INTRODUCTION TO OPENACC

Lecture 2: OpenACC Optimizations, November 2, 2016

NVIDIA

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

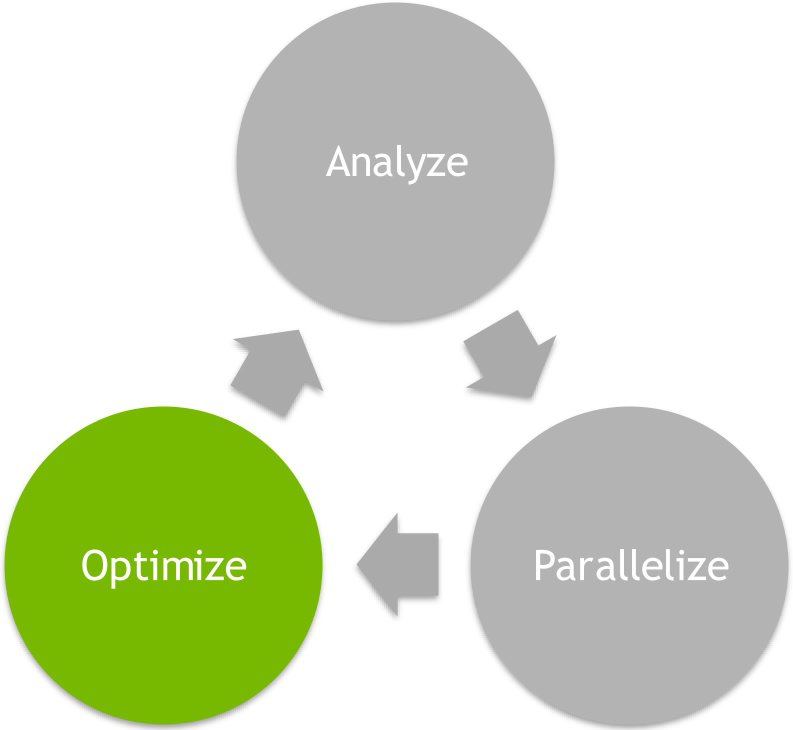
Today's Objectives

Understand OpenACC data directives

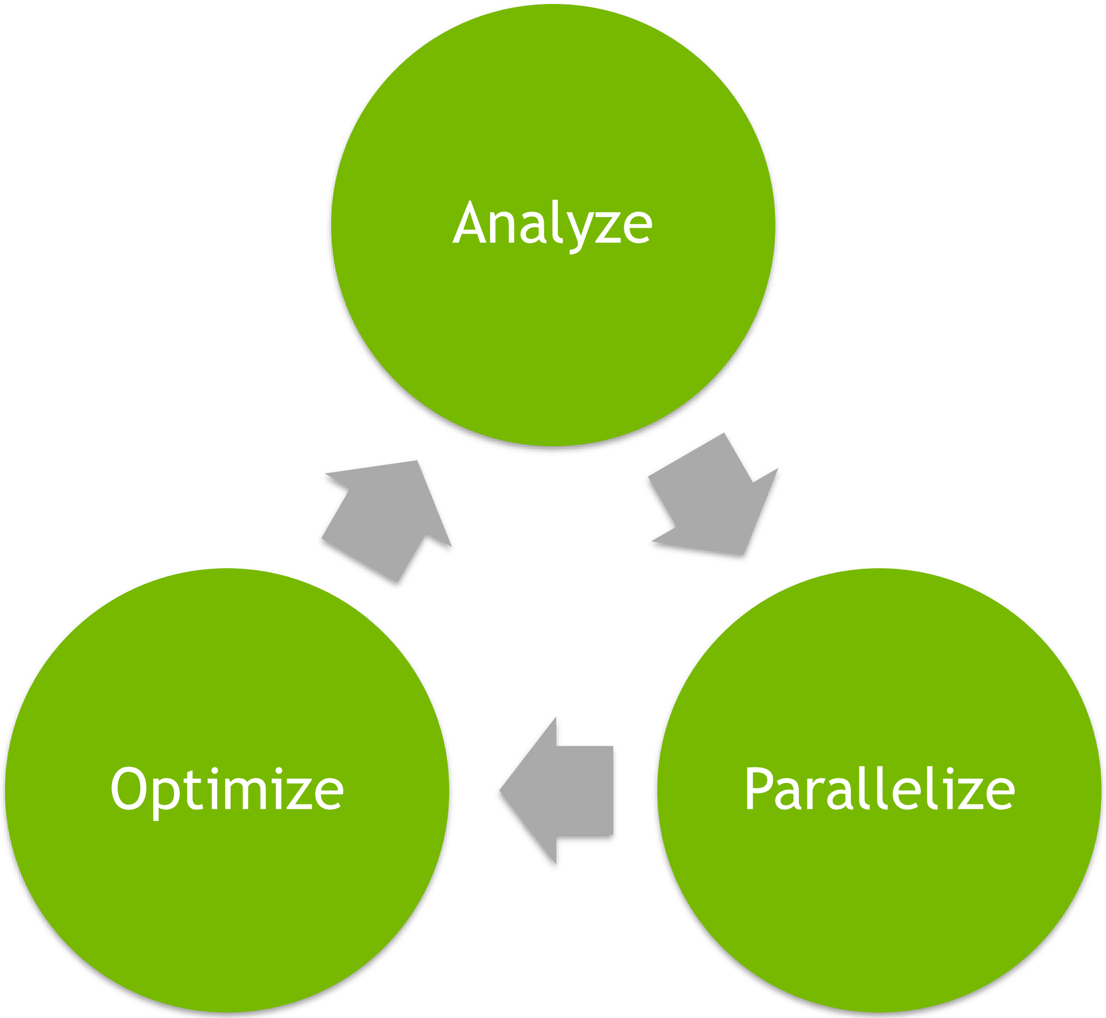
Understand the 3 levels of OpenACC parallelism

Understand how to optimize loop decomposition

Understand other common optimizations to OpenACC codes



3 Steps to Accelerate with OpenACC



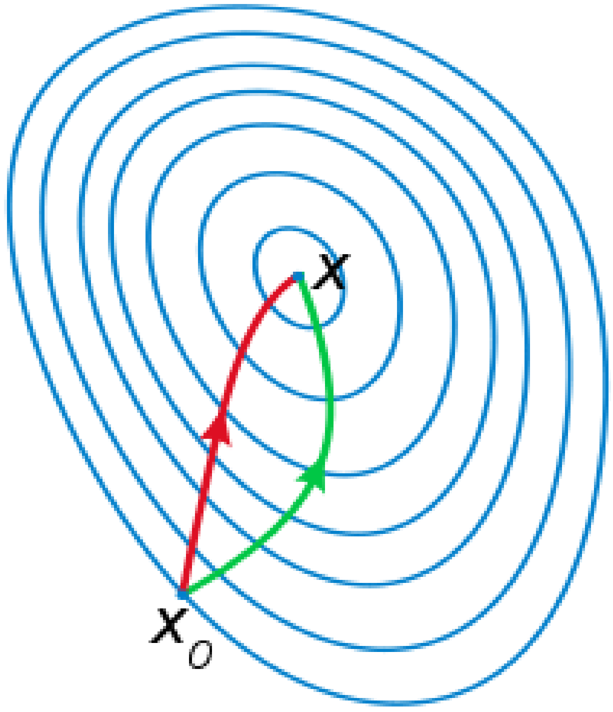
Case Study: Conjugate Gradient

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.

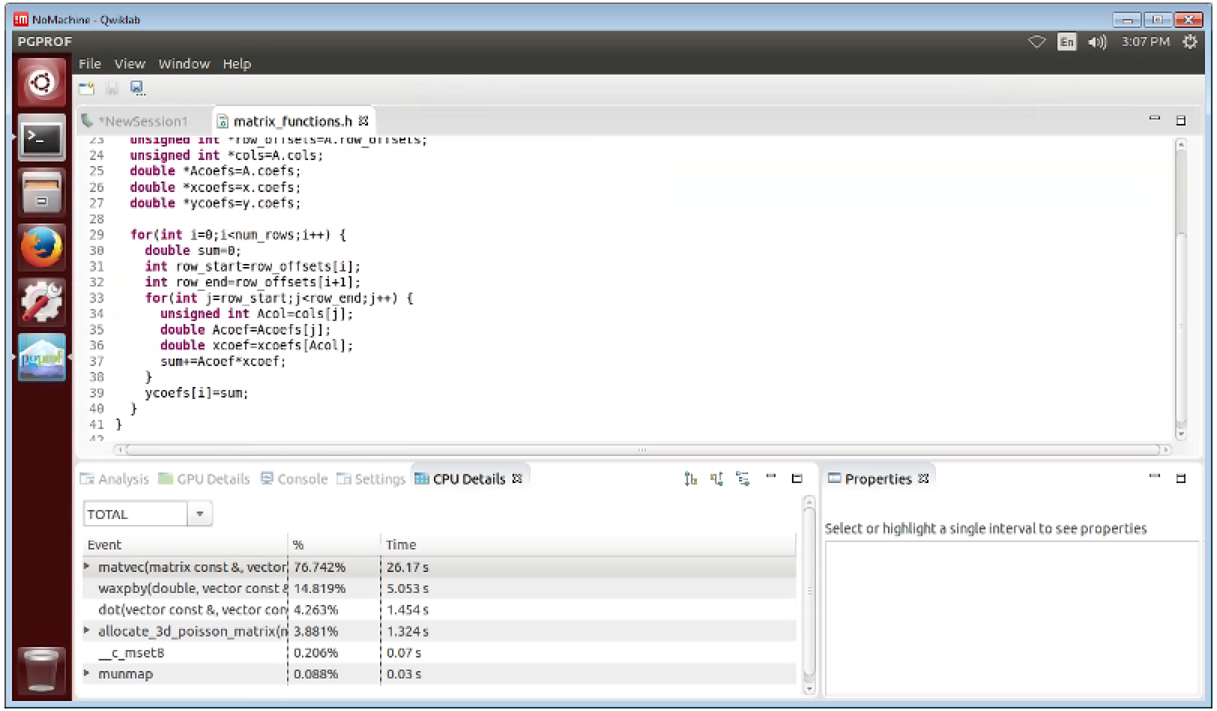
For more information on the CG method, see https://en.wikipedia.org/wiki/Conjugate_gradient_method



Analyze

Analyze

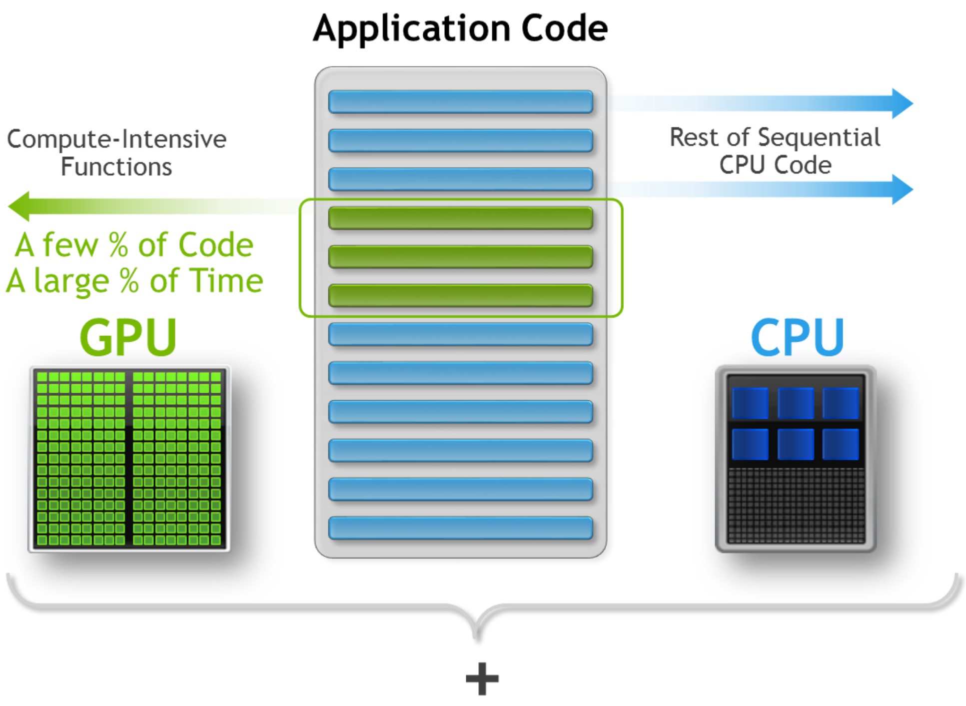
- Obtain a performance profile
- Read compiler feedback
- Understand the code.



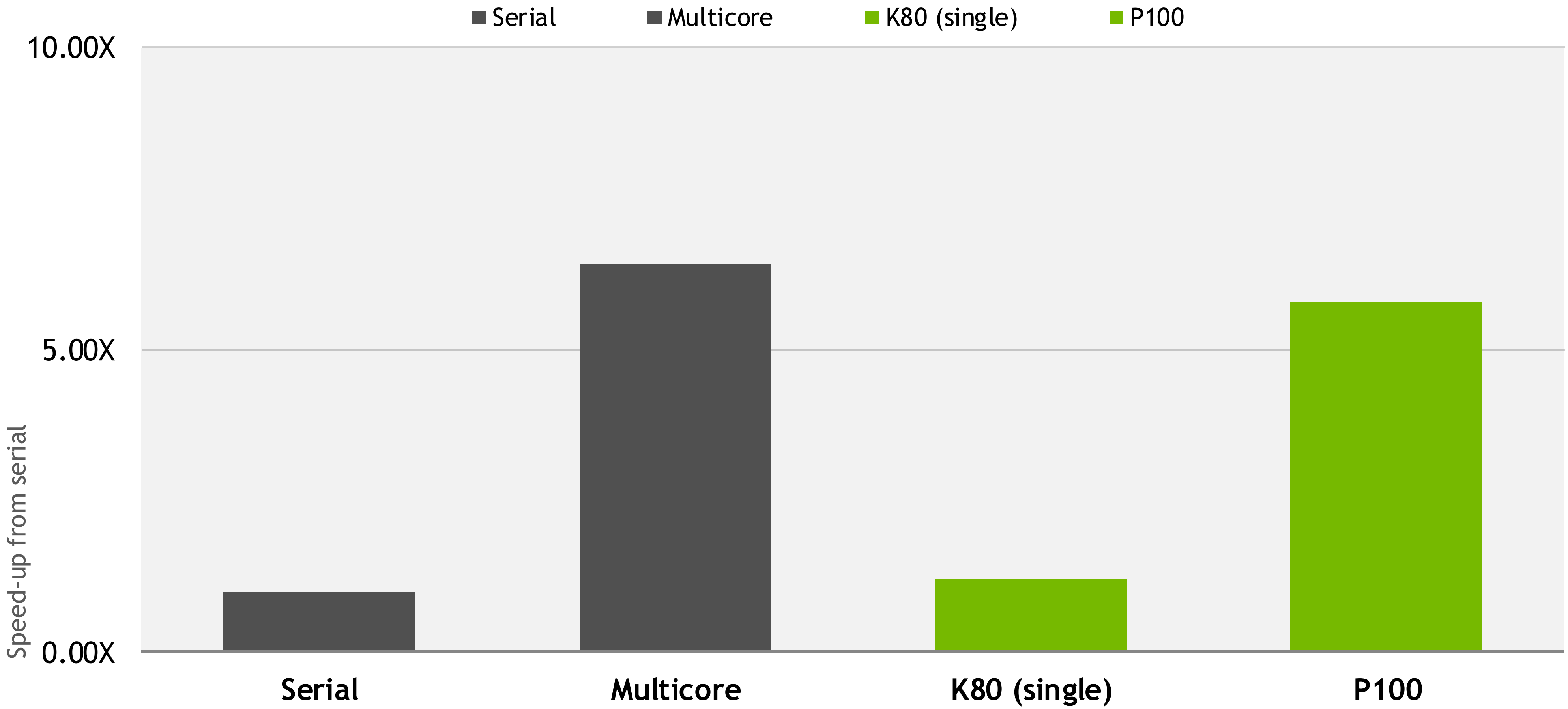
Parallelize

Parallelize

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



Performance after step 2…



Source: PGI 16.9, Multicore: Intel Xeon CPUE5-2698 v3 @ 2.30GHz

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 GPU 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

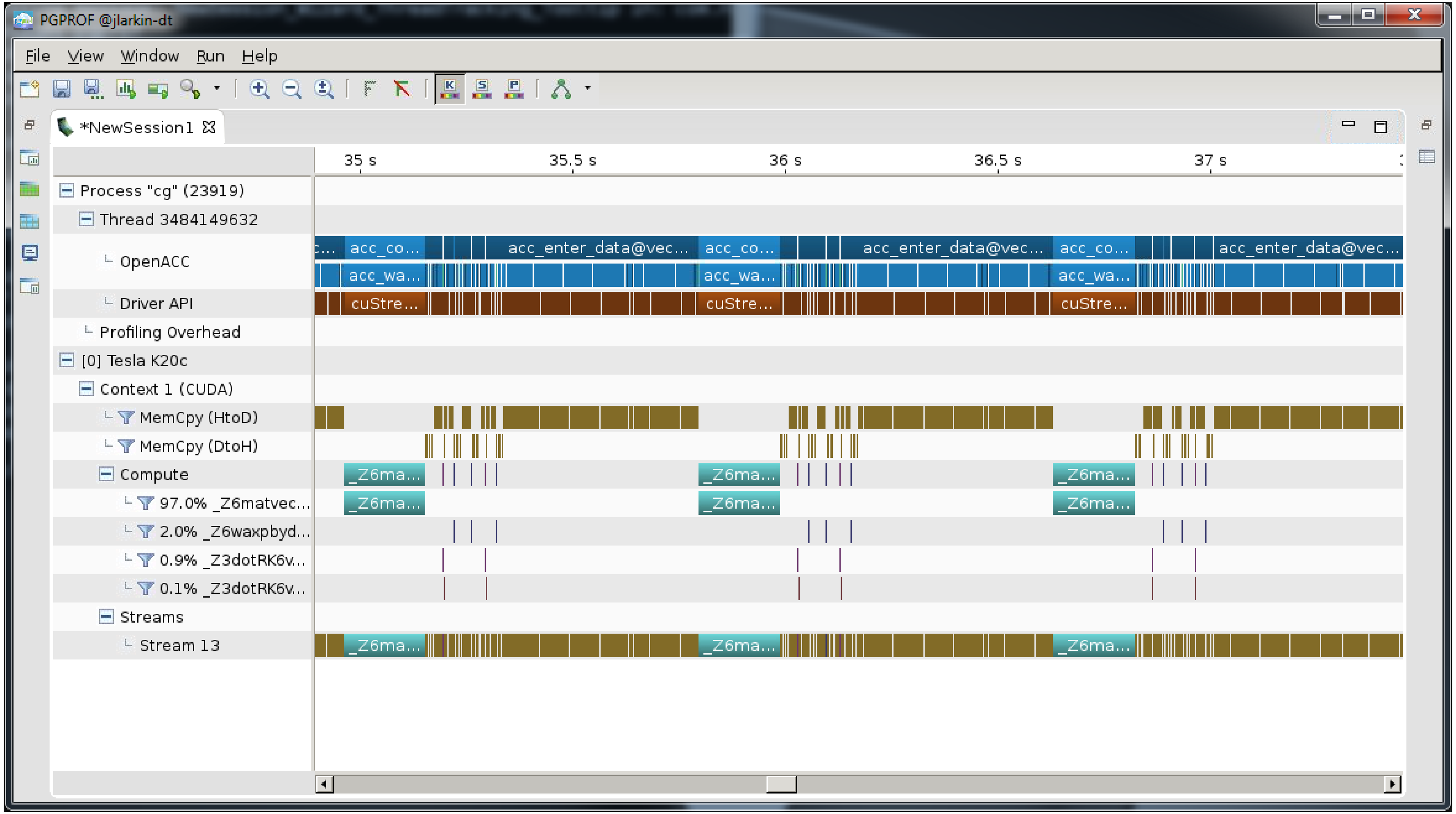
```
!$acc data copyin(a(1:end)) copyout(b(s/4:3*s/4))
```

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

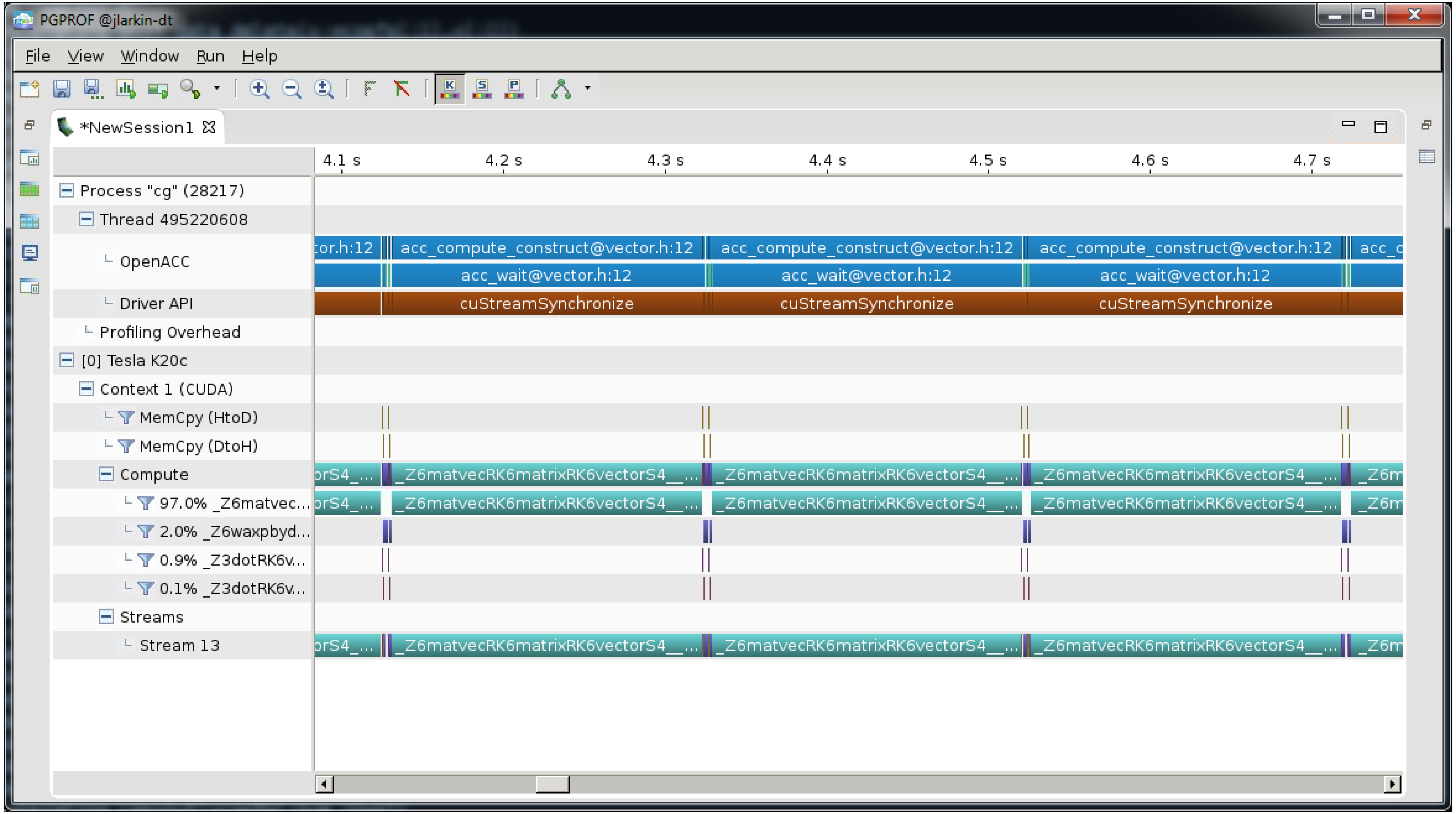
```
void initialize_vector(vector &v,double val)
{

  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     }
```

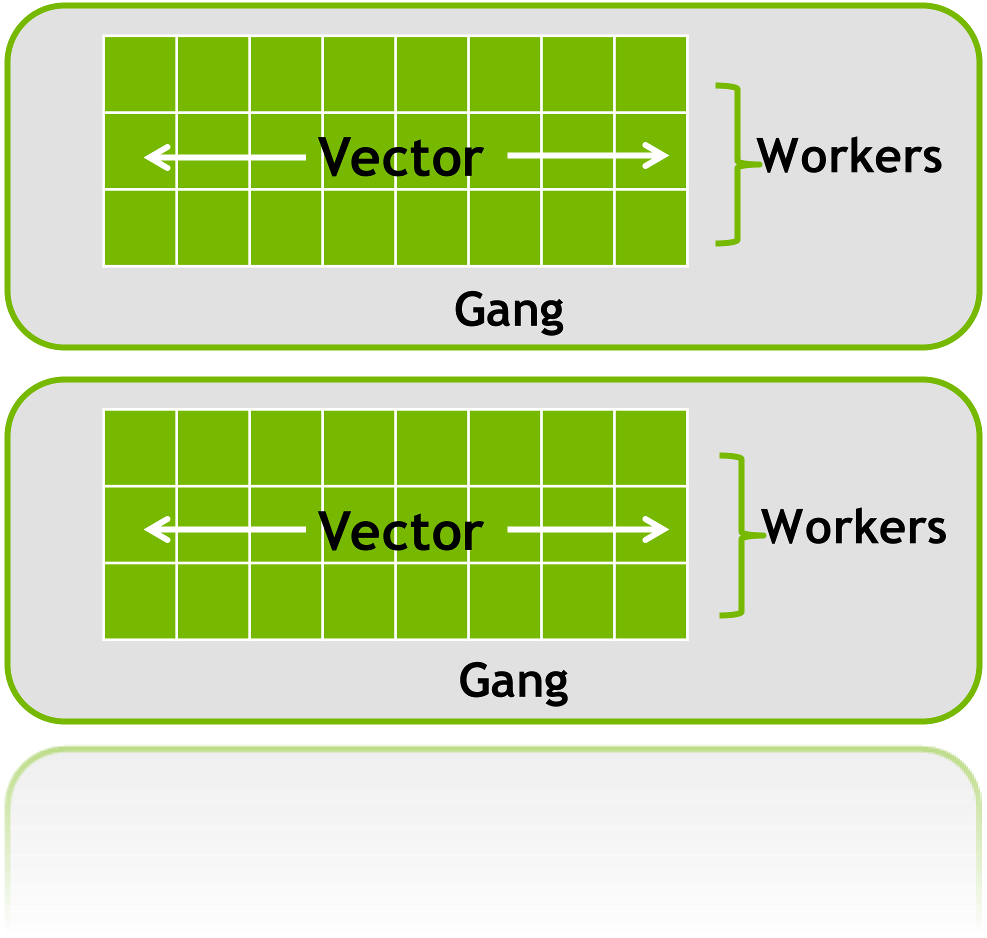
The compiler is vectorizing 128 iterations of this loop. How many iterations does it really do?

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

```
void allocate_3d_poisson_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));
```

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)
   ...
```

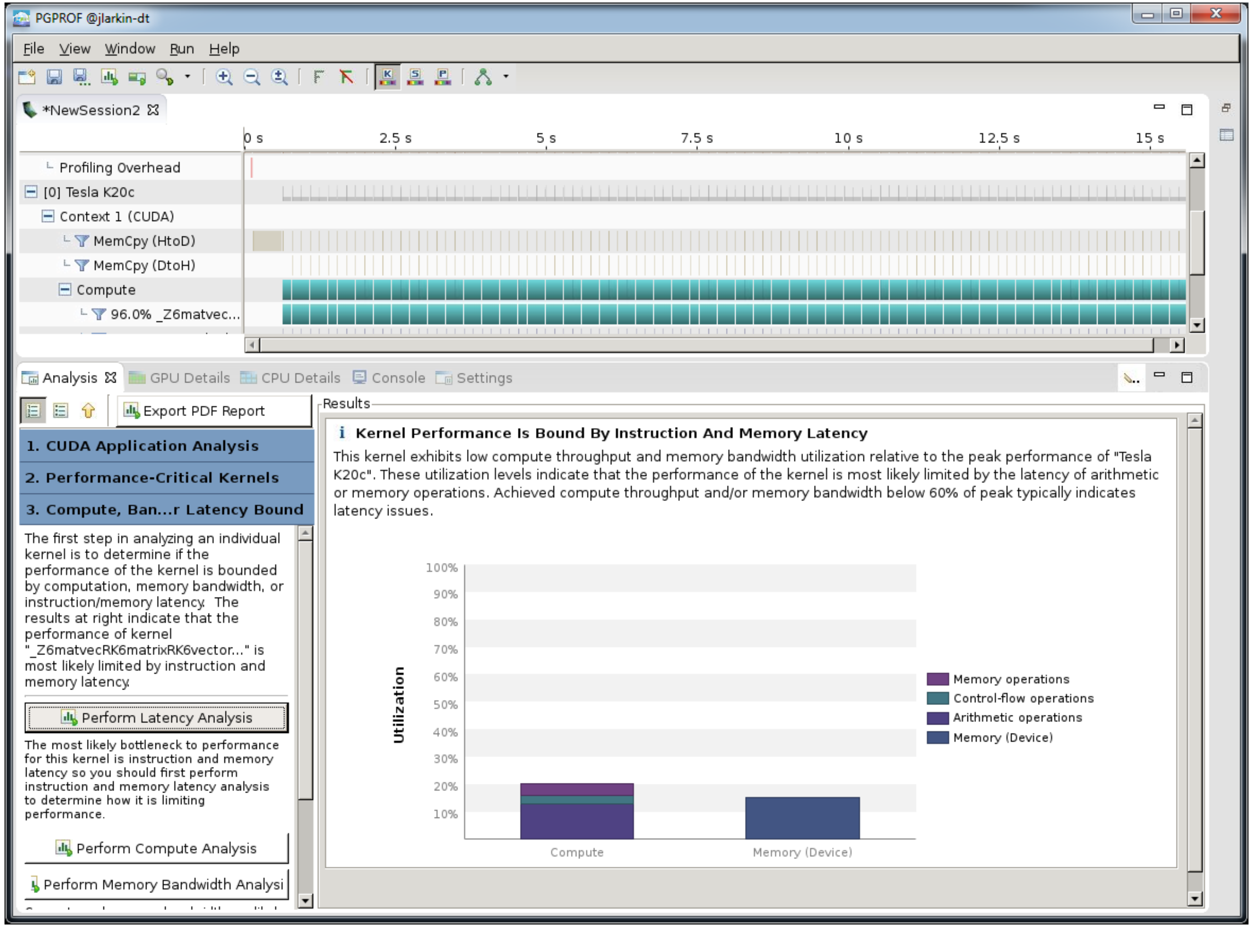
```
#pragma acc parallel vector_length(32)
#pragma acc loop gang worker
for (int i = 0; i < n; ++i)
  #pragma acc loop vector
  for (int j = 0; j < n; ++j)
    ...
```

Optimizing Matvec Loops: Vector Length

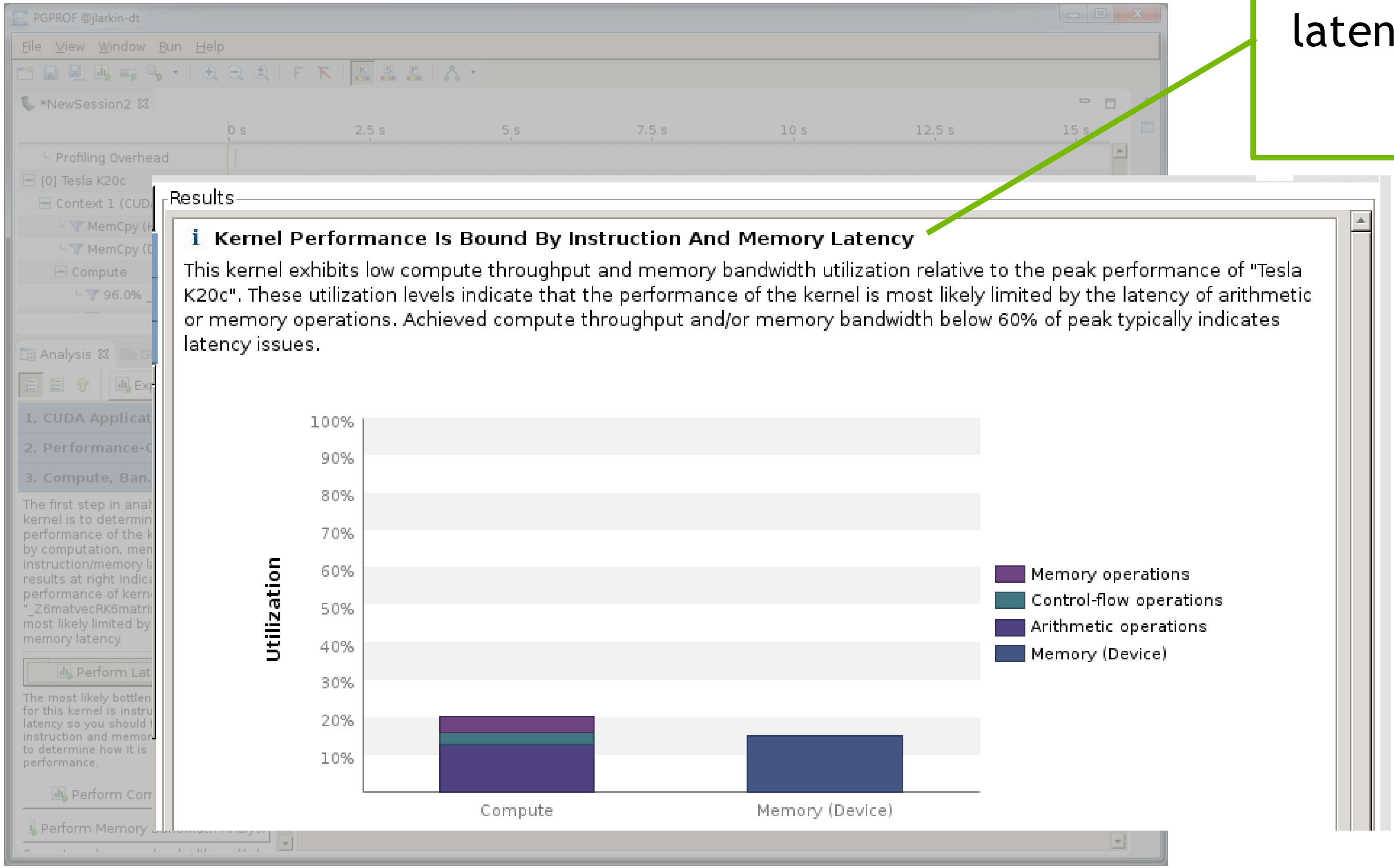
- Use the OpenACC loop directive to force the compiler to vectorizer the inner loop.
- Use vector_length to reduce the vector length closer to actual loop iterations
- Note: NVIDIA GPUs need vector lengths that are multiples of 32 (warp size)

```
#pragma acc parallel loop
vector_length(32) \
  copyout(ycoefs[:num_rows])
copyin(Acoefs[:A.nnz],xcoefs[:num_rows],cols
[:A.nnz])
  for(int i=0;i<num_rows;i++) {
    double sum=0;
    int row_start=row_offsets[i];
    int row_end=row_offsets[i+1];
#pragma acc loop vector reduction(+:sum)
    for(int j=row_start;j<row_end;j++) {
      unsigned int Acol=cols[j];
      double Acoef=Acoefs[j];
      double xcoef=xcoefs[Acol];
      sum+=Acoef*xcoef;
    }
    ycoefs[i]=sum;
  }
```

Matvec Performance Limiter

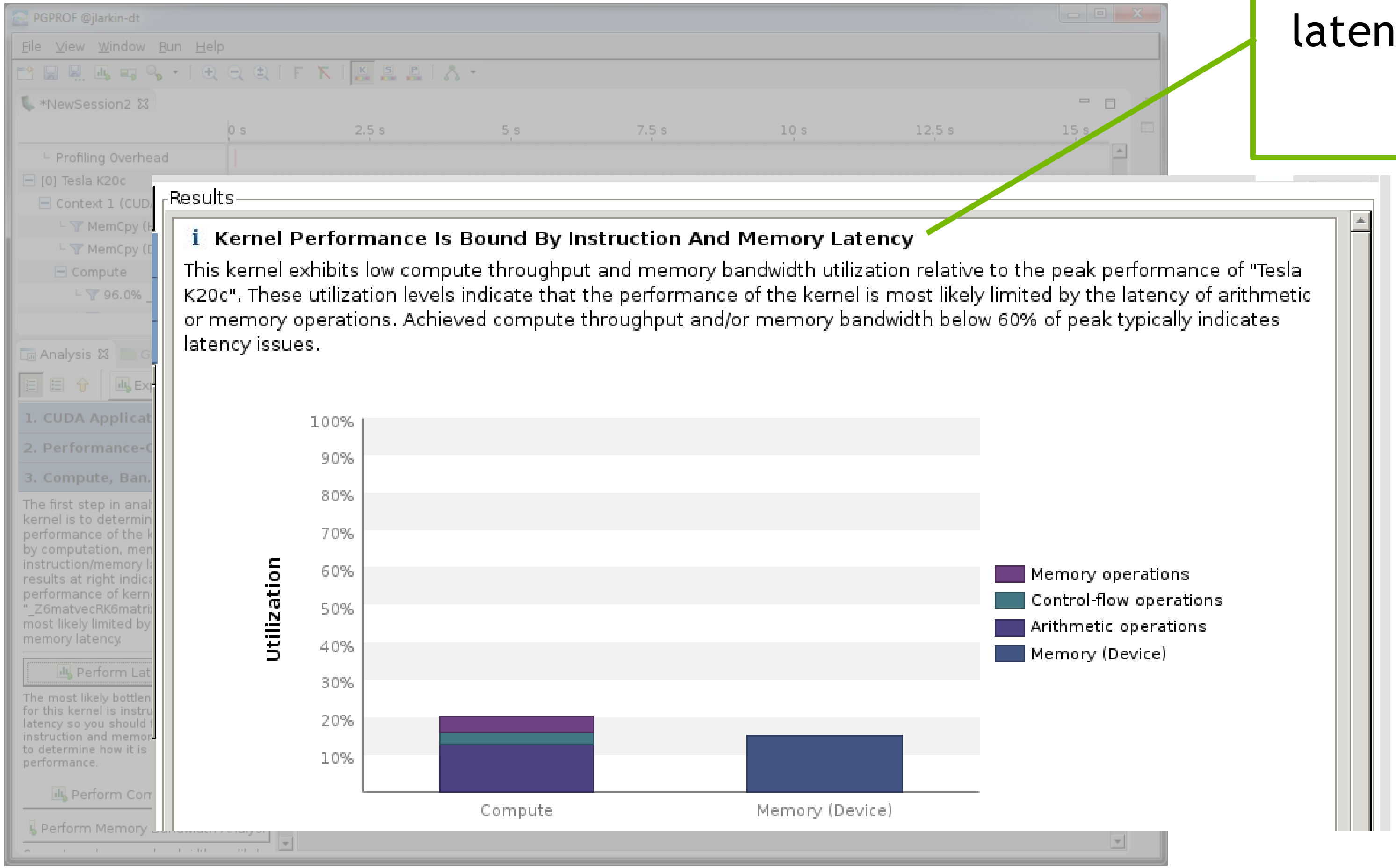


Matvec Performance Limiter



Instruction and Memory latency are limiting kernel performance.

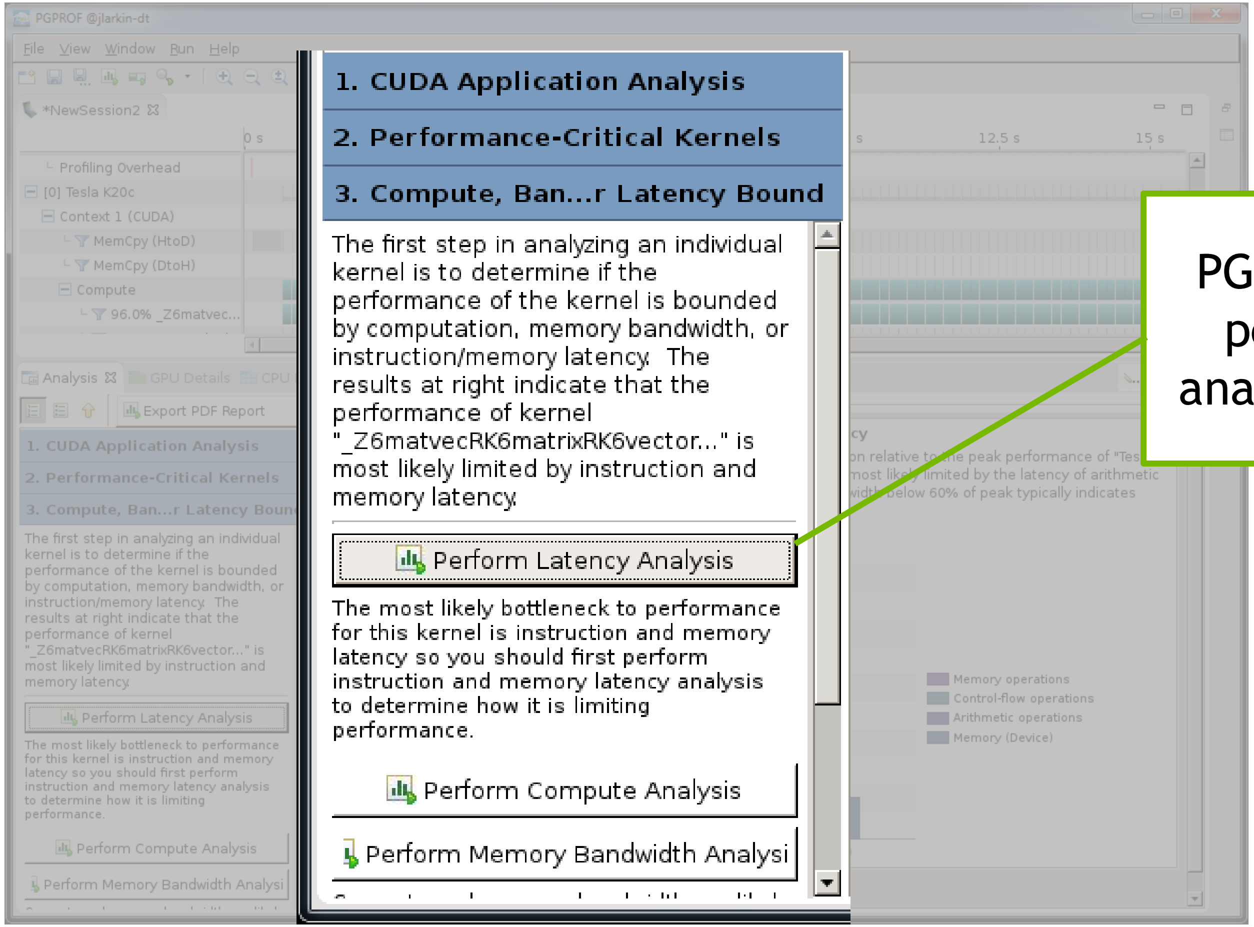
Matvec Performance Limiter



Instruction and Memory latency are limiting kernel performance.

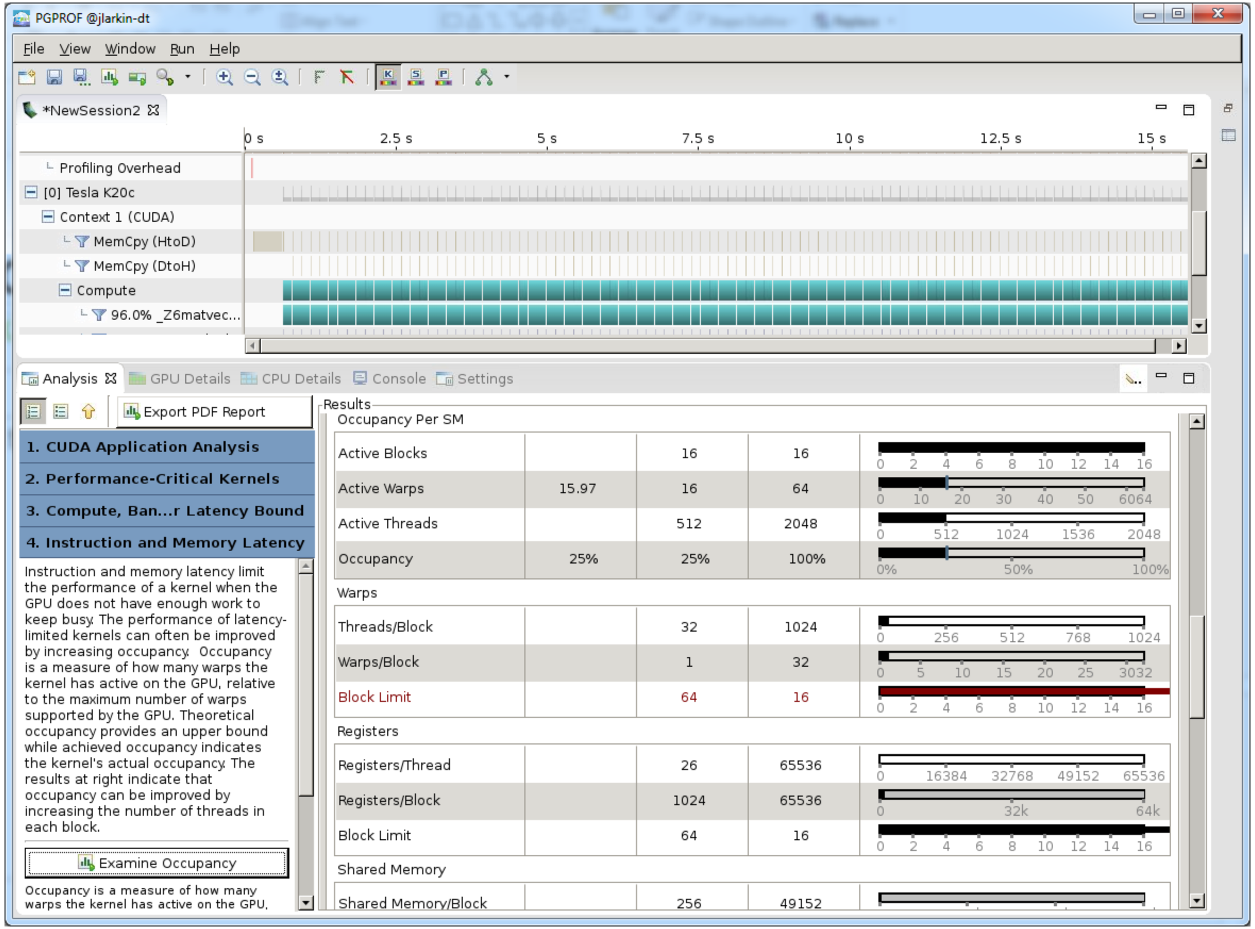
Recall: GPU's tolerate latency by having *enough* parallelism

Matvec Performance Limiter



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

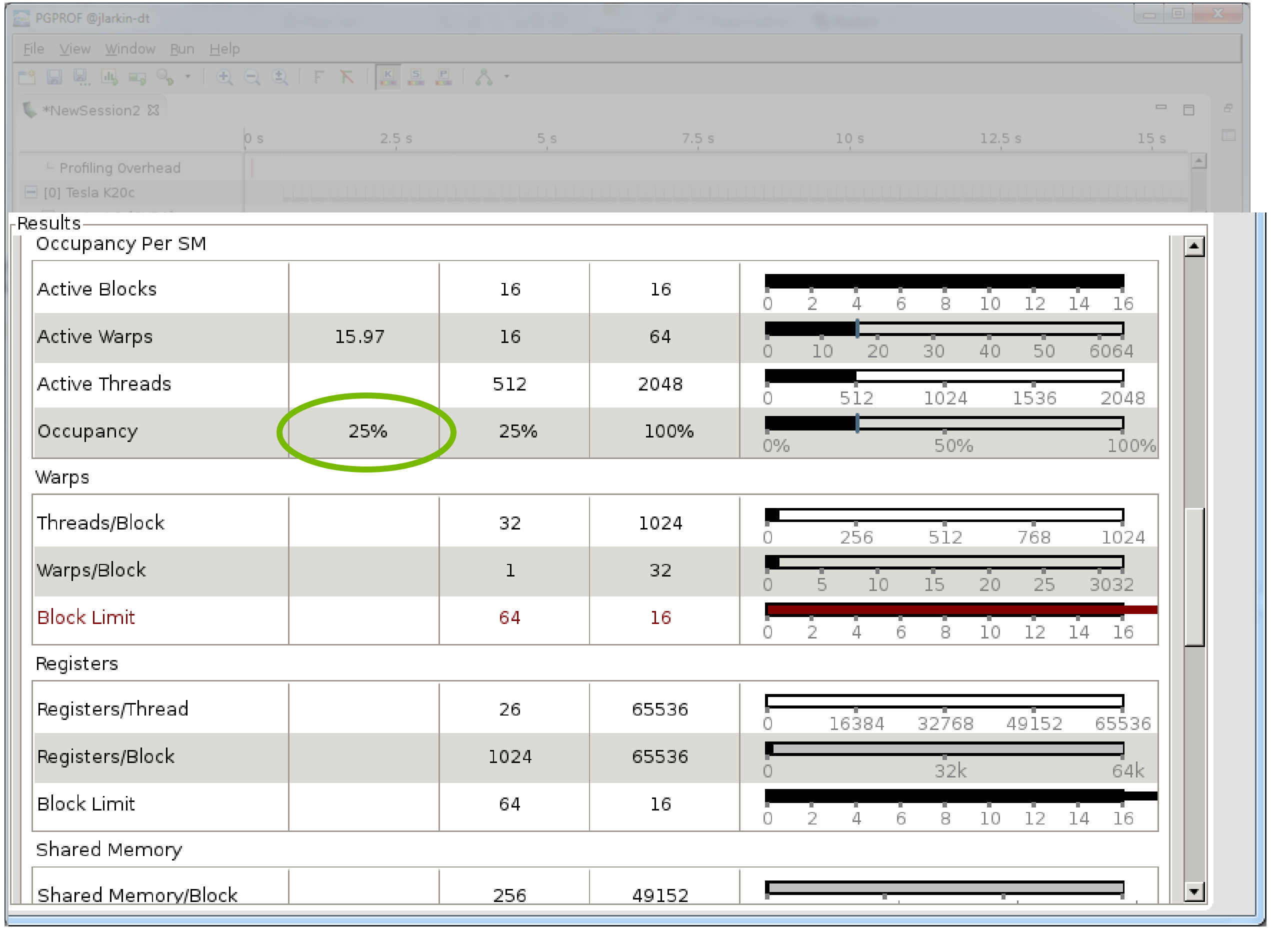
Matvec Occupancy



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 Per SM

Active Blocks		16	16	
Active Warps	15.97	16	64	
Active Threads		512	2048	
Occupancy	25%	25%	100%	

Warps

Threads/Block		32	1024	
Warps/Block		1	32	
Block Limit		64	16	

Registers

Registers/Thread		26	65536	
Registers/Block		1024	65536	
Block Limit		64	16	

Shared Memory

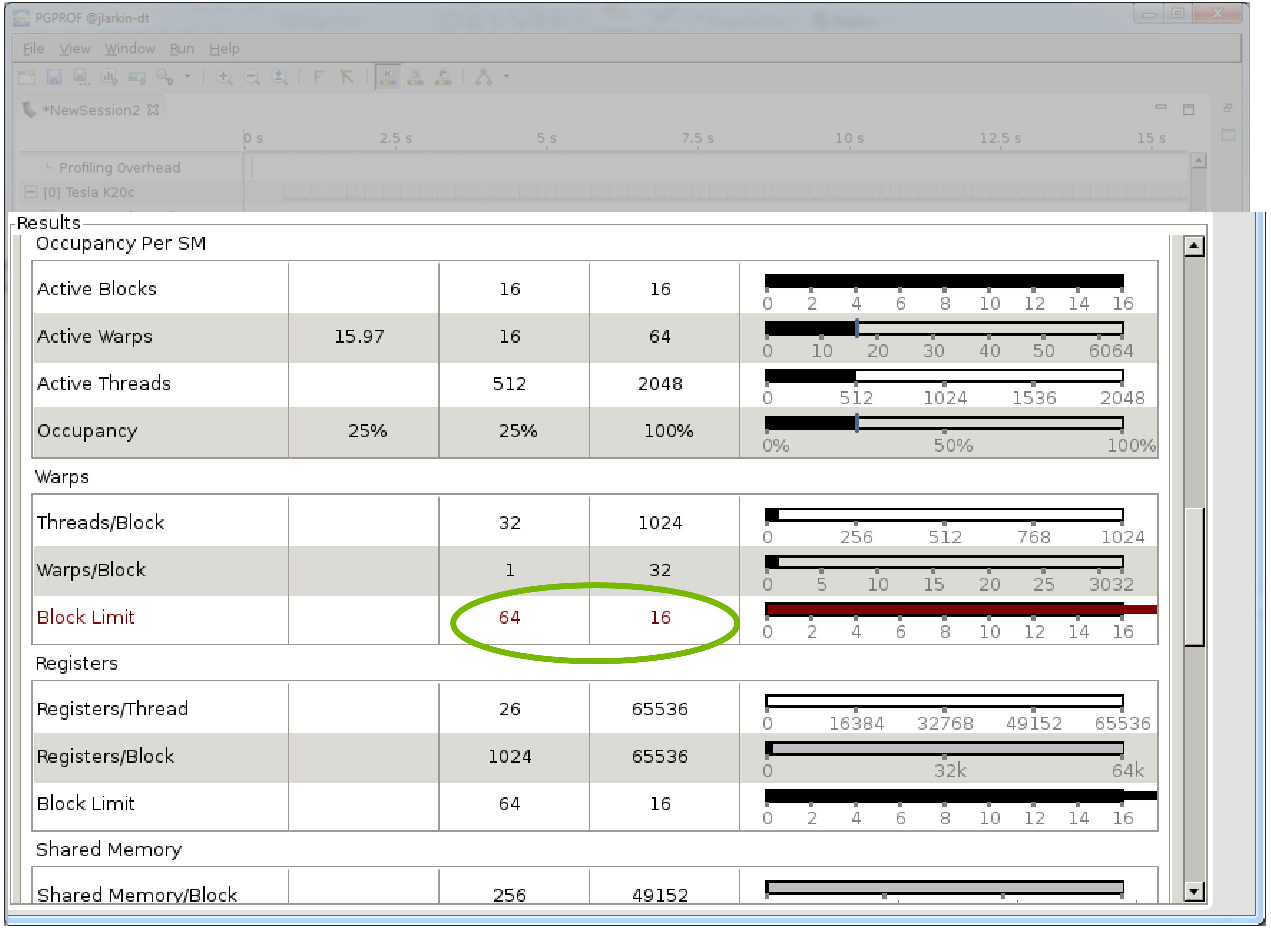
Shared Memory/Block		256	49152	

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



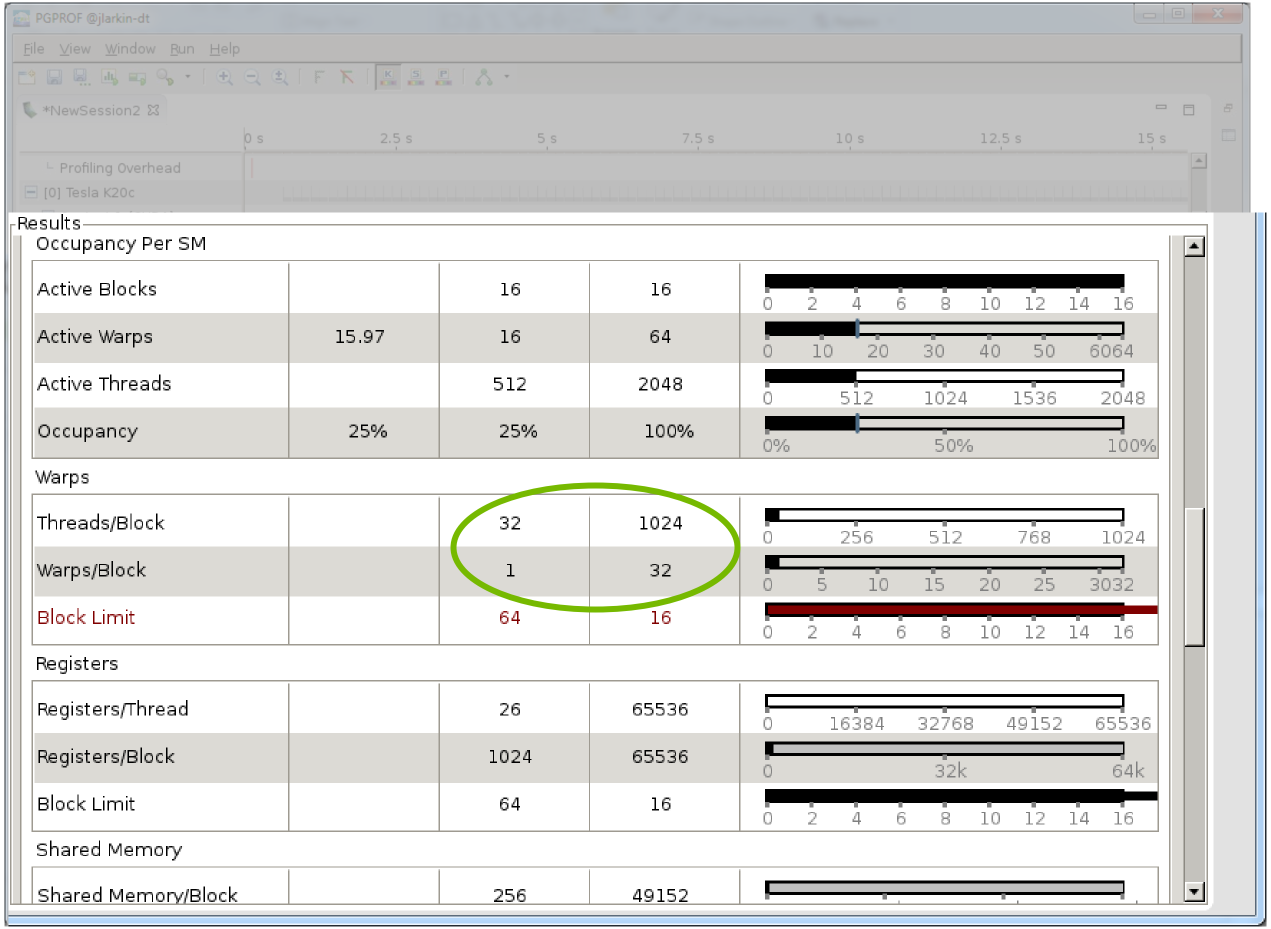
Occupancy is a measure of how well the GPU is being utilized.

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.

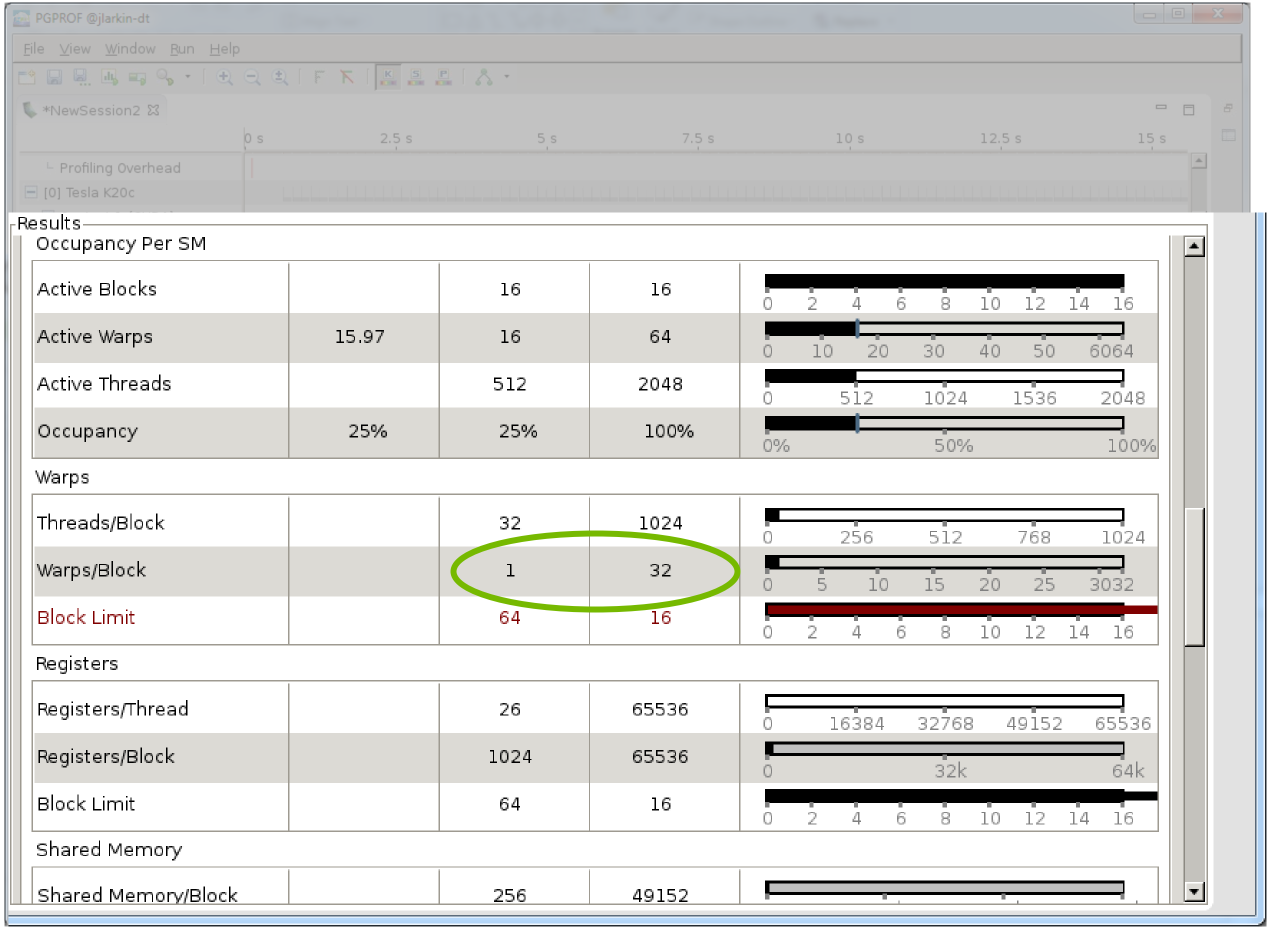
100% occupancy means the GPU is running as many simultaneous threads as it can.

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



Results

Occupancy Per SM

Active Blocks		16	16	0 2 4 6 8 10 12 14 16
Active Warps	15.97	16	64	0 10 20 30 40 50 6064
Active Threads		512	2048	0 512 1024 1536 2048
Occupancy	25%	25%	100%	0% 50% 100%

Warps

Threads/Block		32	1024	0 256 512 768 1024
Warps/Block		1	32	0 5 10 15 20 25 3032
Block Limit		64	16	0 2 4 6 8 10 12 14 16

Registers

Registers/Thread		26	65536	0 16384 32768 49152 65536
Registers/Block		1024	65536	0 32k 64k
Block Limit		64	16	0 2 4 6 8 10 12 14 16

Shared Memory

Shared Memory/Block		256	49152	

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.

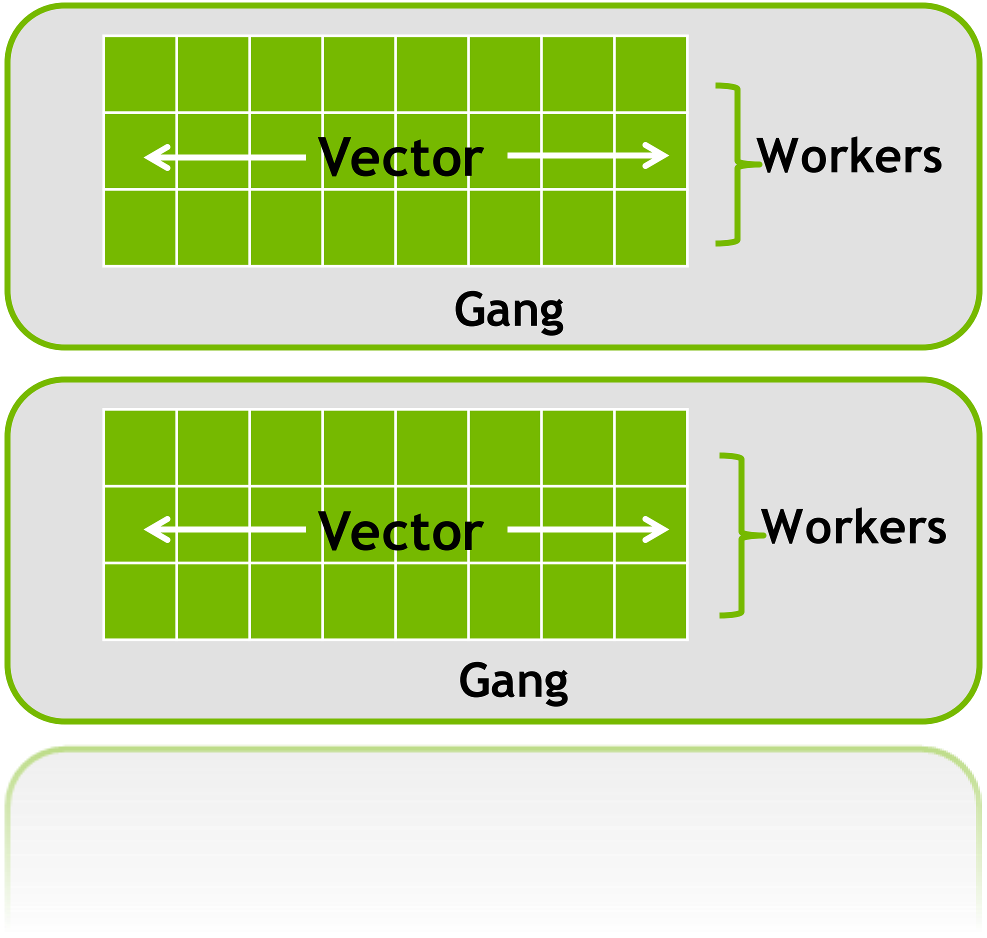
We're only keeping the GPU 25% occupied, why?

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 4x32 = 128 threads in each GPU threadblock

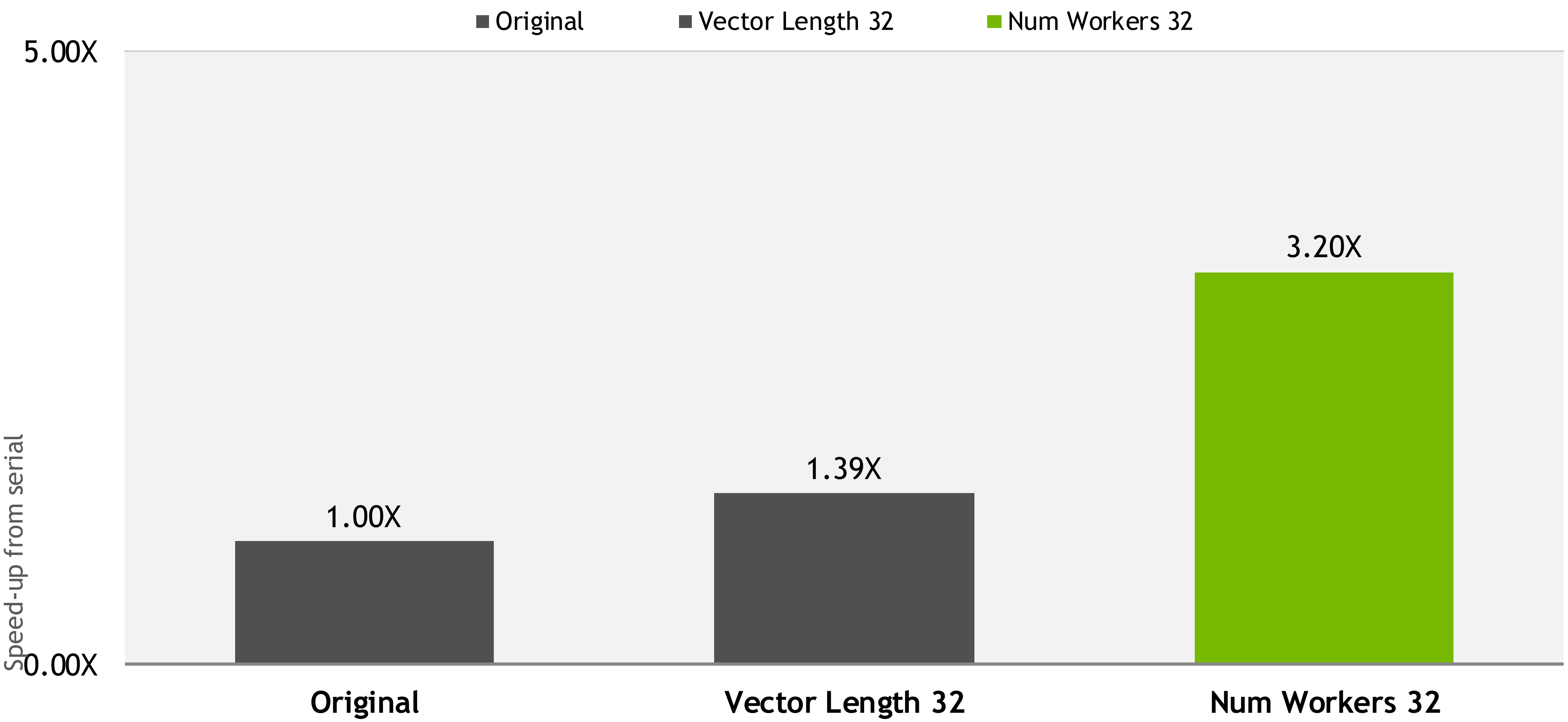
```
#pragma acc parallel loop gang worker \
     num_workers(4) vector_length(32) \
     copyout(ycoefs[:num_rows])
copyin(Acoefs[:A.nnz],xcoefs[:num_rows],cols
[:A.nnz])
     for(int i=0;i<num_rows;i++) {
       double sum=0;
       int row_start=row_offsets[i];
       int row_end=row_offsets[i+1];
#pragma acc loop vector reduction(+:sum)
       for(int j=row_start;j<row_end;j++) {
         unsigned int Acol=cols[j];
         double Acoef=Acoefs[j];
         double xcoef=xcoefs[Acol];
         sum+=Acoef*xcoef;
       }
       ycoefs[i]=sum;
     }
```

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

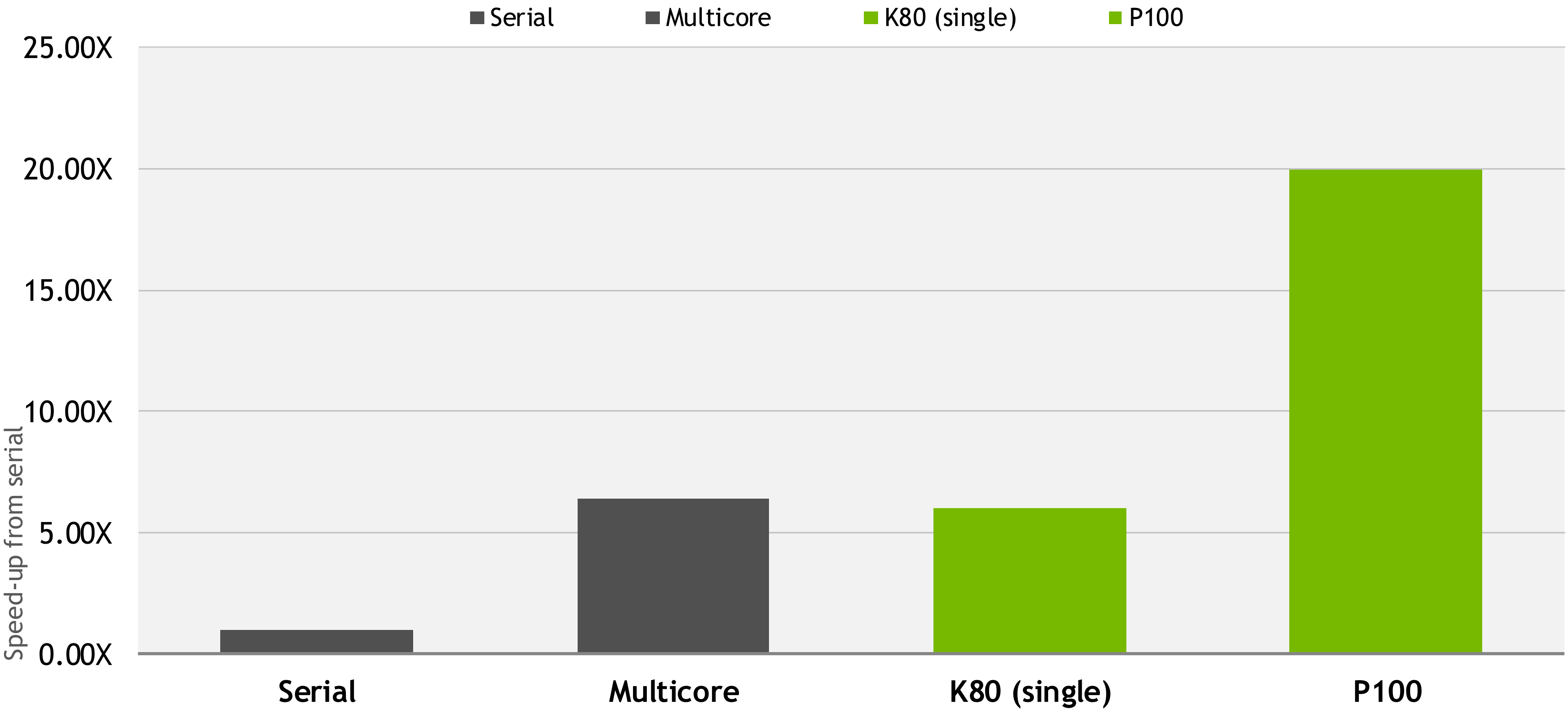
```
matvec(const matrix &, const vector &, const
vector &):
  8, include "matrix_functions.h"
    12, Generating copyin(Acoefs[:A-
>nnz],cols[:A->nnz])
        Generating implicit
copyin(row_offsets[:num_rows+1])
        Generating copyin(xcoefs[:num_rows])
        Generating copyout(ycoefs[:num_rows])
        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



Source: PGI 16.9, NVIDIA Tesla K80

Final Performance



Source: PGI 16.9, Multicore: Intel Xeon CPUE5-2698 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++)
    ...
```

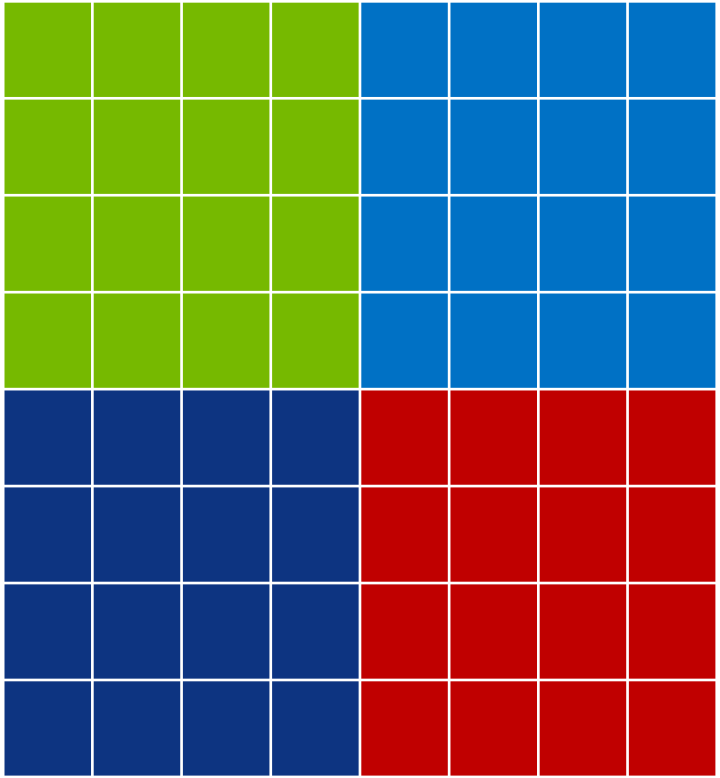
Collapse outer loops to enable creating more gangs.

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 accesses every-other memory element.

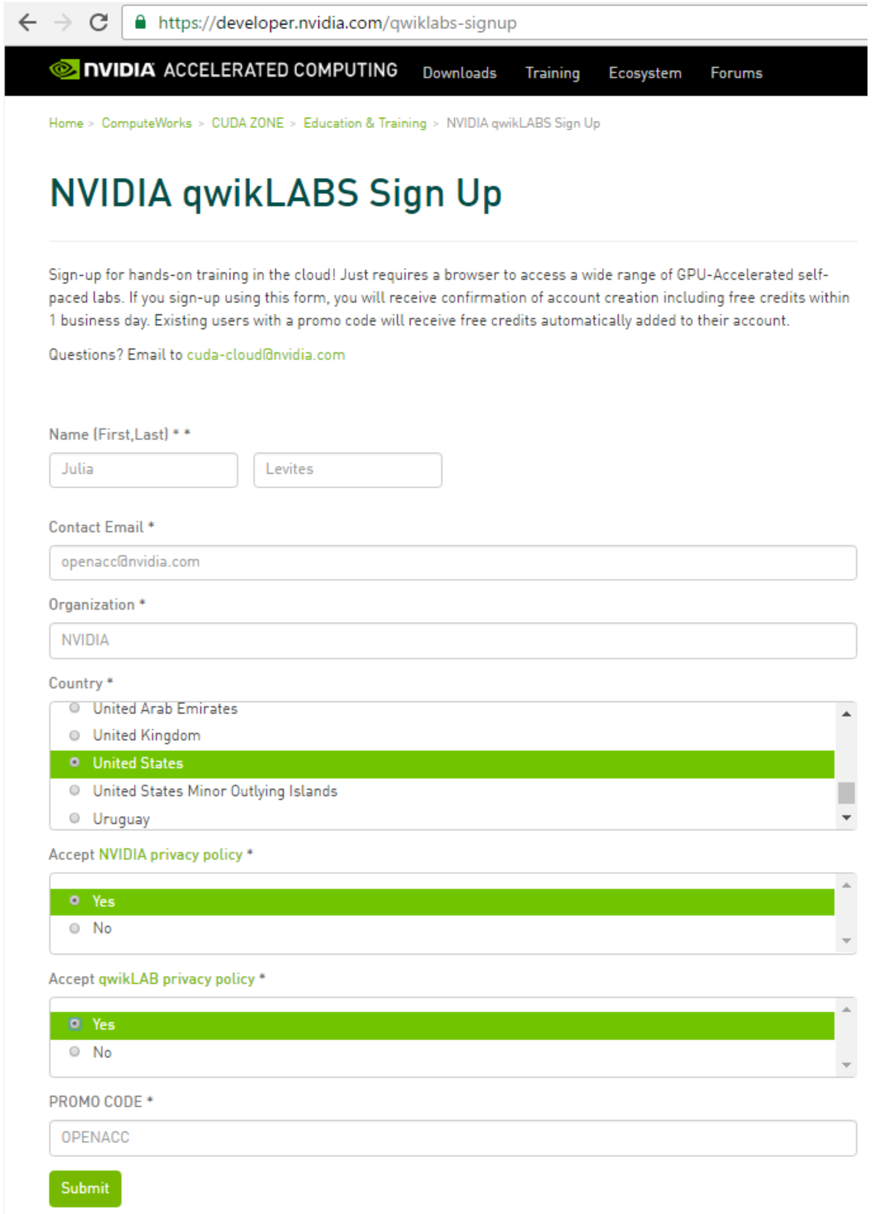
```
for(i=0; i<N; i++)
  for(j=0; j<M; j++)
  {
    Aa[i][j] = 1.0f;
    Ab[i][j] = 0.0f;
  }
}
```

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

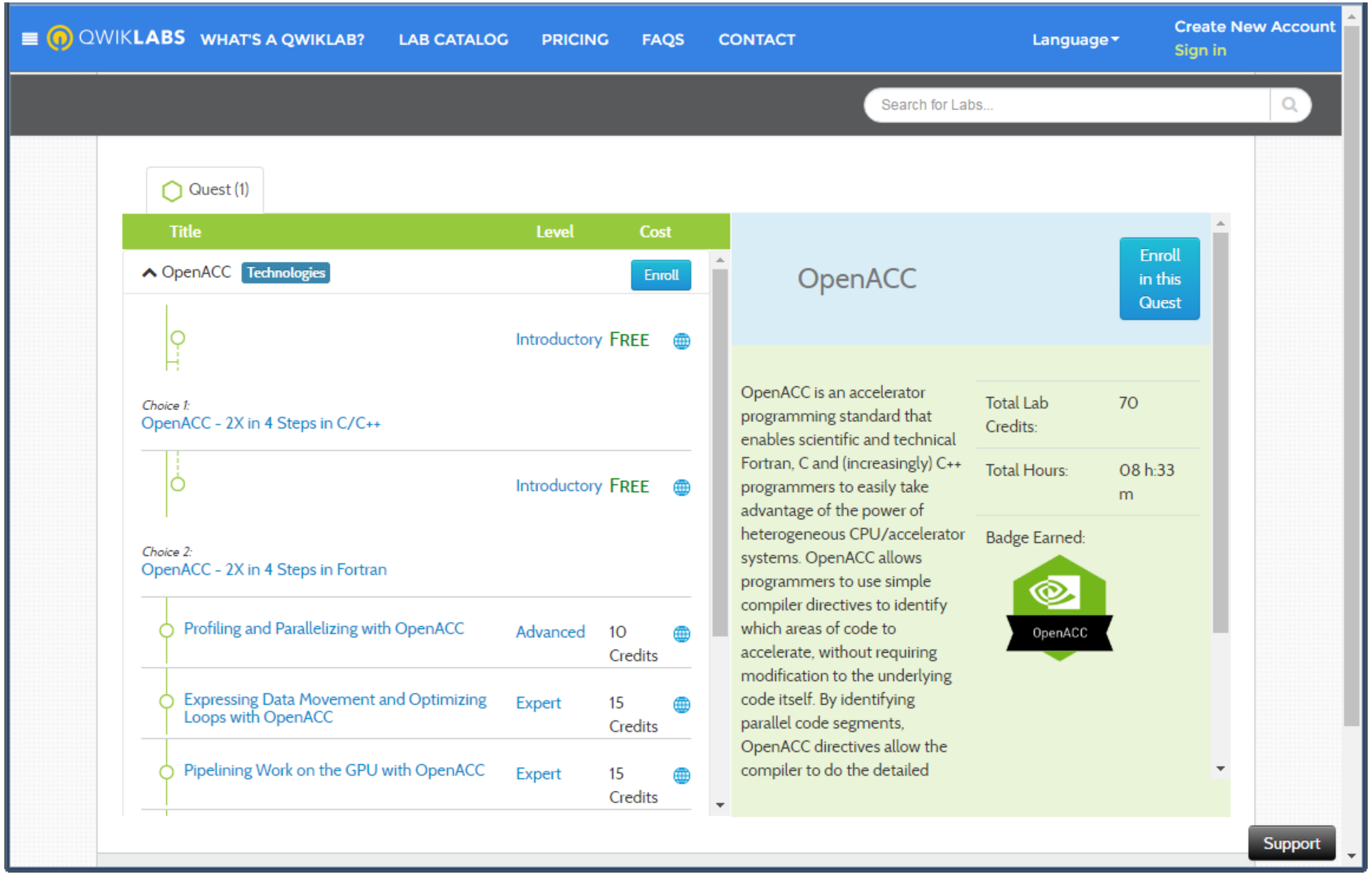
Using QwikLabs

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!



This week's labs

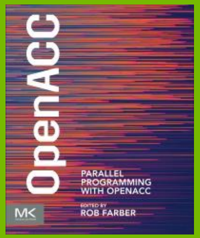


This week you should complete the "Profiling and Parallelizing with OpenACC" and "Expressing Data Movement and Optimizing Loops with OpenACC" labs in qwiklabs.

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 1st, 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

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