OpenACC

Open Accelerators

Matthew Silva
Professor Alvarez
CSC 415
9 April 2018

“What is OpenACC?

OpenACC is a user-driven **directive-based performance-portable parallel programming model** designed for scientists and engineers interested in **porting their code**s to a wide-variety of **heterogeneous [CPU/GPU] HPC hardware** platforms and architectures with **significantly less programming effort than required with a low-level model**.”

Who remembers the programming effort required for other low-level tasks?

(AVX isn’t even very low-level.)

Source: “What Is OpenACC?”

Heterogeneous Computing Review

“ ...**conventional** and **specialized** processors work cooperatively.”

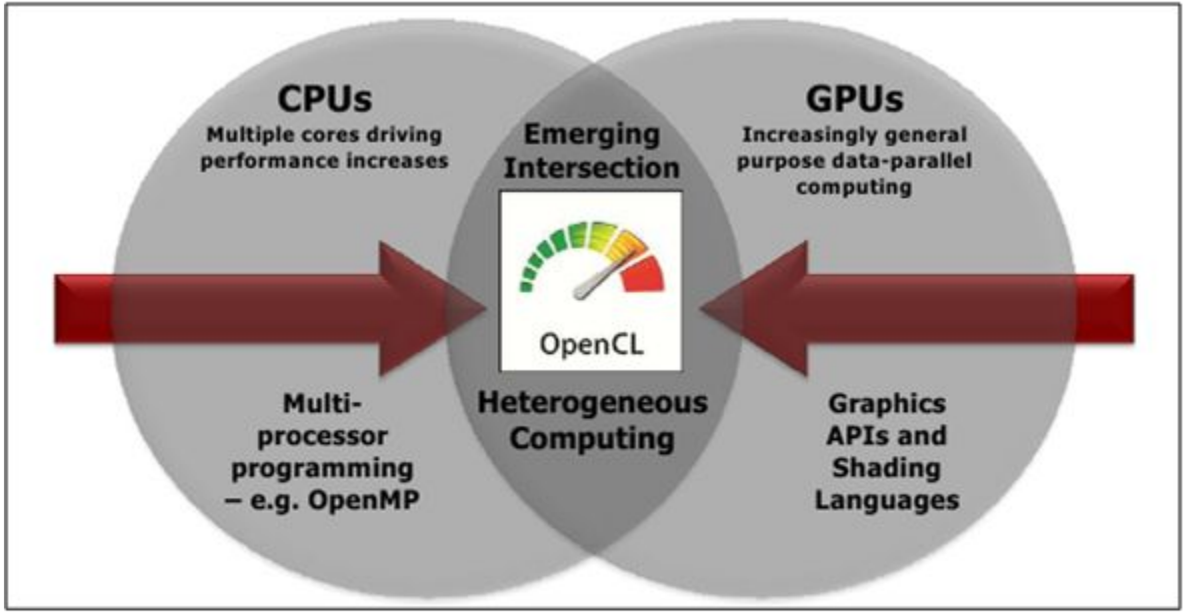
“Conventional processors” generally refers to a regular CPU.

“Specialized processors” may refer to a GPU or other processor meant to perform a certain type of task.

Heterogeneous computing allows conventional and specialized processors to team up, each solving the parts of a problem that they are best at.

OpenACC will accelerate tasks by assigning them to the specialized processor.

Heterogeneous Computing Review



Source: “Research & Innovation”

OpenCL vs OpenACC

OpenCL and OpenACC both deal with parallelizing code for heterogeneous computing applications.

OpenACC is much easier to use, since it relies almost fully on compiler directives that find and create parallelism for you.

(Much less work, but also less effective.)

You can fine-tune OpenACC directives, but many feel that this removes the abstraction that creates appeal for OpenACC over OpenCL.

Source: Melonakos

Directives (pragma)

```
#pragma acc data copy(A) create(Anew)
while ( error > tol  &&  iter  <  iter_max )  {
  error = 0.0;
#pragma acc kernels {
#pragma acc loop independent collapse(2)
  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]);
       error = max ( error, fabs (Anew [j] [i] - A [j] [i]));
      }
    }
  }
}
```

Source: “What is OpenACC?”

Directives Review

Directives are comments that act as commands to the compiler.

Directives are NOT code! (At least, not code in the language being compiled.)

Directives tell the compiler to try to compile the code a certain way (make it faster.)

Directive Results (Compiler Output)

```
 20, Generated vector simd code for the loop containing reductions
     Generated 2 prefetch instructions for the loop
     FMA (fused multiply-add) instruction(s) generated
```

```
matvec(const float *, const float *, unsigned int, unsigned int, float *):
     20, Accelerator kernel generated
         Generating Tesla code
         22, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         24, Generating implicit reduction(+:vecSum)
     20, Generating implicit copyin(A[in:n],x[:n])
```

OpenACC Directives are a portable way to show OpenACC where to automatically generate vectorized code for a specified accelerator

Here, tesla was selected while compiling.

Directive Results (Compiler Output)

```
 20, Generated vector simd code for the loop containing reductions
     Generated 2 prefetch instructions for the loop
     FMA (fused multiply-add) instruction(s) generated
```

```
matvec(const float *, const float *, unsigned int, unsigned int, float *):
     20, Accelerator kernel generated
         Generating Tesla code
         22, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         24, Generating implicit reduction(+:vecSum)
     20, Generating implicit copyin(A[in:n],x[:n])
```

Notice accelerator-specific parallelism being created with SIMD. The tesla is able to use SIMD.

An OpenACC gang is a thread block organized to work together on a task, commonly used for GPUs and other manycore accelerators.

The copyin is used to reduce the disadvantage of accelerator memory latency with the CPU (Move all the data to the GPU at once).

Runtime Functions

OpenACC also provides some runtime functions (like AVX and AVX2).

Examples:

- acc_alloc()
- acc_free()

Source: “OpenACC API”

Who can use OpenACC?

“The OpenACC specification supports **C, C++, Fortran** programming languages and multiple hardware architectures including **X86 & POWER CPUs, NVIDIA GPUs, and Xeon KNL** in the near future.”

Many compilers (both free and commercial) now support OpenACC directives.

Sources: “Get Started” and “Downloads & Tools”

Who can use OpenACC?

OpenACC is generally used for scientific applications.

“ ...take advantage of compilers and libraries to quickly accelerate your codes with CPUs and GPUs so that you can spend more time on real breakthroughs.”

Directives are an abstraction that removes much of the need for knowledge of hardware acceleration.

Scientists get the benefit of acceleration without having to change their programs or know about parallelism/hardware.

Source: “Get Started”

How to use OpenACC

OpenACC.org breaks up usage into three steps that can make optimizing a program for heterogeneous computing systems a simple task:

1. Analyze
2. Parallelize
3. Optimize

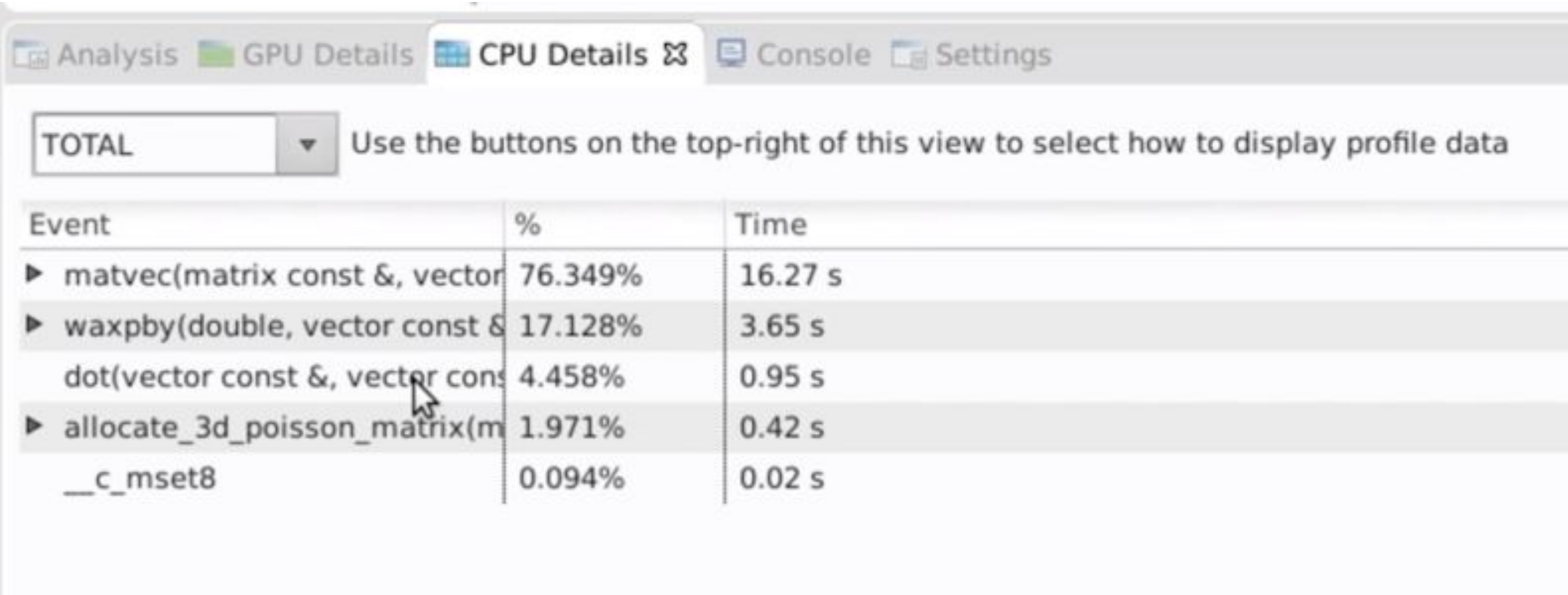
Analyze

OpenACC.org recommends using profiling tools to figure out where a program could benefit from acceleration.

Profiling tools run an executable, recording how much execution time is spent in each function.

Profiling tools can also compare the executable to the source code to see where a compiler made optimizations.

Analyze



Analysis | GPU Details | CPU Details | Console | Settings

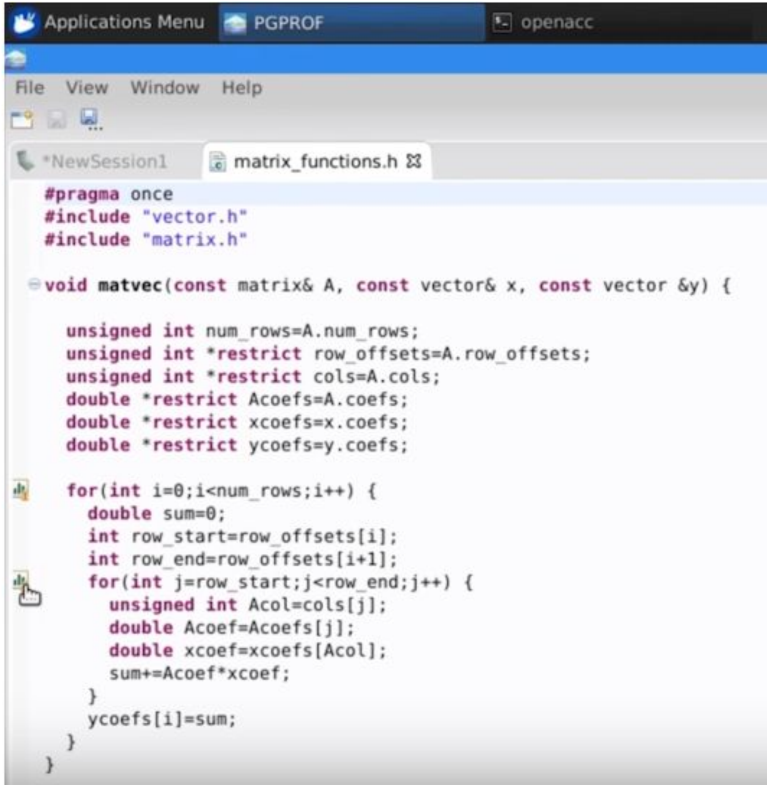
TOTAL — Use the buttons on the top-right of this view to select how to display profile data

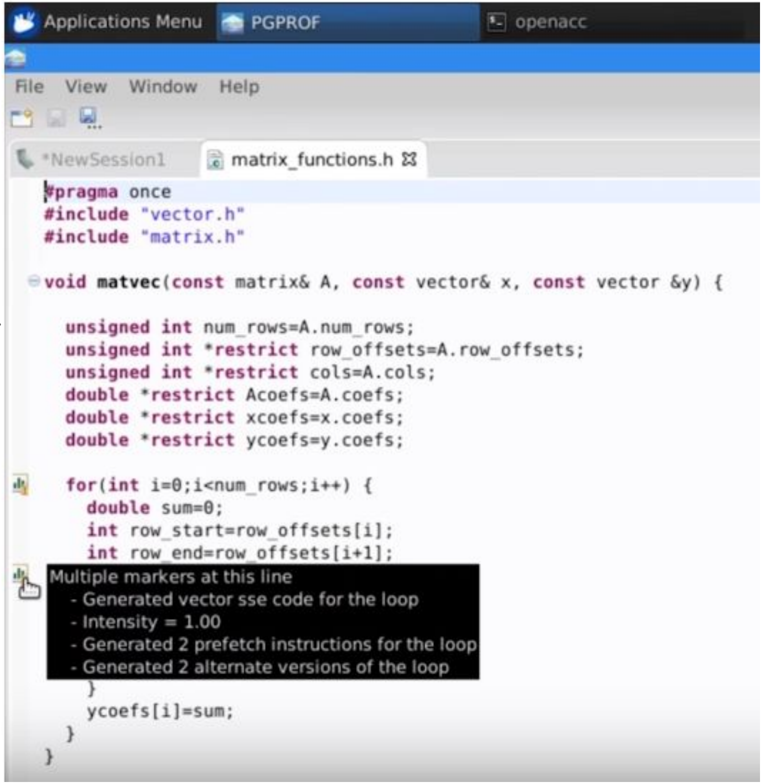
Event	%	Time
▶ matvec(matrix const &, vector	76.349%	16.27 s
▶ waxpby(double, vector const &	17.128%	3.65 s
dot(vector const &, vector cons	4.458%	0.95 s
▶ allocate_3d_poisson_matrix(m	1.971%	0.42 s
__c_mset8	0.094%	0.02 s

" ...by accelerating the three most time consuming routines, I'll accelerate nearly 98% of the application."

Source: "Step 1: Analyze" (2:22)

Analyze





The profiler examines the source code to show where and how the compiler performed optimizations.

Source: “Step 1: Analyze” (2:37)

Parallelize

Knowing where the compiler made optimizations from the profiler, a researcher has an idea of where the most work is being done, as well as loops/operations that can likely be parallelized.

With OpenACC, accelerating the program is now as easy as adding the proper compiler directives.

Parallelize

```
  }
  return sum;
}

void waxpby(double alpha, const vector &x, double beta, const
  unsigned int n=x.n;
  double *restrict xcoefs=x.coefs;
  double *restrict ycoefs=y.coefs;
  double *restrict wcoefs=w.coefs;

#pragma acc parallel loop
  for(int i=0;i<n;i++) {
    wcoefs[i]=alpha*xcoefs[i]+beta*ycoefs[i];
  }
}
```

This pragma will accelerate the loop below it, automatically creating parallelism.

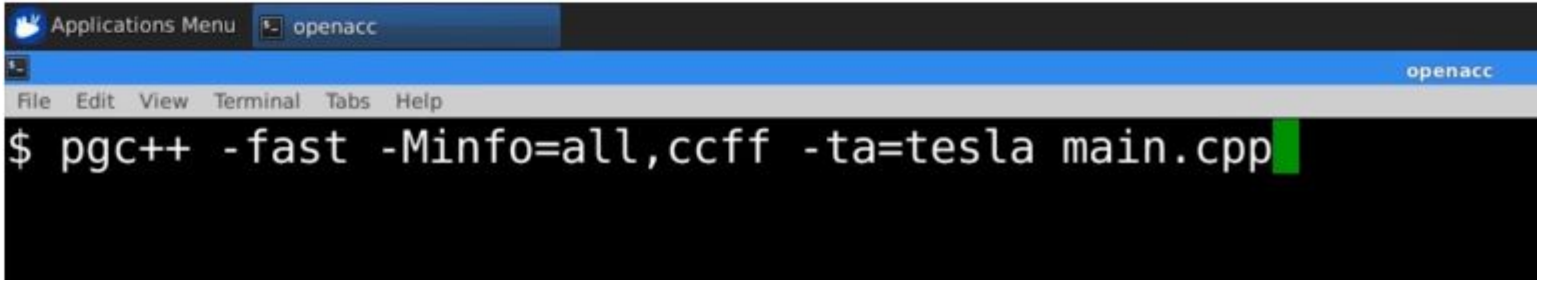
No need to code unrolling or separate accumulators! Compiler does it all for you!

Source: “Step 2: Parallelize” (0:33)

Parallelize

Very important to note that OpenACC will automatically tailor its parallelization to the target hardware.

No knowledge of your hardware is needed besides what kind of hardware it is.



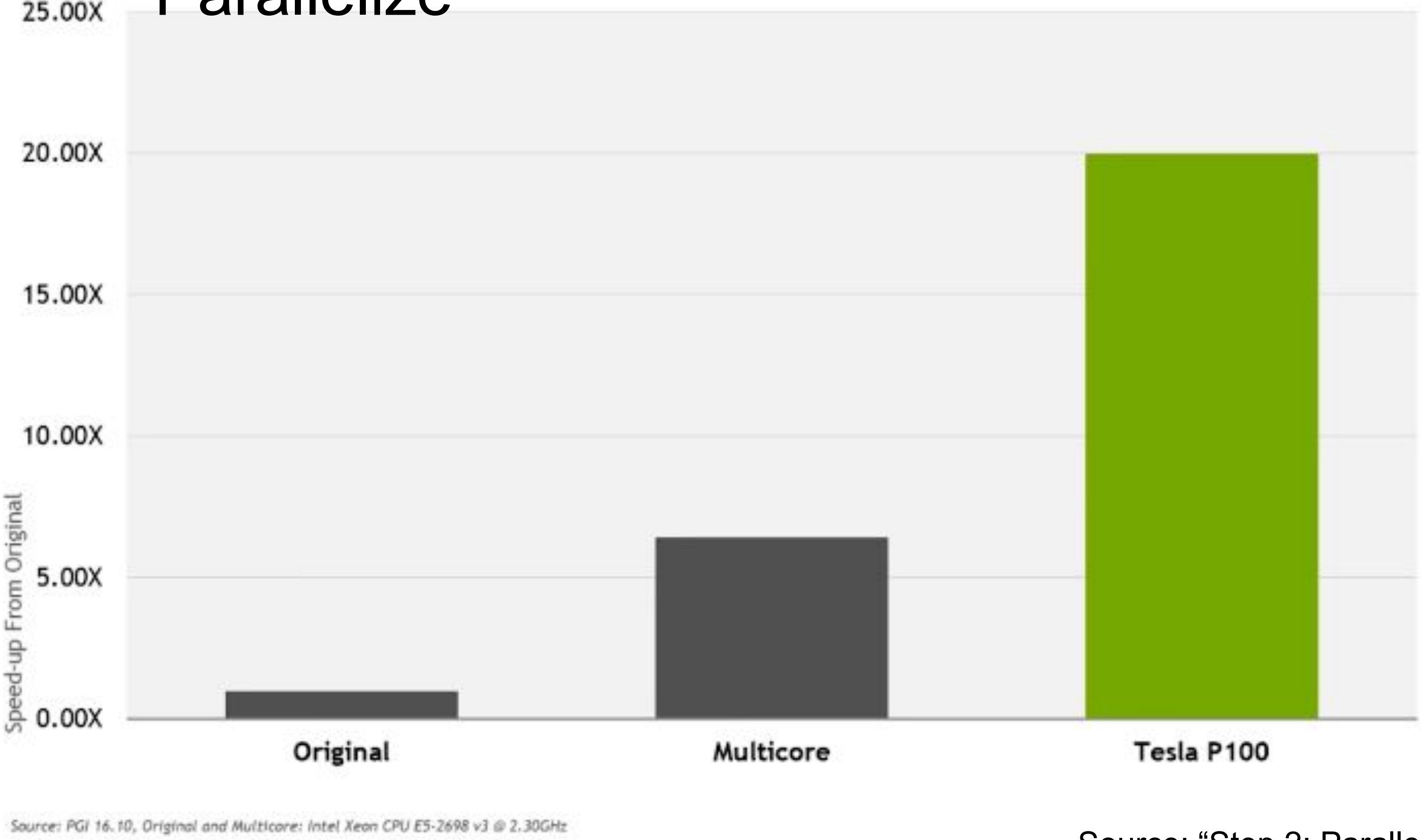
All the researcher has to do is specify their type of specialized processor (target accelerator) with -ta=<processor> when compiling their code.

Source: “Step 2: Parallelize”

Parallelize

The compiler will follow the given directives to produce code that is parallelized where requested for optimal use by the target accelerator when possible.

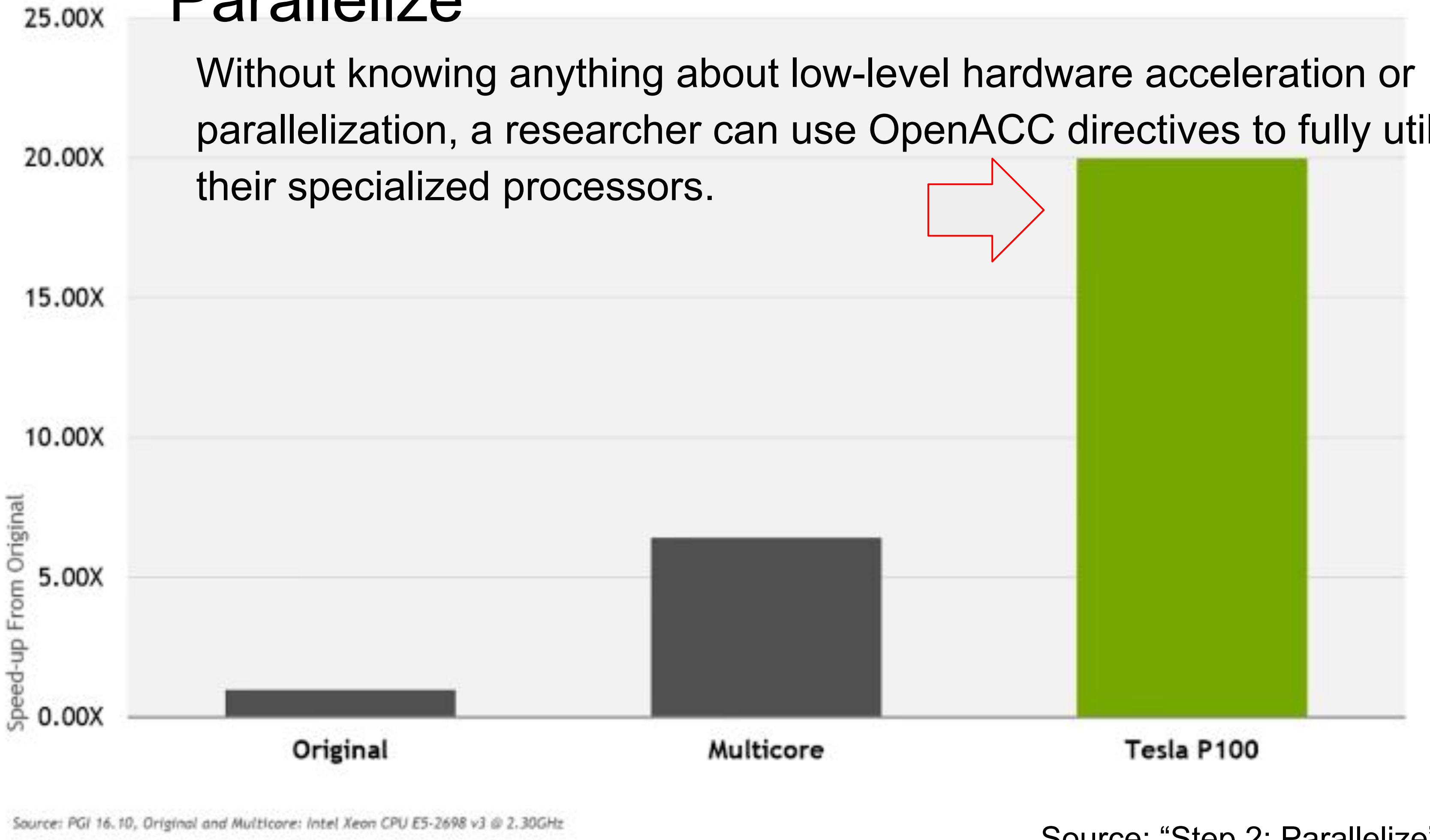
Parallelize



Source: “Step 2: Parallelize”

Parallelize

Without knowing anything about low-level hardware acceleration or parallelization, a researcher can use OpenACC directives to fully utilize their specialized processors.

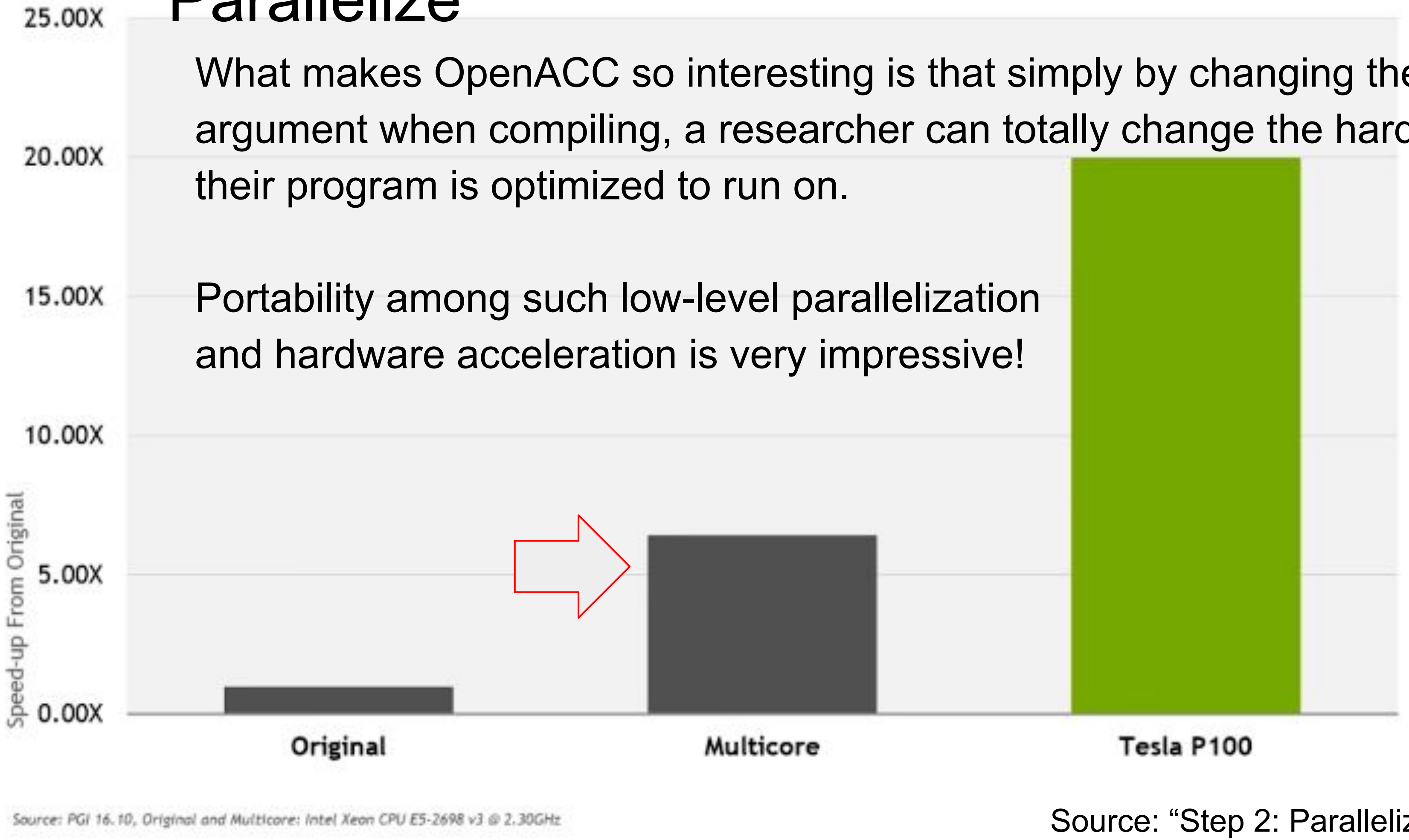


Source: “Step 2: Parallelize”

Parallelize

What makes OpenACC so interesting is that simply by changing the -ta argument when compiling, a researcher can totally change the hardware their program is optimized to run on.

Portability among such low-level parallelization and hardware acceleration is very impressive!



Source: “Step 2: Parallelize”

Parallelize

Inserting pragma directives according to the data given by a profiler can make a program ready to be ported with acceleration across a lot of hardware.

As in Assignment 1 Part 2, it would normally take a bit of research and tricky low-level implementation to get this kind of acceleration on a heterogeneous computing system.

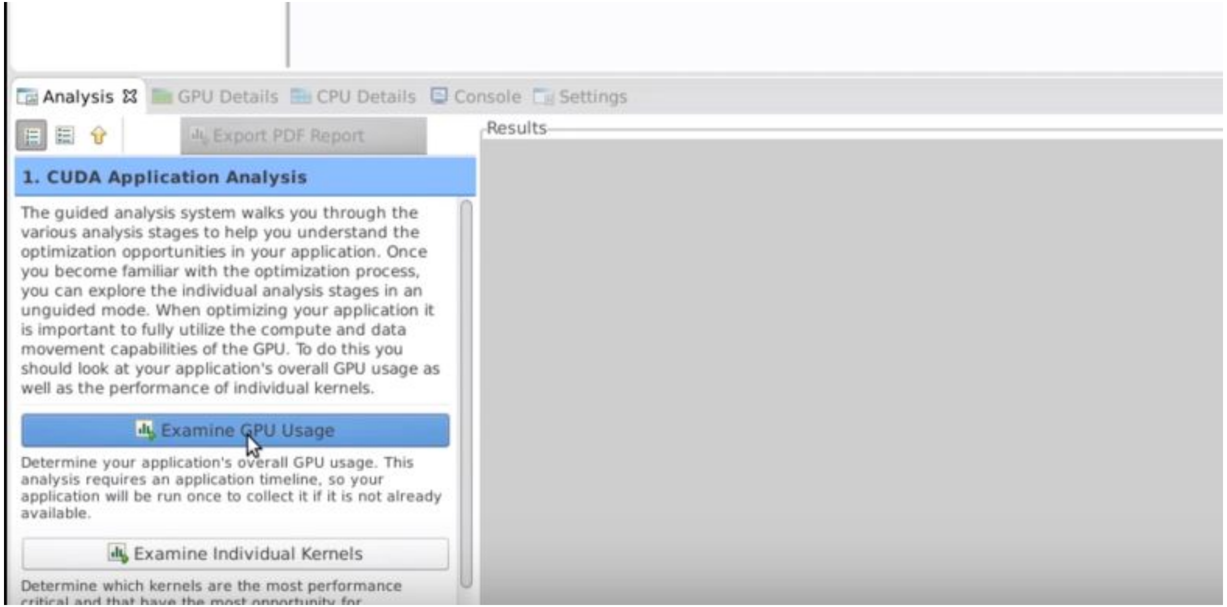
With OpenACC, simple directives allow you to achieve that acceleration across many sets of hardware easily.

Optimize

Optimizing with OpenACC means taking another look at the profiler and improving upon the compiler's best effort to parallelize automatically.

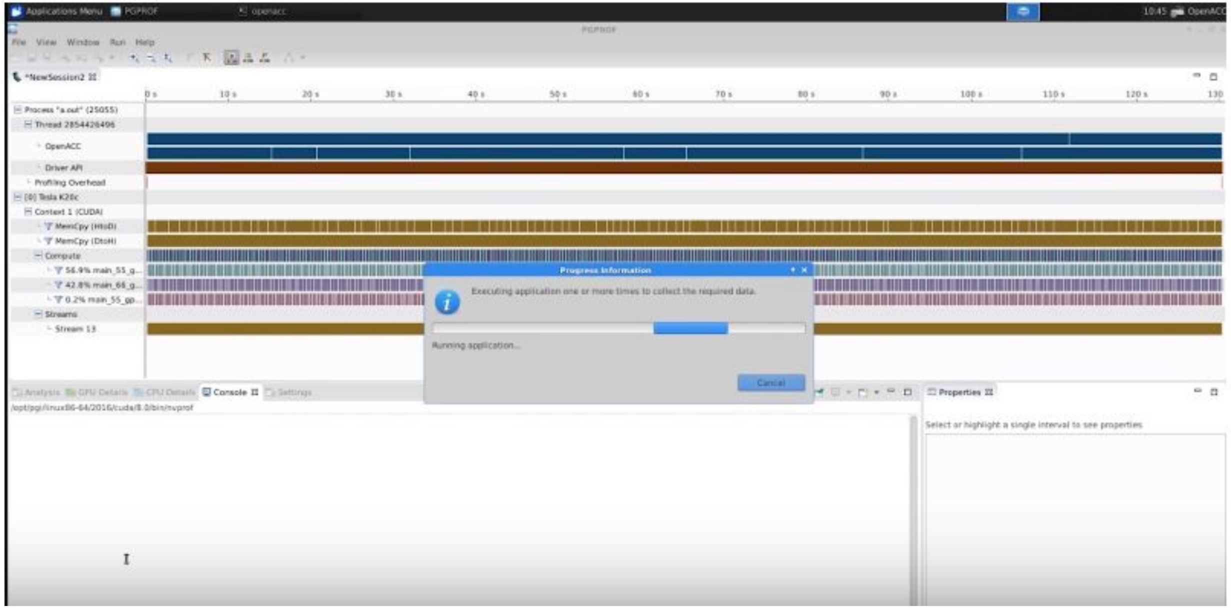
PGPROF will be used as an example.

Optimize



Here, the profiler will examine GPU usage.

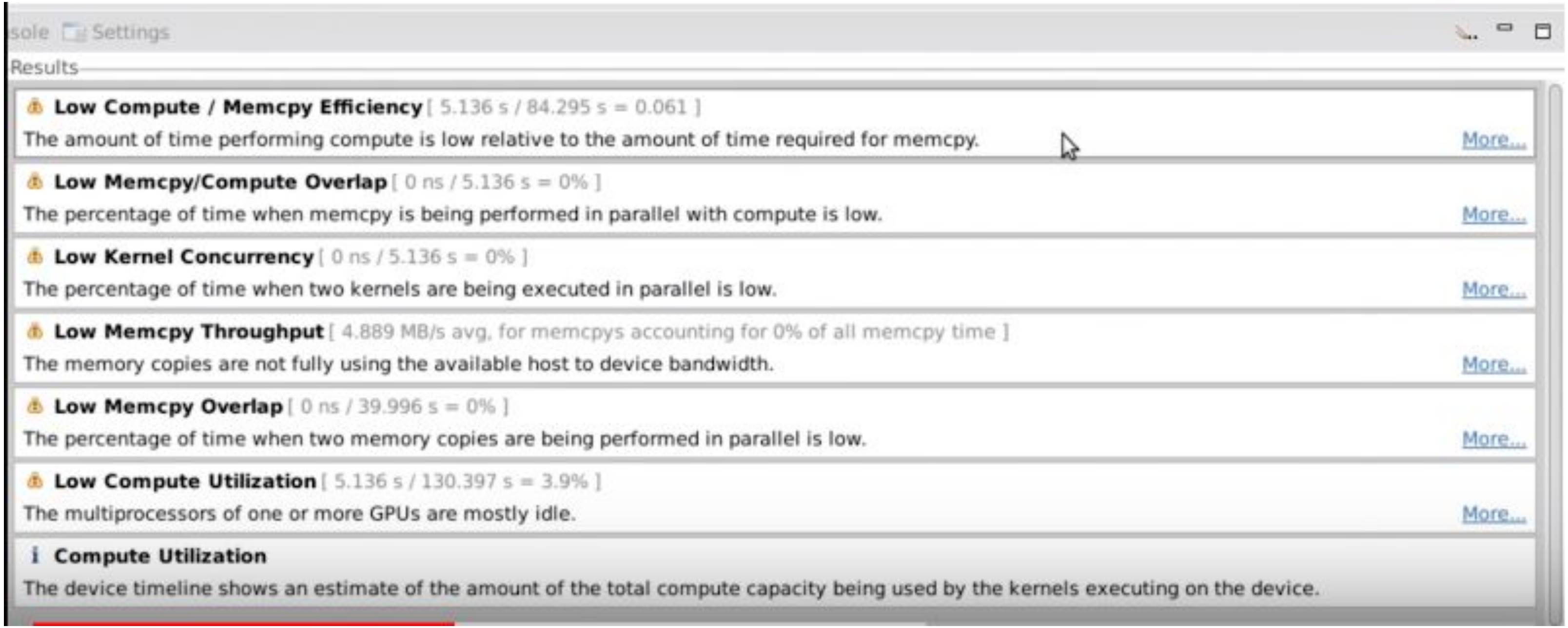
Optimize



"The guided analysis tool will rerun your code several times to help you determine what is limiting the performance of your application."

Source: "Step 3: Optimize" (1:04)

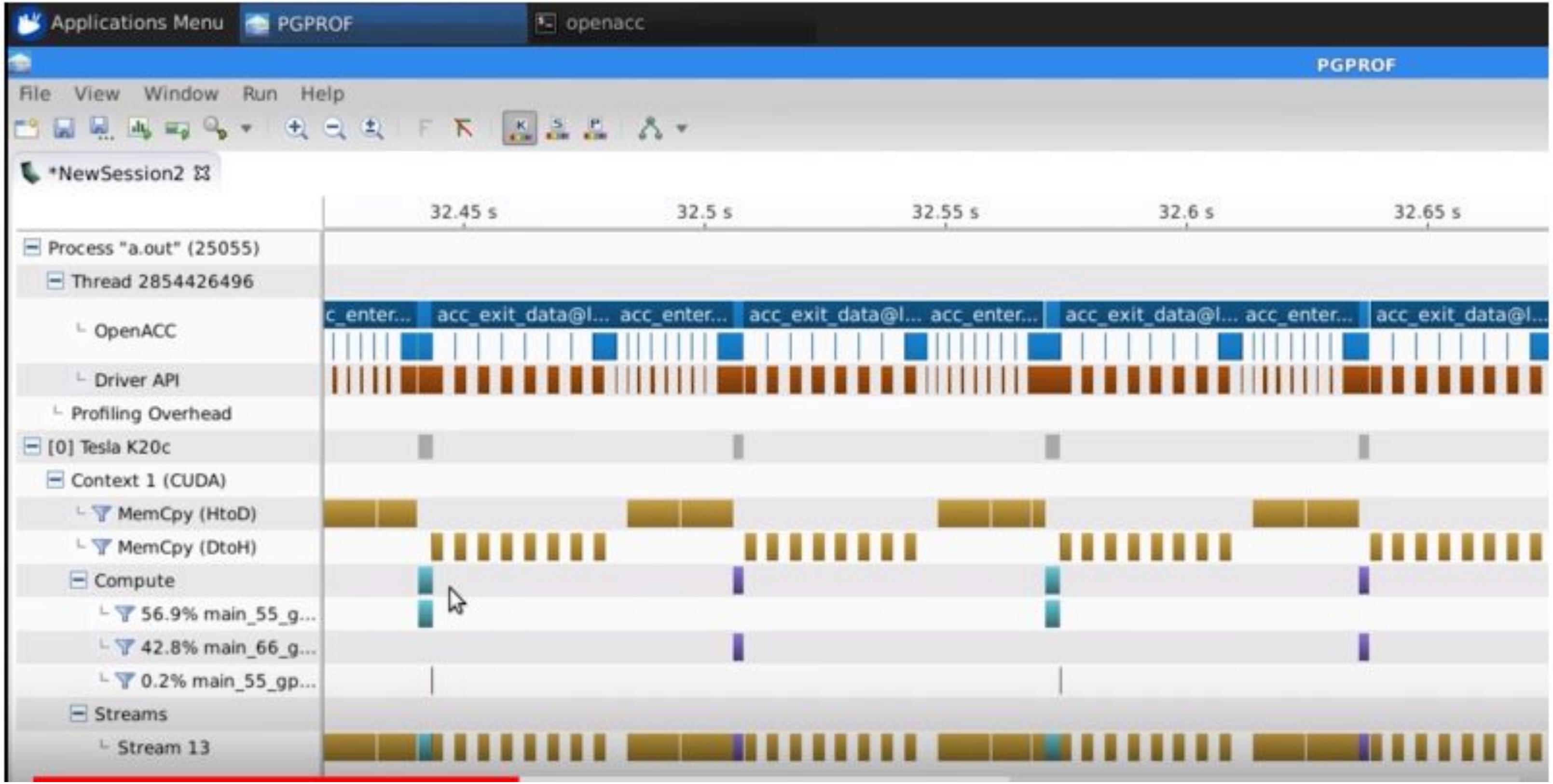
Optimize



The profiler found that data copies between the host and the accelerator were slowing down the code a lot. (Remember how we fixed this with AVX?)

Source: “Step 3: Optimize” (1:29)

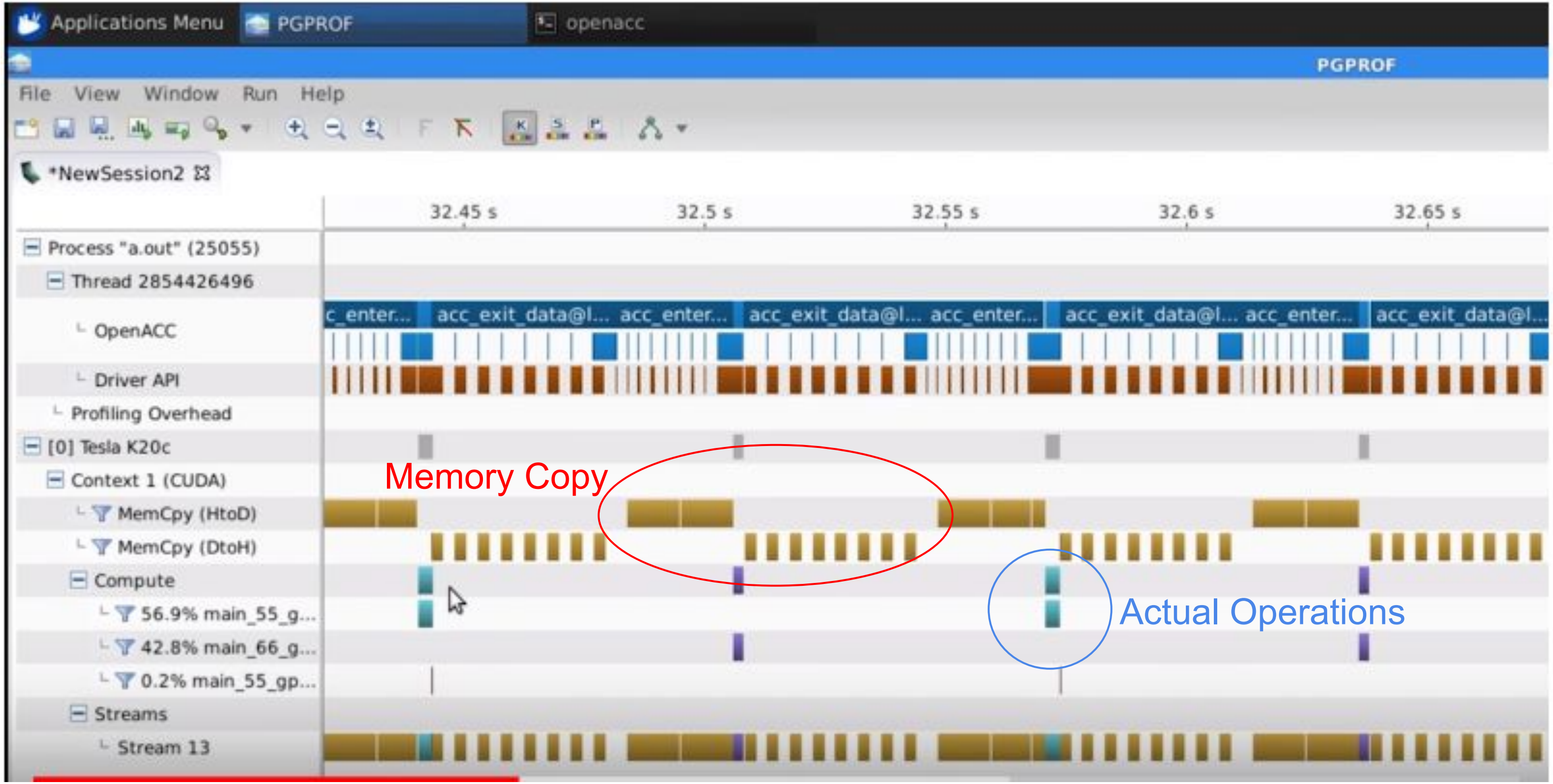
Optimize



The profiler even visualizes the execution time in charts, differentiating different accelerator operations as slices of time.

Source: “Step 3: Optimize”

Optimize



Note the very long MemCpy times compared to the Compute times!

Source: “Step 3: Optimize”

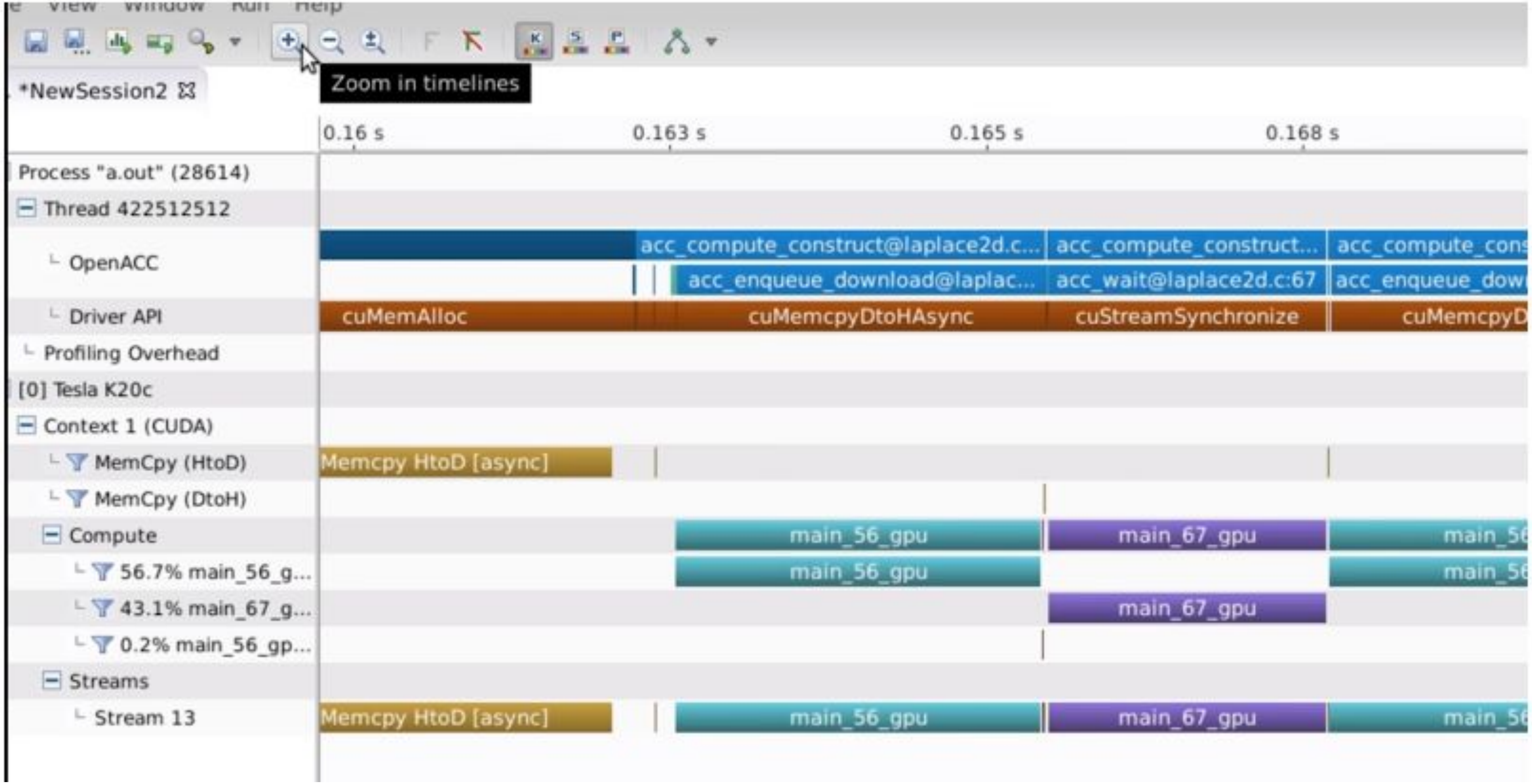
Optimize

```
#pragma acc data copy(A) create(Anew)
    while ( error > tol && iter < iter_max )
    {
        error = 0.0;

#pragma acc parallel loop reduction(max:error)
        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]);
                error = fmax( error, fabs(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];
```

Here, a data directive is used to manually tell the compiler how the data should get loaded to the accelerator (load input at start, unload result at end).

Source: “Step 3: Optimize” (2:18)

Optimize



That one directive "completely eliminated" the memory copy time! (Think about loadu_ps and load_ps).

Source: "Step 3: Optimize" (2:46)

Optimize

It’s time to look at what OpenACC did wrong!

```
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
```

OpenACC is parallelizing the loop with a vector length of 128, which means that it “operates on 128 loop iterations at one time.” (Think of our 256 bit AVX registers that operated 8 loop iterations at once.) Source: “Step 3: Optimize” (2:59)

Optimize

```
#pragma once
#include "vector.h"
#include "matrix.h"

void matvec(const matrix& A, const vector& x, const vector &y)
{

  unsigned int num_rows=A.num_rows;
  unsigned int *restrict row_offsets=A.row_offsets;
  unsigned int *restrict cols=A.cols;
  double *restrict Acoefs=A.coefs;
  double *restrict xcoefs=x.coefs;
  double *restrict ycoefs=y.coefs;

#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++) {
    double sum=0;
    int row_start=row_offsets[i];
    int row_end=row_offsets[i+1];
#pragma acc loop 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;
  }
}
```

However, the loop does not even iterate 128 times, making an attempt to parallelize 128 times over pointless.

Optimize

These are the loops in question:

```
#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++) {
    double sum=0;
    int row_start=row_offsets[i];
    int row_end=row_offsets[i+1];
#pragma acc loop reduction(+:sum)
    for(int j=row_start;j<row_end;j++) {
```

Optimize

“I’ll reduce the vector length to 32, since this is the smallest vector length supported on my GPU.”

```
#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++) {
    double sum=0;
    int row_start=row_offsets[i];
    int row_end=row_offsets[i+1];
#pragma acc loop reduction(+:sum)
    for(int j=row_start;j<row_end;j++) {
```

Note the necessity for knowledge of low-level implementations at this point.

Source: “Step 3: Optimize” (3:26)

Optimize

"I'll reduce the vector length to 32, since this is the smallest vector length supported on my GPU."

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

To fix what the compiler did wrong, we must explicitly vectorize the inner loop and specify a vector length of 32 in the outer loop. (Note that the outer loop directive specifies a "parallel region.")

Source: "Step 3: Optimize" (3:26)

Optimize

“Because 32 is really too little parallelism for this GPU, I’ll tell the compiler to also parallelize the outer loop.”

```
#pragma acc parallel loop copyout(ycoefs[:num_rows])   \
  copyin(Acoefs[:A.nnz],xcoefs[:num_rows],cols[:A.nnz])\
 vector_length(32) worker num_workers(4) gang
  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) vector
    for(int j=row_start;j<row_end;j++) {
      unsigned int Acol=cols[j];
```

This is done by assigning OpenACC “workers” and “gangs” to the loops, which further parallelizes the program.

Source: “Step 3: Optimize” (3:26)

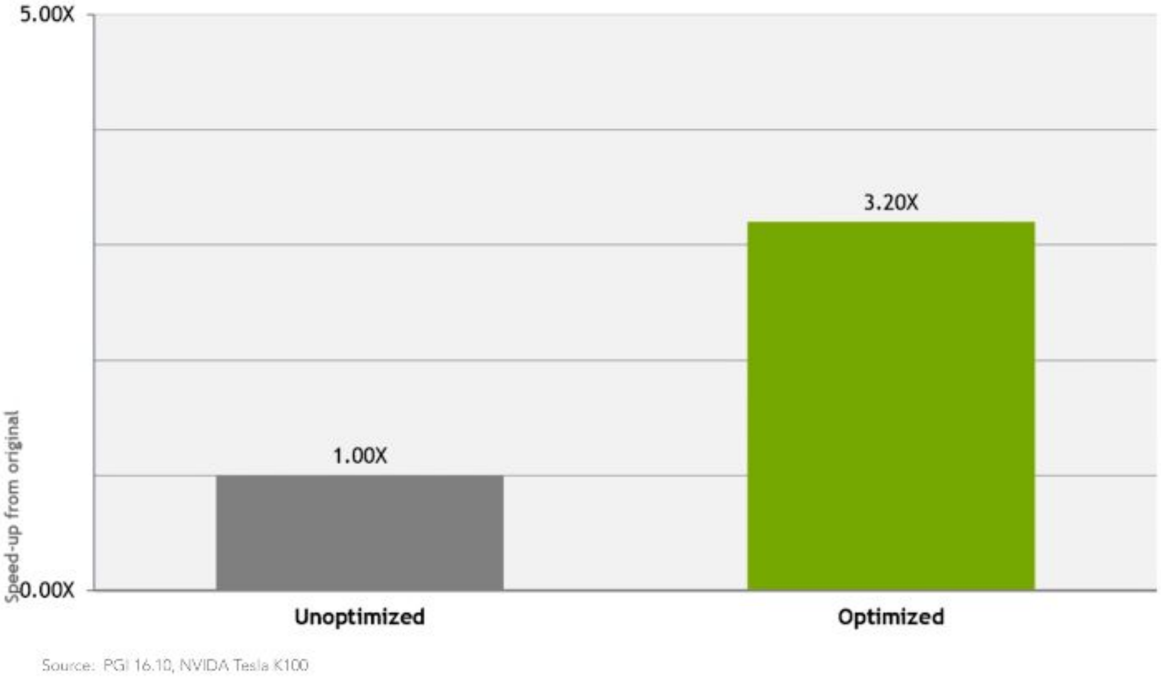
Optimize

Obviously, these optimizations require some low-level knowledge and are much more involved than just looking at the profiler output to find where to put directives.

“There’s no single optimization technique that will work on every code” (“Step 3: Optimize” 5:02).

Portability aside, some argue that it is more than worth the trouble to just learn how to use OpenCL instead, because the compiler is quite bad at automatically generating parallelization much of the time.

Optimize



"I roughly tripled the performance of my code by better mapping my loops to the hardware."

Source: "Step 3: Optimize" (4:48)

Optimize

“I roughly tripled the performance of my code by better mapping my loops to the hardware” (“Step 3: Optimize” 4:48).

True, but isn’t the whole point of OpenACC that your code is portable AND optimized between hardware?

At least OpenACC can handle everything under one set of directives though, and will always be portable across hardware.

Assignment 1: OpenACC Edition

We know firsthand how challenging it can be to write parallelized/vectorized code for a specific set of hardware.

Let’s see how easy it is with OpenACC.

(And whether it works as well as our solutions)

Analyze

Analysis | GPU Details (Summary) | CPU Details | OpenACC Details | Console | Settings

TOTAL — Use the buttons on the top-right of this view to select how to display profile data

Event	%	Time
random	67.647%	2,070.01357 ms
random_r	15.686%	480.00315 ms
main	10.784%	330.00216 ms
rand	2.941%	90.00059 ms
matvec(float const *, float cons	1.961%	60.00039 ms
???	0.98%	30.0002 ms

Analyze

```c
// Sequential implementation of a matrix vectorization
void matvec(const float *A, const float *x, unsigned int m, unsigned int n, float *b) {
    float vecSum;
    unsigned int in;
    for (unsigned int i = 0; i < m; i++) {
        vecSum = 0; // Reset sum to 0 after every row
        in = i*n;
        for (unsigned int j = 0; j < n; j++) {
            // Sum each product of corresponding matrix and vector values
            vecSum += (A[in + j] * x[j]);
        }
        // Record the sum in the resulting vector
        b[i] = vecSum;
    }
}
```

Analyze

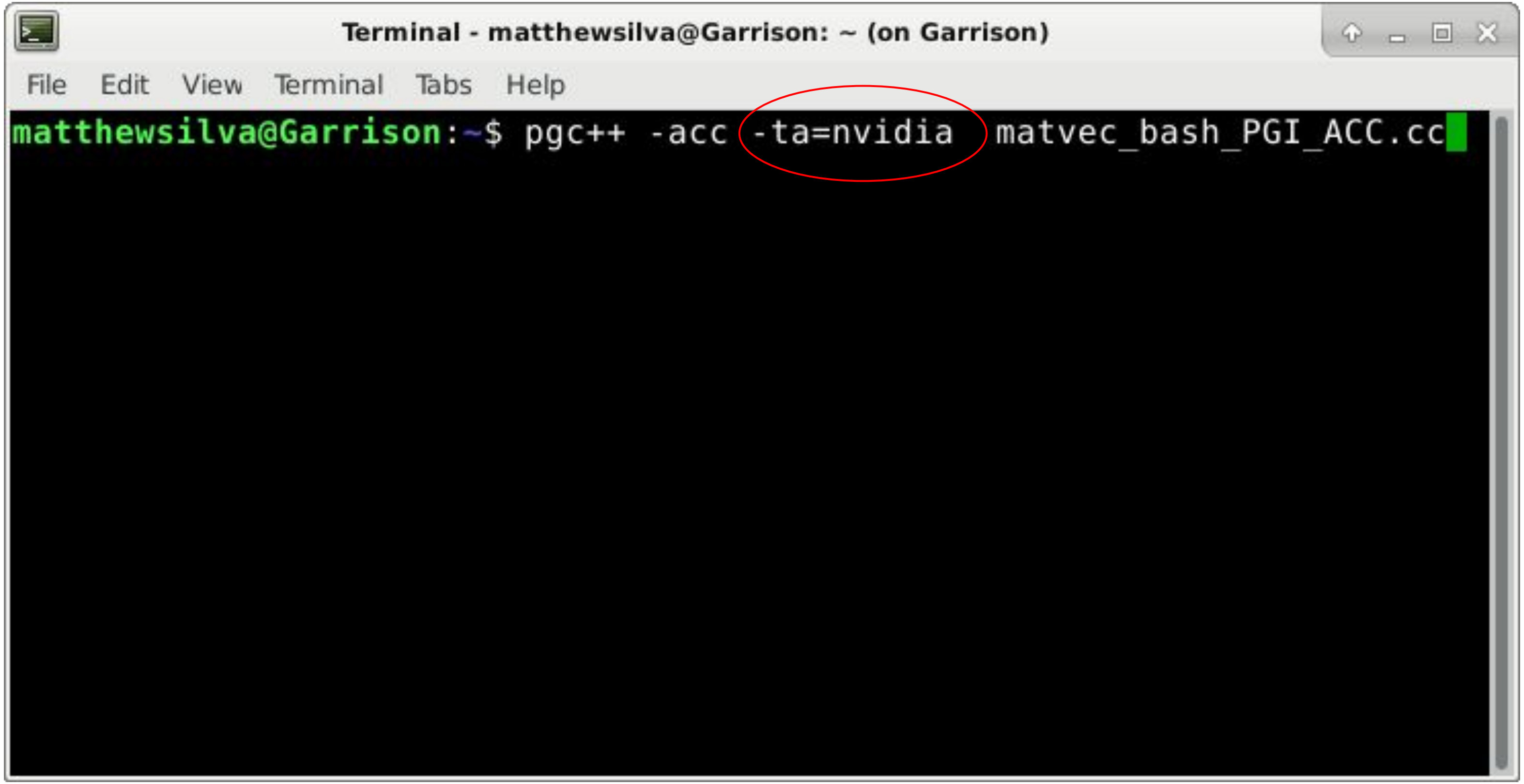
```
// Sequential implementation of a matrix vectorization
void matvec(const float *A, const float *x, unsigned int m, unsigned int n, float *b) {
    float vecSum;
    unsigned int in;
    for (unsigned int i = 0; i < m; i++) {
        vecSum = 0; // Reset sum to 0 after every row
        in = i*n;
        for (unsigned int j = 0; j < n; j++) {
            // Sum each product of correspondin
            vecSum += (A[in + j] * x[j]);
        }
        // Record the sum in the resulting vect
        b[i] = vecSum;
    }
}
```

Multiple markers at this line
- FMA (fused multiply-add) instruction(s) generated
- Generated 2 prefetch instructions for the loop
- Generated vector simd code for the loop containing reductions

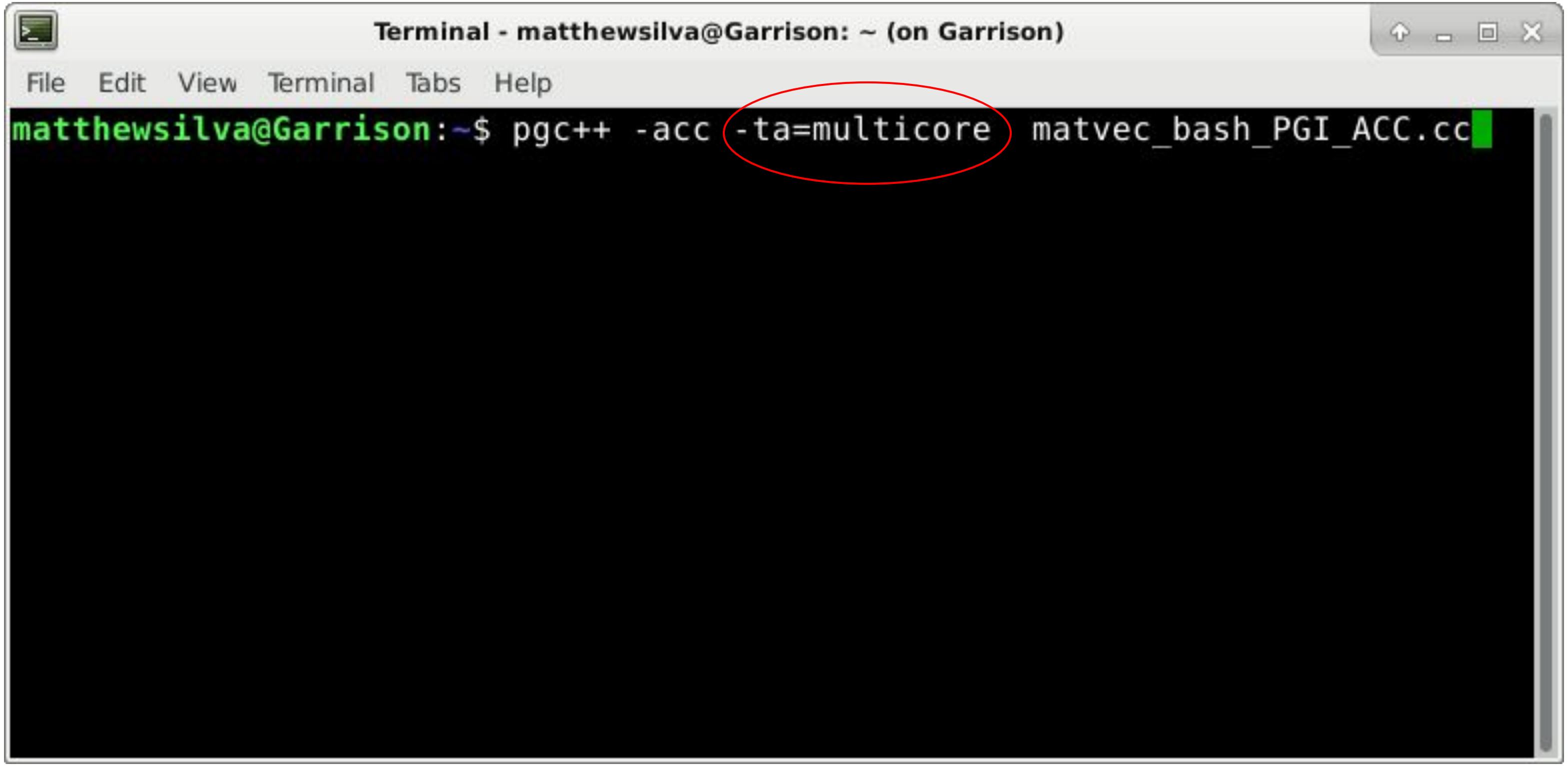
Parallelize

```
  GNU nano 2.7.4                    File: matvec_bash_PGI_ACC.cc



// Sequential implementation of a matrix vectorization
void matvec(const float *A, const float *x, unsigned int m, unsigned int n, flo$
    float vecSum;
    unsigned int in;
    int mn = m*n;
    for (unsigned int i = 0; i < m; i++) {
        vecSum = 0; // Reset sum to 0 after every row
        in = i*n;
        #pragma acc parallel loop
        for (unsigned int j = 0; j < n; j++) {
            // Sum each product of corresponding matrix and vector values
            vecSum += (A[in + j] * x[j]);
        }
        // Record the sum in the resulting vector
        b[i] = vecSum;
    }
}
```

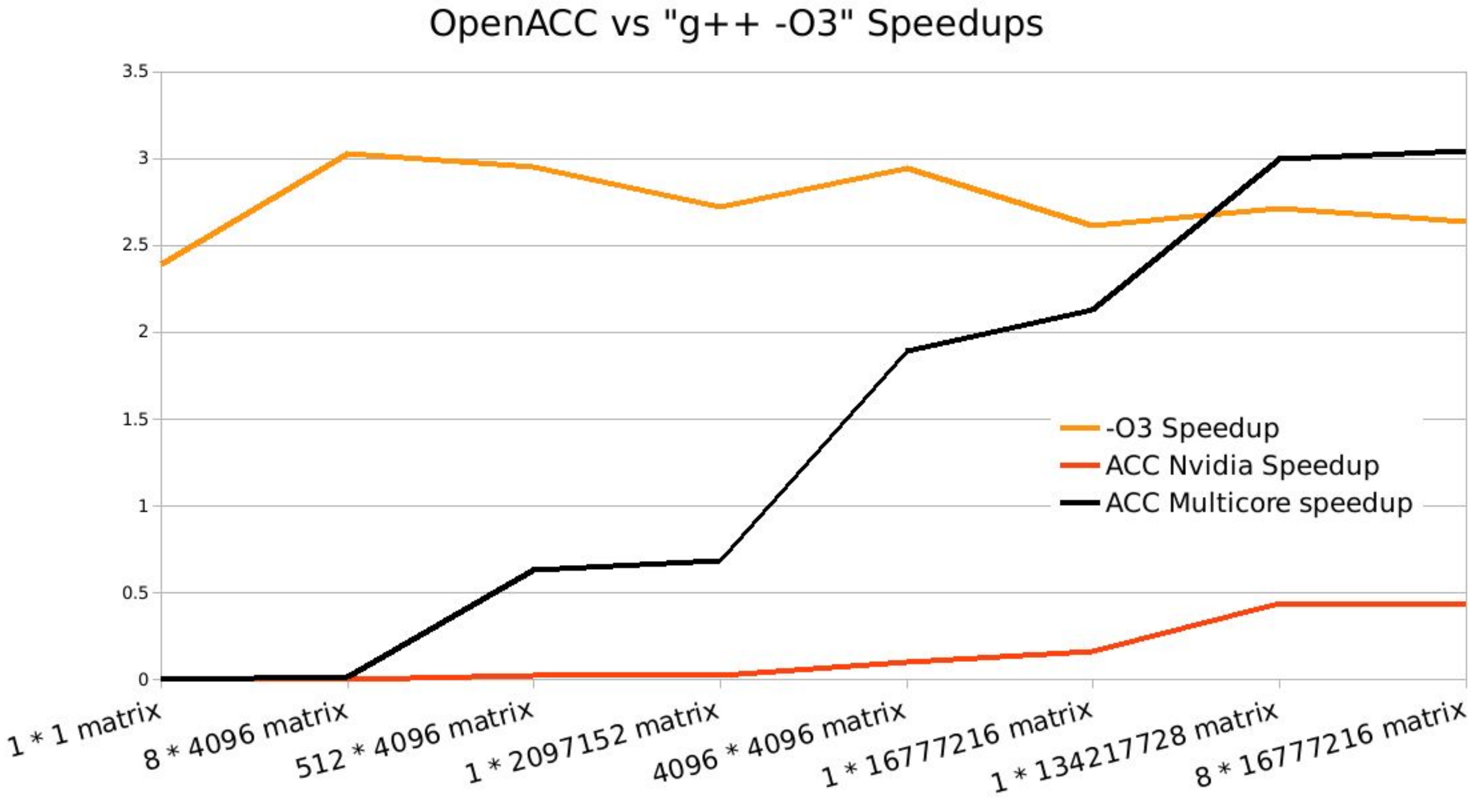
Parallelize (For Nvidia)



Parallelize (for Multicore CPU)

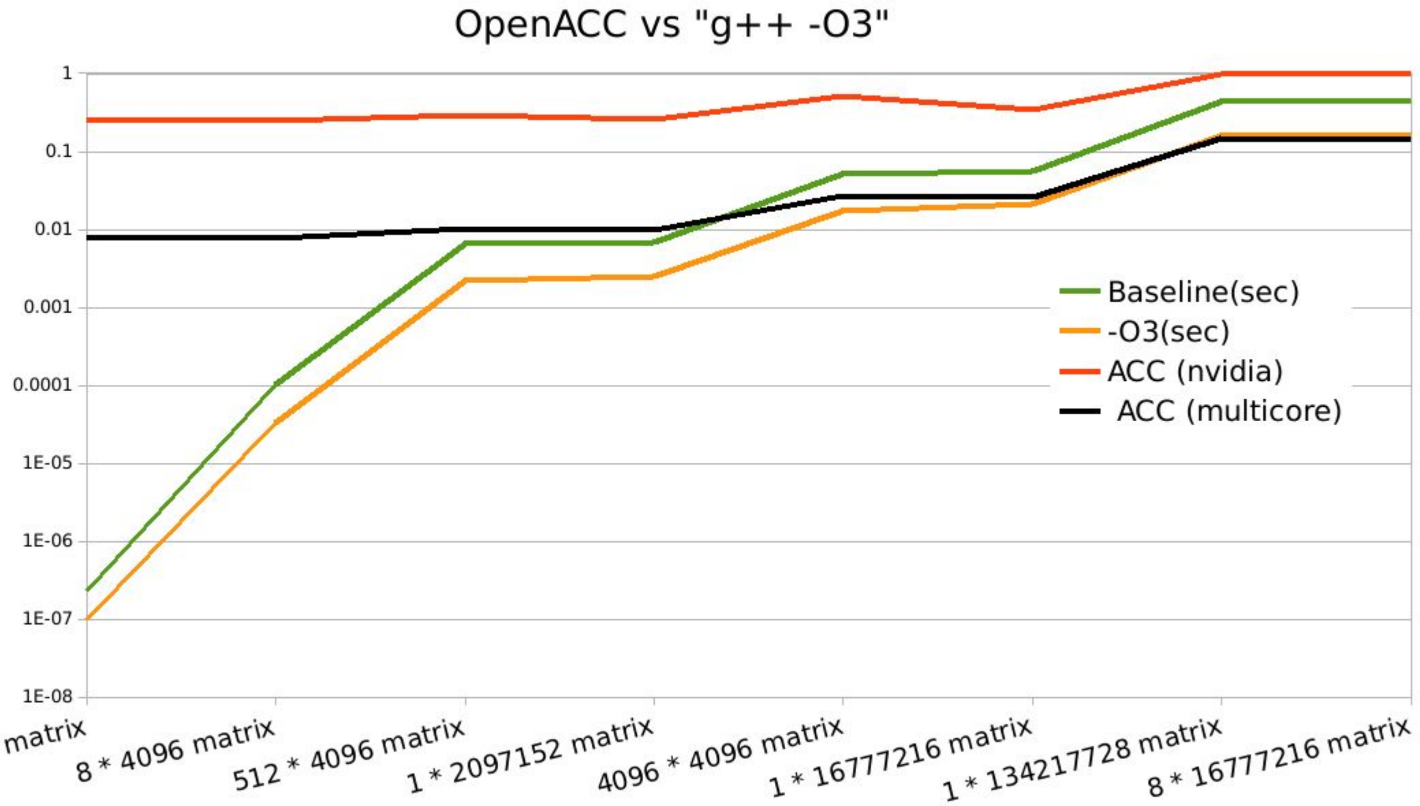


Parallelize



Parallelize

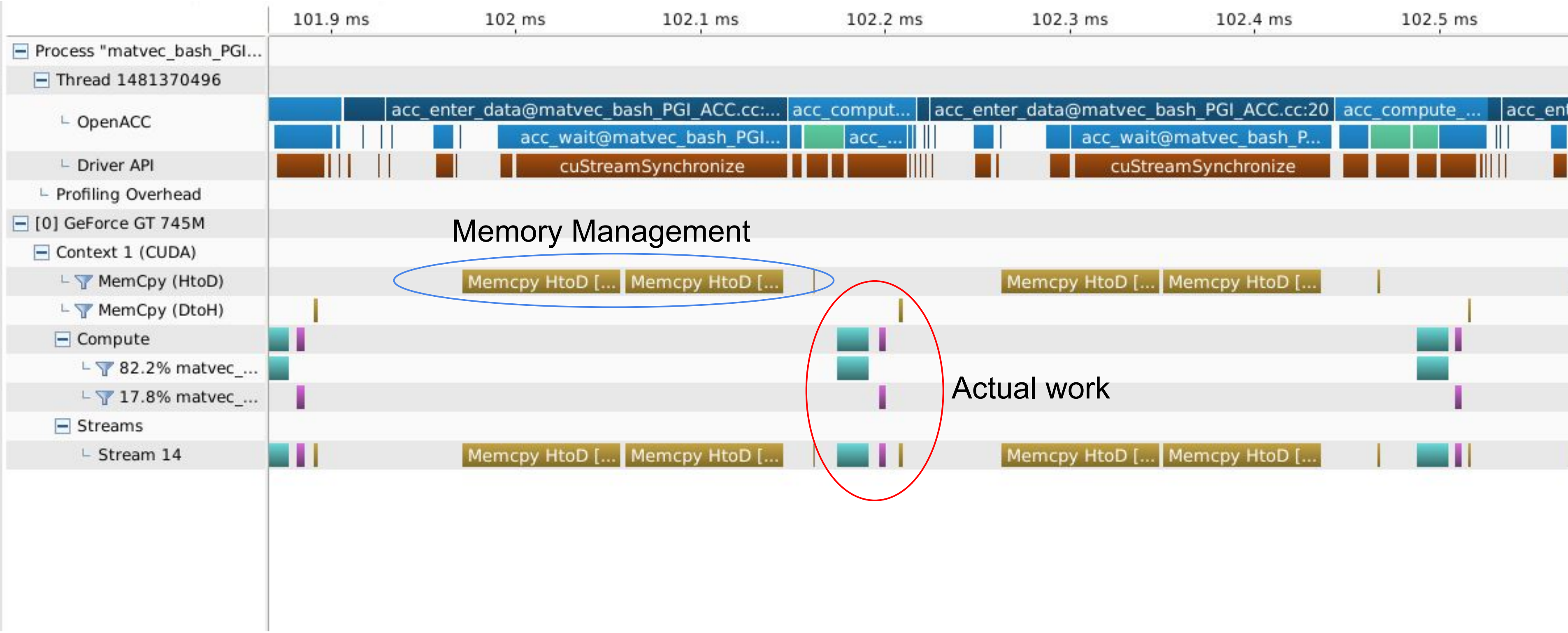
Seconds



Optimize

0 s
0.01 s
0.02 s
0.03 s
0.04 s
0.05 s
0.06 s
0.07 s
0.08 s
0.09 s
0.1 s
0.11 s
0.12 s
0.13 s
0.14 s
0.15 s
Process "matvec_bash_PGI...
Thread 1481370496
OpenACC
Driver API
Profiling Overhead
[0] GeForce GT 745M
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
82.2% matvec_...
17.8% matvec_...
Streams
Stream 14
acc_enter_data@matvec_ba...
cuDevicePrimaryCtxRetain
cuMemHostAlloc
cuMemFreeHost
cuDevicePrimaryCtxRelease
Resource M...
This is all GPU overhead
This is where the memory management and work actually get done

Optimize


101.9 ms
102 ms
102.1 ms
102.2 ms
102.3 ms
102.4 ms
102.5 ms
Process "matvec_bash_PGI...
Thread 1481370496
OpenACC
Driver API
Profiling Overhead
[0] GeForce GT 745M
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
82.2% matvec_...
17.8% matvec_...
Streams
Stream 14
acc_enter_data@matvec_bash_PGI_ACC.cc:...
acc_comput...
acc_enter_data@matvec_bash_PGI_ACC.cc:20
acc_compute_...
acc_wait@matvec_bash_PGI...
acc_...
acc_wait@matvec_bash_P...
cuStreamSynchronize
cuStreamSynchronize
Memcpy HtoD [...
Memcpy HtoD [...
Memcpy HtoD [...
Memcpy HtoD [...
Memory Management
Actual work

Optimize

Results

Low Compute / Memcpy Efficiency [170.183 µs / 1.38581 ms = 0.123]
The amount of time performing compute is low relative to the amount of time required for memcpy. More...

Low Memcpy/Compute Overlap [0 ns / 170.183 µs = 0%]
The percentage of time when memcpy is being performed in parallel with compute is low. More...

Low Kernel Concurrency [0 ns / 170.183 µs = 0%]
The percentage of time when two kernels are being executed in parallel is low. More...

Low Memcpy Throughput [1.513 GB/s avg, for memcpys accounting for 100% of all memcpy time]
The memory copies are not fully using the available host to device bandwidth. More...

Low Compute Utilization [170.183 µs / 155.17625 ms = 0.1%]
The multiprocessors of one or more GPUs are mostly idle. More...

Absolutely terrible!

Compute Utilization
The device timeline shows an estimate of the amount of the total compute capacity being used by the kernels executing on the device.

Optimize

```
  GNU nano 2.7.4              File: matvec_bash_PGI_ACC_red_copy.cc


// Sequential implementation of a matrix vectorization
void matvec(const float *A, const float *x, unsigned int m, unsigned int n, float *$
    float vecSum;
    unsigned int in;
    int mn = m*n;
    #pragma acc data copy(A[0:mn]) copy(x[0:n])
    for (unsigned int i = 0; i < m; i++) {
        vecSum = 0; // Reset sum to 0 after every row
        in = i*n;
        #pragma acc parallel loop reduction(+:vecSum)
        for (unsigned int j = 0; j < n; j++) {
            // Sum each product of corresponding matrix and vector values
            vecSum += (A[in + j] * x[j]);
        }
        // Record the sum in the resulting vector
        b[i] = vecSum;
    }
}
```

Optimize

Results

Low Compute / Memcpy Efficiency [48.75105 ms / 742.64716 ms = 0.066]
The amount of time performing compute is low relative to the amount of time required for memcpy. More...

Low Memcpy/Compute Overlap [0 ns / 48.75105 ms = 0%]
The percentage of time when memcpy is being performed in parallel with compute is low. More...

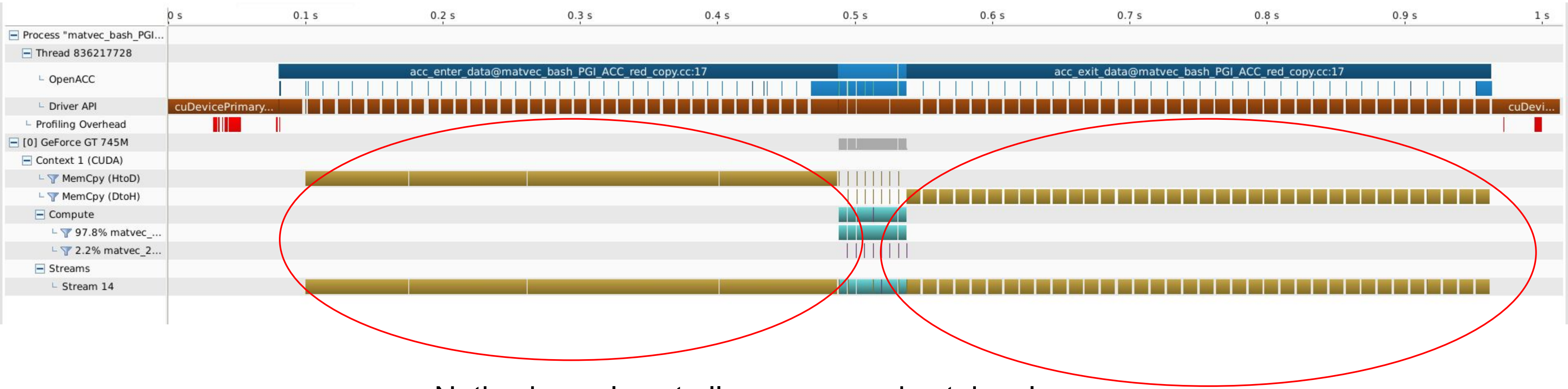
Low Kernel Concurrency [0 ns / 48.75105 ms = 0%]
The percentage of time when two kernels are being executed in parallel is low. More...

Low Memcpy Throughput [1.627 GB/s avg, for memcpys accounting for 100% of all memcpy time]
The memory copies are not fully using the available host to device bandwidth. More...

Low Compute Utilization [48.75105 ms / 1.01286 s = 4.8%] **Still terrible but 48 times better!**
The multiprocessors of one or more GPUs are mostly idle. More...

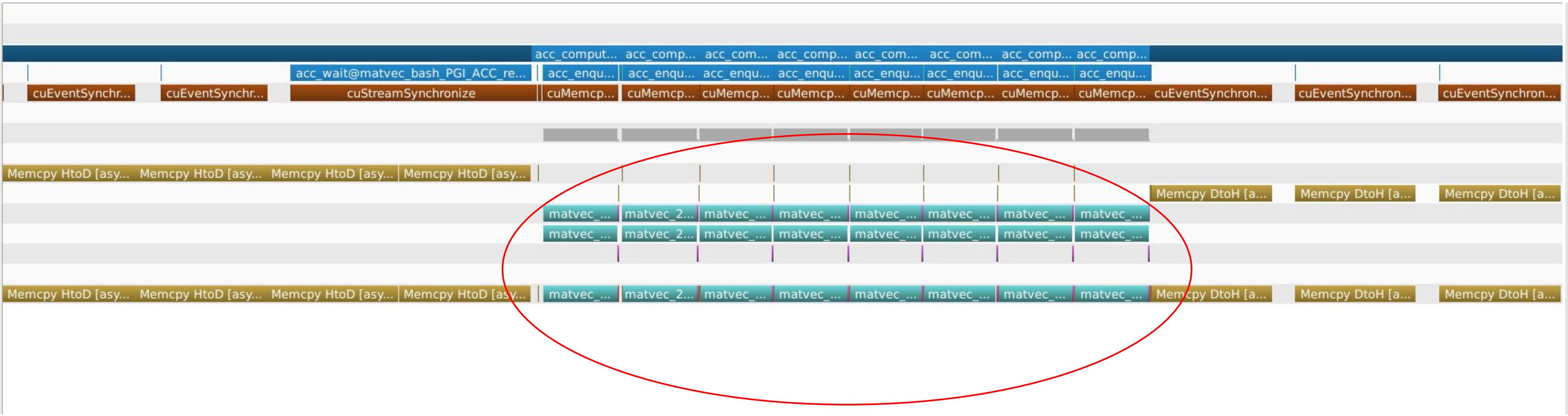
Compute Utilization
The device timeline shows an estimate of the amount of the total compute capacity being used by the kernels executing on the device.

Optimize



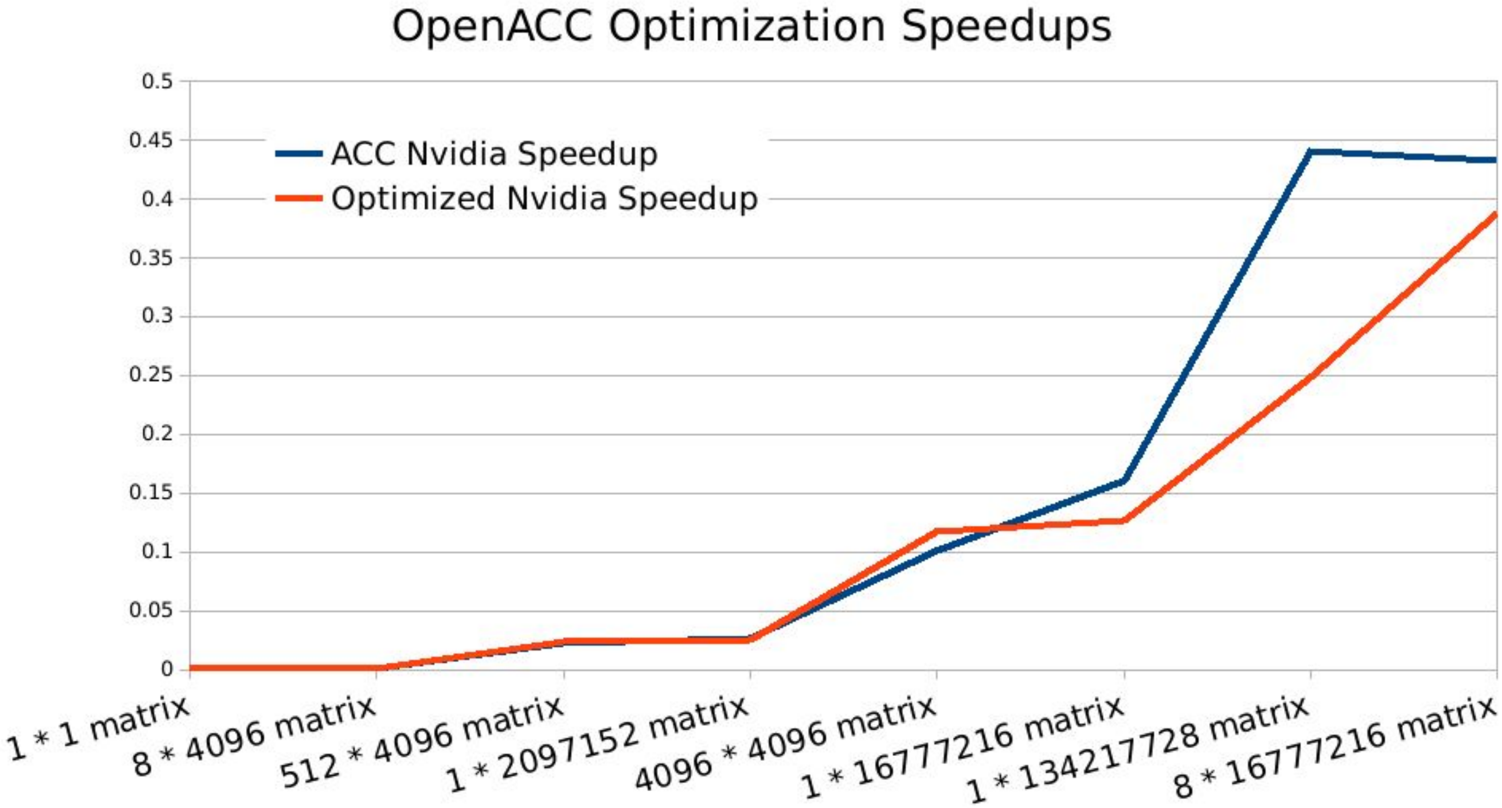
Notice how almost all memory copies take place before and after the computations

Optimize



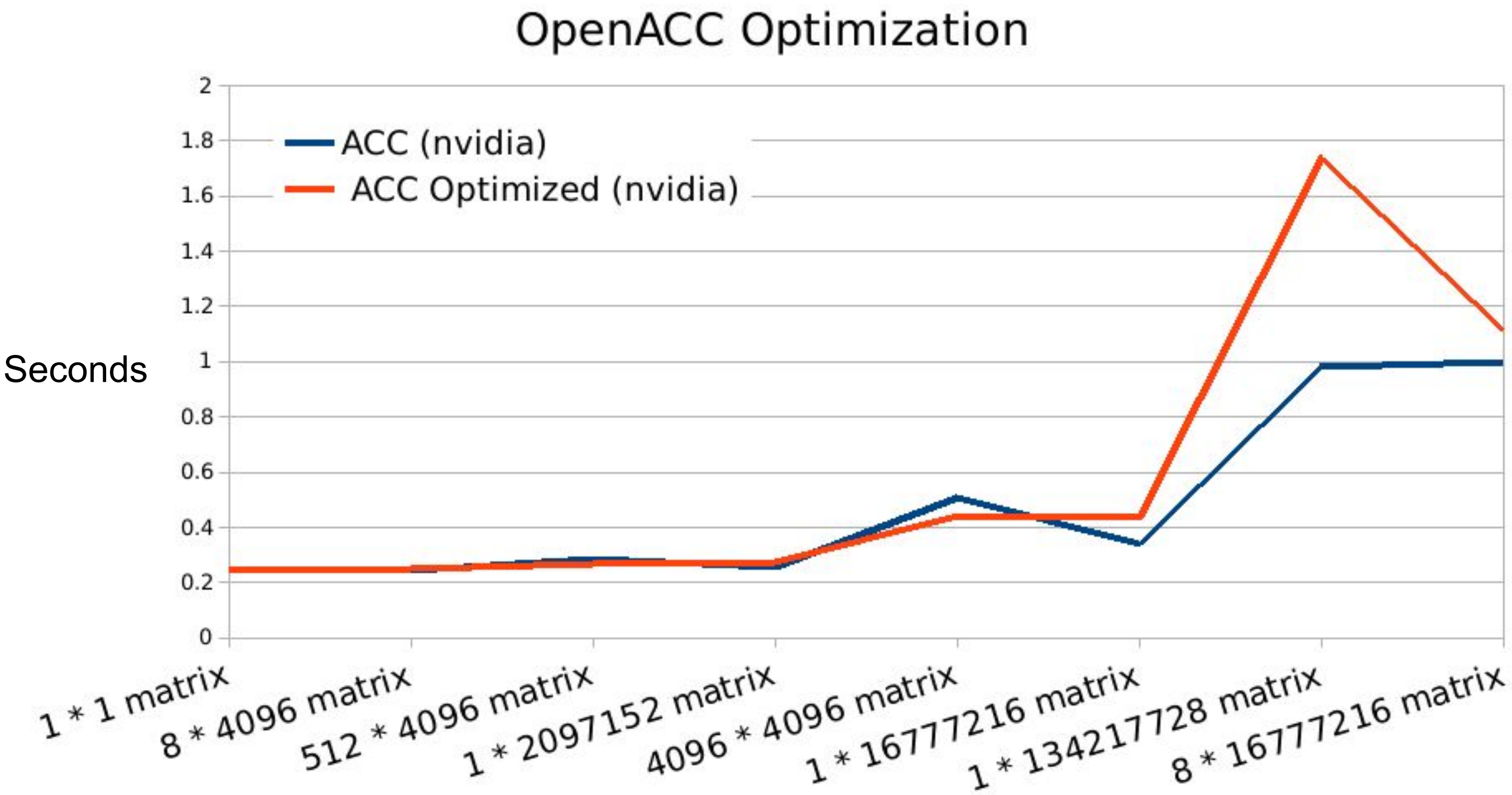
All uninterrupted computations!

Optimize



OpenACC Optimization Speedups
ACC Nvidia Speedup
Optimized Nvidia Speedup
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
1 * 1 matrix
8 * 4096 matrix
512 * 4096 matrix
1 * 2097152 matrix
4096 * 4096 matrix
1 * 16777216 matrix
1 * 134217728 matrix
8 * 16777216 matrix

Optimize



OpenACC Optimization
Seconds
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
ACC (nvidia)
ACC Optimized (nvidia)
1 * 1 matrix
8 * 4096 matrix
512 * 4096 matrix
1 * 2097152 matrix
4096 * 4096 matrix
1 * 16777216 matrix
1 * 134217728 matrix
8 * 16777216 matrix

OpenACC Overview

- OpenACC: Compiler directives for automatically parallelizing code
- Performance-portable: Can optimize for different hardware (heterogeneous) automatically based on the same directives
- Profilers: Automatically point to parallelizable sections of code to guide directives
- Abstraction: More high-level than most attempts to accelerate code using specialized hardware
- Drawbacks: Compilers do not always follow the directives intelligently with respect to the hardware available

References

“Downloads & Tools.” *OpenACC*, OpenACC-standard.org, 2017.

“Get Started.” *OpenACC*, OpenACC-standard.org, 2017.

Melonakos. “No Free Lunch for GPU Compiler Directives.” *GPU and Accelerator Software Blog RSS*, GPU and Accelerator Software Blog, 11 Apr. 2012.

“The OpenACC API Quick Reference Guide.” *OpenACC,* OpenACC-standard.org, 2011.

“Research & Innovation.” *Softeco Sismat,* Softeco Sismat InformationTechnology, 2018.

“Step 1: Analyze, Three Steps to More Science Guide.” *NVIDIADeveloper,* NVIDIA, 13 Jan. 2017.

“Step 2: Parallelize, Three Steps to More Science Guide.” *NVIDIADeveloper,* NVIDIA, 13 Jan. 2017.

“Step 3: Optimize, Three Steps to More Science Guide.” *NVIDIADeveloper,* NVIDIA, 13 Jan. 2017.

“What Is OpenACC?” *OpenACC*, OpenACC-standard.org, 2017.