OpenACC Fundamentals

Steve Abbott <sabbott@nvidia.com>, November 15, 2017

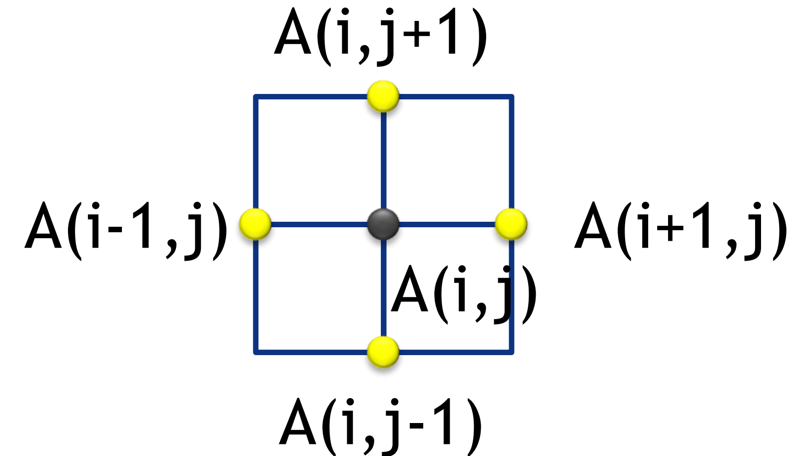
AGENDA

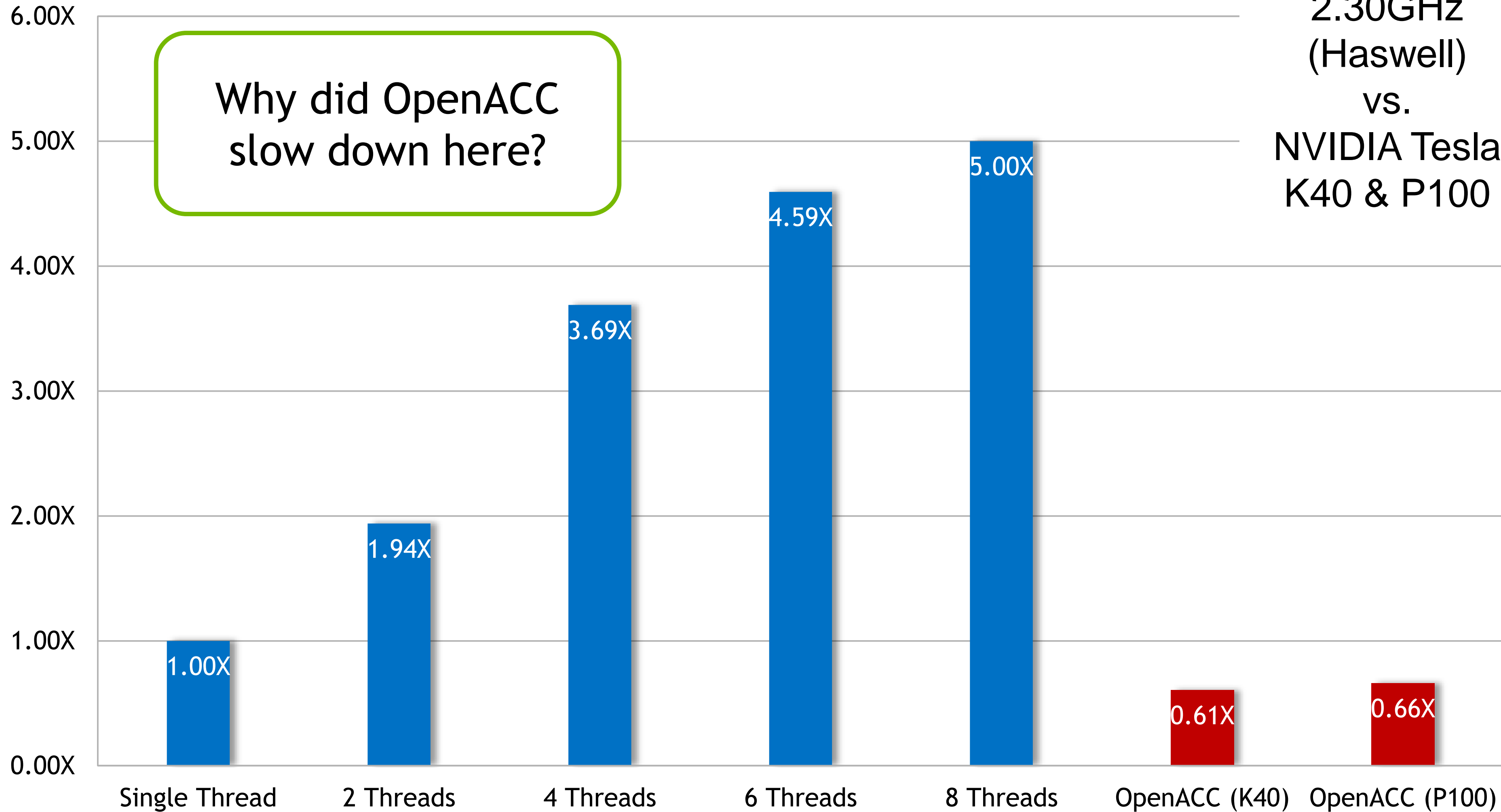
Data Regions

Deep Copy

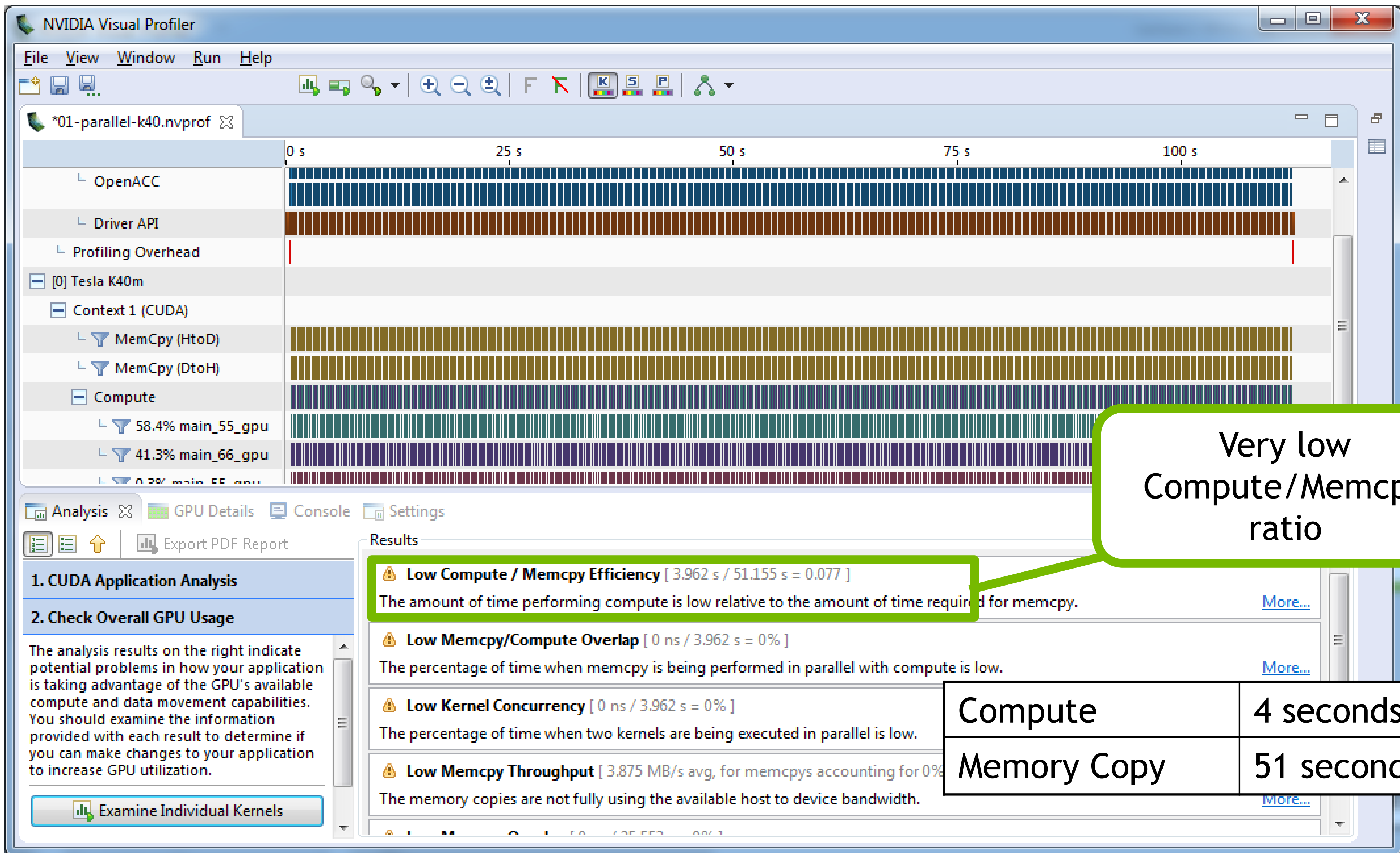
JACOBI ITERATION

```
while ( err > tol && iter < iter_max ) {
  err=0.0;

#pragma acc parallel loop reduction(max:err)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }

#pragma acc parallel loop
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```





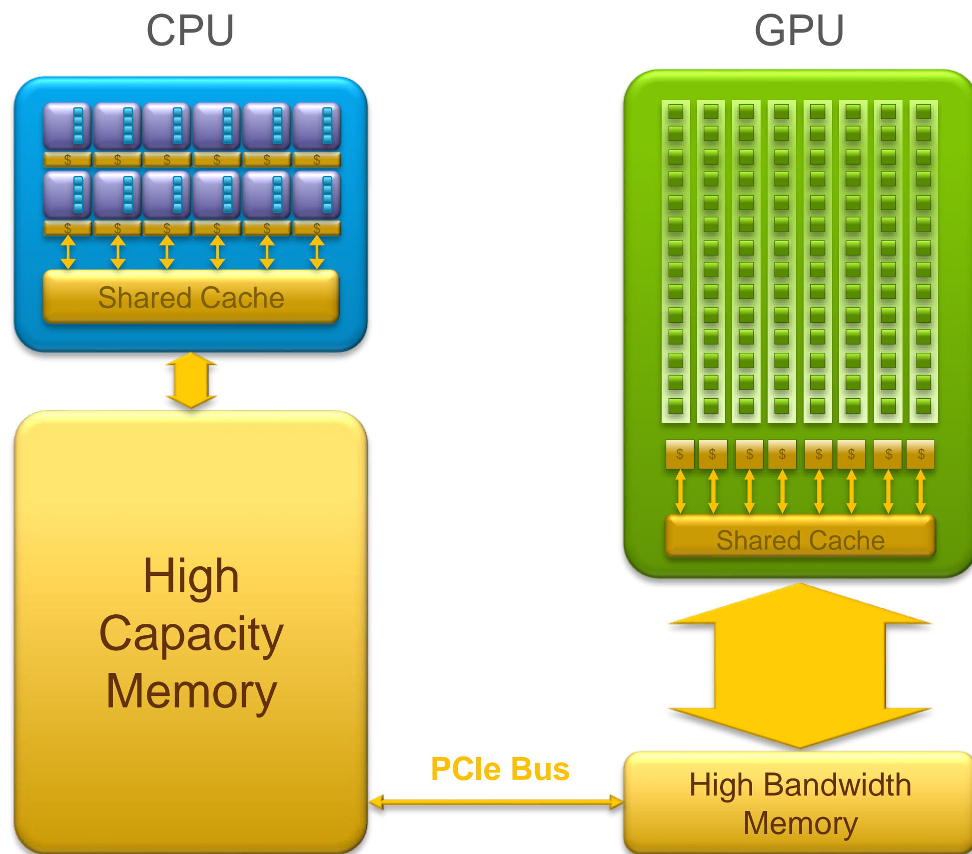
Compiler: PGI 16.10

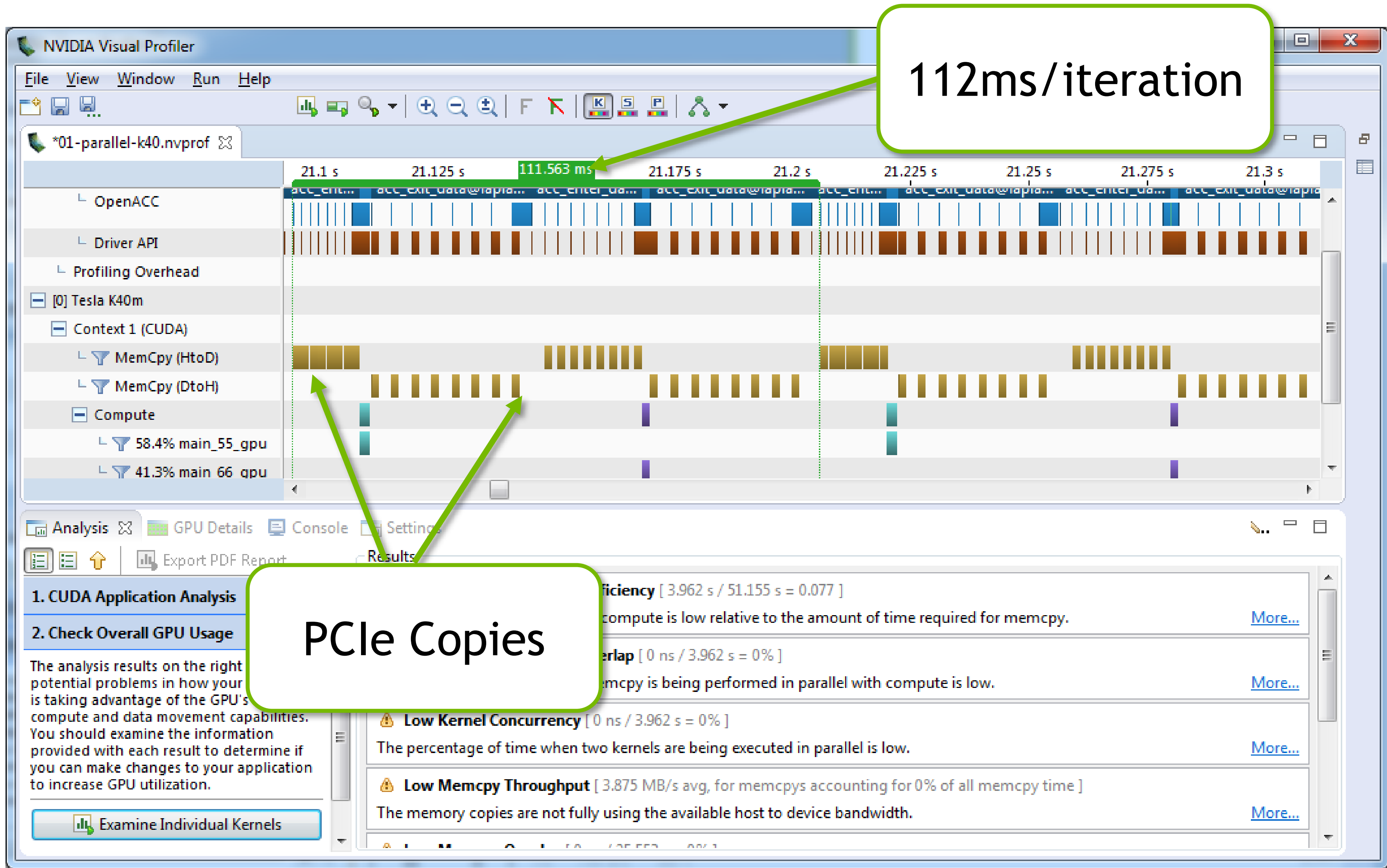


Compute	4 seconds
Memory Copy	51 seconds

ARCHITECTURE BASIC CONCEPT

SIMPLIFIED, BUT SADLY TRUE





NVIDIA Visual Profiler
File View Window Run Help
*01-parallel-k40.nvprof
112ms/iteration
111.563 ms
21.1 s
21.125 s
21.175 s
21.2 s
21.225 s
21.25 s
21.275 s
21.3 s
OpenACC
Driver API
Profiling Overhead
[0] Tesla K40m
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
58.4% main_55_gpu
41.3% main_66_gpu
PCIe Copies
Analysis
GPU Details
Console
Export PDF Report
Results
1. CUDA Application Analysis
2. Check Overall GPU Usage
The analysis results on the right
potential problems in how your
is taking advantage of the GPU's
compute and data movement capabilities.
You should examine the information
provided with each result to determine if
you can make changes to your application
to increase GPU utilization.
Examine Individual Kernels
ficiency [3.962 s / 51.155 s = 0.077]
compute is low relative to the amount of time required for memcpy.
erlap [0 ns / 3.962 s = 0%]
mcpy is being performed in parallel with compute is low.
Low Kernel Concurrency [0 ns / 3.962 s = 0%]
The percentage of time when two kernels are being executed in parallel is low.
Low Memcpy Throughput [3.875 MB/s avg, for memcpys accounting for 0% of all memcpy time]
The memory copies are not fully using the available host to device bandwidth.
More...

Excessive Data Transfers

```
while ( err > tol && iter < iter_max )
{
  err=0.0;
```

A, Anew resident on host

Copy

```
#pragma acc parallel loop
```

A, Anew resident on accelerator

```
for( int j = 1; j < n-1; j++) {
  for(int i = 1; i < m-1; i++) {
    Anew[j][i] = 0.25 * (A[j][i+1] +
                 A[j][i-1] + A[j-1][i] +
                 A[j+1][i]);
    err = max(err, abs(Anew[j][i] -
                        A[j][i]);
  }
}
...
```

These copies happen every iteration of the outer while loop!

Copy

A, Anew resident on accelerator

A, Anew resident on host

```
  ...
}
```

Evaluate Data Locality

```
while ( err > tol && iter < iter_max ) {
  err=0.0;

#pragma acc parallel loop
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }

#pragma acc parallel loop
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

Does the CPU need the data between these loop nests?

Does the CPU need the data between iterations of the convergence loop?

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.

Data Clauses

`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.
`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.
`create ( list )`	Allocates memory on GPU but does not copy.
`present ( list )`	Data is already present on GPU from another containing data region.
`deviceptr( list )`	The variable is a device pointer (e.g. CUDA) and can be used directly on the device.

Array Shaping

Compiler sometimes cannot determine size of arrays

Must specify explicitly using data clauses and array "shape"

<u>C/C++</u>

```
#pragma acc data copyin(a[0:nelem]) copyout(b[s/4:3*s/4])
```

<u>Fortran</u>

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

Note: data clauses can be used on `data`, `parallel`, or `kernels`

DATA REGIONS HAVE REAL CONSEQUENCES

Simplest Kernel

```
int main(int argc, char** argv){

float A[1000];


    #pragma acc kernels
    for( int iter = 1; iter < 1000 ; iter++){
      A[iter] = 1.0;
    }

    A[10] = 2.0;


  printf("A[10] = %f", A[10]);
}
```

A[ ] Copied To GPU

A[ ] Copied To Host

Runs On Host

Output:
A[10] = 2.0

With Global Data Region

```
int main(int argc, char** argv){

float A[1000];

#pragma acc data copy(A)
{

    #pragma acc kernels
    for( int iter = 1; iter < 1000 ; iter++){
      A[iter] = 1.0;
    }

    A[10] = 2.0;

}

  printf("A[10] = %f", A[10]);
}
```

A[ ] Copied To GPU

Still Runs On Host

A[ ] Copied To Host

Output:
A[10] = 1.0

Add Data Clauses

```
#pragma acc data copy(A) create(Anew)
while ( err > tol && iter < iter_max ) {
  err=0.0;
#pragma acc parallel loop
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }

#pragma acc parallel loop
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

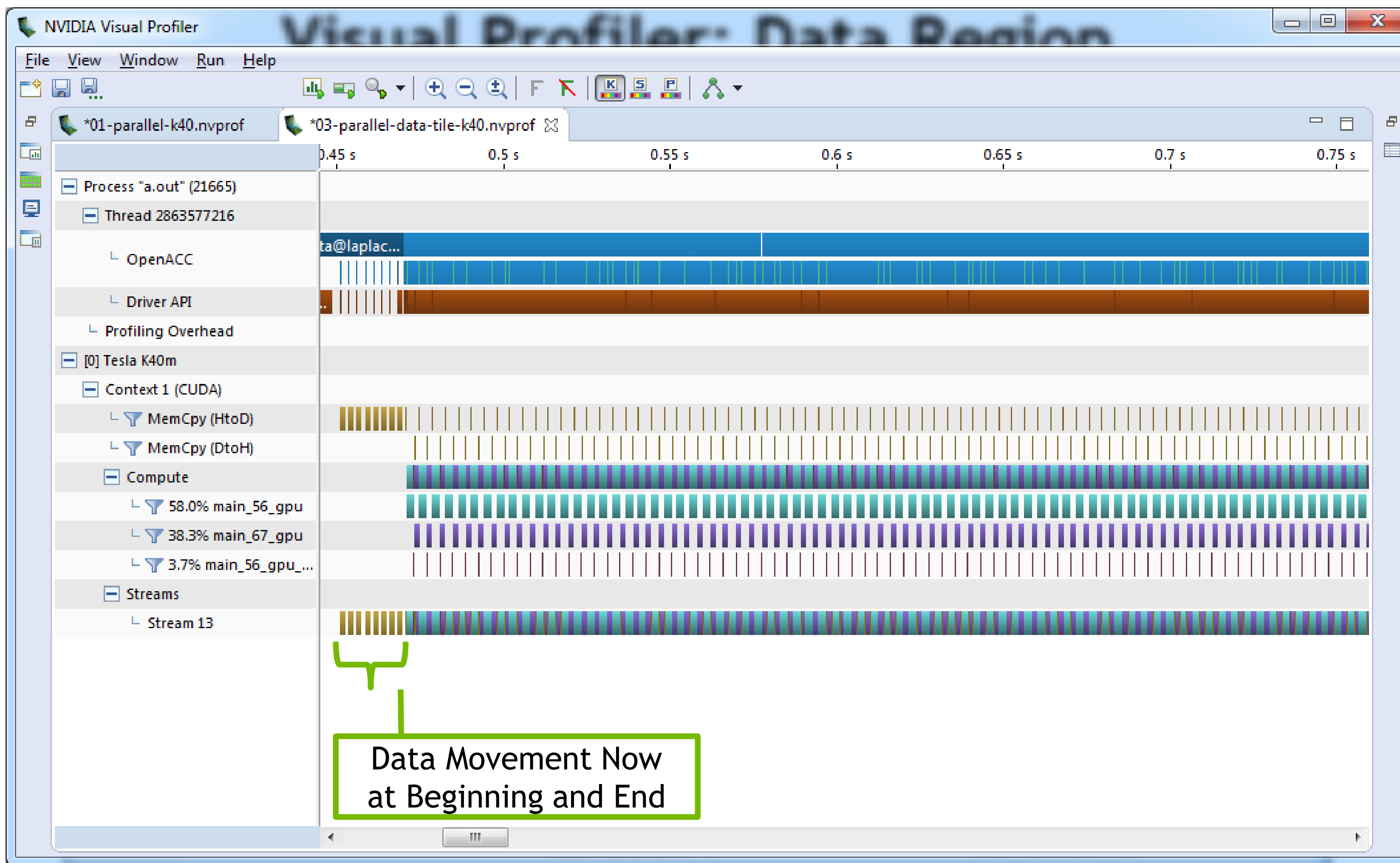
Copy A to/from the accelerator only when needed.

Create Anew as a device temporary.

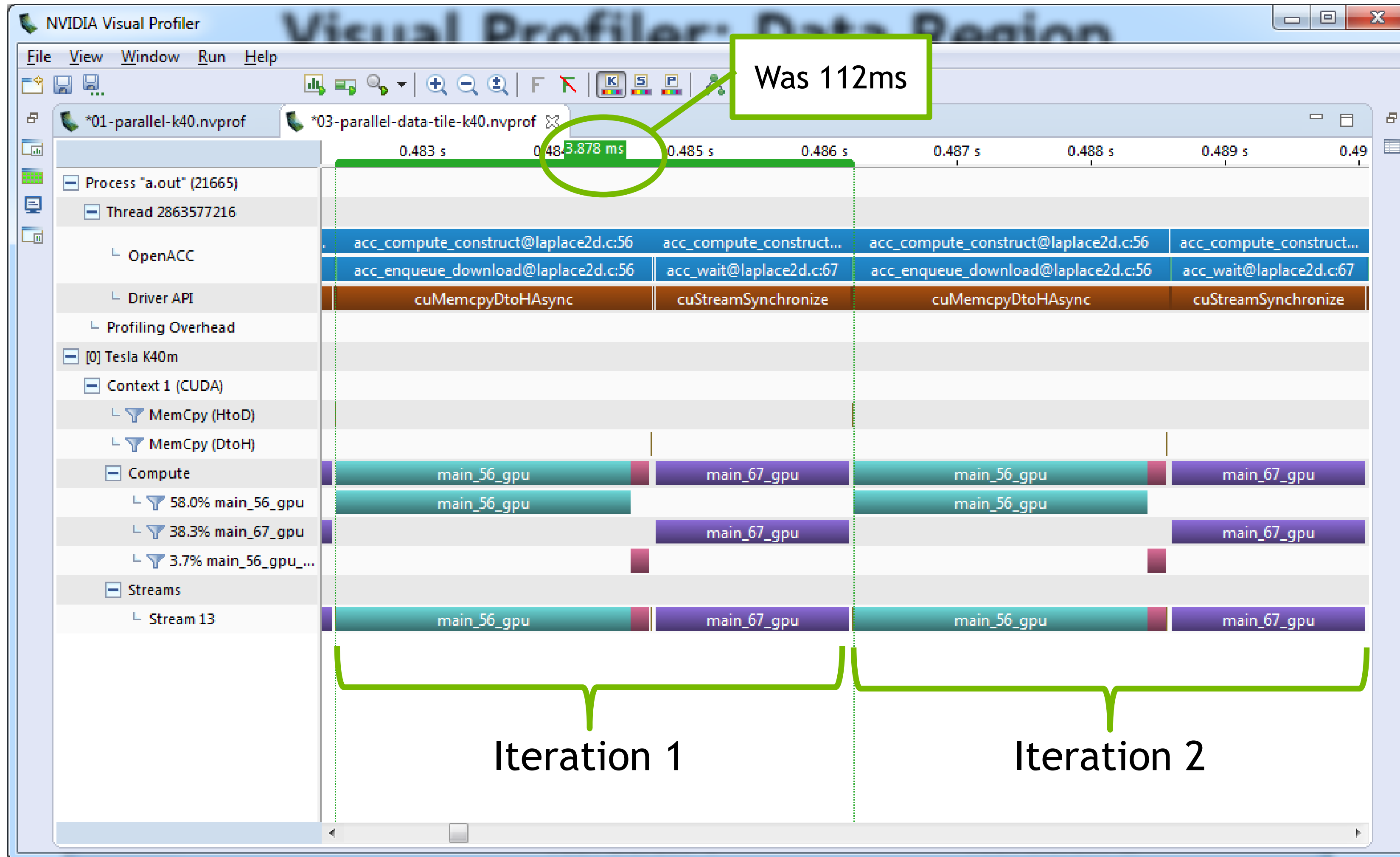
Rebuilding the code

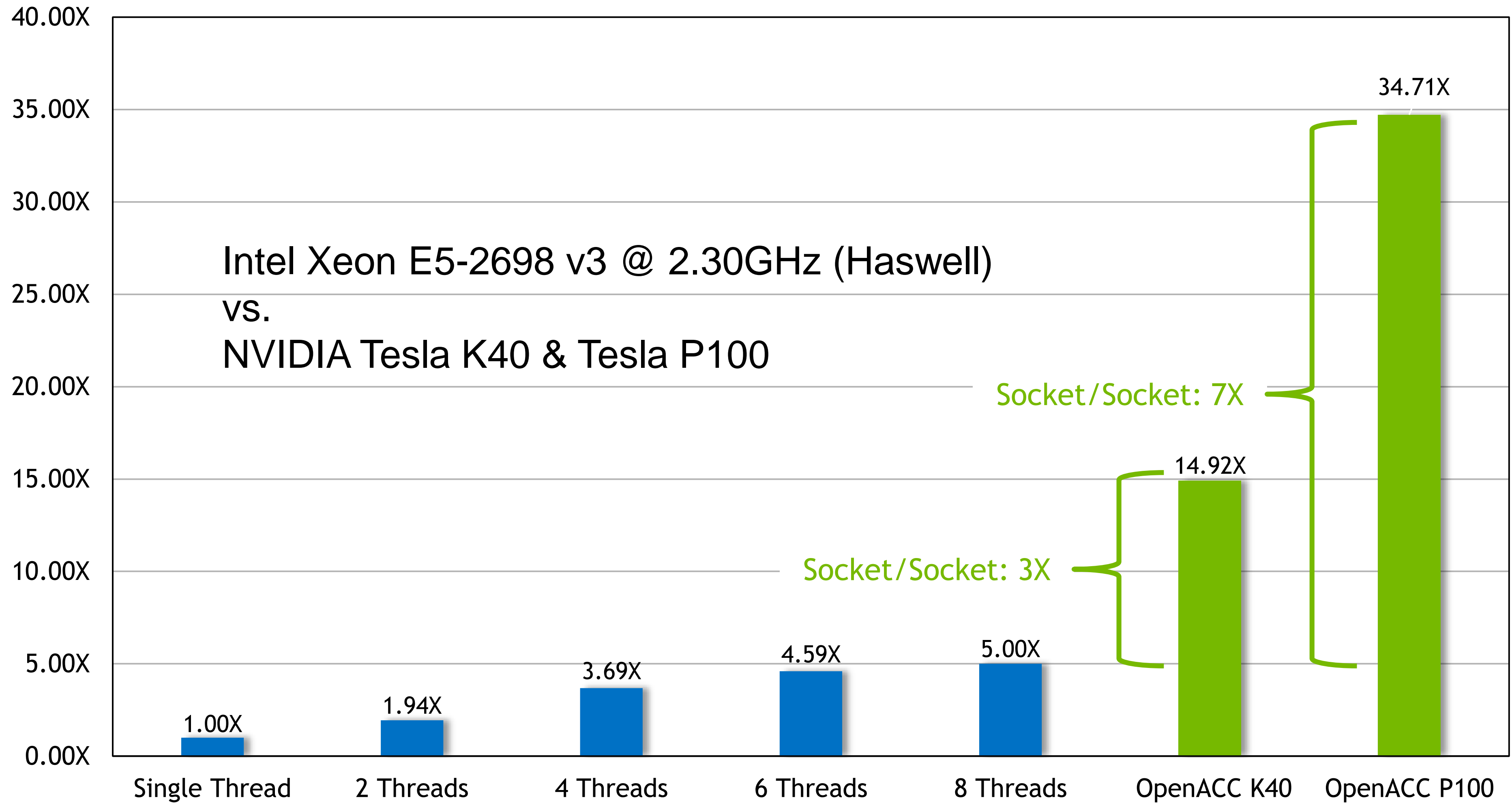
```
$ pgcc -fast -acc -ta=tesla -Minfo=all laplace2d.c
main:
     40, Loop not fused: function call before adjacent loop
         Generated vector sse code for the loop
     51, Generating copy(A[:][:])
         Generating create(Anew[:][:])
         Loop not vectorized/parallelized: potential early exits
     56, Accelerator kernel generated
         56, Max reduction generated for error
         57, #pragma acc loop gang /* blockIdx.x */
         59, #pragma acc loop vector(256) /* threadIdx.x */
     56, Generating Tesla code
     59, Loop is parallelizable
     67, Accelerator kernel generated
         68, #pragma acc loop gang /* blockIdx.x */
         70, #pragma acc loop vector(256) /* threadIdx.x */
     67, Generating Tesla code
     70, Loop is parallelizable
```

Visual Profiler: Data Region



Visual Profiler: Data Region





Compiler: PGI 16.10

unStructured data Directives

Enter Data Directive

The **enter data** directive handles device memory **allocation**

You may use either the **create** or the **copyin** clause for memory allocation

You may allocate more than one array at a time, and you may allocate arrays in any function

The enter data directive is **not** the start of a data region, because you may have multiple enter data directives

```
#pragma acc enter data clauses

  < Sequential and/or Parallel code >

#pragma acc exit data clauses
```

```
!$acc enter data clauses

  < Sequential and/or Parallel code >

!$acc exit data clauses
```


NVIDIA.

unStructured data Directives

Exit Data Directive

The **exit data** directive handles device memory **deallocation**

You may use either the **delete** or the **copyout** clause for memory deallocation

You may use the exit data directive to deallocate any array that was previously allocated with the enter data directive

One of the biggest advantages of using unstructured data directives is the ability to do device memory allocation and deallocation in **completely different functions**

```
#pragma acc enter data clauses

  < Sequential and/or Parallel code >

#pragma acc exit data clauses
```

```
!$acc enter data clauses

  < Sequential and/or Parallel code >

!$acc exit data clauses
```

unstructured vs structured

With a simple code

Unstructured

Can have multiple starting/ending points

Memory exists until explicitly deallocated

Can branch across multiple functions

```
#pragma acc enter data copyin(a[0:N],b[0:N]) \
  create(c[0:N])

  #pragma acc parallel loop
  for(int i = 0; i < N; i++){
    c[i] = a[i] + b[i];
  }

#pragma acc exit data copyout(c[0:N]) \
  delete(a,b)
```

Structured

- Must have a explicit start/end point
- Memory only exists within the data region
- Must be within a single function

```
#pragma acc data copyin(a[0:N],b[0:N]) \
  copyout(c[0:N])
{
  #pragma acc parallel loop
  for(int i = 0; i < N; i++){
    c[i] = a[i] + b[i];
  }

}
```

unStructured data Directives

Branching across multiple functions

```
int* allocate_array(int N){
   int* ptr = (int *) malloc(N * sizeof(int));
   #pragma acc enter data create(ptr[0:N])
   return ptr;
}

void deallocate_array(int* ptr){
   #pragma acc exit data delete(ptr)
   free(ptr);
}

int main(){
   int* a = allocate_array(100);
   #pragma acc kernels
   {
      a[0] = 0;
   }
   deallocate_array(a);
}
```

This is an example code where the memory allocation/deallocation is handled in separate functions

The data region is not explicitly defined by a starting point and an ending point

The data enter and exit will be decided by whenever the programmer calls the allocate/deallocate functions

C structs

Without dynamic data members

Dynamic data members are anything contained within a struct that can have a **variable size** (dynamically allocated arrays)

OpenACC is easily able to copy our struct to device memory because everything in our float3 struct has a **fixed size**

If float3 has any members with a varying size, then the programmer will need to explicitly allocate that member in device memory

```
typedef struct {
  float x, y, z;
} float3;

int main(int argc, char* argv[]){
  int N = 10;
  float3* f3 = malloc(N * sizeof(float3));

  #pragma acc enter data create(f3[0:N])

  #pragma acc kernels
  for(int i = 0; i < N; i++){
    f3[i].x = 0.0f;
    f3[i].y = 0.0f;
    f3[i].z = 0.0f;
  }

  #pragma acc exit data delete(f3)
  free(f3);
}
```

C structs

With dynamic data members

OpenACC is not automatically able to copy dynamic pointers to the device

You must first copy the struct into device memory

Then you must allocate/copy the dynamic members into device memory

To deallocate, you must first deallocate the dynamic members

Then deallocate the struct

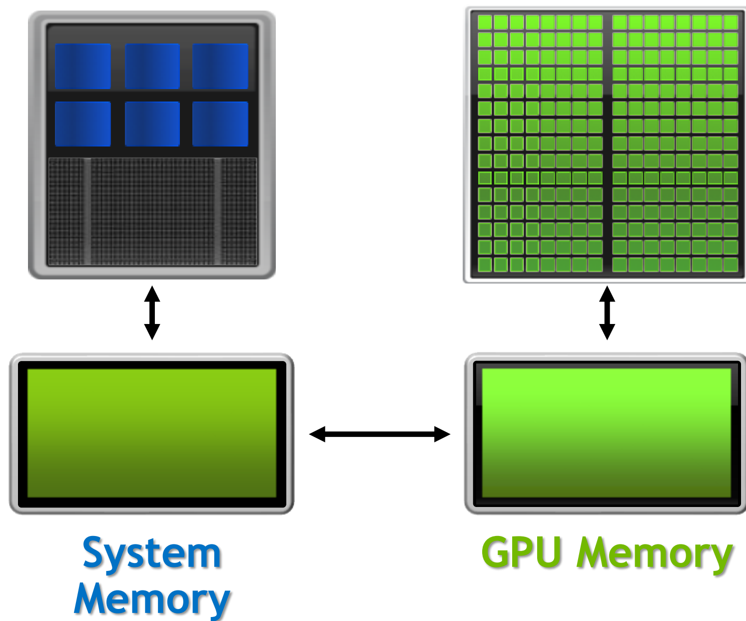
```
typedef struct {
  float *arr;
  int n;
} vector;

int main(int argc, char* argv[]){

  vector v;
  v.n = 10;
  v.arr = (float*) malloc(v.n*sizeof(float));

  #pragma acc enter data copyin(v)
  #pragma acc enter data create(v.arr[0:v.n])

  ...

  #pragma acc exit data delete(v.arr)
  #pragma acc exit data delete(v)
  free(v.arr);
}
```

Cuda managed memory

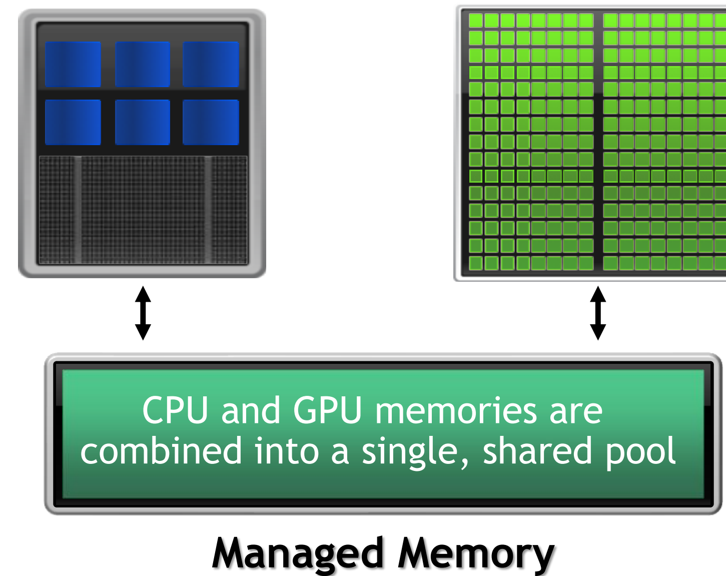
Simplified Developer

Commonly referred to as "unified memory."

Without Managed Memory



With Managed Memory



Managed memory

Limitations

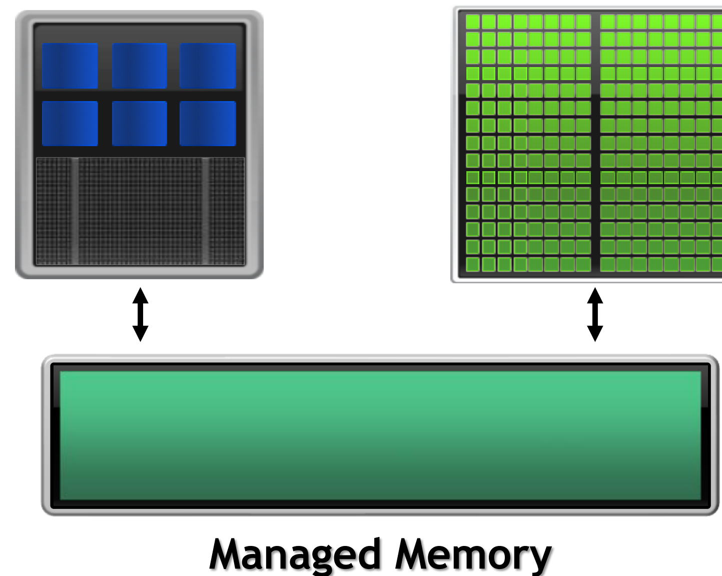
The programmer will almost always be able to get better performance by manually handling data transfers

Memory allocation/deallocation takes longer with managed memory

Cannot transfer data asynchronously

Cannot be used with static memory*

Performance depends on how the data is accessed



Managed memory

Why and How to Use It

Handling explicit data transfers between the host and device (CPU and GPU) can be difficult

PGI provides the managed target option for NVIDIA Tesla GPUs

This will tell the compiler allocate all memory as CUDA Managed Memory

This generally means that the programmer will do less work, but the code is less portable.

```
$ pgcc -acc -ta=tesla:managed -Minfo=accel main.c
```

Pros & Cons of Managed Memory

Pro

Simple porting of complex data structures

Concentrate on parallelism first and data later

Con

- Limited to the PGI compiler and NVIDIA GPUs, no portability
- Performance will depend heavily on access pattern

Using managed memory should be a stepping stone to quickly port the code.

Next Lecture

Friday – Advanced OpenACC

NVIDIA.