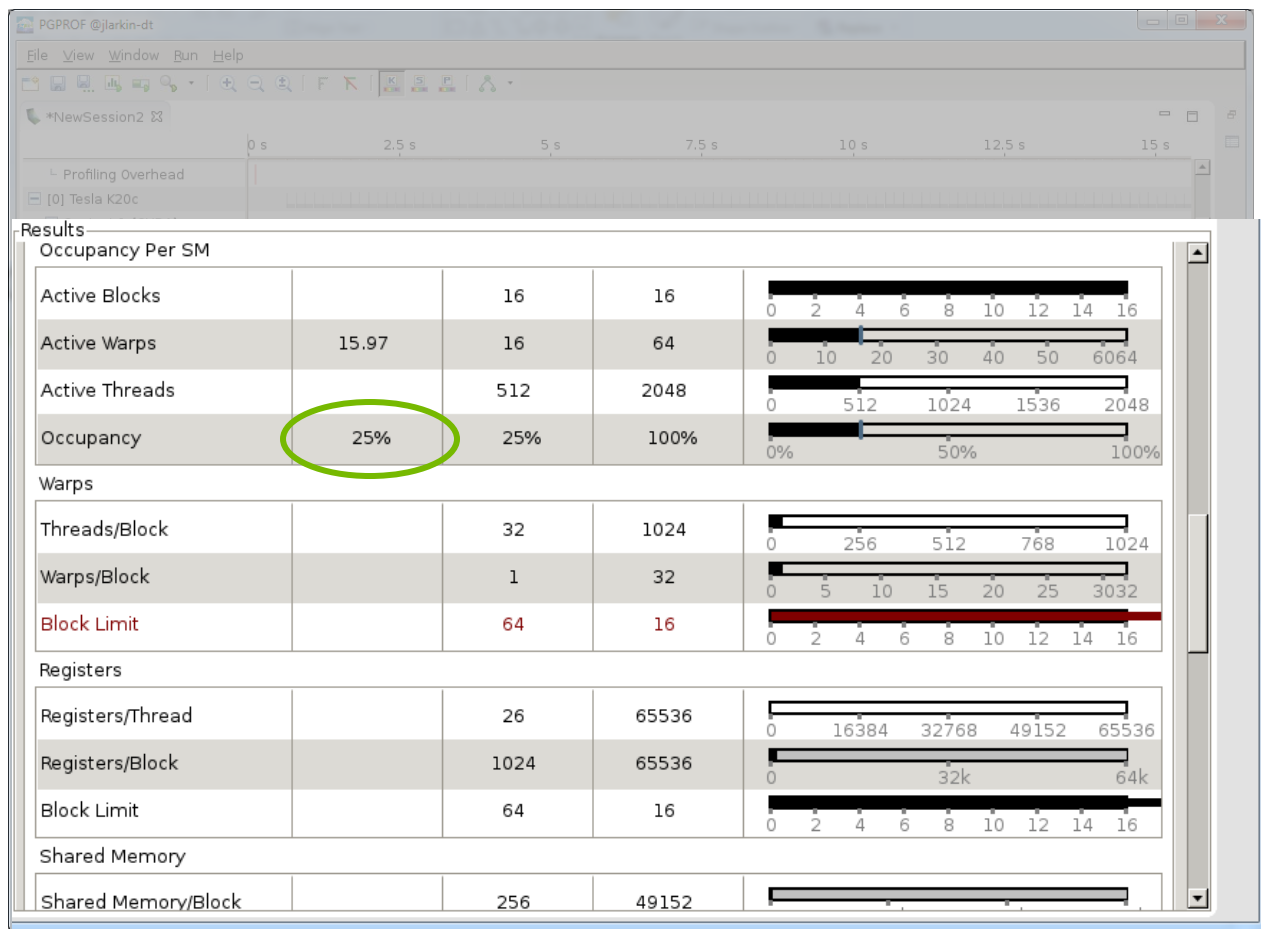
# Matvec Occupancy



Occupancy is a measure of how well the GPU is being utilized.

100% occupancy means the GPU is running as many simultaneous threads as it can.

# Matvec Occupancy



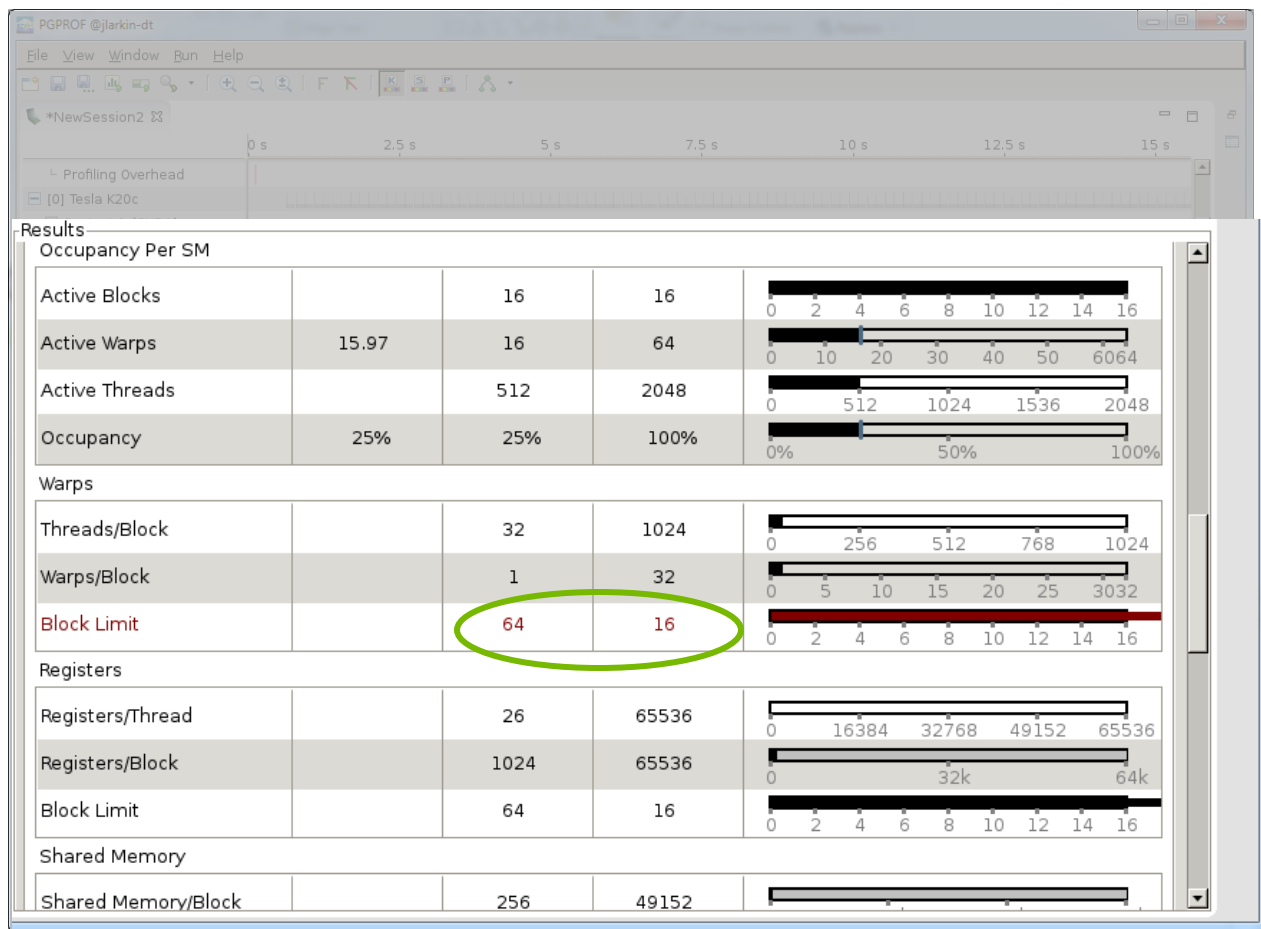
Occupancy is a measure of how well the GPU is being utilized.

100% occupancy means the GPU is running as many simultaneous threads as it can.

We're only keeping the GPU 25% occupied, why?



# Matvec Occupancy



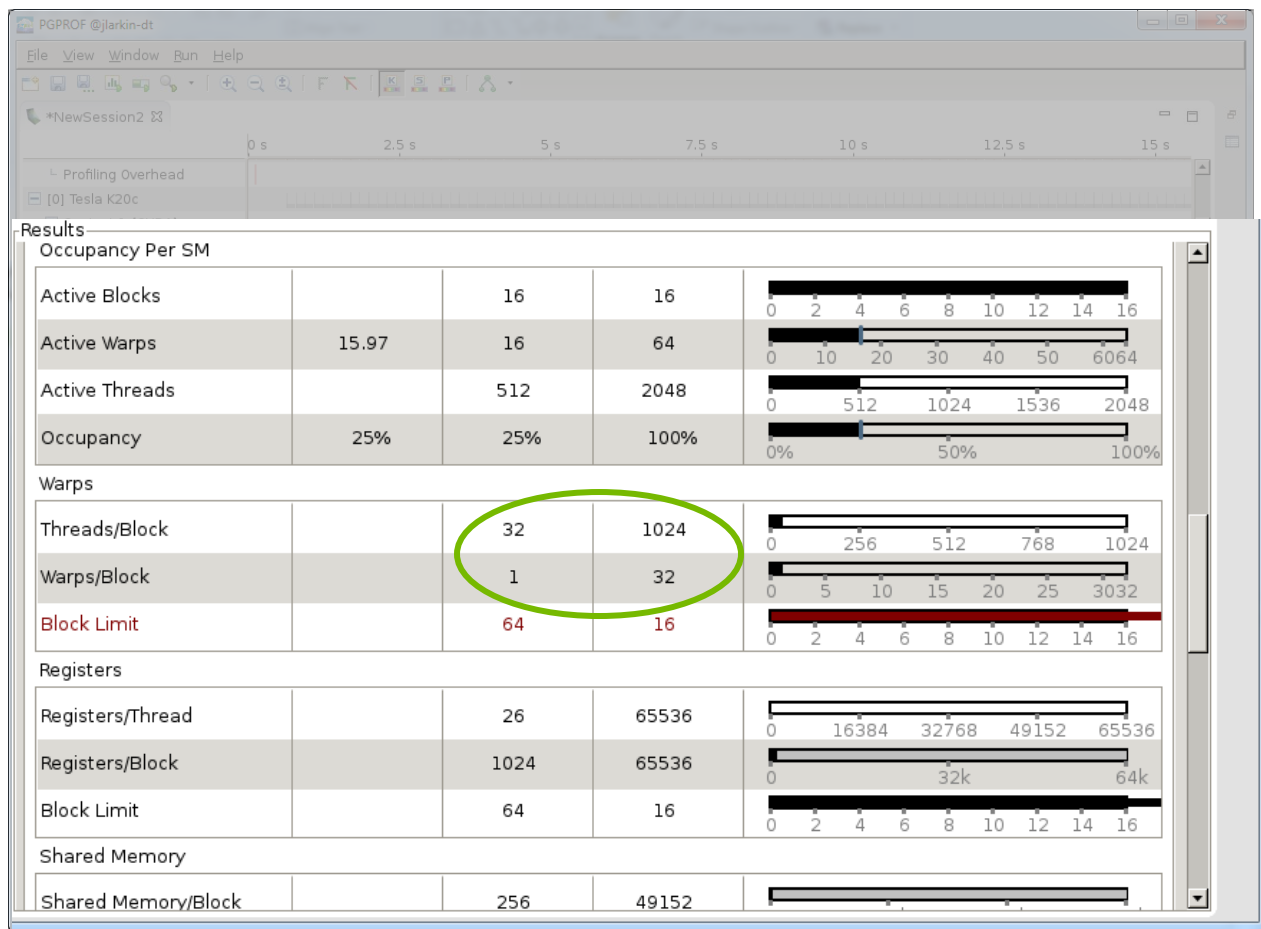
Occupancy is a measure of how well the GPU is being utilized.

100% occupancy means the GPU is running as many simultaneous threads as it can.

We're only keeping the GPU 25% occupied, why?

We need 64 threadblocks to get 100% occupancy, but the hardware can only manage 16. Why?

# Matvec Occupancy



Occupancy is a measure of how well the GPU is being utilized.

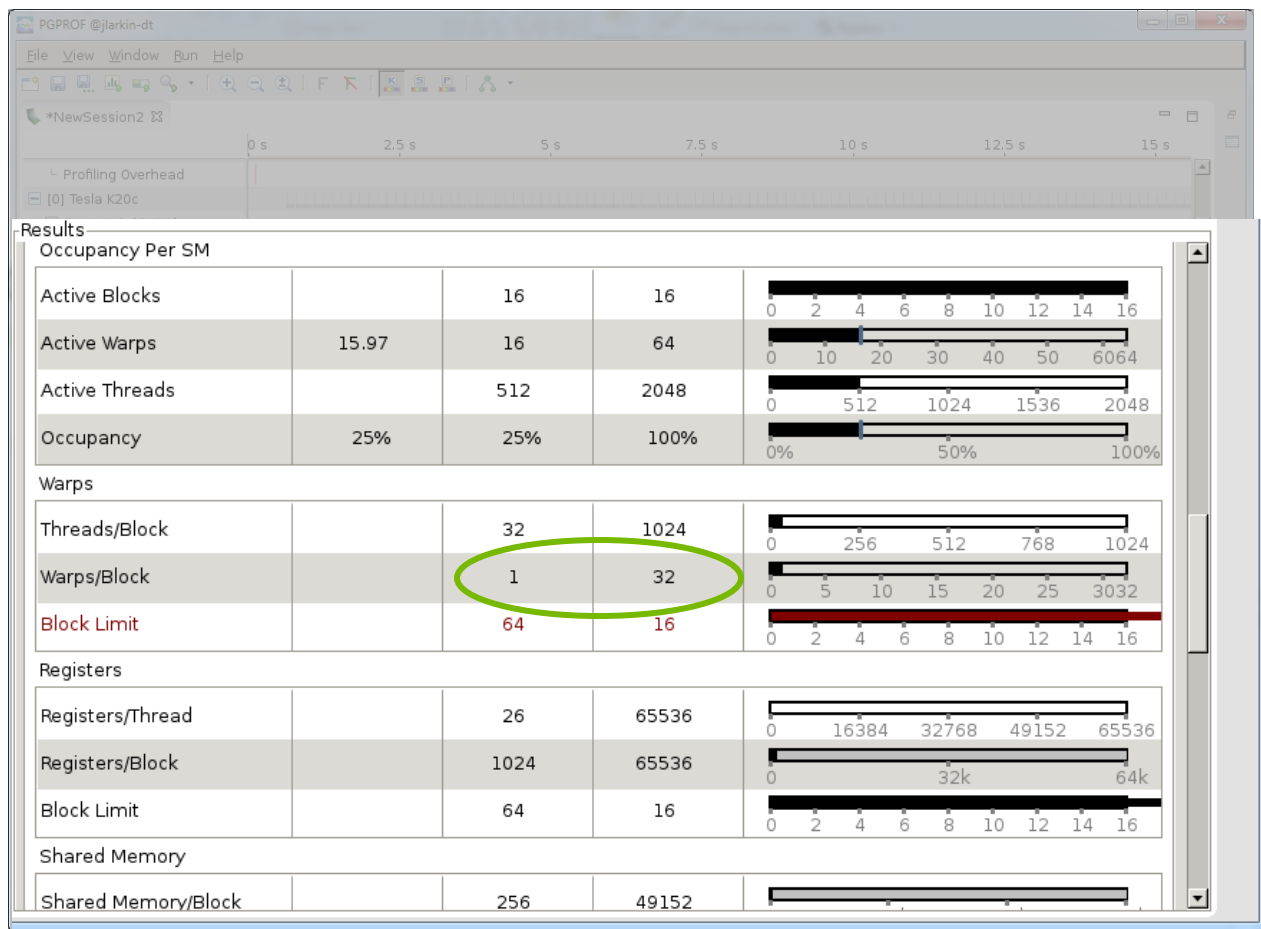
100% occupancy means the GPU is running as many simultaneous threads as it can.

We're only keeping the GPU 25% occupied, why?

We need 64 threadblocks to get 100% occupancy, but the hardware can only manage 16. Why?

We've reduced our vector length so that each block has only 1 warp

# Matvec Occupancy



Occupancy is a measure of how well the GPU is being utilized.

100% occupancy means the GPU is running as many simultaneous threads as it can.

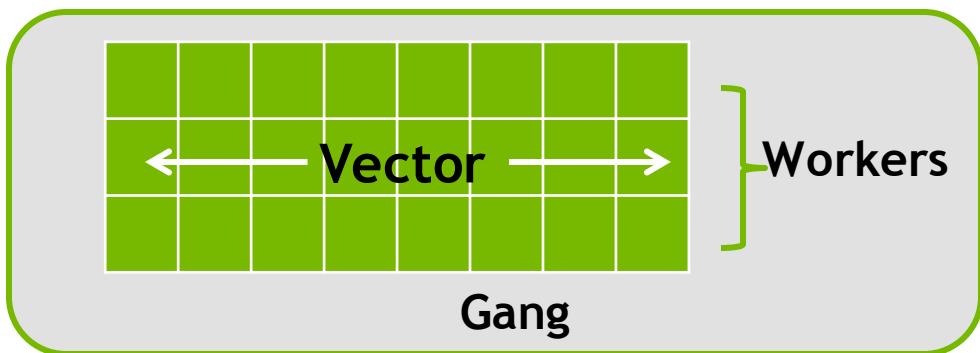
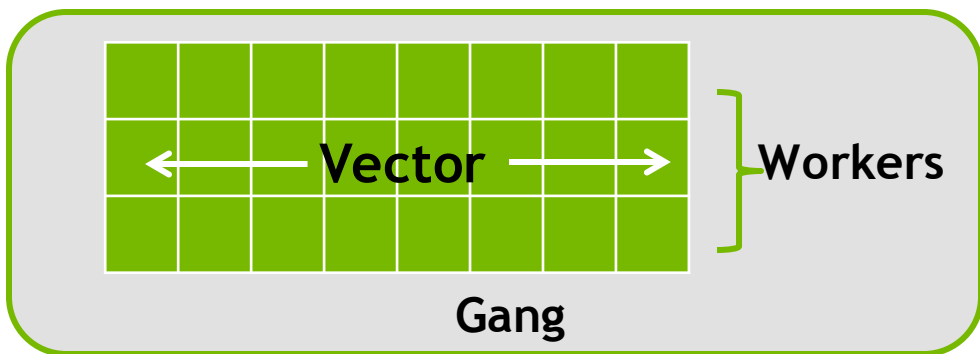
We're only keeping the GPU 25% occupied, why?

We need 64 threadblocks to get 100% occupancy, but the hardware can only manage 16. Why?

We've reduced our vector length so that each block has only 1 warp

So we need at least 4X more parallelism per gang

# Increasing per-gang parallelism



- The inner loop lacks sufficient parallelism to occupy the GPU
- We can increase the size of the gang by increasing the number of workers

# Optimizing Matvec Loops: Increase Workers

- ▶ By splitting the iterations of the outer loop among workers and gangs, we'll increase the size of the gangs.
- ▶ The compiler will handle breaking up the outer loop so you don't have to.
- ▶ Now we should have  $4 \times 32 = 128$  threads in each GPU threadblock

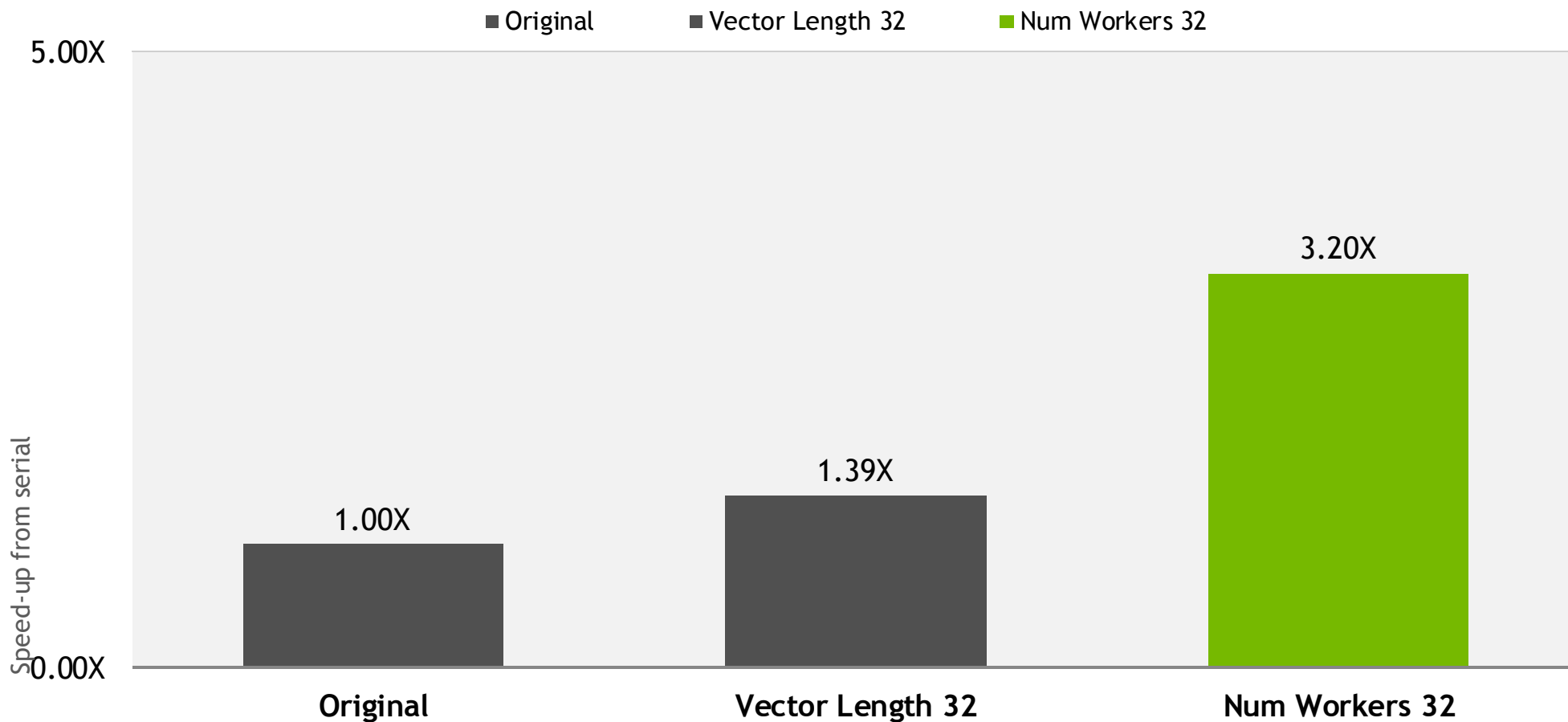
```
14 #pragma acc parallel loop gang worker \
    num_workers(4) vector_length(32) \
15     copyout(ycoefs[:num_rows])
copyin(Acoefs[:A.nnz],xcoefs[:num_rows],cols
[:A.nnz])
16     for(int i=0;i<num_rows;i++) {
17         double sum=0;
18         int row_start=row_offsets[i];
19         int row_end=row_offsets[i+1];
20 #pragma acc loop vector reduction(+:sum)
21         for(int j=row_start;j<row_end;j++) {
22             unsigned int Acol=cols[j];
23             double Acoef=Acoefs[j];
24             double xcoef=xcoefs[Acol];
25             sum+=Acoef*xcoef;
26         }
27         ycoefs[i]=sum;
28     }
```

# Increase Workers: Compiler feedback

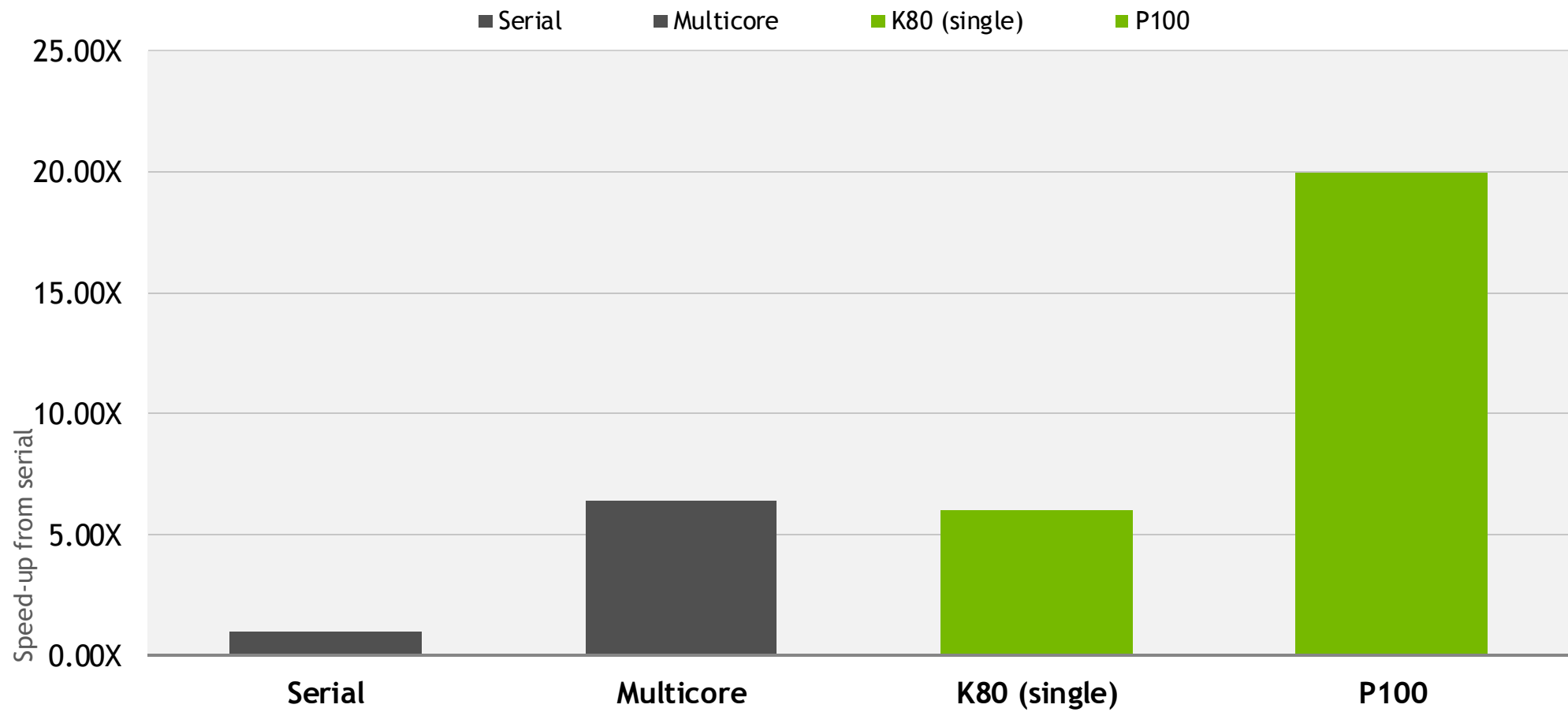
- ▶ The compiler will tell you that it has honored your loop directives.
- ▶ If you're familiar with CUDA, it'll also tell you how the loops are mapped the CUDA thread blocks

```
matvec(const matrix &, const vector &, const
vector &):
    8, include "matrix_functions.h"
    12, Generating copyin(Acoefs[:A-
>nnz],cols[:A->nnz])
    Generating implicit
copyin(row_offsets[:num_rows+1])
    Generating copyin(xcoefs[:num_rows])
    Generating copyout(ycoefs[:num_rows])
    Accelerator kernel generated
    Generating Tesla code
    5, Vector barrier inserted due to
potential dependence into a vector loop
    16, #pragma acc loop gang, worker(4)
/* blockIdx.x threadIdx.y */
    21, #pragma acc loop vector(32) /*
threadIdx.x */
    Generating reduction(+:sum)
    Vector barrier inserted due to
potential dependence out of a vector loop
    21, Loop is parallelizable
```

# Tuning Matvec



# Final Performance



Source: PGI 16.9, Multicore: Intel Xeon CPU E5-2698 v3 @ 2.30GHz



# The device\_type clause

- ▶ Use `device_type` to specialize optimizations to specific hardware.
- ▶ The compiler will choose values for all other targets.

```
#pragma acc parallel loop \  
    device_type(nvidia) vector_length(256)  
    device_type(radeon) vector_length(512)  
for(int i = 0; i < n ; i++)  
{  
    ...;  
}
```

# Common optimizations

# The collapse Clause

**collapse(*n*):** Applies the associated directive to the following *n* tightly nested loops.

```
#pragma acc parallel loop \  
    collapse(2)  
for(int i=0; i<N; i++)  
    for(int j=0; j<M; j++)  
        ...
```



```
#pragma acc parallel loop  
for(int ij=0; ij<N*N; ij++)  
    ...
```

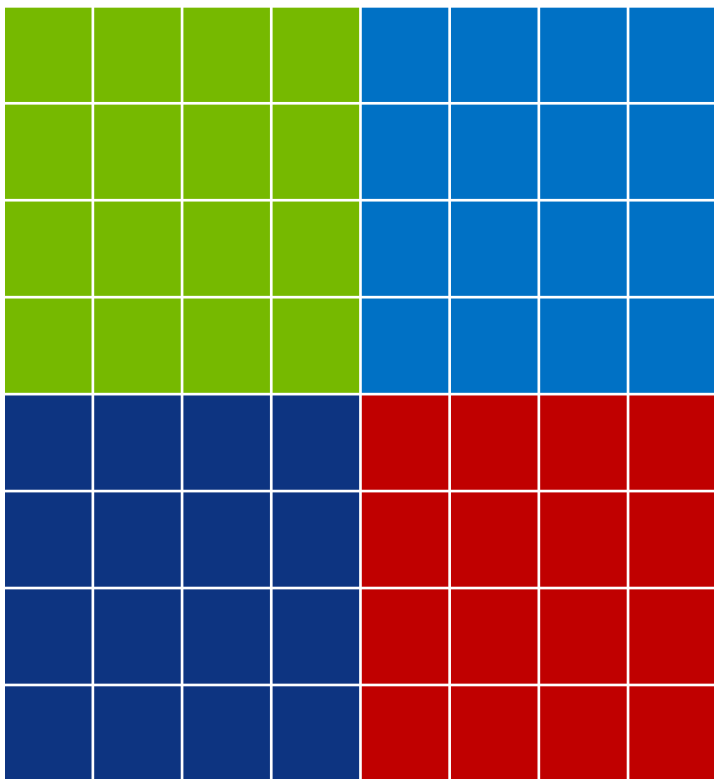
Collapse outer loops to enable creating more gangs.

Collapse inner loops to enable longer vector lengths.

Collapse all loops, when possible, to do both.

# 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]);
    }
}
```

# Stride-1 Memory Accesses

```
for(i=0; i<N; i++)  
  for(j=0; j<M; j++)  
  {  
    A[i][j][1] = 1.0f;  
    A[i][j][2] = 0.0f;  
  }  
}
```

The fastest dimension is length 2  
and fastest loop strides by 2.

```
for(i=0; i<N; i++)  
  for(j=0; j<M; j++)  
  {  
    A[1][i][j] = 1.0f;  
    A[2][i][j] = 0.0f;  
  }  
}
```

Now the inner loop is the fastest  
dimension through memory.

# Stride-1 Memory Accesses

```
for(i=0; i<N; i++)  
  for(j=0; j<M; j++)  
  {  
    A[i][j].a= 1.0f;  
    A[i][j].b = 0.0f;  
  }  
}
```

If all threads access the “a” element, they will be accessing every-other memory element.

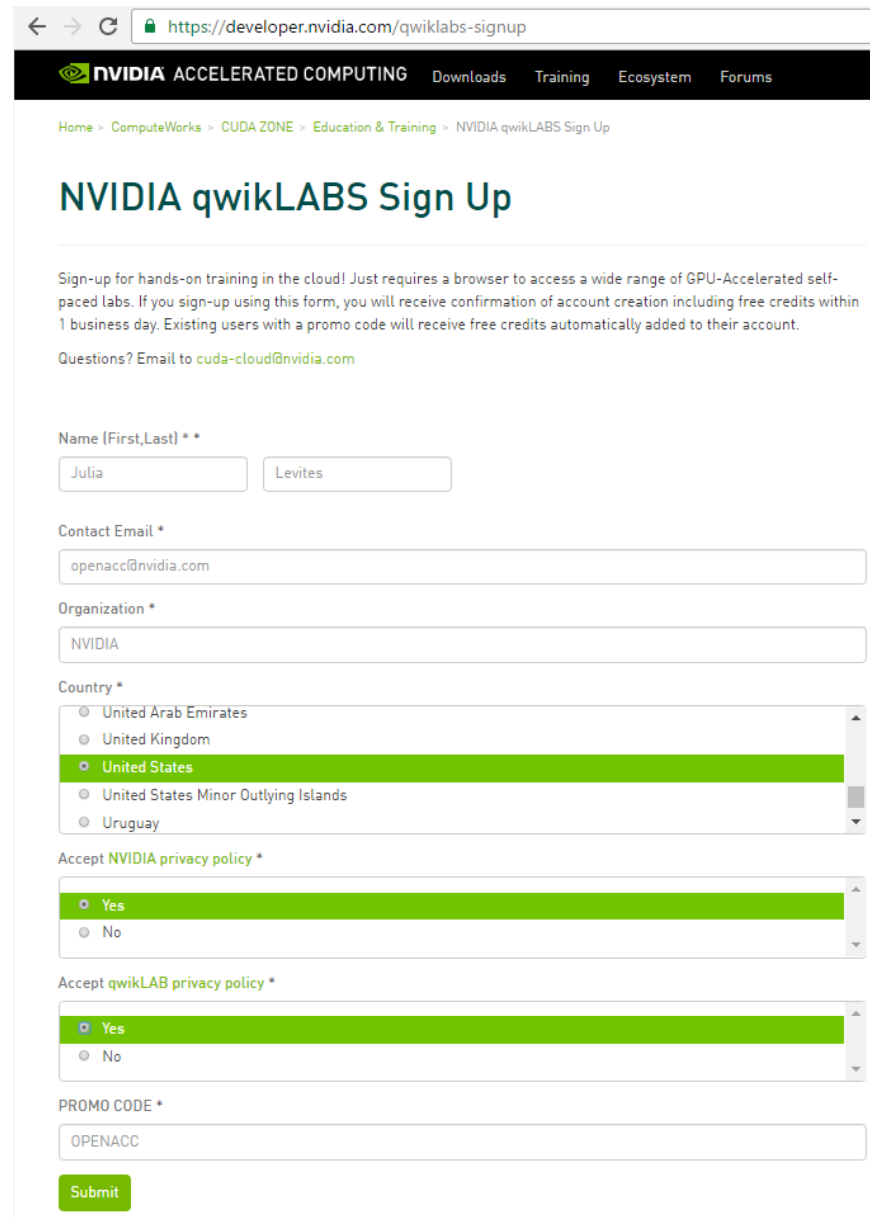
```
for(i=0; i<N; i++)  
  for(j=0; j<M; j++)  
  {  
    Aa[i][j] = 1.0f;  
    Ab[i][j] = 0.0f;  
  }  
}
```

Now all threads are accessing contiguous elements of Aa and Ab.

# Using QwikLabs

# Getting access

1. Create an account with NVIDIA qwikLABS <https://developer.nvidia.com/qwiklabs-signup>
2. Enter a promo code OPENACC16 before submitting the form
3. Free credits will be added to your account
4. Start using OpenACC!



The screenshot shows the NVIDIA qwikLABS Sign Up page. The browser address bar displays <https://developer.nvidia.com/qwiklabs-signup>. The page header includes the NVIDIA logo and navigation links: ACCELERATED COMPUTING, Downloads, Training, Ecosystem, and Forums. A breadcrumb trail reads: Home > ComputeWorks > CUDA ZONE > Education & Training > NVIDIA qwikLABS Sign Up.

## NVIDIA qwikLABS Sign Up

Sign-up for hands-on training in the cloud! Just requires a browser to access a wide range of GPU-Accelerated self-paced labs. If you sign-up using this form, you will receive confirmation of account creation including free credits within 1 business day. Existing users with a promo code will receive free credits automatically added to their account.

Questions? Email to [cuda-cloud@nvidia.com](mailto:cuda-cloud@nvidia.com)

Name (First,Last) \* \*

Julia Levites

Contact Email \*

openacc@nvidia.com

Organization \*

NVIDIA

Country \*

- United Arab Emirates
- United Kingdom
- United States**
- United States Minor Outlying Islands
- Uruguay

Accept NVIDIA privacy policy \*

**Yes**

No

Accept qwikLAB privacy policy \*

**Yes**

No

PROMO CODE \*

OPENACC

Submit



# This week's labs

The screenshot displays the Qwiklabs website interface. At the top, a blue navigation bar contains the Qwiklabs logo, links for 'WHAT'S A QWIKLAB?', 'LAB CATALOG', 'PRICING', 'FAQS', and 'CONTACT', a 'Language' dropdown, and buttons for 'Create New Account' and 'Sign in'. Below the navigation bar is a search bar labeled 'Search for Labs...'. The main content area is divided into two sections. On the left, a 'Quest (1)' section titled 'OpenACC Technologies' lists a series of labs. The first two labs are introductory and free, while the last three are advanced or expert level and cost credits. On the right, a detailed view of the 'OpenACC' lab is shown, including a description, a table of lab credits and hours, and a badge earned.

Title	Level	Cost
OpenACC - 2X in 4 Steps in C/C++	Introductory	FREE
OpenACC - 2X in 4 Steps in Fortran	Introductory	FREE
Profiling and Parallelizing with OpenACC	Advanced	10 Credits
Expressing Data Movement and Optimizing Loops with OpenACC	Expert	15 Credits
Pipelining Work on the GPU with OpenACC	Expert	15 Credits

**OpenACC**

OpenACC is an accelerator programming standard that enables scientific and technical Fortran, C and (increasingly) C++ programmers to easily take advantage of the power of heterogeneous CPU/accelerator systems. OpenACC allows programmers to use simple compiler directives to identify which areas of code to accelerate, without requiring modification to the underlying code itself. By identifying parallel code segments, OpenACC directives allow the compiler to do the detailed

Total Lab Credits:	70
Total Hours:	08 h:33 m

Badge Earned:

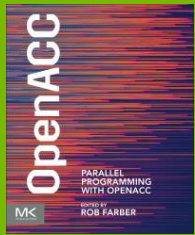
OpenACC

This week you should complete the “Profiling and Parallelizing with OpenACC” and “Expressing Data Movement and Optimizing Loops with OpenACC” labs in qwiklabs.

# CERTIFICATION

Available after November 9th

1. Attend live lectures
2. Complete the test
3. Enter for a chance to win a Titan X or an OpenACC Book



Official rules:

<http://developer.download.nvidia.com/compute/OpenACC-Toolkit/docs/TITANX-GIVEAWAY-OPENACC-Official-Rules-2016.pdf>

# OPENACC TOOLKIT

Free for Academia

Download link:

<https://developer.nvidia.com/openacc-toolkit>

---

## NEW OPENACC BOOK

Parallel Programming with OpenACC

Available starting Nov 1<sup>st</sup>, 2016:

<http://store.elsevier.com/Parallel-Programming-with-OpenACC/Rob-Farber/isbn-9780124103979/>

# 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>
- OpenACC Toolkit - <http://developer.nvidia.com/openacc-toolkit>
- 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](mailto:openacc@nvidia.com)

# Course Syllabus

Oct 26: Analyzing and Parallelizing with OpenACC

Nov 2: OpenACC Optimizations

Nov 9: Advanced OpenACC

Recordings:

<https://developer.nvidia.com/intro-to-openacc-course-2016>

Questions? Email [openacc@nvidia.com](mailto:openacc@nvidia.com)