

# Expressing Data Locality and Optimizations with OpenACC

OpenACC Course: Lecture 3, October 29, 2015



# Course Syllabus

Oct 1: Introduction to OpenACC

Oct 6: Office Hours

Oct 15: Profiling and Parallelizing with the OpenACC Toolkit

Oct 20: Office Hours

Oct 29: Expressing Data Locality and Optimizations with OpenACC

Nov 3: Office Hours

Nov 12: Advanced OpenACC Techniques

Nov 24: Office Hours

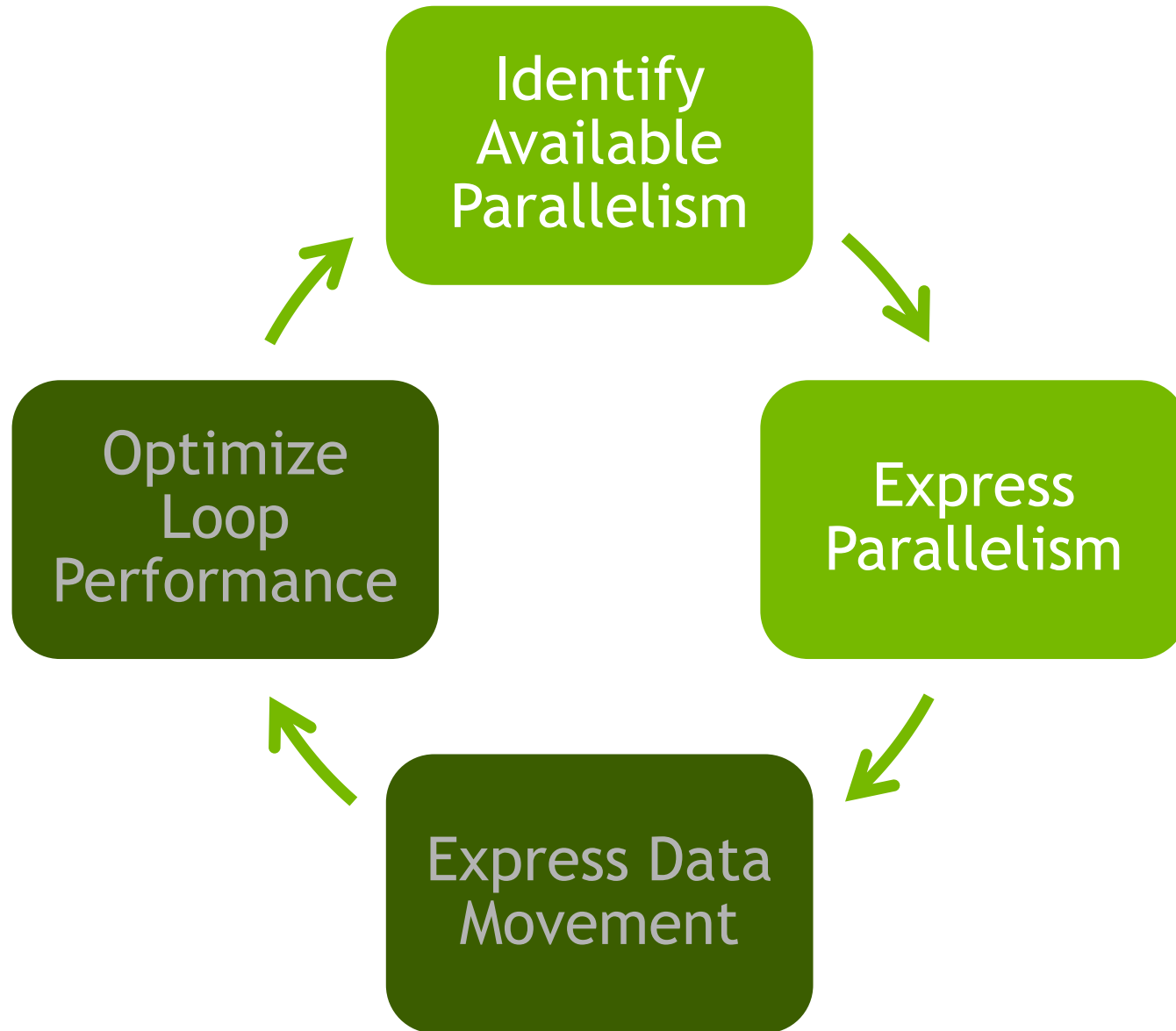
# Agenda

Review: Parallelizing with OpenACC and Unified Memory

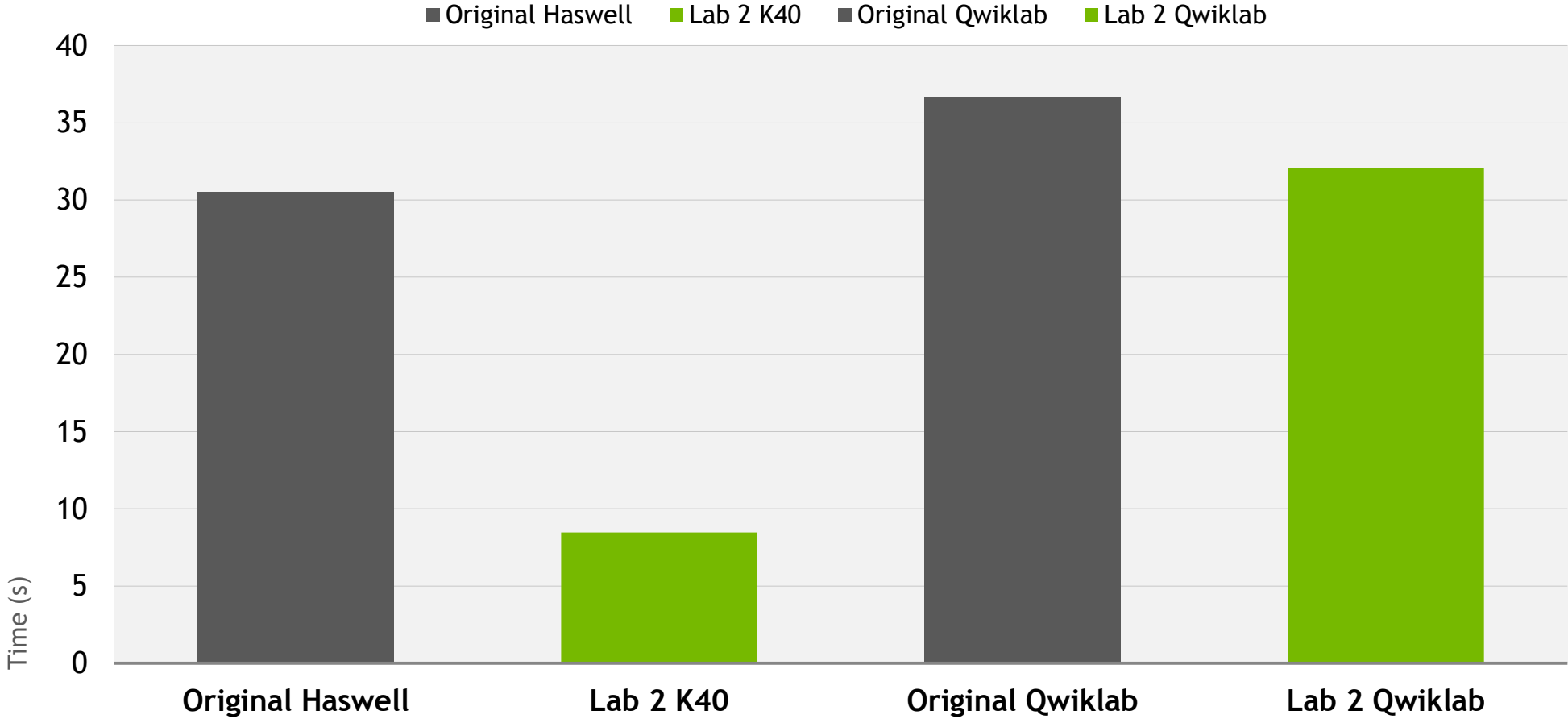
Expressing Data Movement

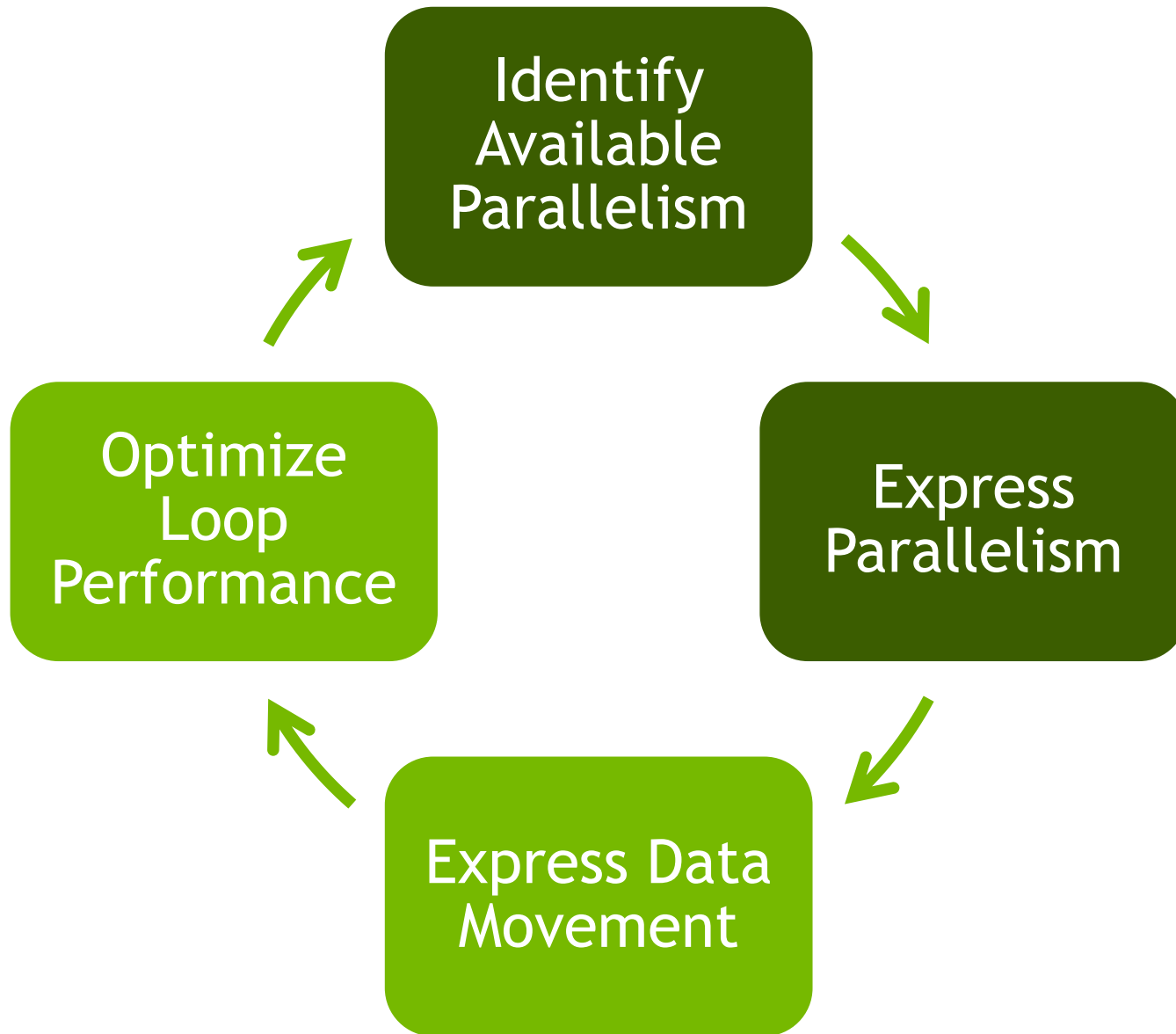
Optimizing Loops

Next Steps & Homework



# Lab 2 Results (Lower is better)





Identify  
Available  
Parallelism

Express  
Parallelism

Express Data  
Movement

Optimize  
Loop  
Performance

# Expressing Data Management

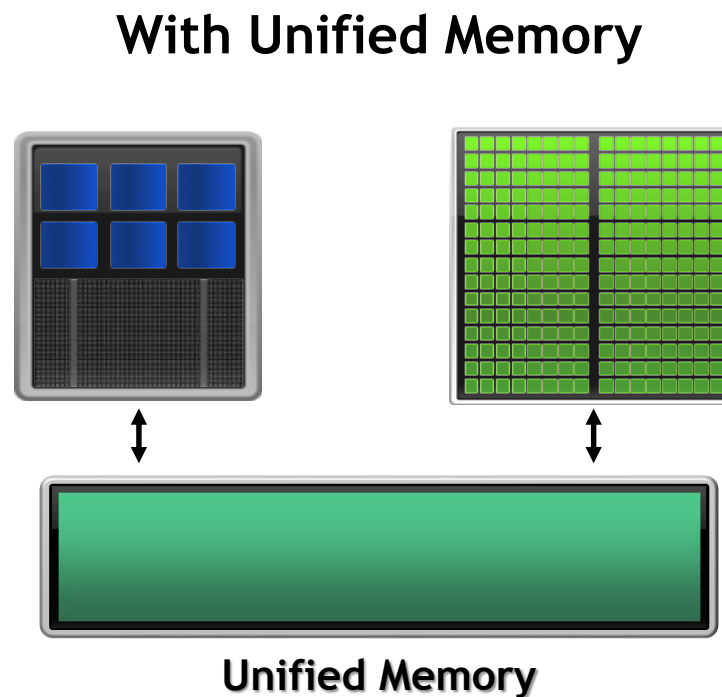
# Making Data Management Explicit

We used CUDA Unified Memory to simplify the first steps in accelerating our code.

This made the process simple, but it also made the code not portable

- PGI-only: `-ta=tesla:managed` flag
- NVIDIA-only: CUDA Unified Memory

Explicitly managing data will make the code portable and may improve 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)`

```
#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;  
};
```

# 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.

# 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))
```

# Coursework : Expressing Data Movement

# Explicit Data Movement: Copy In Matrix

```
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));

    // 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.
- ▶ Lab 3: Next do the same for the vector.

# OpenACC present clause

When managing the memory at a higher level, it's necessary to inform the compiler that data is already present on the device.

Local variables should still be declared in the function where they're used.

High-level data management and the present clause are often *critical* to good performance.

```
function main(int argc, char **argv)
{
    #pragma acc data copy(A)
    {
        laplace2D(A,n,m);
    }
}
```

```
function laplace2D(double[N][M] A,n,m)
{
    #pragma acc data present(A[n][m]) create(Anew)
    while ( err > tol && iter < iter_max ) {
        err=0.0;
        ...
    }
}
```

# Explicit Data Movement: Present Clause

```
#pragma acc kernels \  
    present(row_offsets,cols,Acoefs,xcoefs,ycoefs)  
{  
    for(int i=0;i<num_rows;i++) {  
        double sum=0;  
        int row_start=row_offsets[i];  
        int row_end=row_offsets[i+1];  
        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;  
    }  
}
```

- ▶ At the compute regions (kernels or parallel loop) inform the compiler that the data is already present.
- ▶ Lab 3: Do the same for the waxpy and dot functions on the vector.

# 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 self(a)
```



Copy "a" from GPU to CPU

```
do_something_on_host()
```

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



Copy "a" from CPU to GPU

# Explicit Data Movement: Update Vector

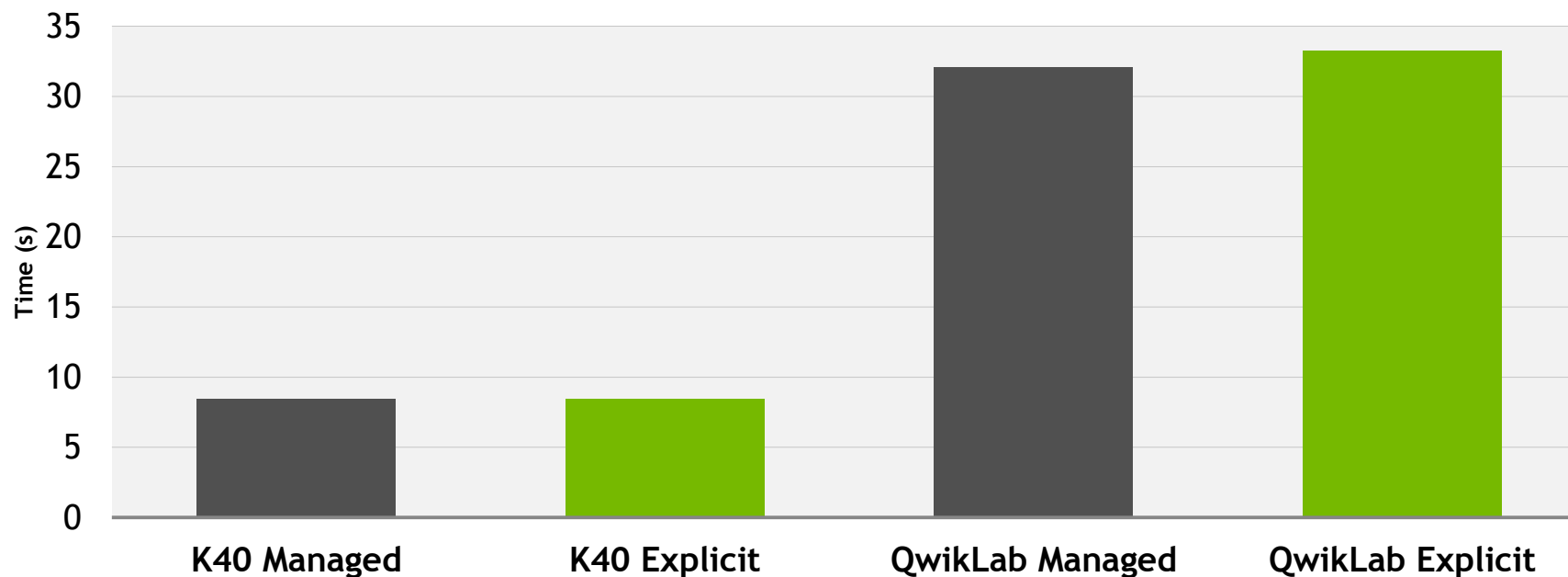
```
void initialize_vector(vector &v, double val)
{
    for(int i=0; i<v.n; i++)
        v.coefs[i]=val;
    #pragma acc update device(v.coefs[:v.n])
}
```

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

# Build & Run without Unified Memory

Rebuild the code without managed memory.

- Change `-ta=tesla:managed` to just `-ta=tesla`



Optimize Loops



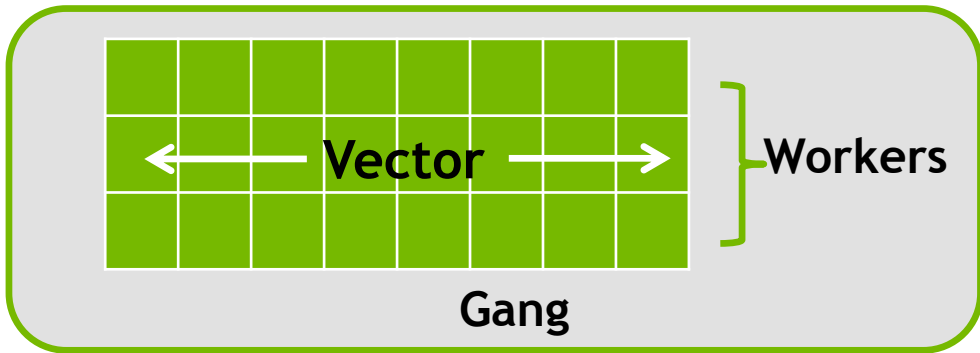
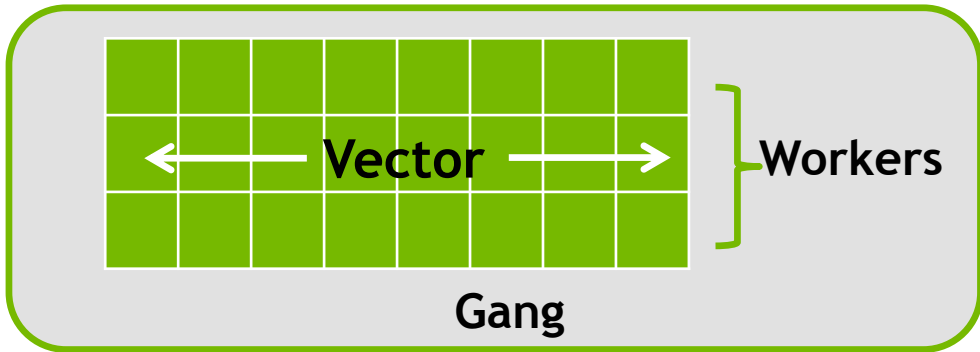
# Apply Application Knowledge

What do you know that the compiler doesn't?

```
matvec(const matrix &, const vector &, const vector &):  
    8, include "matrix_functions.h"  
    15, Generating present(row_offsets[:],  
cols[:],Acoefs[:],xcoefs[:],ycoefs[:])  
    16, Loop is parallelizable  
    Accelerator kernel generated  
    Generating Tesla code  
    16, #pragma acc loop gang, vector(128)  
/* blockIdx.x threadIdx.x */  
    20, Loop is parallelizable
```

- ▶ We know that each row of the matrix has 27 elements.
- ▶ The compiler generates vectors of length 128.
- ▶ This means each vector has 101 empty elements, wasting compute resources.

# 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 loop directive: gang, worker, vector, seq

The loop directive gives the compiler additional information about the *next* loop.

- **gang** - Apply gang-level parallelism to this loop
- **worker** - Apply worker-level parallelism to this loop
- **vector** - Apply vector-level parallelism to this loop
- **seq** - Do not apply parallelism to this loop, run it sequentially

Multiple levels can be applied to the same loop, but the levels must be applied in a top-down order.

# Optimize Loops: Vector Length

```
#pragma acc kernels \
    present(row_offsets,cols,Acoefs,xcoefs,ycoefs)
{
    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 device_type(nvidia) vector(32)
        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;
    }
}
```

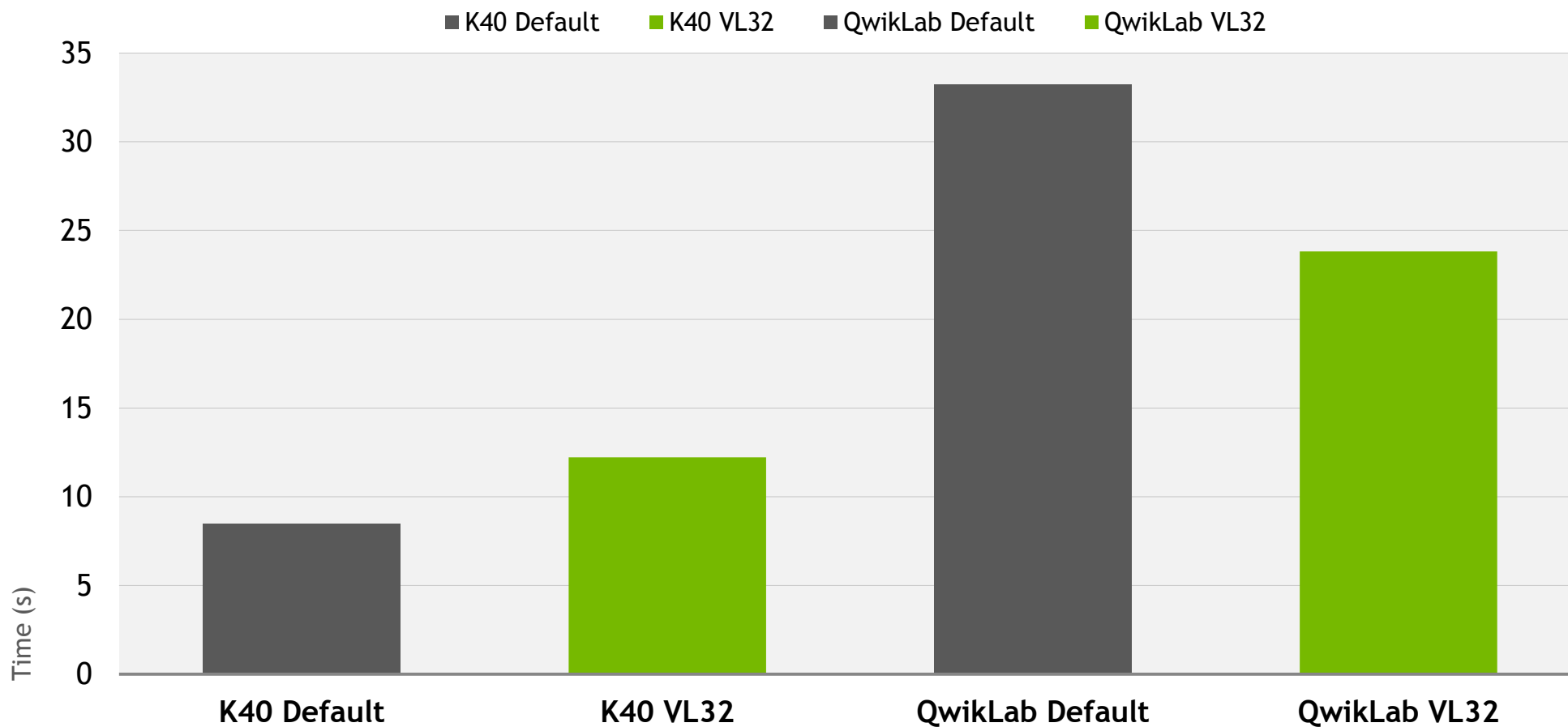
- ▶ Inform the compiler that on NVIDIA devices it should use a vector length of 32 on the innermost loop.
- ▶ On NVIDIA devices, vector lengths *must* be multiples of 32 (up to 1024)

# Optimize Loops: Parallel Loop Vector Length

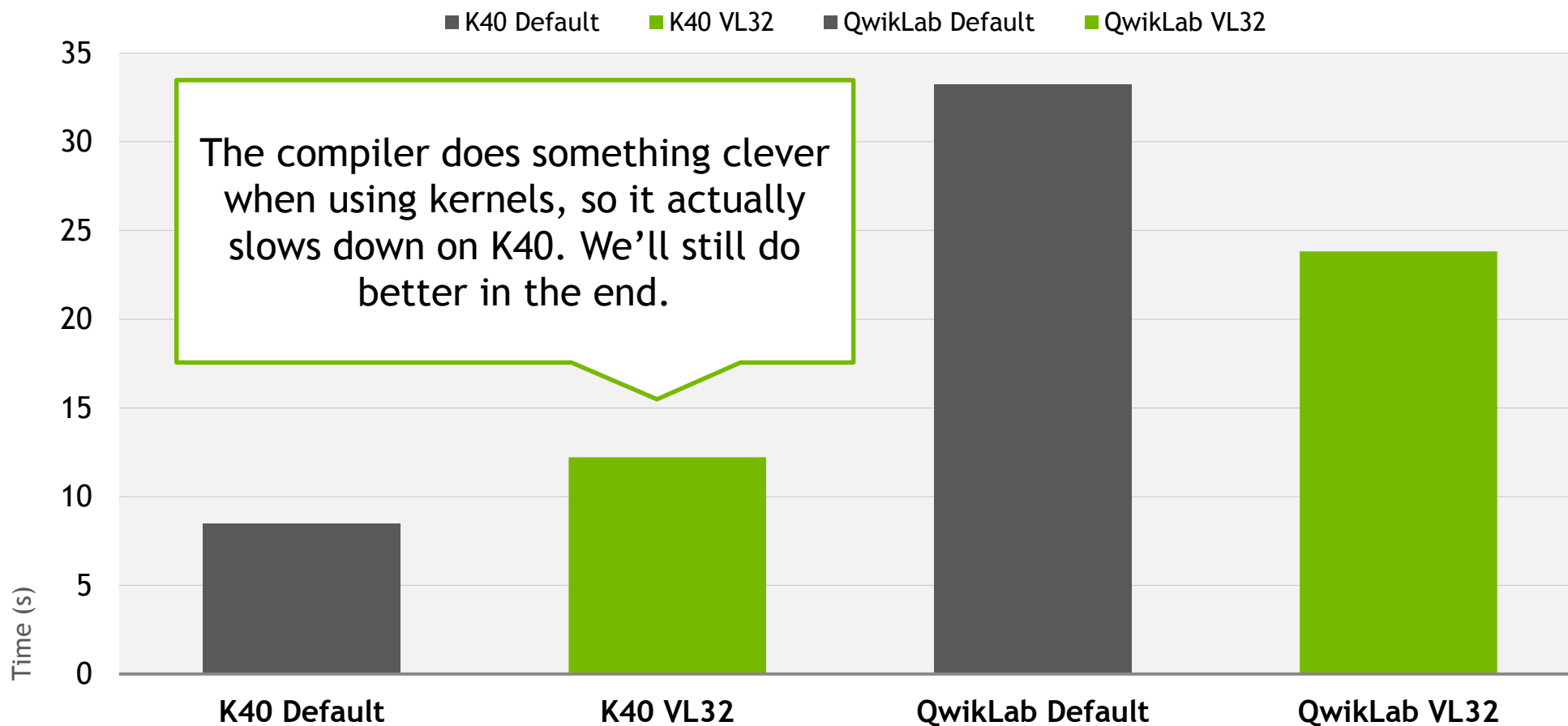
```
#pragma acc parallel loop
present(row_offsets,cols,Acoefs,xcoefs,ycoefs) \
        device_type(nvidia) vector_length(32)
for(int i=0;i<num_rows;i++) {
    double sum=0;
    int row_start=row_offsets[i];
    int row_end=row_offsets[i+1];
#pragma acc loop reduction(+:sum) \
        device_type(nvidia) vector
    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;
}
```

- ▶ When using **parallel loop**, the vector length is specified at the top of the region.
- ▶ The **vector** clause is then used to inform the compiler which loop to vectorize.

# Optimize Loops: Adjust Vector Length



# Optimize Loops: Adjust Vector Length



# Profiling with Visual Profiler

## Visual Profiler Guided Analysis

**GPU Utilization Is Limited By Block Size**

The kernel has a block size of 32 threads. This block size is likely preventing the kernel from fully utilizing the GPU. Device "Tesla K20c" can simultaneously execute up to 16 blocks on each SM. Because each block uses 1 warp to execute the block's 32 threads, the kernel is using only 16 warps on each SM. Chart "Varying Block Size" below shows how changing the block size will change the number of warps that can execute on each SM.

*Optimization: Increase the number of threads in each block to increase the number of warps that can execute on each SM.*

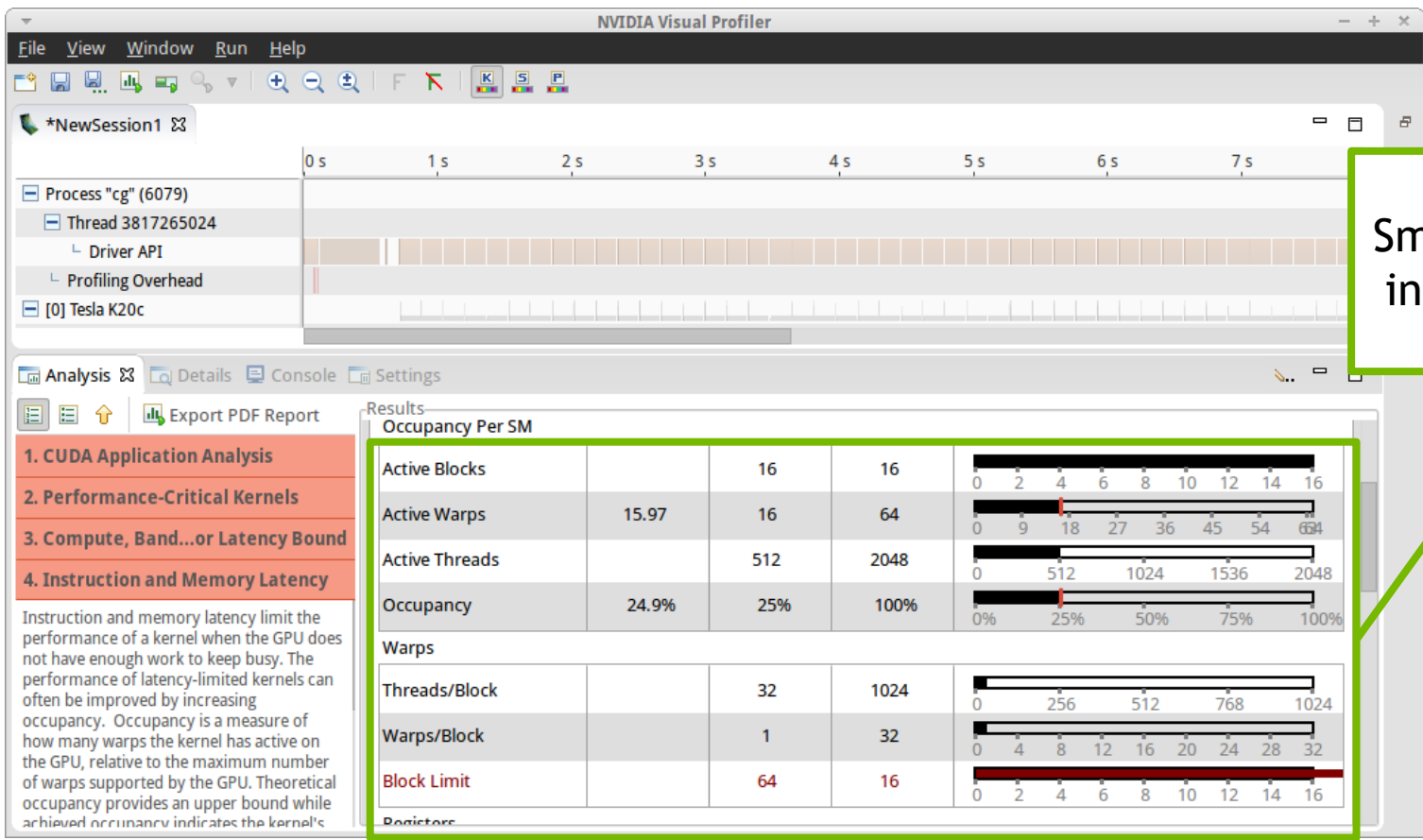
Variable	Achieved	Theoretical	Device Limit	Grid Size: [ 65535,1,1 ] (65535 blocks)Block Size: [
Occupancy Per SM				
Active Blocks		16	16	
Active Warps	15.97	16	64	
Active Threads		512	2048	

Performance limited by gang size.



# Profiling with Visual Profiler

## Visual Profiler Guided Analysis



Small gangs results in low occupancy.

# GPU Occupancy

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

Roughly speaking: Occupancy is...

How much parallelism *is* running / How much parallelism the hardware *could* run

- 100% occupancy is not required for, nor does it guarantee best performance.
- Less than 50% occupancy is often a red flag, our occupancy is 25%

# Profiling with Visual Profiler

## Visual Profiler Guided Analysis

Results

### Occupancy Per SM

Active Blocks		16	16	
Active Warps	15.97	16	64	
Active Threads		512	2048	
Occupancy	24.9%	25%	100%	

The GPU could run this...

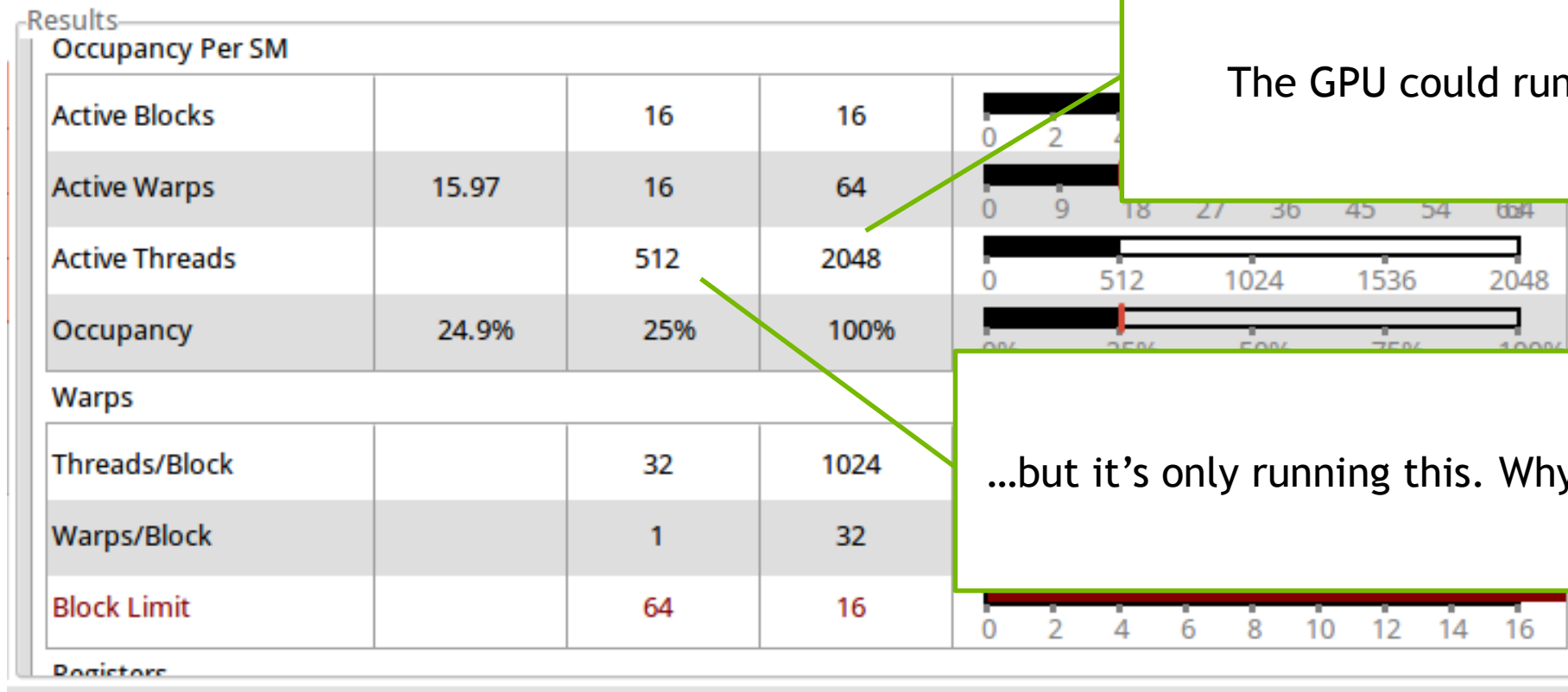
### Warps

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

### Registers

# Profiling with Visual Profiler

## Visual Profiler Guided Analysis



The GPU could run this...





...but it's only running this. Why?

# Profiling with Visual Profiler

## Visual Profiler Guided Analysis


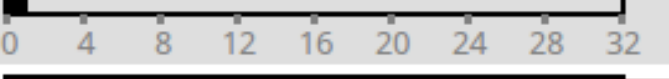

Results

### Occupancy Per SM

Active Blocks		16	16	
Active Warps	15.97	16	64	
Active Threads		512	2048	
Occupancy	24.9%	25%	100%	

Each gang could have as many as 1024 threads...

### Warps

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





### Registers

# Profiling with Visual Profiler

## Visual Profiler Guided Analysis


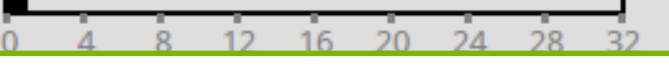
Results

### Occupancy Per SM

Active Blocks		16	16	
Active Warps	15.97	16	64	
Active Threads		512	2048	
Occupancy	24.9%	25%	100%	

Each gang could have as many as 1024 threads...

### Warps

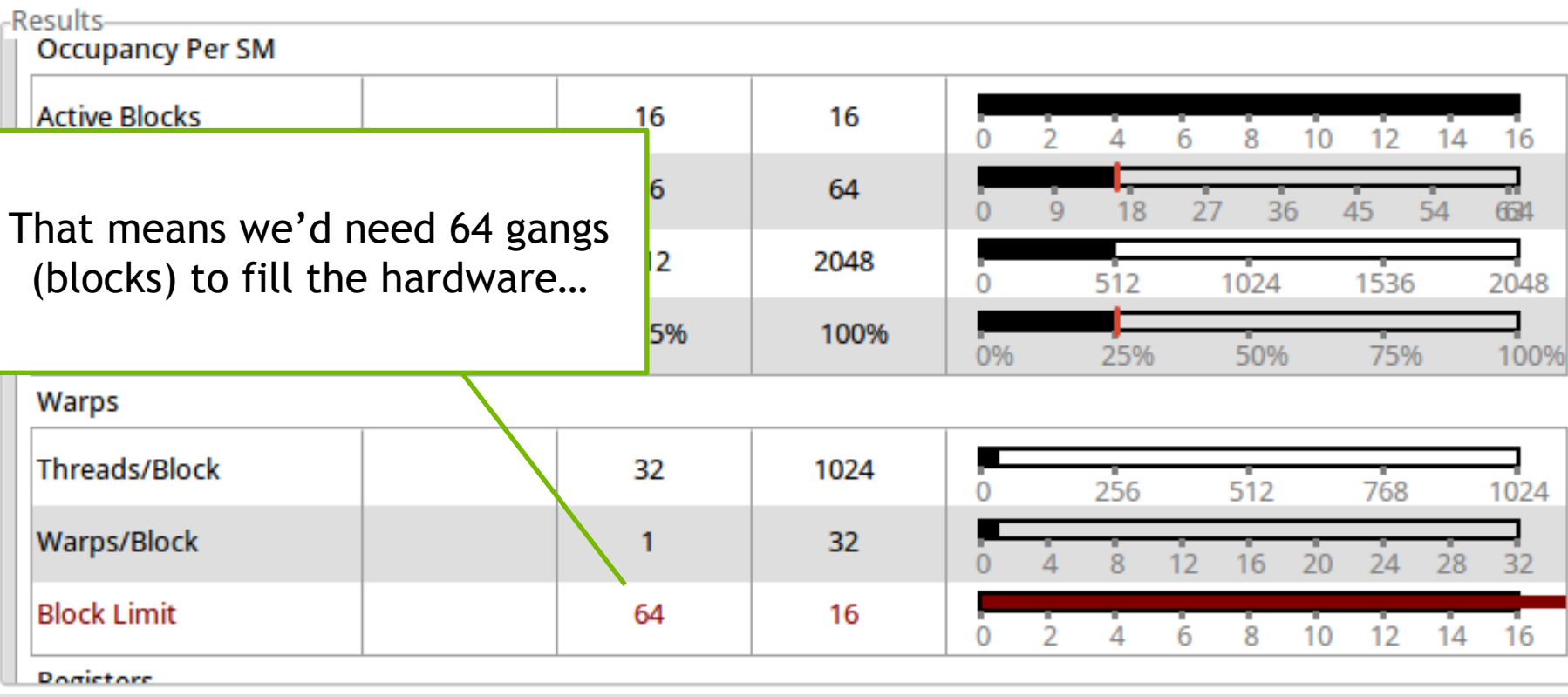
Threads/Block		32	1024	
Warps/Block		1	32	
Block Limit		64	16	

...but only has 32 now.

Registers

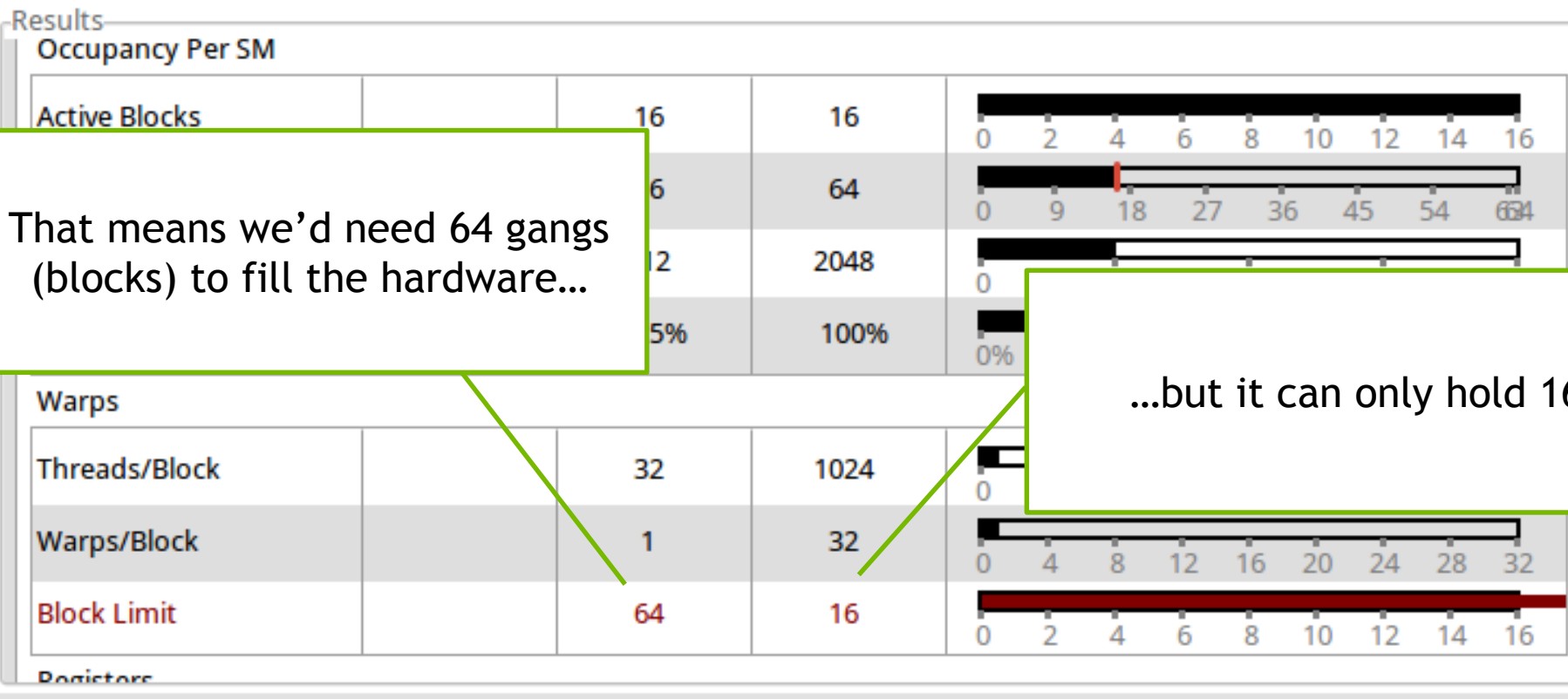
# Profiling with Visual Profiler

## Visual Profiler Guided Analysis



# Profiling with Visual Profiler

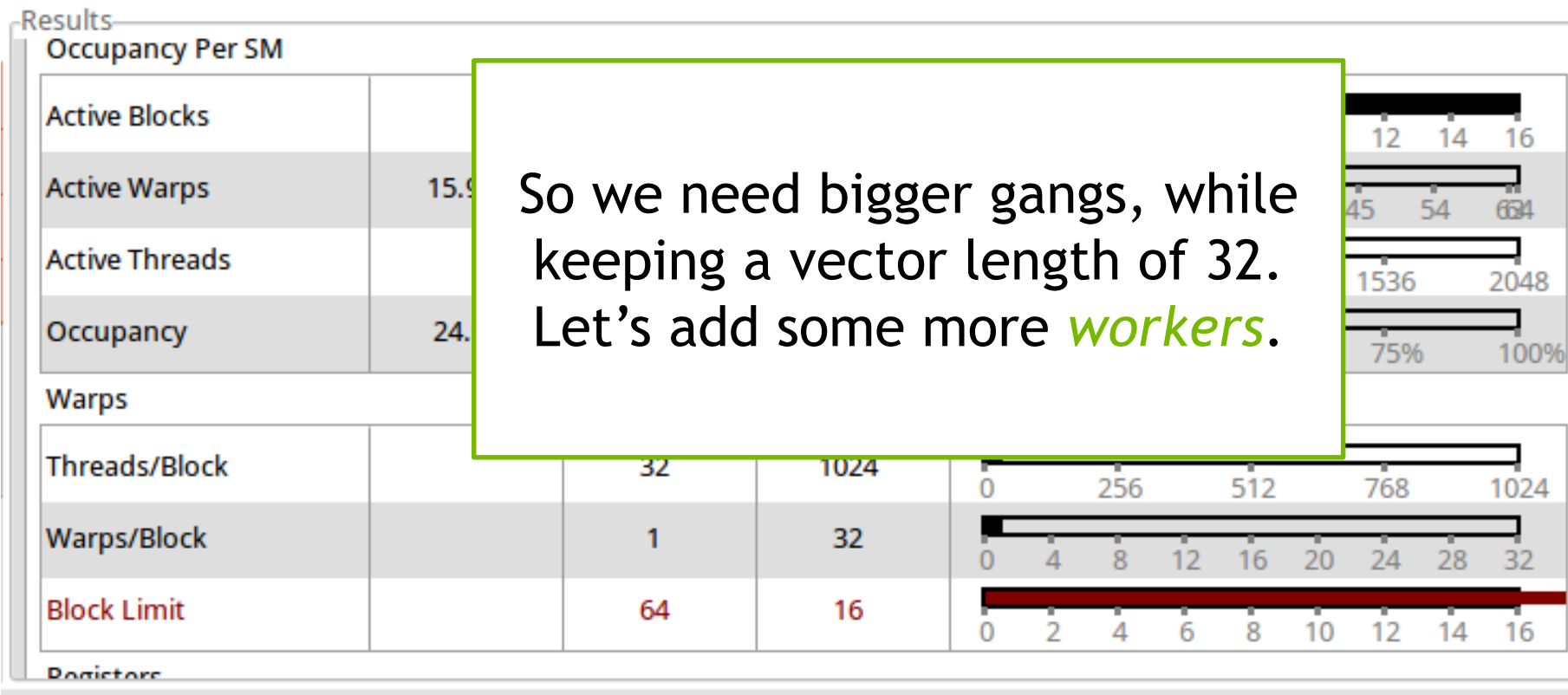
## Visual Profiler Guided Analysis





# Profiling with Visual Profiler

## Visual Profiler Guided Analysis



# Optimize Loops: Adding Workers

```
#pragma acc kernels \
    present(row_offsets,cols,Acoefs,xcoefs,ycoefs)
{
#pragma acc loop device_type(nvidia) gang worker(32)
    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 device_type(nvidia) vector(32)
        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;
    }
}
```

- ▶ Inform the compiler that on NVIDIA GPUs, the outer loop should be broken over gangs and workers, with 32 workers per gang.
- ▶ Gang size is Workers X Vector Length
- ▶ This must be no more than 1024 on NVIDIA GPUs

# Optimize Loops: Parallel Loop Adding Workers

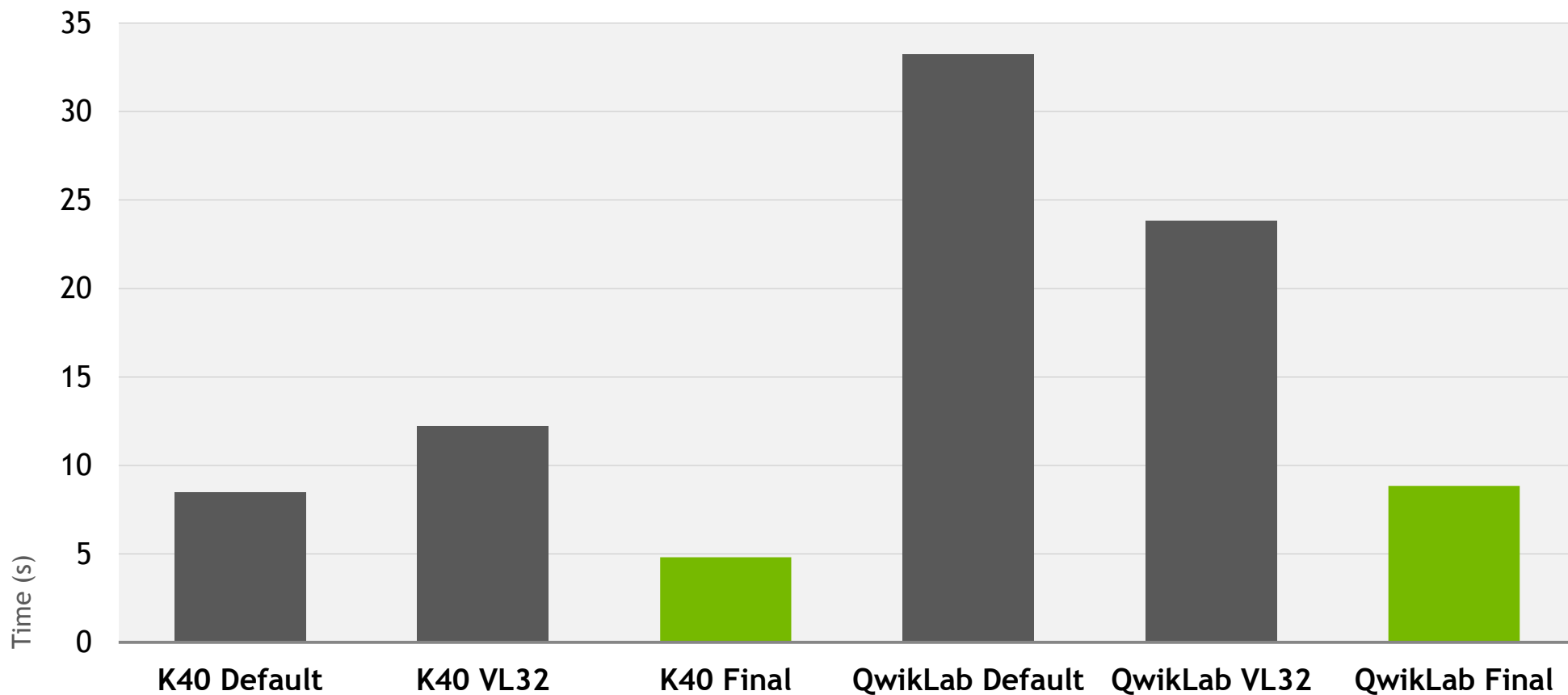
```
#pragma acc parallel loop
present(row_offsets,cols,Acoefs,xcoefs,ycoefs) \
        device_type(nvidia) vector_length(32) \
        gang worker num_workers(32)
for(int i=0;i<num_rows;i++) {
    double sum=0;
    int row_start=row_offsets[i];
    int row_end=row_offsets[i+1];
#pragma acc loop reduction(+:sum) \
        device_type(nvidia) vector
    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;
}
```

- ▶ When using **parallel loop**, the number of workers is specified at the top of the region.
- ▶ The **gang** and **worker** clauses are then used to inform the compiler to break the loop over workers and gangs.

# Optimize Loops: Compiler Output

```
matvec(const matrix &, const vector &, const vector &):  
    8, include "matrix_functions.h"  
    15, Generating present(row_offsets[:], cols[:], Acoefs[:], xcoefs[:], ycoefs[:])  
    17, Loop is parallelizable  
        Accelerator kernel generated  
        Generating Tesla code  
    17, #pragma acc loop gang, worker(32) /* blockIdx.x threadIdx.y */  
    22, #pragma acc loop vector(32) /* threadIdx.x */  
    26, Sum reduction generated for sum  
    22, Loop is parallelizable
```

# Optimize Loops: Final Performance



# Additional Loop Optimizations

Important clauses not used in this example

`collapse(N)`

Take the next  $N$  *tightly nested* loops and turn them into 1, flattened loop.

- This is really useful when you have many ( $> 2-3$ ) nested loops or when your loops are very small.

`tile(N[,M,...])`

Break the next loops into tiles/blocks before parallelizing the loops. (See lecture 1 for an example)

- This is useful for algorithms with high locality, because it encourages reuse of nearby data within each tile.

# In Summary

In this lecture we discussed the final 2 steps of accelerating an application using OpenACC

- Express Data Movement - Move your data to/from the GPU at a high enough level that the arrays can be reused between functions. (PCIe transfers are expensive!)
- Optimize Loops - Apply your knowledge of the code plus feedback from profiling tools to adjust the way that loop iterations are distributed to the hardware.

In the next lecture we'll discuss asynchronous execution as a way to further improve data movement and performance and also how to interoperate with accelerated libraries and CUDA.

# Next Steps & Homework



# Homework

This week's homework will build upon the previous lab by applying the two steps discussed today: Express Data Movement and Optimize Loops.

Go to <http://bit.ly/nvoacclab3> from your web browser to take the free lab, or download the code from <https://github.com/NVIDIA-OpenACC-Course/nvidia-openacc-course-sources/> to do the lab on your own machine.

If you have not already completed the previous labs, it's highly recommended that you do <http://bit.ly/nvoacclab2> first.

# Office Hours Next Week

Next week's session will be an office hours session.

Bring your questions from this week's lecture and homework to next week's session.

If you can't wait until then, post a question on StackOverflow tagged with openacc.

# Course Syllabus

Oct 1: Introduction to OpenACC

Oct 6: Office Hours

Oct 15: Profiling and Parallelizing with the OpenACC Toolkit

Oct 20: Office Hours

Oct 29: Expressing Data Locality and Optimizations with OpenACC

Nov 3: Office Hours

Nov 12: Advanced OpenACC Techniques

Nov 24: Office Hours