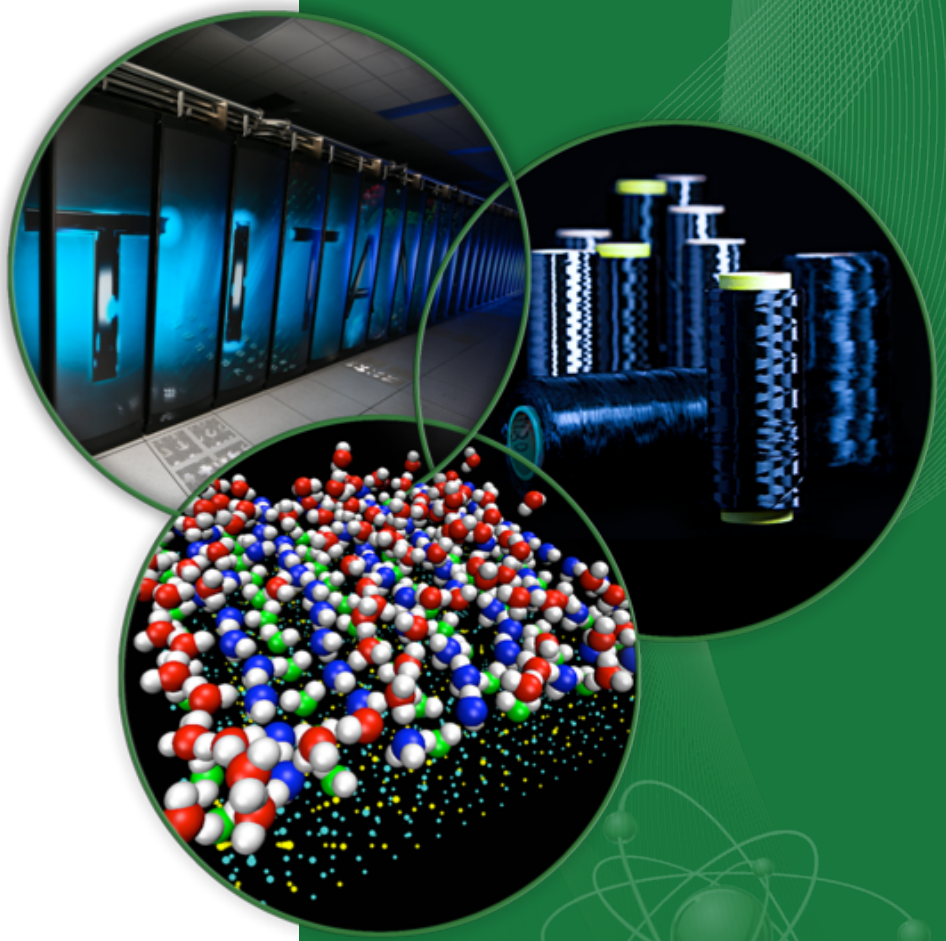
Introduction to the NVIDIA Profilers (nvprof/nvvp)

Tom Papatheodore
HPC User Support Specialist/Programmer

February 8, 2018



ORNL is managed by UT-Battelle
for the US Department of Energy

OAK RIDGE National Laboratory | OAK RIDGE LEADERSHIP COMPUTING FACILITY

First Things First…

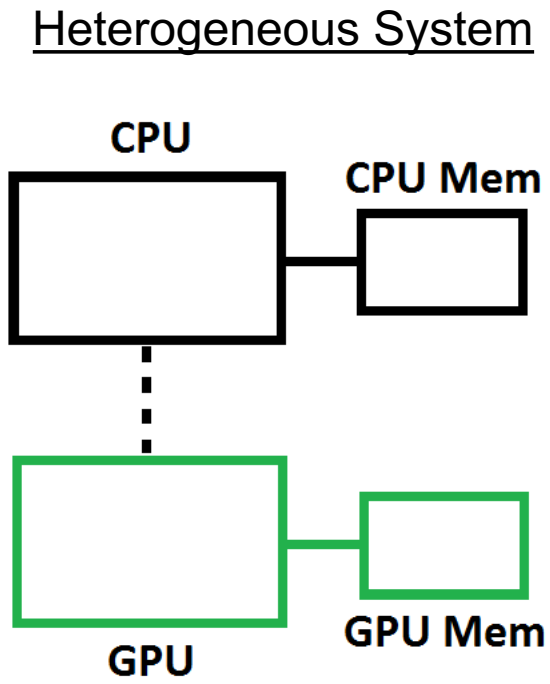
- Do you have latest CUDA Toolkit installed on your local machine?
  - https://developer.nvidia.com/nvidia-visual-profiler
- **Everyone** will need to use a training token to run on SummitDev during this tutorial, even if you already have access to the machine.
- We will be running pre-made examples to introduce features of the profilers.
  - There is a wide range of GPU-programming experience, so we will not assume a knowledge of any specific model.
- We will not be altering the code, Makefiles, or job launch scripts.
  - Some participants have never run on SummitDev before, and learning about LSF, jsrun, etc. would be a separate tutorial.

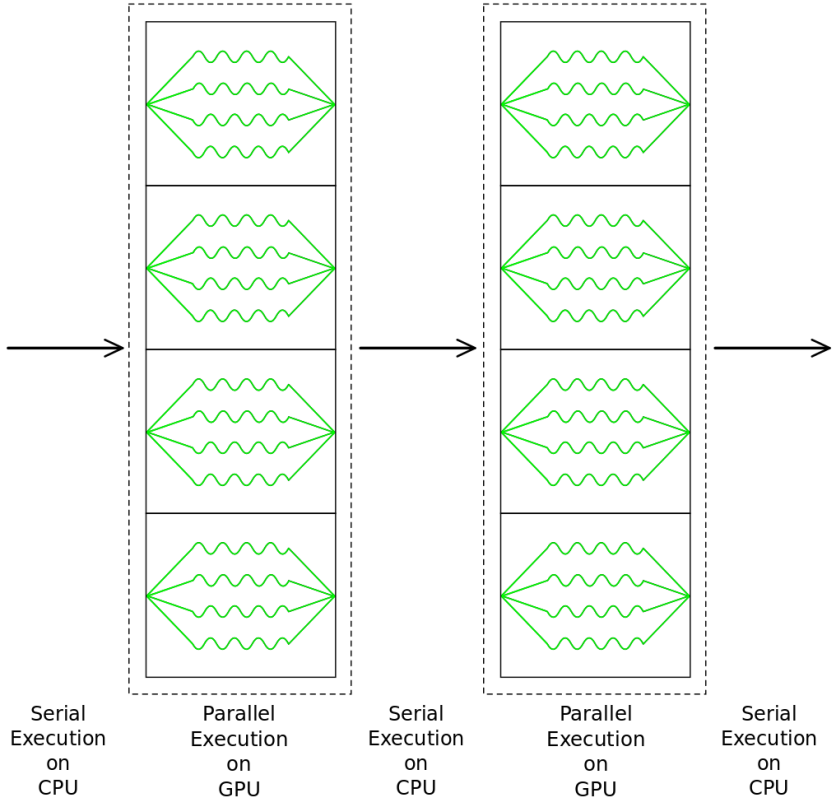
Outline

- **Programming model for heterogeneous architectures**
- **SummitDev System**
- **Simple Example – vector addition**
  - Use to demonstrate workflow & basic features of profiler
- **Poisson Equation w/Jacobi Iteration**
  - Show how profiler can be used when accelerating this code
  - Single-GPU: naïve versus improved data transfers
  - Multiple GPUs with OpenMP
  - Multiple GPUs with MPI
- **Guided Analysis**
  - Look for optimization opportunities in your code

Heterogeneous Programming Model

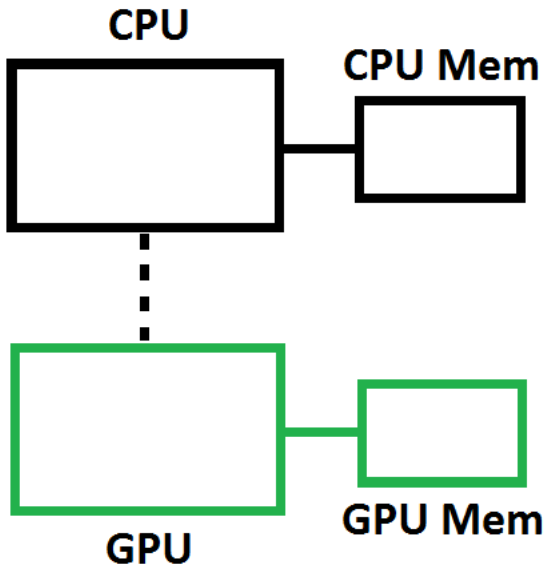
- Compute-intensive portions of an application get offloaded to run on the GPUs, while the remainder of the code continues to run on the CPUs.





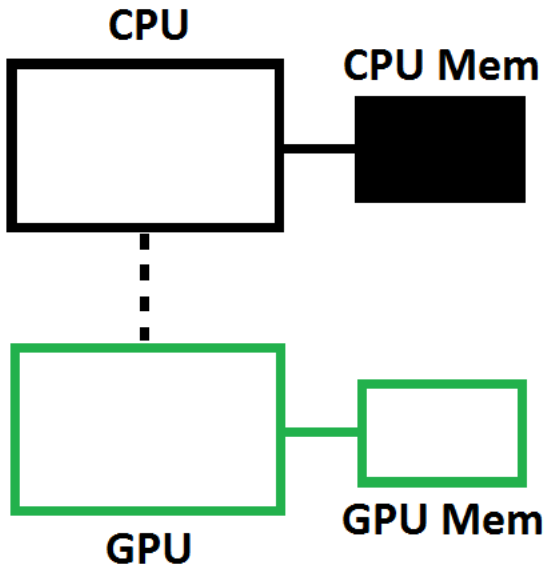
A Basic Hybrid Program Outline

```
int main(){
  // Allocate memory for array on host
  // Allocate memory for array on device
  // Fill array on host
  // Copy data from host array to device array
  // Do something on device (e.g. vector addition)
  // Copy data from device array to host array
  // Check data for correctness
  // Free Device Memory
  // Free Host Memory
}
```



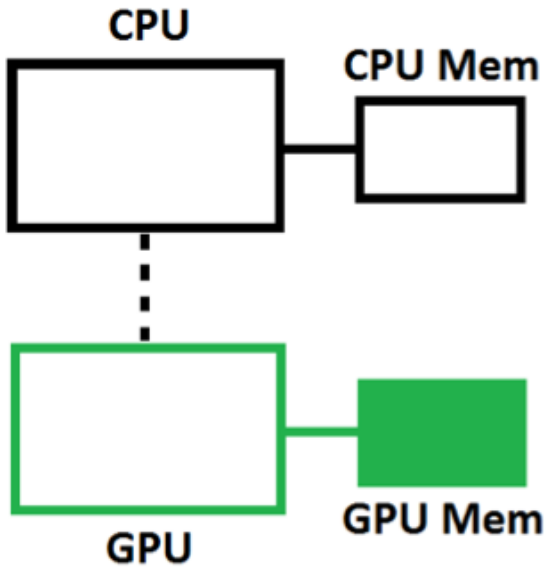
A Basic Hybrid Program Outline

```
int main(){
  // Allocate memory for array on host
  // Allocate memory for array on device
  // Fill array on host
  // Copy data from host array to device array
  // Do something on device (e.g. vector addition)
  // Copy data from device array to host array
  // Check data for correctness
  // Free Device Memory
  // Free Host Memory
}
```



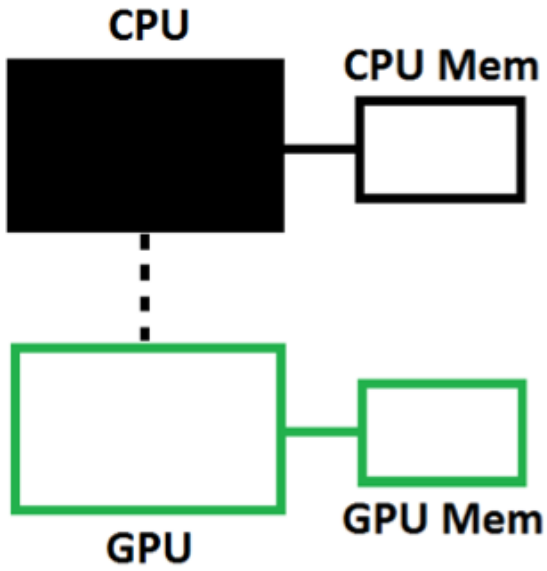
A Basic Hybrid Program Outline

```
int main(){
  // Allocate memory for array on host
  // Allocate memory for array on device
  // Fill array on host
  // Copy data from host array to device array
  // Do something on device (e.g. vector addition)
  // Copy data from device array to host array
  // Check data for correctness
  // Free Device Memory
  // Free Host Memory
}
```



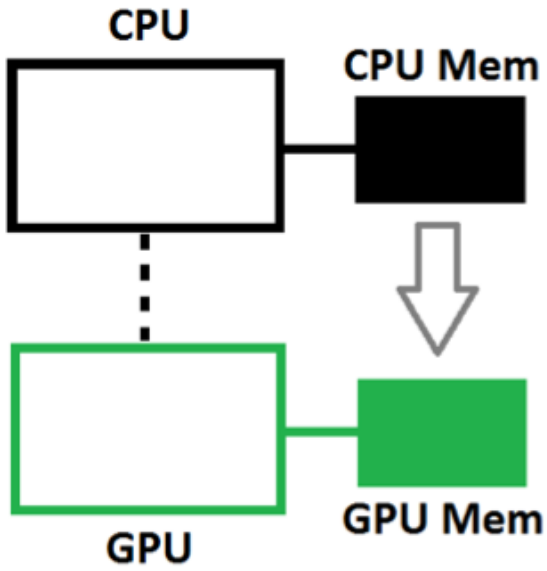
A Basic Hybrid Program Outline

```
int main(){
  // Allocate memory for array on host
  // Allocate memory for array on device
  // Fill array on host
  // Copy data from host array to device array
  // Do something on device (e.g. vector addition)
  // Copy data from device array to host array
  // Check data for correctness
  // Free Device Memory
  // Free Host Memory
}
```



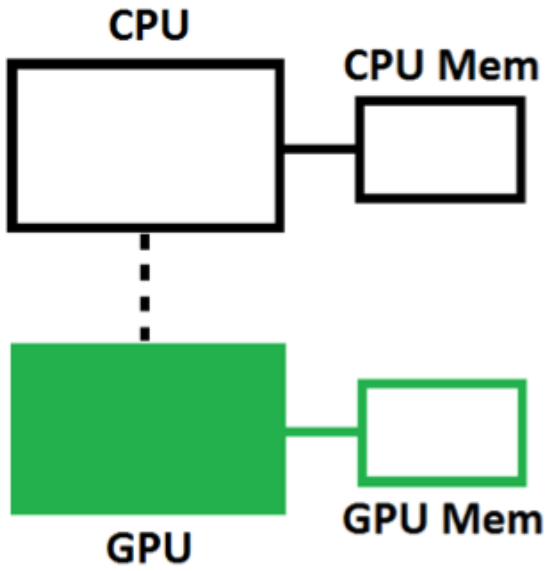
A Basic Hybrid Program Outline

```
int main(){
  // Allocate memory for array on host
  // Allocate memory for array on device
  // Fill array on host
  // Copy data from host array to device array
  // Do something on device (e.g. vector addition)
  // Copy data from device array to host array
  // Check data for correctness
  // Free Device Memory
  // Free Host Memory
}
```



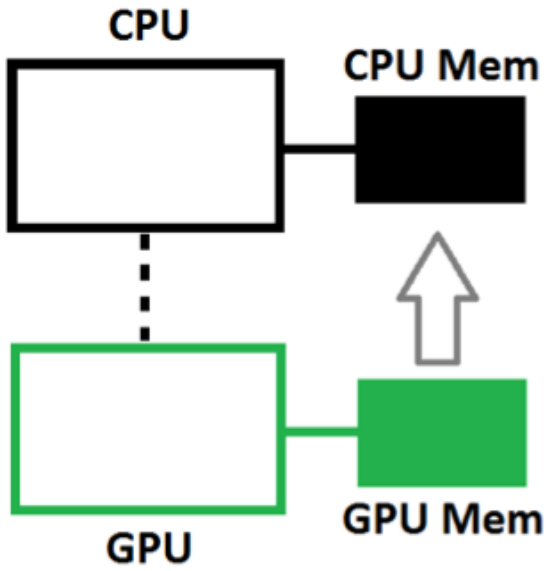
A Basic Hybrid Program Outline

```
int main(){
  // Allocate memory for array on host
  // Allocate memory for array on device
  // Fill array on host
  // Copy data from host array to device array
  // Do something on device (e.g. vector addition)
  // Copy data from device array to host array
  // Check data for correctness
  // Free Device Memory
  // Free Host Memory
}
```



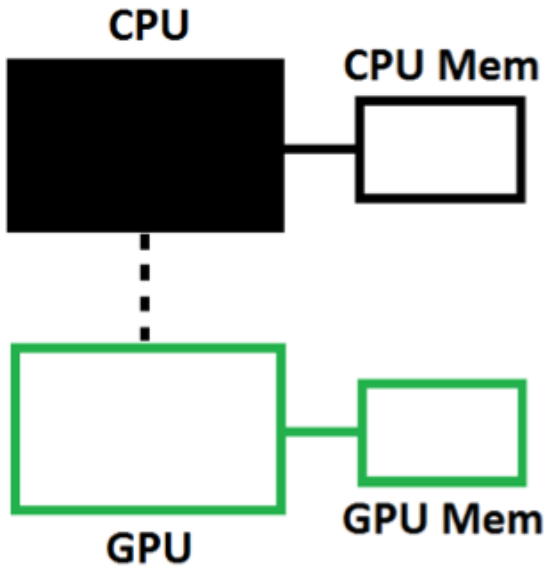
A Basic Hybrid Program Outline

```
int main(){
  // Allocate memory for array on host
  // Allocate memory for array on device
  // Fill array on host
  // Copy data from host array to device array
  // Do something on device (e.g. vector addition)
  // Copy data from device array to host array
  // Check data for correctness
  // Free Device Memory
  // Free Host Memory
}
```



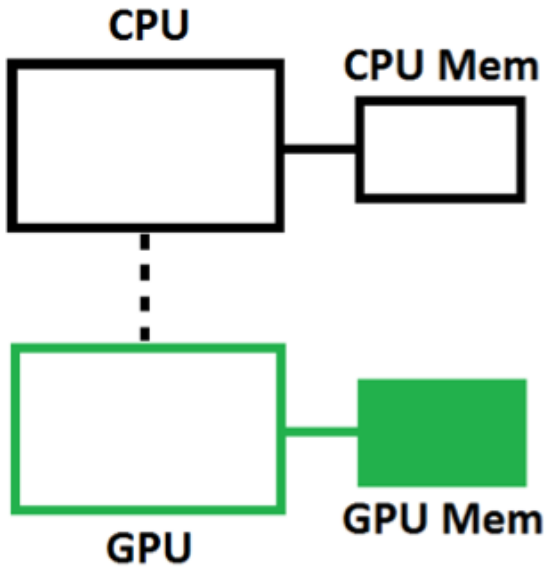
A Basic Hybrid Program Outline

```
int main(){
  // Allocate memory for array on host
  // Allocate memory for array on device
  // Fill array on host
  // Copy data from host array to device array
  // Do something on device (e.g. vector addition)
  // Copy data from device array to host array
  // Check data for correctness
  // Free Device Memory
  // Free Host Memory
}
```



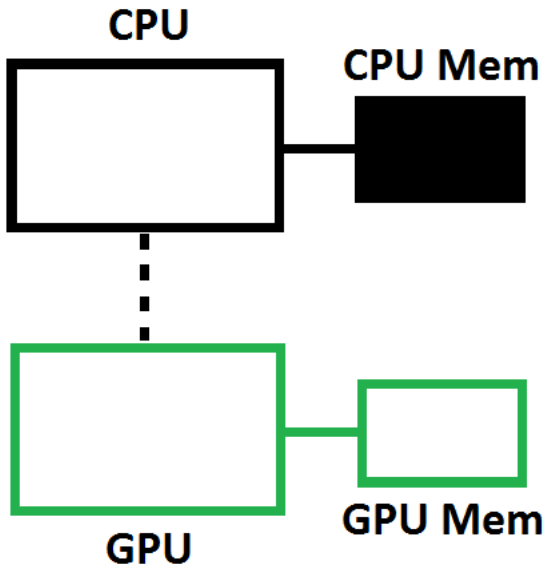
A Basic Hybrid Program Outline

```
int main(){
  // Allocate memory for array on host
  // Allocate memory for array on device
  // Fill array on host
  // Copy data from host array to device array
  // Do something on device (e.g. vector addition)
  // Copy data from device array to host array
  // Check data for correctness
  // Free Device Memory
  // Free Host Memory
}
```



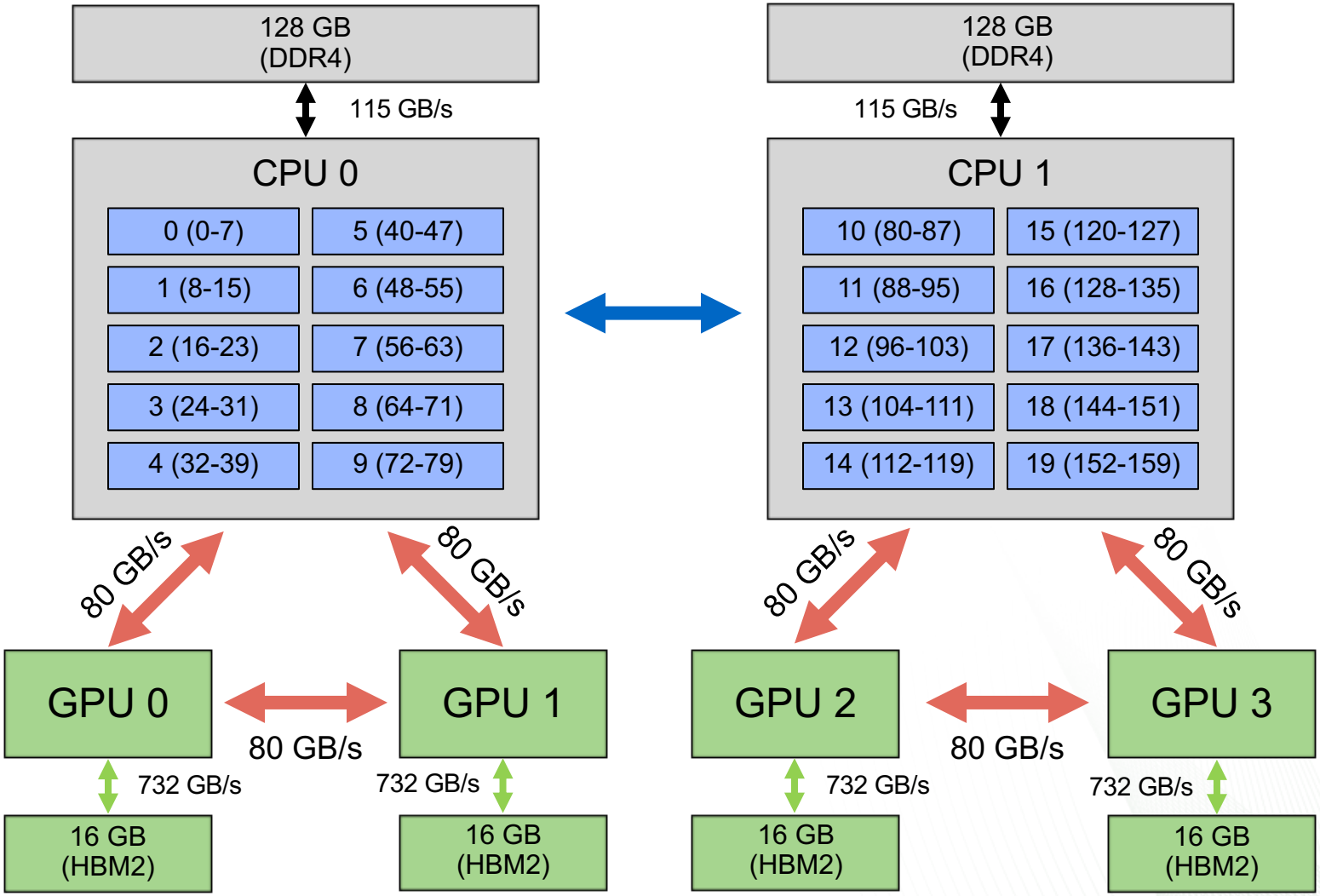
A Basic Hybrid Program Outline

```
int main(){
  // Allocate memory for array on host
  // Allocate memory for array on device
  // Fill array on host
  // Copy data from host array to device array
  // Do something on device (e.g. vector addition)
  // Copy data from device array to host array
  // Check data for correctness
  // Free Device Memory
  // Free Host Memory
}
```



SummitDev "Minsky" Node

(2) IBM Power8 + (4) NVIDIA Pascal P100



Activate RSA Training Token

Activating your SecurID Token:

1. Initiate an SSH connection to home.ccs.ornl.gov using your OLCF username.
2. When prompted for a PASSCODE, enter the 6-digit token code displayed on your fob.
3. When asked if you are ready to set your PIN, answer with 'y'.
4. You will then be prompted to enter a PIN. Enter a 4-8 character alphanumeric pin you can remember. You will then be prompted to re-enter your PIN.
5. A message will appear stating that your PIN has been accepted. Press enter to continue.
6. Finally, you will be prompted again with "Enter PASSCODE". This time enter both your PIN and the 6-digit token code displayed on your fob before pressing enter.
7. Your PIN is now set and you are logged into home.ccs.ornl.gov

To use:
When prompted for your PASSCODE, enter your PIN + the currently displayed token code. For example, if your PIN is 1234 and the token code is 987654, enter 1234987654.

Note: The 6-digit code displayed on the SecurID fob can only be used once. If prompted for multiple PASSCODE entries, always allow the 6-digit code to change between entries. Re-using the 6-digit code can cause your account to be automatically disabled.

Logging in to SummitDev

1. `ssh username@home.ccs.ornl.gov`
2. `ssh summitdev`

NOTE: Use the csepXXX username on the front of the envelope your token came in.

Clone Repository & Setup Environment

- `git clone https://github.com/tpapathe/2018_ECP_nvprof_tutorial.git`
- `cd 2018_ECP_nvprof_tutorial`
- `./interactive_job.sh`
- `source environment.sh`

This will start an interactive job on SummitDev

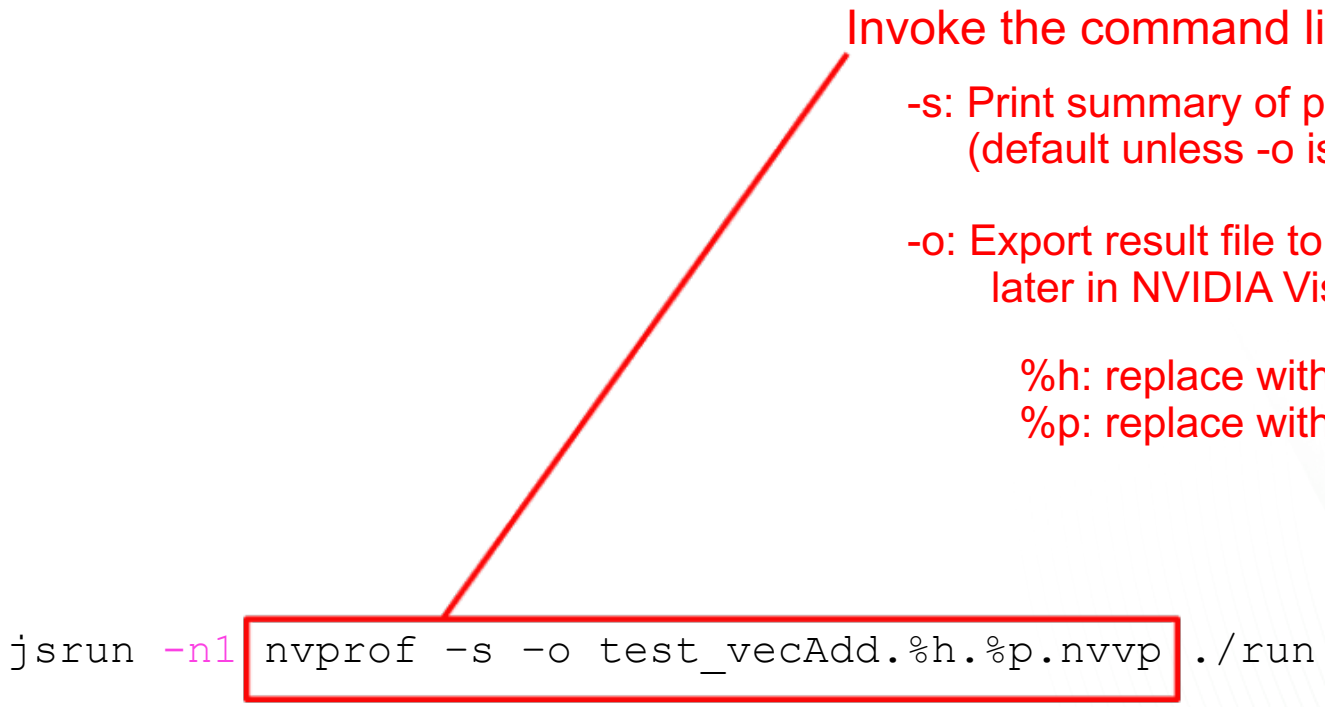
This will change from xl to pgi/17.10, load CUDA9, and change prompt

Simple Example: Vector Addition

- Show basic workflow we'll be following
- Introduce general features of the text-based (nvprof) and visual profilers (nvvp).

Simple Example: Vector Addition

- `cd 2018_ECP_nvprof_tutorial/01_vector_addition`
- `make`
- `./launch.sh`



Simple Example: Vector Addition

- nvprof results (text only)

```
==101250== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   49.62%  602.57us         1  602.57us  602.57us  602.57us  [CUDA memcpy DtoH]
                   48.18%  585.10us         2  292.55us  273.51us  311.59us  [CUDA memcpy HtoD]
                    2.20%  26.657us         1  26.657us  26.657us  26.657us  add_vectors(int*, int*, int*)
      API calls:   97.06%  227.58ms         3  75.861ms  679.13us  226.22ms  cudaMalloc
                    1.09%  2.5516ms       376  6.7860us     255ns  278.18us  cuDeviceGetAttribute
                    0.80%  1.8653ms         3  621.76us  336.15us  1.1088ms  cudaMemcpy
                    0.65%  1.5215ms         4  380.38us  374.56us  395.25us  cuDeviceTotalMem
                    0.28%  660.41us         3  220.14us  192.70us  267.78us  cudaFree
                    0.09%  222.25us         4  55.561us  51.912us  63.914us  cuDeviceGetName
                    0.02%  45.061us         1  45.061us  45.061us  45.061us  cudaLaunch
                    0.01%  17.531us         3  5.8430us     494ns  16.438us  cudaSetupArgument
                    0.00%  4.1950us         3  1.3980us     397ns  3.2300us  cuDeviceGetCount
                    0.00%  3.6890us         8     461ns     275ns     698ns  cuDeviceGet
                    0.00%  3.1950us         1  3.1950us  3.1950us  3.1950us  cudaConfigureCall
```

GPU activities

Simple Example: Vector Addition

- nvprof results (text only)

```
==101250== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   49.62%  602.57us         1  602.57us  602.57us  602.57us  [CUDA memcpy DtoH]
                   48.18%  585.10us         2  292.55us  273.51us  311.59us  [CUDA memcpy HtoD]
                    2.20%  26.657us         1  26.657us  26.657us  26.657us  add_vectors(int*, int*, int*)
      API calls:   97.06%  227.58ms         3  75.861ms  679.13us  226.22ms  cudaMalloc
                    1.09%  2.5516ms       376  6.7860us     255ns  278.18us  cuDeviceGetAttribute
                    0.80%  1.8653ms         3  621.76us  336.15us  1.1088ms  cudaMemcpy
                    0.65%  1.5215ms         4  380.38us  374.56us  395.25us  cuDeviceTotalMem
                    0.28%  660.41us         3  220.14us  192.70us  267.78us  cudaFree
                    0.09%  222.25us         4  55.561us  51.912us  63.914us  cuDeviceGetName
                    0.02%  45.061us         1  45.061us  45.061us  45.061us  cudaLaunch
                    0.01%  17.531us         3  5.8430us     494ns  16.438us  cudaSetupArgument
                    0.00%  4.1950us         3  1.3980us     397ns  3.2300us  cuDeviceGetCount
                    0.00%  3.6890us         8     461ns     275ns     698ns  cuDeviceGet
                    0.00%  3.1950us         1  3.1950us  3.1950us  3.1950us  cudaConfigureCall
```

API Calls

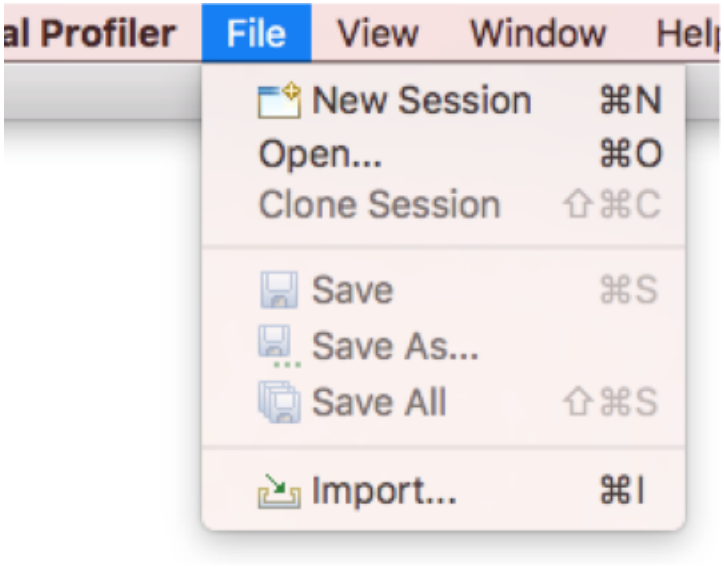
Simple Example: Vector Addition

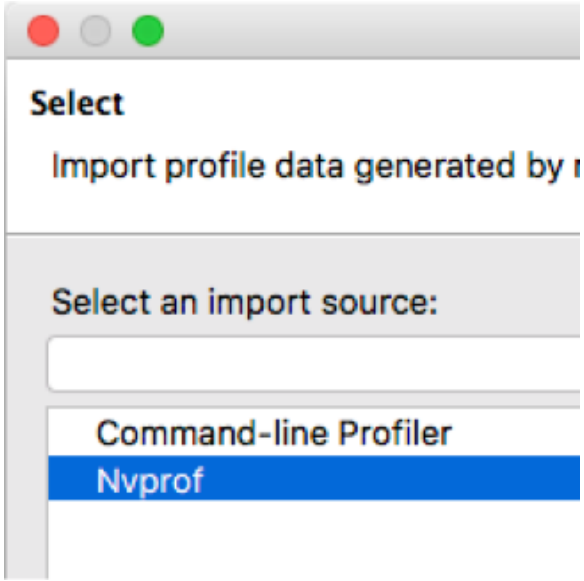
Now transfer .nvvp file to your local machine to view in NVIDIA Visual Profiler.

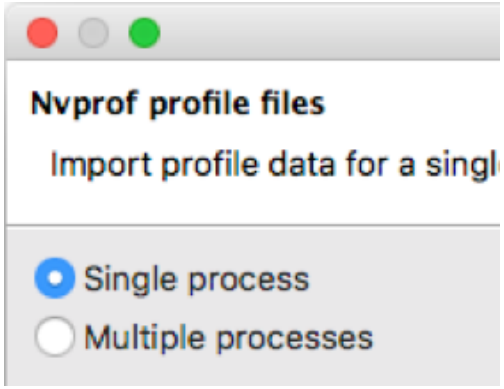
- `scp username@dtn.ccs.ornl.gov:/path/to/file/remote /path/to/desired/location/local`

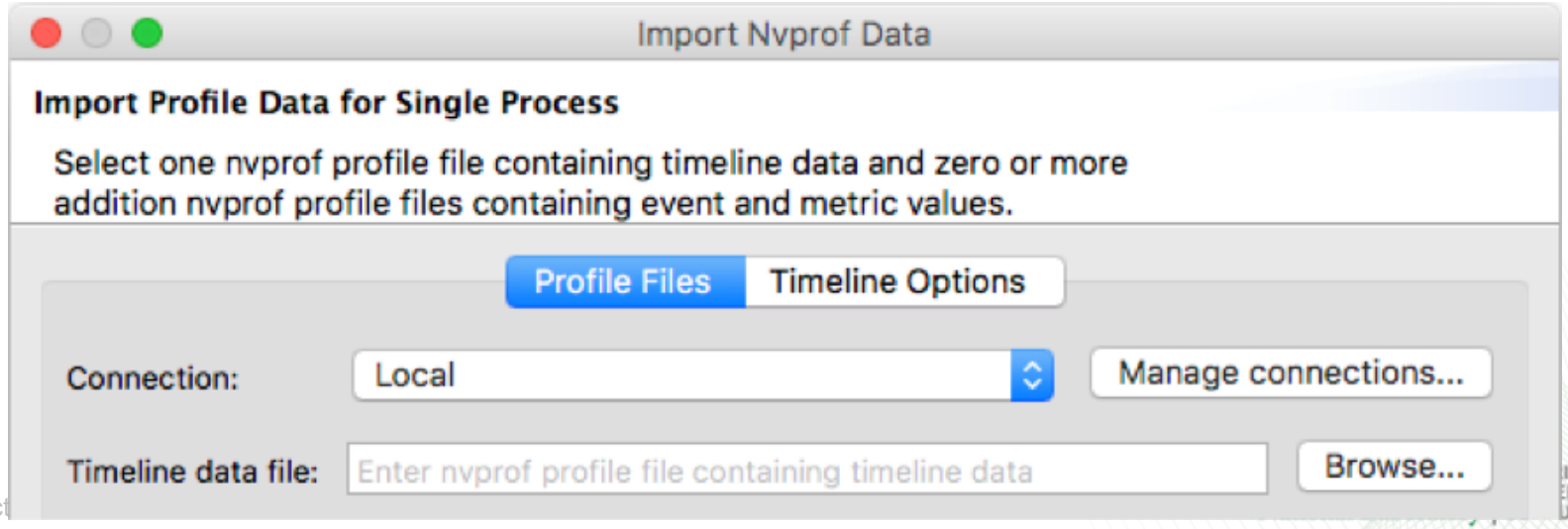
Simple Example: Vector Addition

Now import .nvvp file into NVIDIA Visual Profiler.

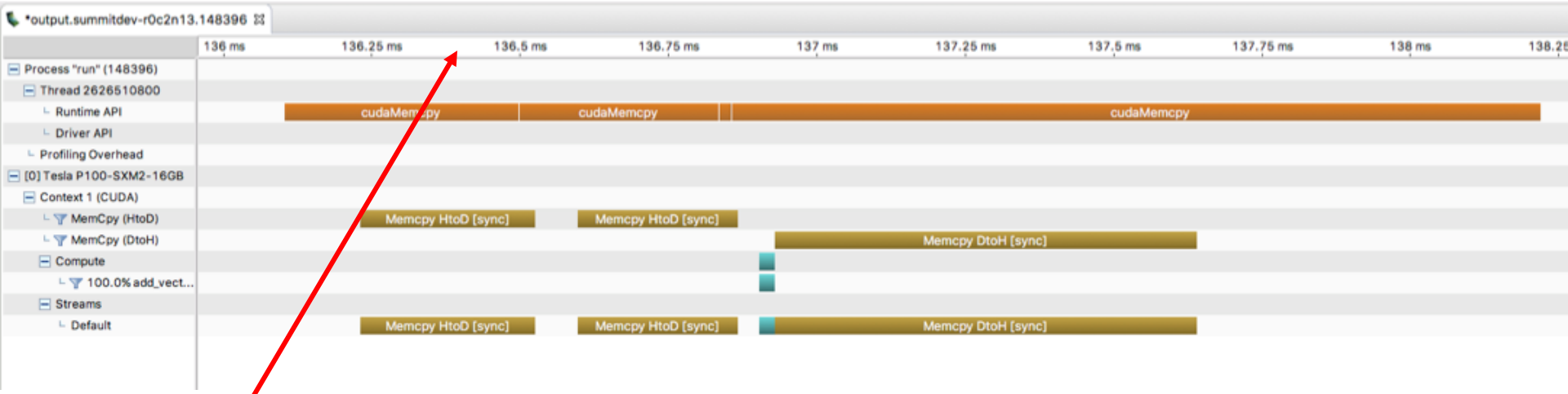








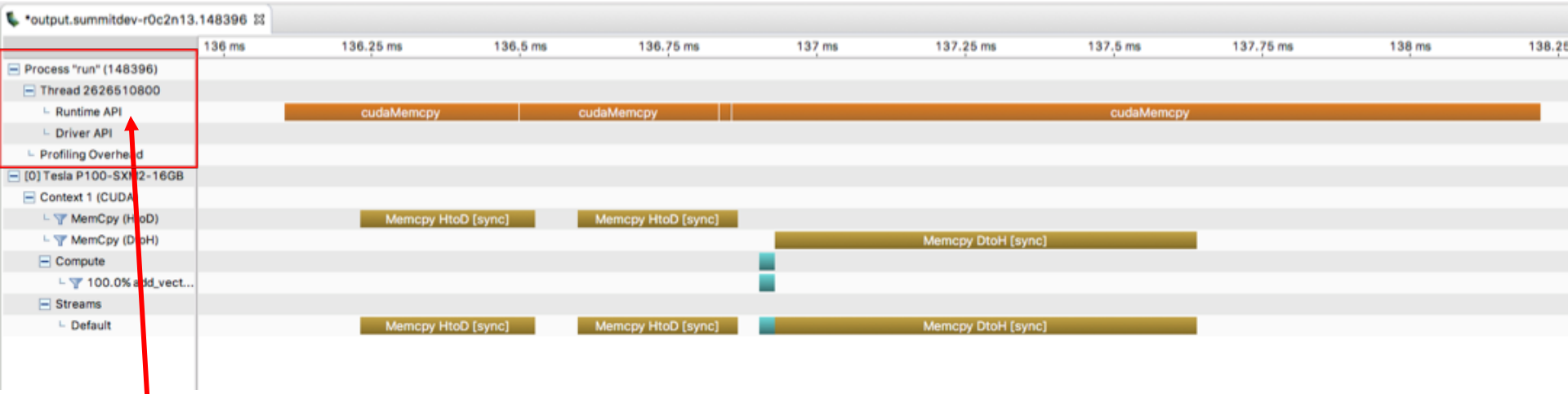
Simple Example: Vector Addition



Timeline

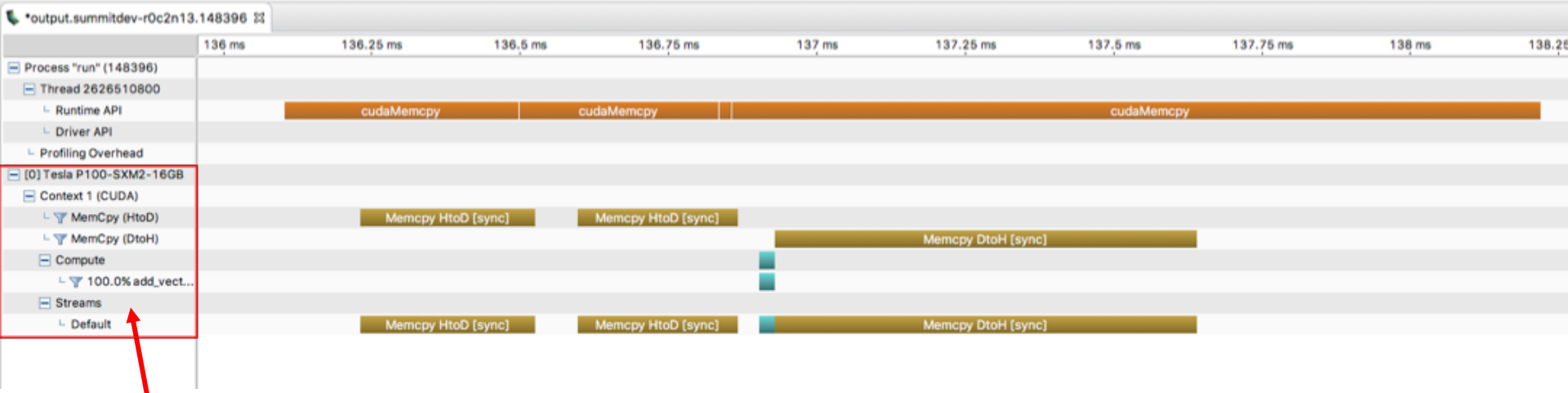
- Zoom in: hold Ctrl + drag mouse (Cmd for Mac)
- Left-click drag mouse to measure specific activities

Simple Example: Vector Addition



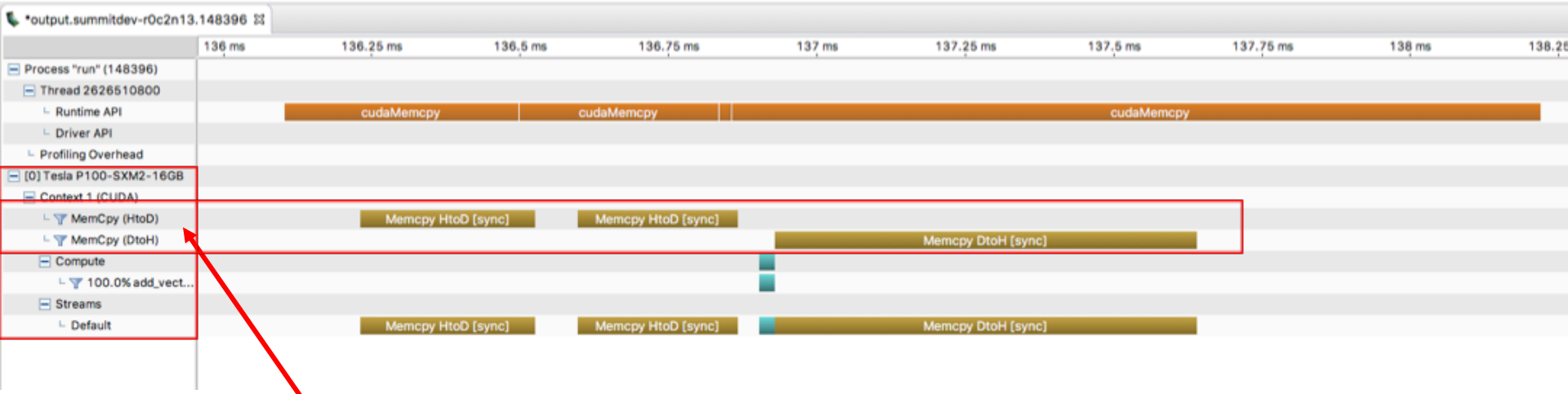
CUDA API Activity from CPU process

Simple Example: Vector Addition



GPU Activity

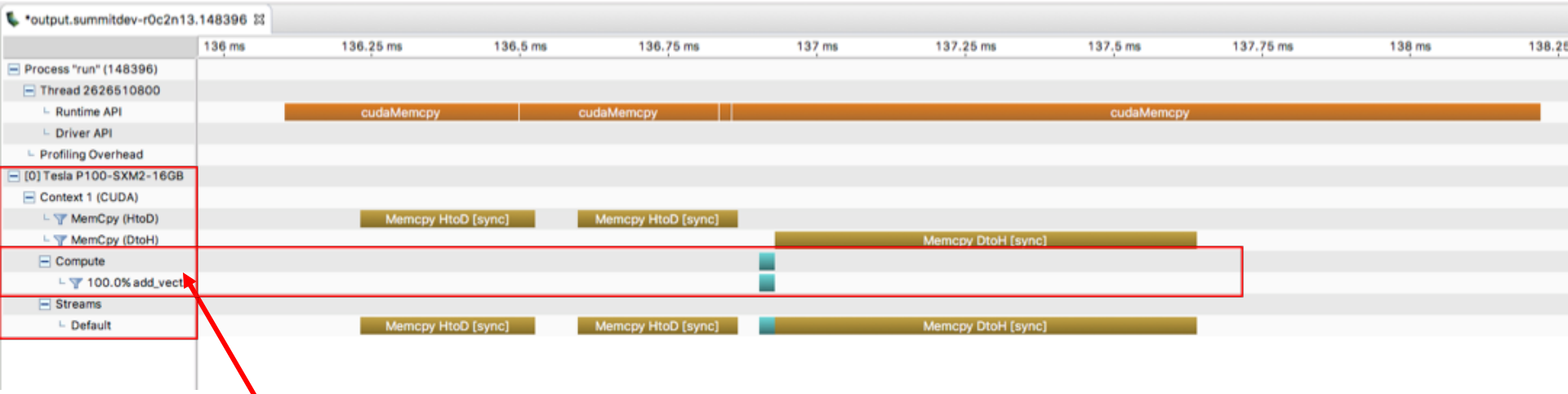
Simple Example: Vector Addition



GPU Activity

- HtoD and DtoH memory copies

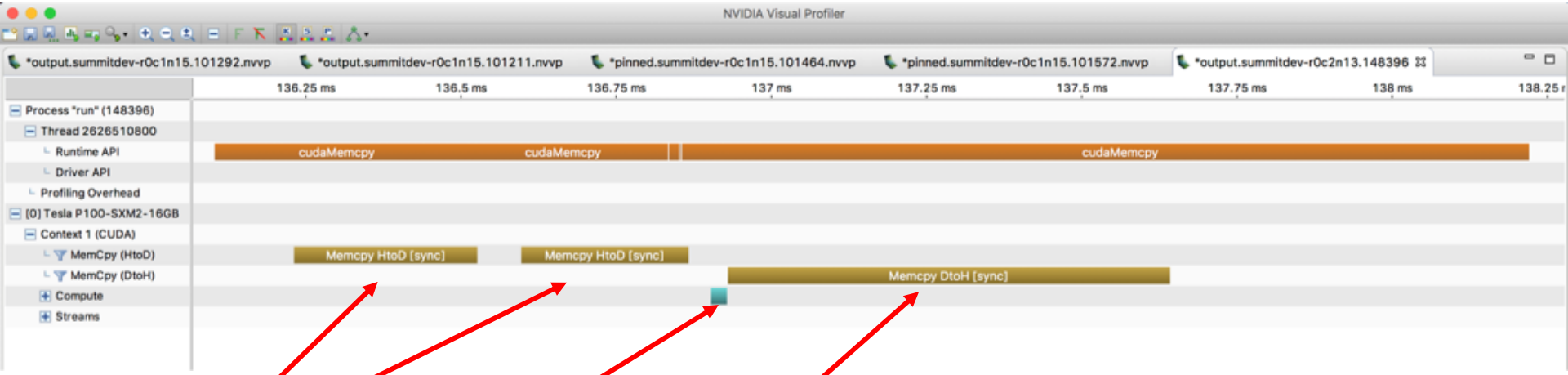
Simple Example: Vector Addition



GPU Activity

- Compute activities (in this case, our add_vectors kernel)

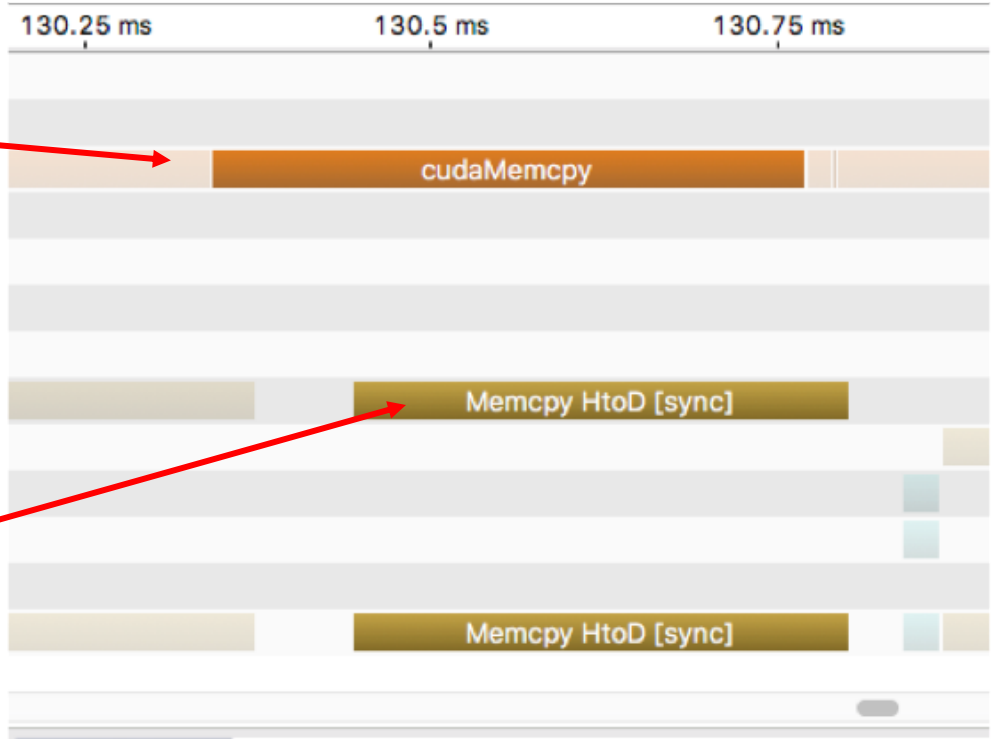
Simple Example: Vector Addition




```
// Copy data from host arrays A and B to device arrays d_A and d_B
cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);

// Set execution configuration parameters
//    thr_per_blk: number of CUDA threads per grid block
//    blk_in_grid: number of blocks in grid
int thr_per_blk = 256;
int blk_in_grid = ceil( float(N) / thr_per_blk );

// Launch kernel
add_vectors<<< blk_in_grid, thr_per_blk >>>(d_A, d_B, d_C);

// Copy data from device array d_C to host array C
cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);
```

```
==101250== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   49.62%  602.57us         1  602.57us  602.57us  602.57us  [CUDA memcpy DtoH]
                   48.18%  585.10us         2  292.55us  273.51us  311.59us  [CUDA memcpy HtoD]
                    2.20%  26.657us         1  26.657us  26.657us  26.657us  add_vectors(int*, int*, int*)
```

CUDA API Call

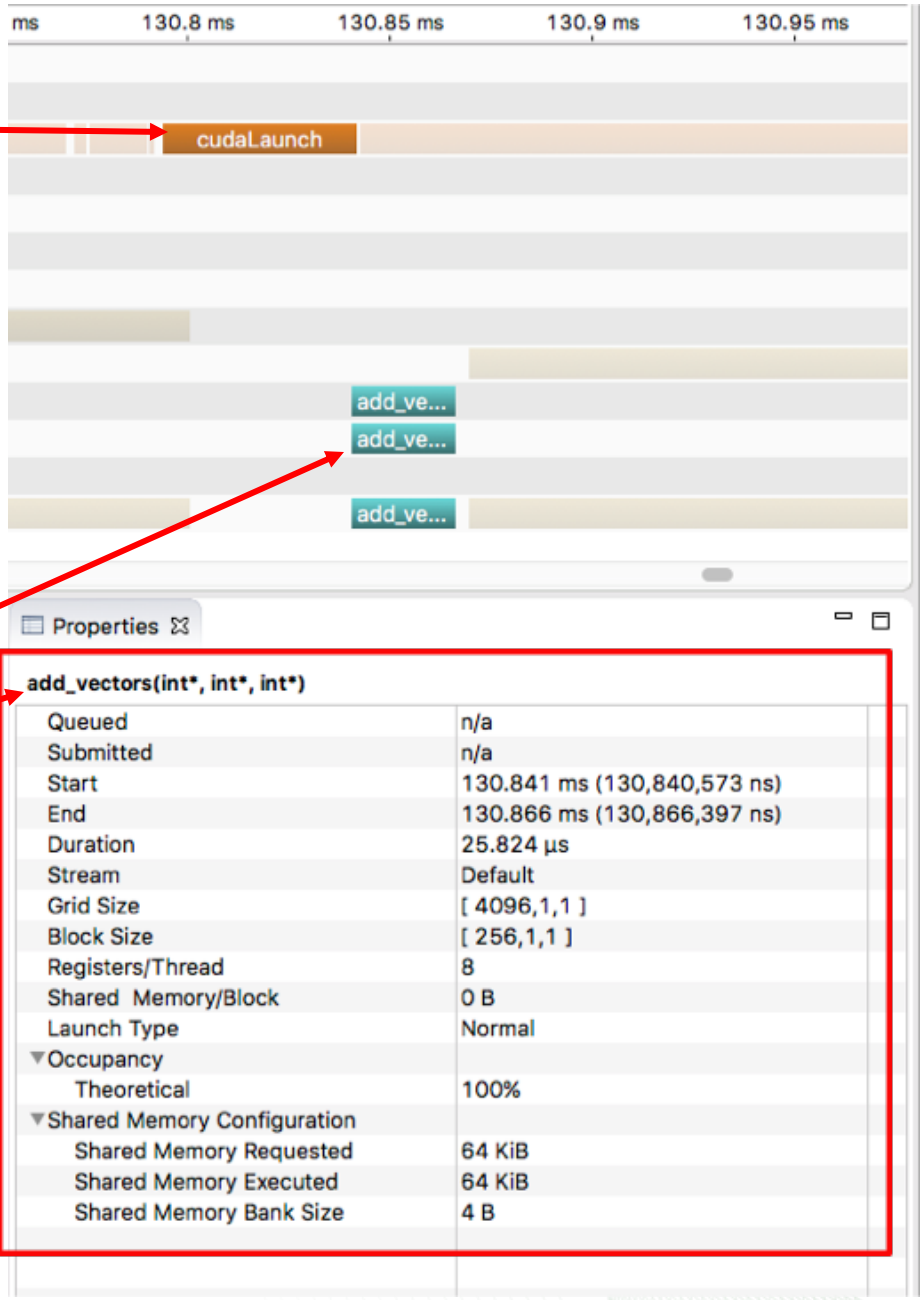



```
// Copy data from host arrays A and B to device arrays d_A and d_B
cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);

// Set execution configuration parameters
//    thr_per_blk: number of CUDA threads per grid block
//    blk_in_grid: number of blocks in grid
int thr_per_blk = 256;
int blk_in_grid = ceil( float(N) / thr_per_blk );

// Launch kernel
add_vectors<<< blk_in_grid, thr_per_blk >>>(d_A, d_B, d_C);

// Copy data from device array d_C to host array C
cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);
```

Properties

Memcpy HtoD [sync]	
Start	130.444 ms (130,444,344 ns)
End	130.801 ms (130,800,540 ns)
Duration	356.196 µs
Size	4.194 MB
Throughput	11.775 GB/s
Stream	Default
▼Memory Type	
Source	Pageable
Destination	Device

Kernel Launch




```
// Copy data from host arrays A and B to device arrays d_A and d_B
cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);

// Set execution configuration parameters
//    thr_per_blk: number of CUDA threads per grid block
//    blk_in_grid: number of blocks in grid
int thr_per_blk = 256;
int blk_in_grid = ceil( float(N) / thr_per_blk );

// Launch kernel
add_vectors<<< blk_in_grid, thr_per_blk >>>(d_A, d_B, d_C);

// Copy data from device array d_C to host array C
cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);
```

Simple Example: Vector Addition

Steps of workflow for using Visual Profiler in this tutorial

1. Create profiler output file (.nvvp) on SummitDev
2. scp output file to local machine
   - `scp username@dtn.ccs.ornl.gov:/path/to/file/remote /path/to/desired/location/local`
3. Import output file into NVIDIA Visual Profiler

Example: Jacobi Iteration

- Show how profilers might be used when attempting to port code to GPU
  - Problem description
  - Profile CPU-only
  - Profile naïve single GPU implementation
  - Profile single GPU with data directives
  - Multiple GPUs with OpenMP
  - Multiple GPUs with MPI

Thank You, NVIDIA.

```
/* Copyright (c) 2016, NVIDIA CORPORATION. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *  * Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *  * Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *  * Neither the name of NVIDIA CORPORATION nor the names of its
 *    contributors may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS ``AS IS'' AND ANY
 * EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY
 * OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */
```

Jacobi Iteration – Problem Description

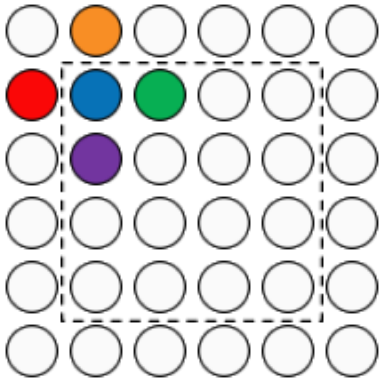
Example: Jacobi Iteration

Use Jacobi Iteration to solve 2D Poisson equation with periodic boundary conditions

$$\Delta A(y,x) = e^{-10(x*x + y*y)}$$

Example: Jacobi Iteration

- Execute a Jacobi step on inner points.



$$A_{k+1}(iy,ix) = \frac{rhs(iy,ix) - (A_k(iy,ix-1) + A_k(iy,ix+i) + A_k(iy-1,ix) + A_k(iy+1,ix))}{4}$$

```
for (int iy = 1; iy < NY-1; iy++)
{
    for( int ix = 1; ix < NX-1; ix++ )
    {
        Anew[iy][ix] = -0.25 * (rhs[iy][ix] - ( A[iy][ix+1] + A[iy][ix-1]
                                              + A[iy-1][ix] + A[iy+1][ix] ));
        error = fmaxr( error, fabsr(Anew[iy][ix]-A[iy][ix]));
    }
}
```

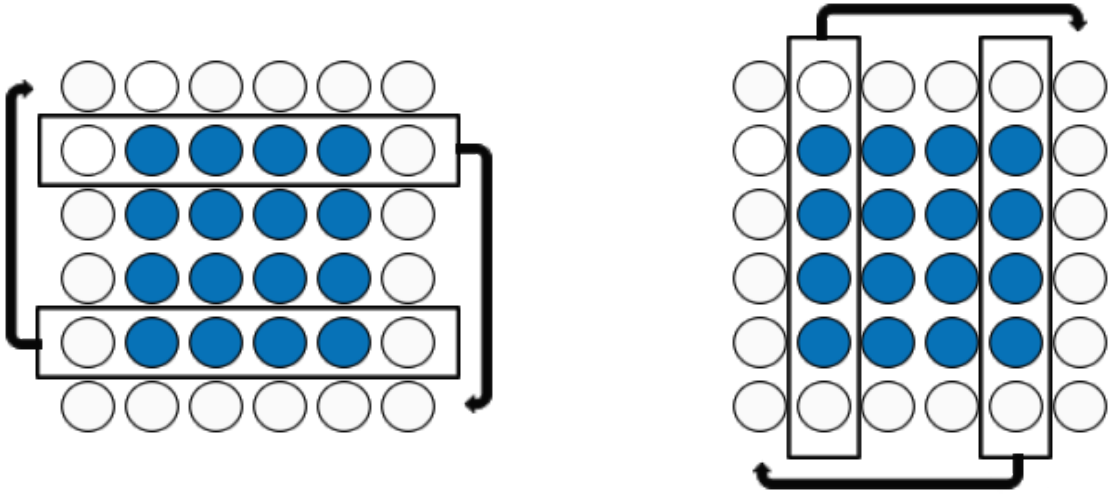
Example: Jacobi Iteration

- Copy new values of Anew to A.

```
for (int iy = 1; iy < NY-1; iy++)
{
    for( int ix = 1; ix < NX-1; ix++ )
    {
        A[iy][ix] = Anew[iy][ix];
    }
}
```

Example: Jacobi Iteration

- Apply Periodic Boundary Conditions



```
//Periodic boundary conditions
for( int ix = 1; ix < NX-1; ix++ )
{
        A[0][ix]      = A[(NY-2)][ix];
        A[(NY-1)][ix] = A[1][ix];
}
for (int iy = 1; iy < NY-1; iy++)
{
        A[iy][0]      = A[iy][(NX-2)];
        A[iy][(NX-1)] = A[iy][1];
}
```

Jacobi Iteration – CPU Profiling

Example: Jacobi Iteration

Run CPU-only version (nvprof)

- `cd 02_Jacobi_single_CPU`
- `make`
- `./launch.sh`

```
jsrun -n1 nvprof --cpu-profiling on ./run
```

Enable CPU profiling

```
Jacobi relaxation Calculation: 4096 x 4096 mesh
    0, 0.250000
  100, 0.249940
  200, 0.249880
  300, 0.249821
  400, 0.249761
  500, 0.249702
  600, 0.249642
  700, 0.249583
  800, 0.249524
  900, 0.249464
4096x4096: 1 CPU:  49.7727 s

======== CPU profiling result (bottom up):
Time(%)      Time  Name
 78.12%    39.06s  main
 78.12%    39.06s  | generic_start_main.isra.0
 21.50%    10.75s  __c_mcopy8
 21.50%    10.75s  | generic_start_main.isra.0
  0.28%     140ms  __xlmass_expd2
  0.26%     130ms  | __fd_exp_2
  0.26%     130ms  | | main
  0.26%     130ms  | |   generic_start_main.isra.0
  0.02%      10ms  | main
  0.02%      10ms  |   generic_start_main.isra.0
  0.04%      20ms  00000017.plt_call.__c_mcopy8
  0.04%      20ms  | generic_start_main.isra.0
  0.04%      20ms  __fd_exp_2
  0.02%      10ms  | generic_start_main.isra.0
  0.02%      10ms  | main
  0.02%      10ms  |   generic_start_main.isra.0
  0.02%      10ms  __memset_power7
  0.02%      10ms    main
  0.02%      10ms      generic_start_main.isra.0

======== Data collected at 100Hz frequency
```

Profile only contains main since individual loops are not in separate functions.

- But most codes don't consist of only a main function

Example: Jacobi Iteration

Run CPU-only version (nvprof - with separate functions)

- `cd 03_Jacobi_single_CPU_separateFunctions`
- `make`
- `./launch.sh`

```
Jacobi relaxation Calculation: 4096 x 4096 mesh
    0, 0.250000
  100, 0.249940
  200, 0.249880
  300, 0.249821
  400, 0.249761
  500, 0.249702
  600, 0.249642
  700, 0.249583
  800, 0.249524
  900, 0.249464
4096x4096: 1 CPU: 215.9252 s

======== CPU profiling result (bottom up):
Time(%)      Time  Name
 88.20%   190.75s  calculate_jacobi_iteration
 88.20%   190.75s  | generic_start_main.isra.0
 11.51%    24.89s  __c_mcopy8
 11.51%    24.89s  | main
 11.51%    24.89s  |   generic_start_main.isra.0
  0.10%     220ms  periodic_BC
  0.10%     220ms  | generic_start_main.isra.0
  0.07%     160ms  __xlmass_expd2
  0.07%     160ms  | __fd_exp_2
  0.07%     160ms  |   set_rhs
  0.07%     160ms  |     main
  0.07%     160ms  |       generic_start_main.isra.0
  0.06%     120ms  set_rhs
  0.06%     120ms  | main
  0.06%     120ms  |   generic_start_main.isra.0
  0.01%      30ms  Anew_to_A
  0.01%      30ms  | main
  0.01%      30ms  |   generic_start_main.isra.0
  0.01%      30ms  00000017.plt_call.__c_mcopy8
```

Now the individual functions are visible in the profile

- Allows you to find the longest running functions.

NOTE: the longer runtime in this example is due to a “false” data dependency that I am still unable to eliminate.

```
calculate_jacobi_iteration:
     55, Loop not vectorized: data dependency
```

Jacobi Iteration – OpenACC (naïve)

Simply added `#pragma acc kernels` to the for loops within the main while loop

Example: Jacobi Iteration

Run naïve GPU version

(place kernels directive around each step of an iteration)

- `cd 04_Jacobi_single_device_naive`
- `make`
- `./launch.sh`

About 2.5x slower than CPU only.

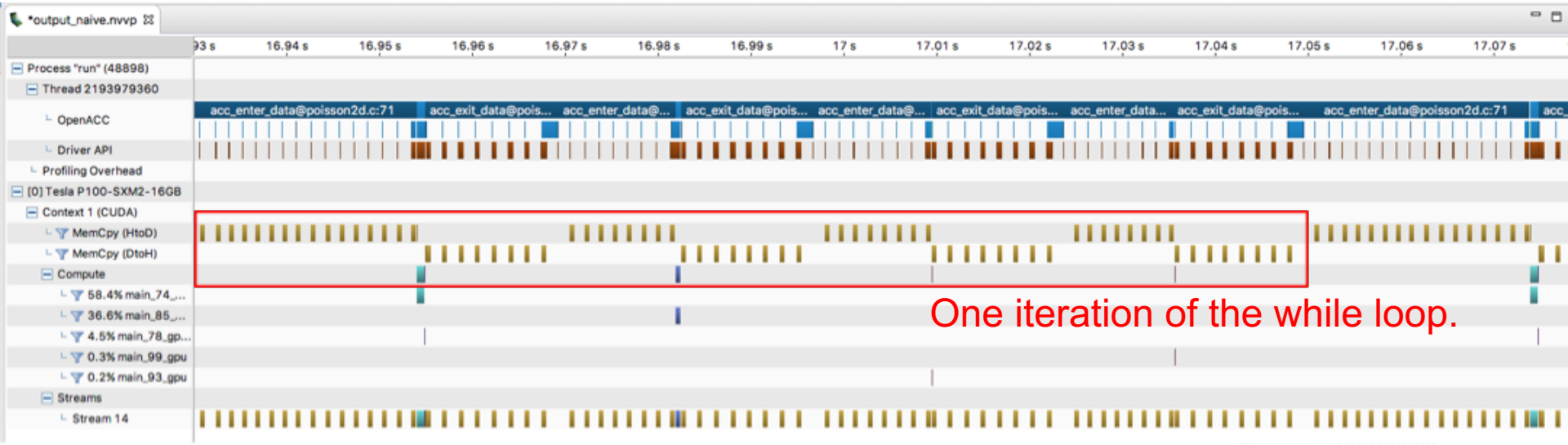
Do we really need all these data transfers?

```
==48898== NVPROF is profiling process 48898, command: ./run
==48898== Profiling application: ./run
Jacobi relaxation Calculation: 4096 x 4096 mesh
    0, 0.250000
  100, 0.249940
  200, 0.249880
  300, 0.249821
  400, 0.249761
  500, 0.249702
  600, 0.249642
  700, 0.249583
  800, 0.249524
  900, 0.249464
4096x4096: 1 GPU: 122.7411 s
==48898== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   55.14%  21.5064s     41000  524.55us     960ns  550.12us  [CUDA memcpy HtoD]
                   41.13%  16.0427s     33000  486.14us     896ns  520.07us  [CUDA memcpy DtoH]
                    2.18%  848.78ms      1000  848.78us  841.29us  855.91us  main_74_gpu
                    1.36%  531.49ms      1000  531.49us  528.93us  535.04us  main_85_gpu
                    0.17%  65.708ms      1000  65.707us  63.520us  67.649us  main_78_gpu__red
                    0.01%  4.5522ms      1000  4.5520us  3.9680us  5.4720us  main_99_gpu
                    0.01%  2.7420ms      1000  2.7410us  2.2080us  3.2640us  main_93_gpu
```

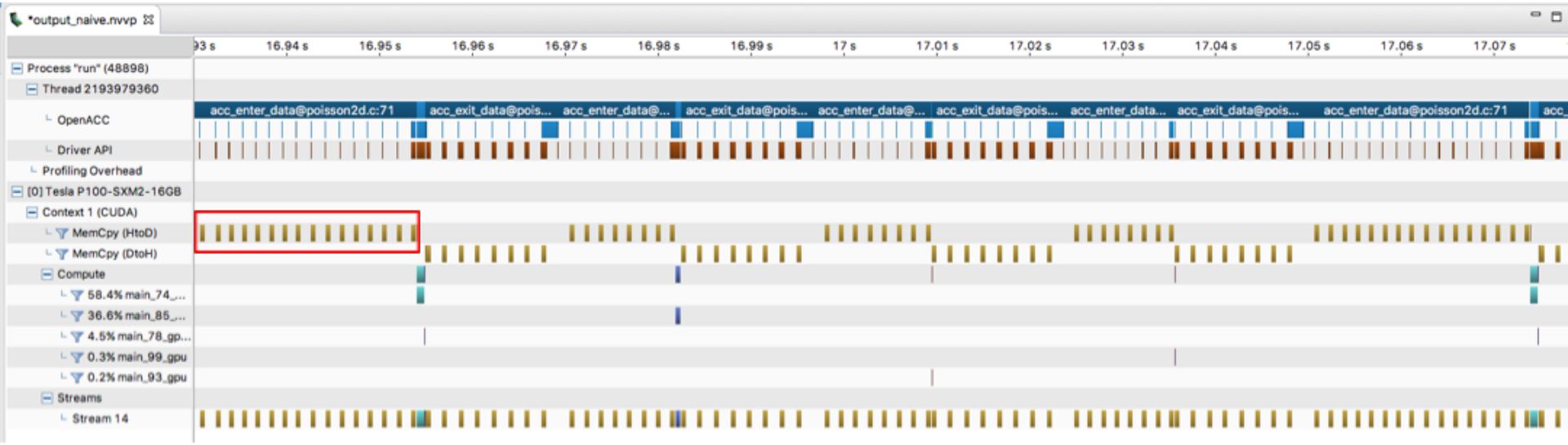
Example: Jacobi Iteration

1. Create profiler output file (.nvvp) on SummitDev
2. scp output file to local machine
3. Import output file into NVIDIA Visual Profiler

```
scp username@dtn.ccs.ornl.gov:/path/to/file/remote  /path/to/desired/location/local
```

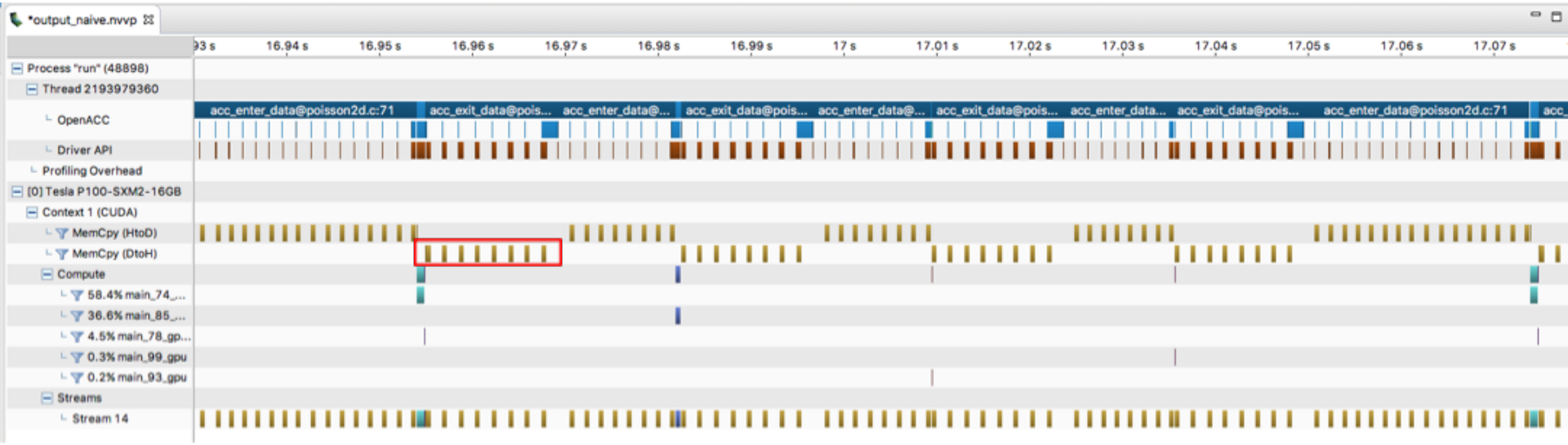


Many unnecessary HtoD and DtoH data transfers



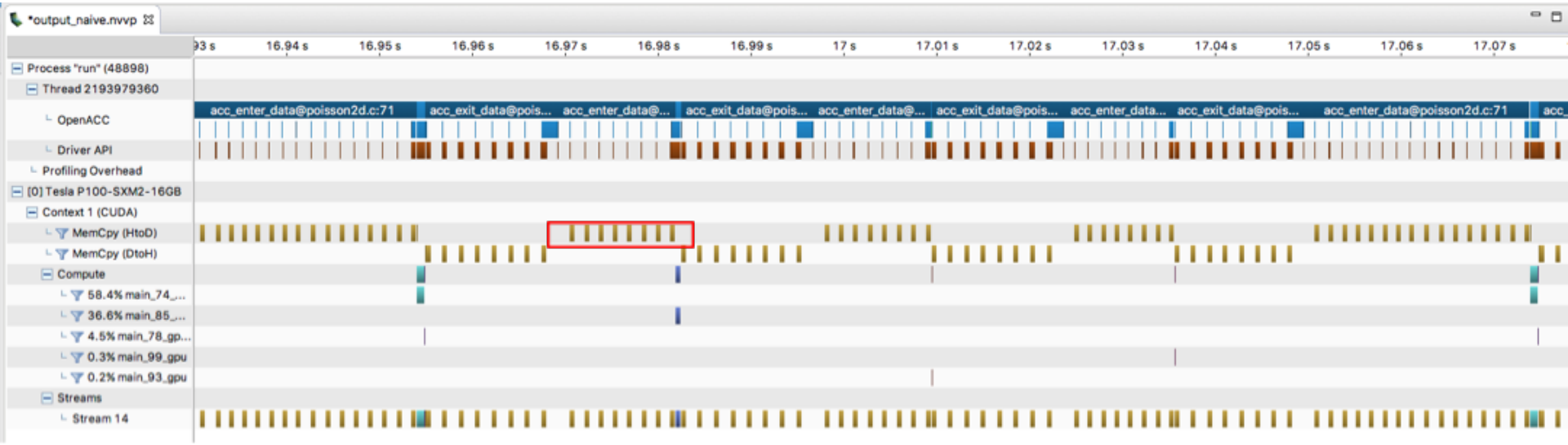

```
pgcc -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc60 -c  poisson2d.c
main:
     71, Generating implicit copyin(A[:][:])
         Generating implicit copyout(Anew[1:4094][1:4094])
         Generating implicit copyin(rhs[1:4094][1:4094])
     72, Loop is parallelizable
     74, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         72, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         74, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
         78, Generating implicit reduction(max:error)
     82, Generating implicit copyin(Anew[1:4094][1:4094])
         Generating implicit copyout(A[1:4094][1:4094])
     83, Loop is parallelizable
     85, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         83, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         85, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
     92, Generating implicit copy(A[:][1:4094])
     93, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         93, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     98, Generating implicit copy(A[1:4094][:])
     99, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         99, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
pgcc -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc60 poisson2d.o -o run
```

Transfer A and rhs from HtoD




```
pgcc -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc60 -c  poisson2d.c
main:
     71, Generating implicit copyin(A[:][:])
         Generating implicit copyout(Anew[1:4094][1:4094])
         Generating implicit copyin(rhs[1:4094][1:4094])
     72, Loop is parallelizable
     74, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         72, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         74, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
         78, Generating implicit reduction(max:error)
     82, Generating implicit copyin(Anew[1:4094][1:4094])
         Generating implicit copyout(A[1:4094][1:4094])
     83, Loop is parallelizable
     85, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         83, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         85, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
     92, Generating implicit copy(A[:][1:4094])
     93, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         93, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     98, Generating implicit copy(A[1:4094][:])
     99, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         99, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
pgcc -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc60 poisson2d.o -o run
```

Transfer Anew from DtoH




```
pgcc -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc60 -c  poisson2d.c
main:
     71, Generating implicit copyin(A[:][:])
         Generating implicit copyout(Anew[1:4094][1:4094])
         Generating implicit copyin(rhs[1:4094][1:4094])
     72, Loop is parallelizable
     74, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         72, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         74, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
         78, Generating implicit reduction(max:error)
     82, Generating implicit copyin(Anew[1:4094][1:4094])
         Generating implicit copyout(A[1:4094][1:4094])
     83, Loop is parallelizable
     85, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         83, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         85, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
     92, Generating implicit copy(A[:][1:4094])
     93, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         93, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     98, Generating implicit copy(A[1:4094][:])
     99, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         99, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
pgcc -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc60 poisson2d.o -o run
```

Transfer Anew from HtoD

…and so on.

Jacobi Iteration – OpenACC (naïve + Unified Memory)

Just add unified memory flag to the compile line

```
-ta=tesla:cc60,managed
```

See how UM handles that data transfers to inform explicit data transfers

Example: Jacobi Iteration

Run naïve GPU version with Unified Memory

- `cd 05_Jacobi_single_device_naive_unifiedMemory`
- `make`
- `./launch.sh`

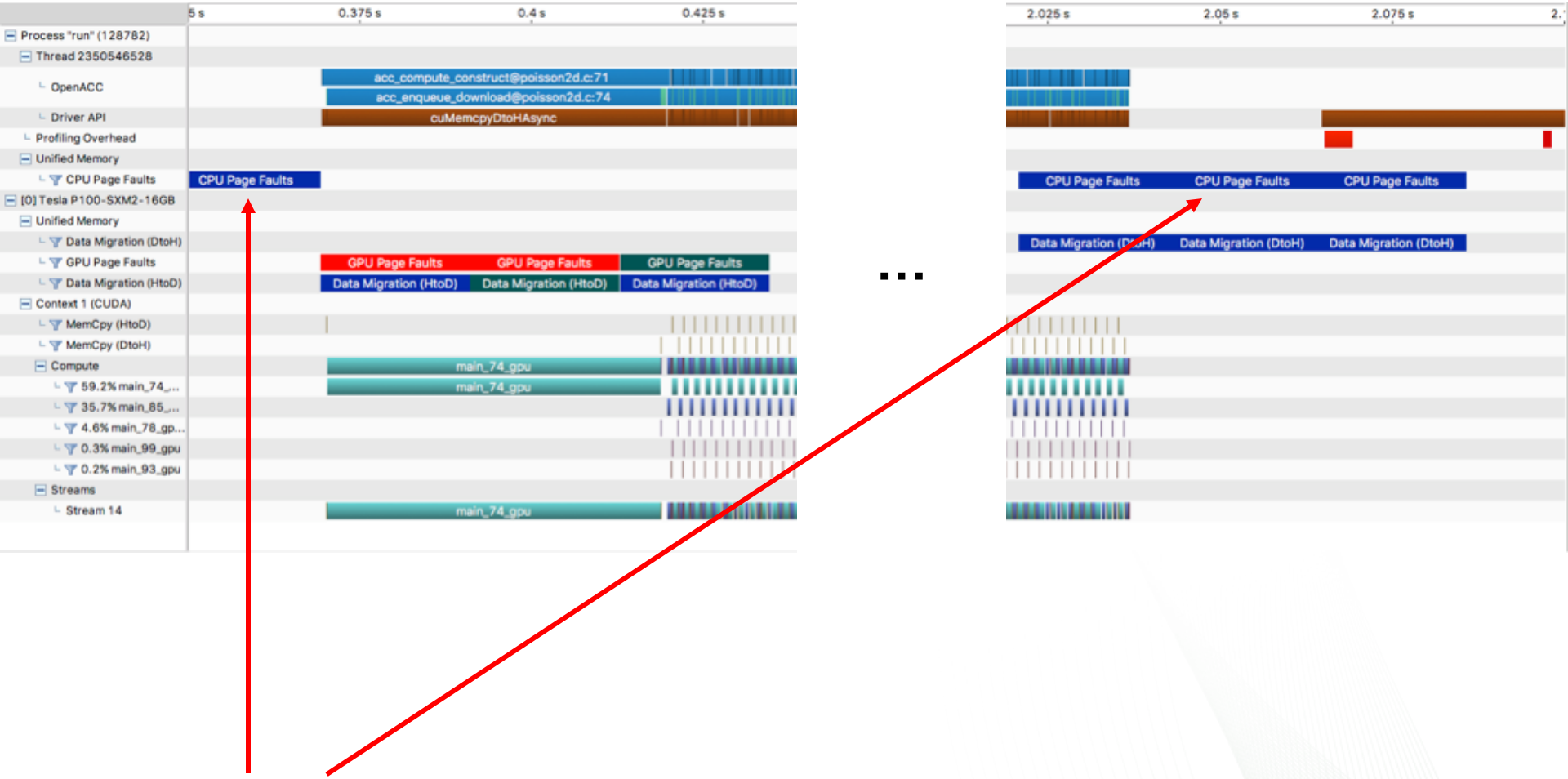
Now about 30x faster than CPU only.

```
==155941== NVPROF is profiling process 155941, command: ./run
==155941== Profiling application: ./run
Jacobi relaxation Calculation: 4096 x 4096 mesh
    0, 0.250000
  100, 0.249940
  200, 0.249880
  300, 0.249821
  400, 0.249761
  500, 0.249702
  600, 0.249642
  700, 0.249583
  800, 0.249524
  900, 0.249464
4096x4096: 1 GPU:   1.6682 s
==155941== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   59.09%  875.99ms      1000  875.99us  823.02us  47.803ms  main_74_gpu
                   35.77%  530.29ms      1000  530.29us  526.09us  537.54us  main_85_gpu
                    4.53%  67.124ms      1000  67.123us  63.585us  70.976us  main_78_gpu__red
                    0.31%  4.5944ms      1000  4.5940us  4.0960us  5.5040us  main_99_gpu
                    0.18%  2.6577ms      1000  2.6570us  2.3040us  3.2640us  main_93_gpu
                    0.07%  989.23us      1000     989ns     960ns  1.5360us  [CUDA memcpy HtoD]
                    0.06%  899.40us      1000     899ns     864ns     929ns  [CUDA memcpy DtoH]

Device "Tesla P100-SXM2-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    1229  213.72KB  64.000KB  960.00KB  256.5000MB  9.432416ms  Host To Device
     644  203.93KB  64.000KB  960.00KB  128.2500MB  4.599023ms  Device To Host
     457         -         -         -           -  48.09904ms  Gpu page fault groups
Total CPU Page faults: 1160
```

Due to having fewer data transfers

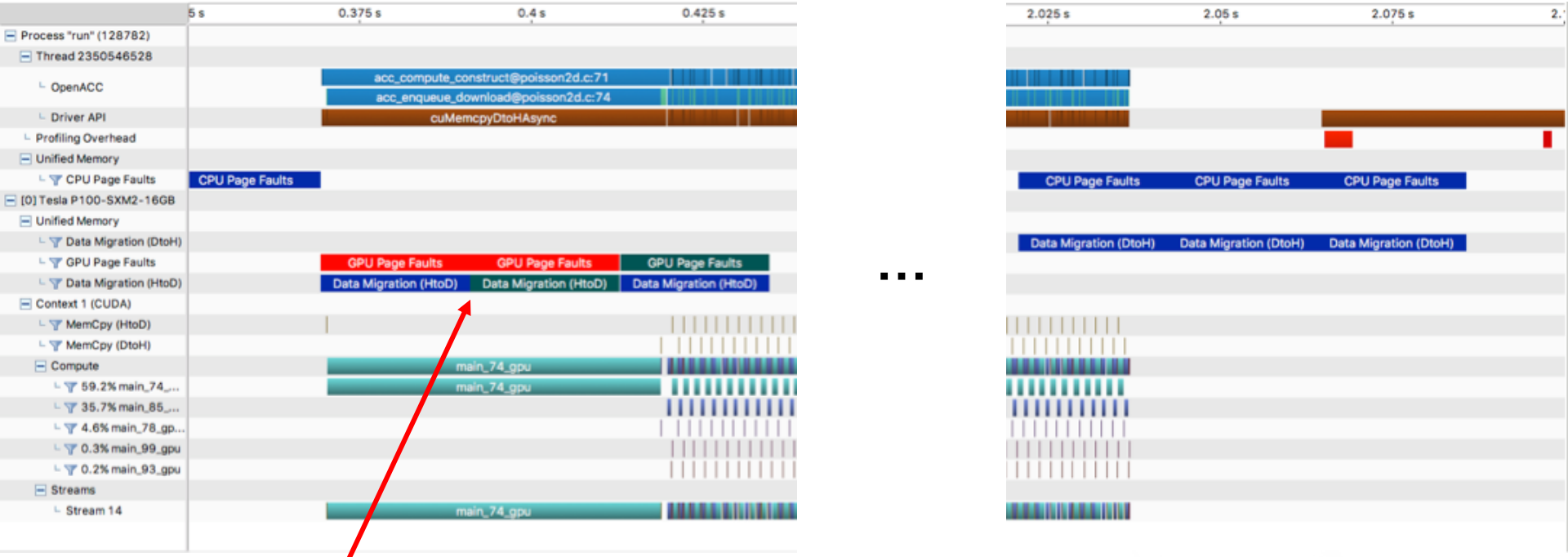
Unified Memory

Example: Jacobi Iteration



CPU Page Fault Groups (bins) and associated DtoH Data Migrations

Example: Jacobi Iteration



GPU Page Fault Groups (bins) and associated HtoD Data Migrations

Example: Jacobi Iteration

These results can help show us when and where the data is actually needed, so we can add in our own manual data movements.

…although this problem is simple enough that it is obvious from looking at the code.

A: copy in before while loop and out after
rhs: copy in before while loop, but not back out
Anew: only needed on device

Jacobi Iteration – OpenACC (data)

Specify when/where the data is actually needed:

A:	copy in before while loop and out after
rhs:	copy in before while loop, but not back out
Anew:	only needed on device

```
#pragma acc data copy(A) copyin(rhs) create(Anew)
while (error > tol && iter < iter_max){
    ...
}
```

Example: Jacobi Iteration

Run single GPU version

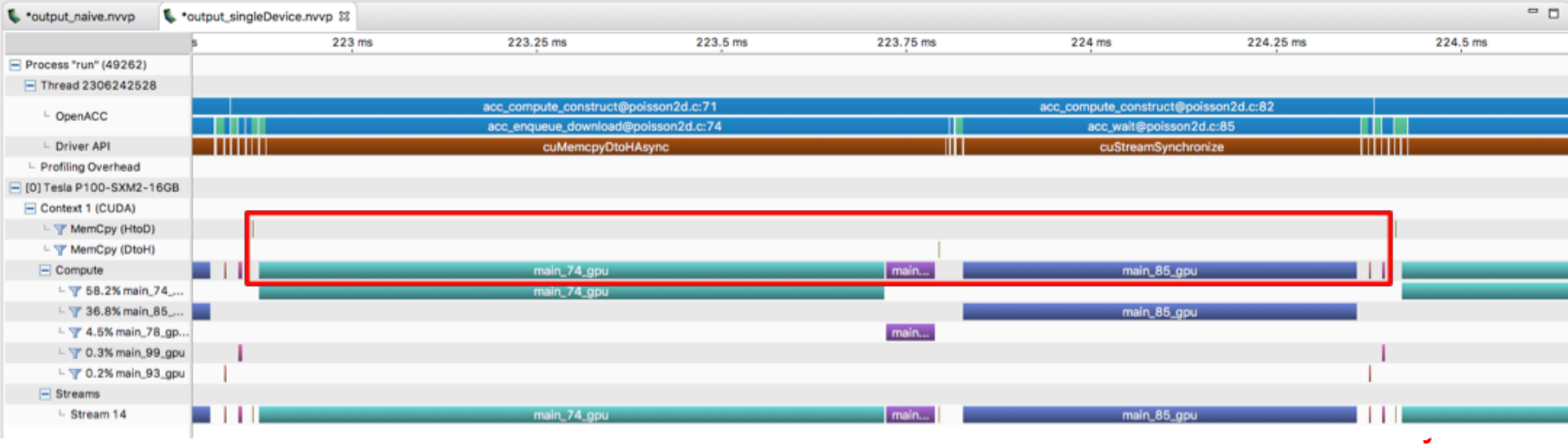
(add data directive along with kernels constructs)

- `cd 06_Jacobi_single_device`
- `make`
- `./launch.sh`

About 30X faster than CPU only.

Time spent on (and calls to) data transfers are now greatly reduced!

```
==85396== NVPROF is profiling process 85396, command: ./run
==85396== Profiling application: ./run
Jacobi relaxation Calculation: 4096 x 4096 mesh
    0, 0.250000
  100, 0.249940
  200, 0.249880
  300, 0.249821
  400, 0.249761
  500, 0.249702
  600, 0.249642
  700, 0.249583
  800, 0.249524
  900, 0.249464
4096x4096: 1 GPU:   1.8526 s
==85396== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   57.65%  841.39ms      1000  841.39us  835.88us  849.07us  main_74_gpu
                   36.47%  532.25ms      1000  532.25us  528.33us  536.52us  main_85_gpu
                    4.45%  64.898ms      1000  64.897us  62.560us  68.033us  main_78_gpu__red
                    0.66%  9.6297ms      1016  9.4780us     960ns  548.65us  [CUDA memcpy HtoD]
                    0.33%  4.8889ms      1009  4.8450us     896ns  499.53us  [CUDA memcpy DtoH]
                    0.26%  3.7868ms      1000  3.7860us  3.1040us  4.6080us  main_99_gpu
                    0.18%  2.5869ms      1000  2.5860us  2.2720us  3.0080us  main_93_gpu
```




```
pgcc -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc60 -c  poisson2d.c
main:
     66, Generating create(Anew[:][:])
         Generating copyin(rhs[:][:])
         Generating copy(A[:][:])
     72, Loop is parallelizable
     74, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         72, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         74, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
         78, Generating implicit reduction(max:error)
     83, Loop is parallelizable
     85, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         83, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         85, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
     93, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         93, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     99, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         99, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
pgcc -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc60 poisson2d.o -o run
```

Only transfer, create data before and after main while loop.

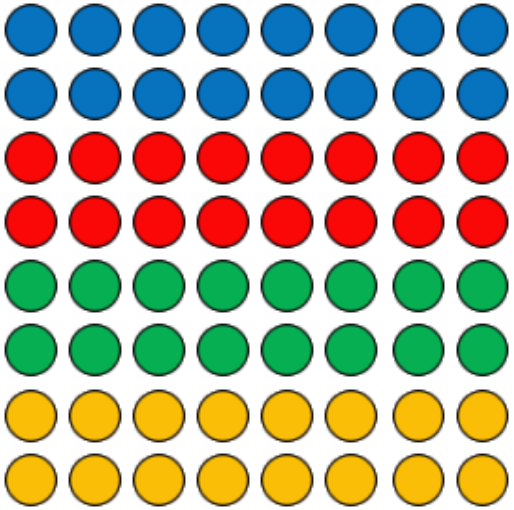
Instead of between each kernel during every iteration of the loop.

Jacobi Iteration

OpenACC+OpenMP (4 GPUs)

Example: Jacobi Iteration

- Each OpenMP thread calculates its own loop bounds for its portion of the domain, and uses its own GPU



OpenMP Thread 0 ⇒ GPU 0

OpenMP Thread 1 ⇒ GPU 1

OpenMP Thread 2 ⇒ GPU 2

OpenMP Thread 3 ⇒ GPU 3

Example: Jacobi Iteration

Run multi-GPU version with OpenMP

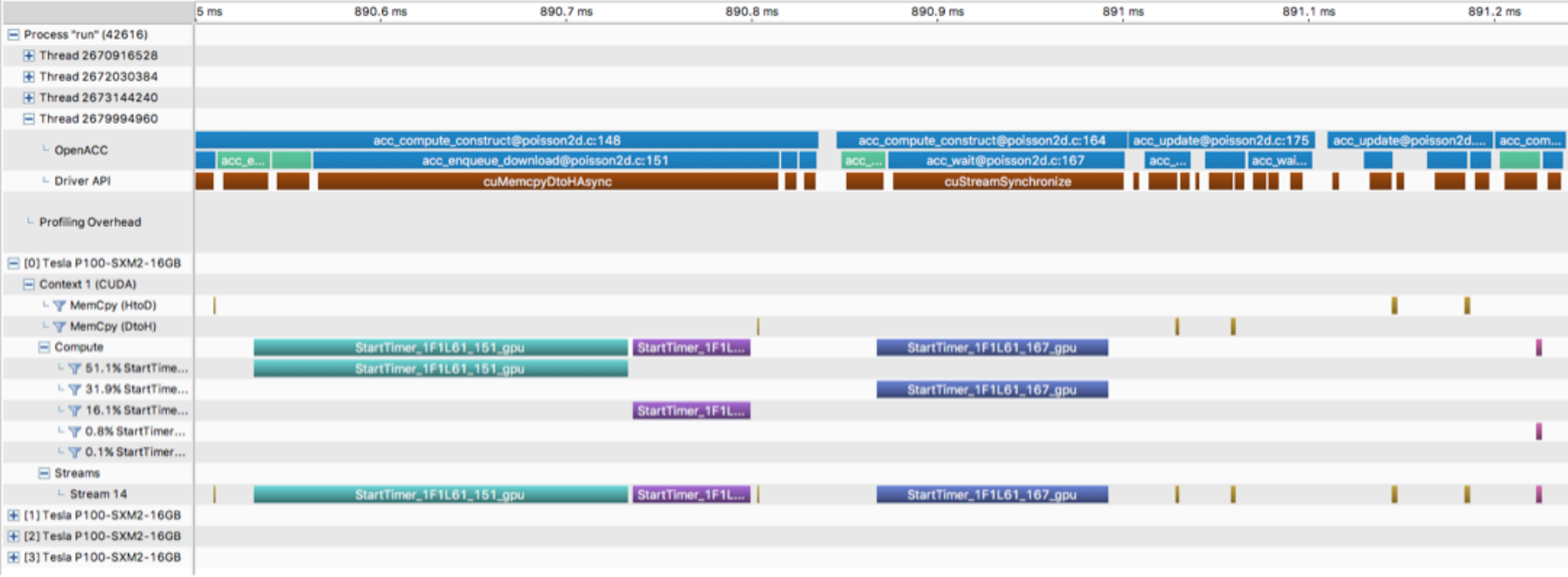
(Each of the 4 OpenMP threads targets a separate GPU)

- `cd 07_Jacobi_OpenMP_multi_device`
- `make`
- `./launch.sh`

About 3X faster than single GPU.

Somehow StartTimer_ got appended…

```
Jacobi relaxation Calculation: 4096 x 4096 mesh
Parallel execution.
    0, 0.250000
  100, 0.249940
  200, 0.249880
  300, 0.249821
  400, 0.249761
  500, 0.249702
  600, 0.249642
  700, 0.249583
  800, 0.249524
  900, 0.249464
Num GPUs: 4.
4096x4096: 4 GPUs:   0.8192 s

            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
GPU activities:   47.47%  801.25ms      4000  200.31us  197.67us  204.71us  StartTimer_1F1L61_151_gpu
                  29.59%  499.45ms      4000  124.86us  123.20us  127.33us  StartTimer_1F1L61_167_gpu
                  15.25%  257.42ms      4000  64.354us  61.440us  69.633us  StartTimer_1F1L61_155_gpu__red
                   4.62%  77.955ms     12084  6.4510us     864ns  2.0066ms  [CUDA memcpy DtoH]
                   2.26%  38.206ms     12020  3.1780us     960ns  578.79us  [CUDA memcpy HtoD]
                   0.69%  11.634ms      4000  2.9080us  2.4960us  3.6480us  StartTimer_1F1L61_193_gpu
                   0.12%  2.0525ms         4  513.12us  505.51us  534.05us  StartTimer_1F1L61_109_gpu
```



- This is basically the same profile as the single-device version, but each OpenMP thread has its own section.
- Here, I have collapsed all activity other than OpenMP thread 0
- By hovering over a GPU activity segment, you can find associated host thread activity.

Jacobi Iteration

CUDA+MPI (4 GPUs)

Example: Jacobi Iteration

Run multi-GPU version with MPI

(Each of the 4 MPI ranks performs same Jacobi solve on different GPU)

- `cd 09_Jacobi_MPI_cuda`
- `make`
- `./launch.sh`

```
jsrun -n1 -a4 -c20 ./set_ulimit.sh nvprof -s -o mpi_multiGPU.%h.%q{PMIX_RANK}.nvvp ./run
```

Increases stack size limit since current value is too small

Expands to node-local MPI rank ID (so each MPI ranks writes its own .nvvp file)

Example: Jacobi Iteration

Run multi-GPU version with MPI

(Each of the 4 MPI ranks performs same Jacobi solve on different GPU)

- `cd Jacobi_MPI_cuda`
- `make`
- `./launch.sh`

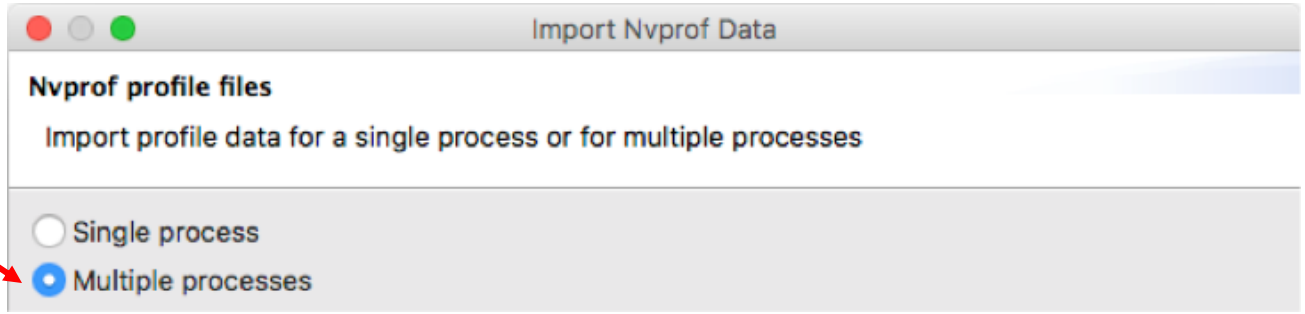
Not optimized implementation

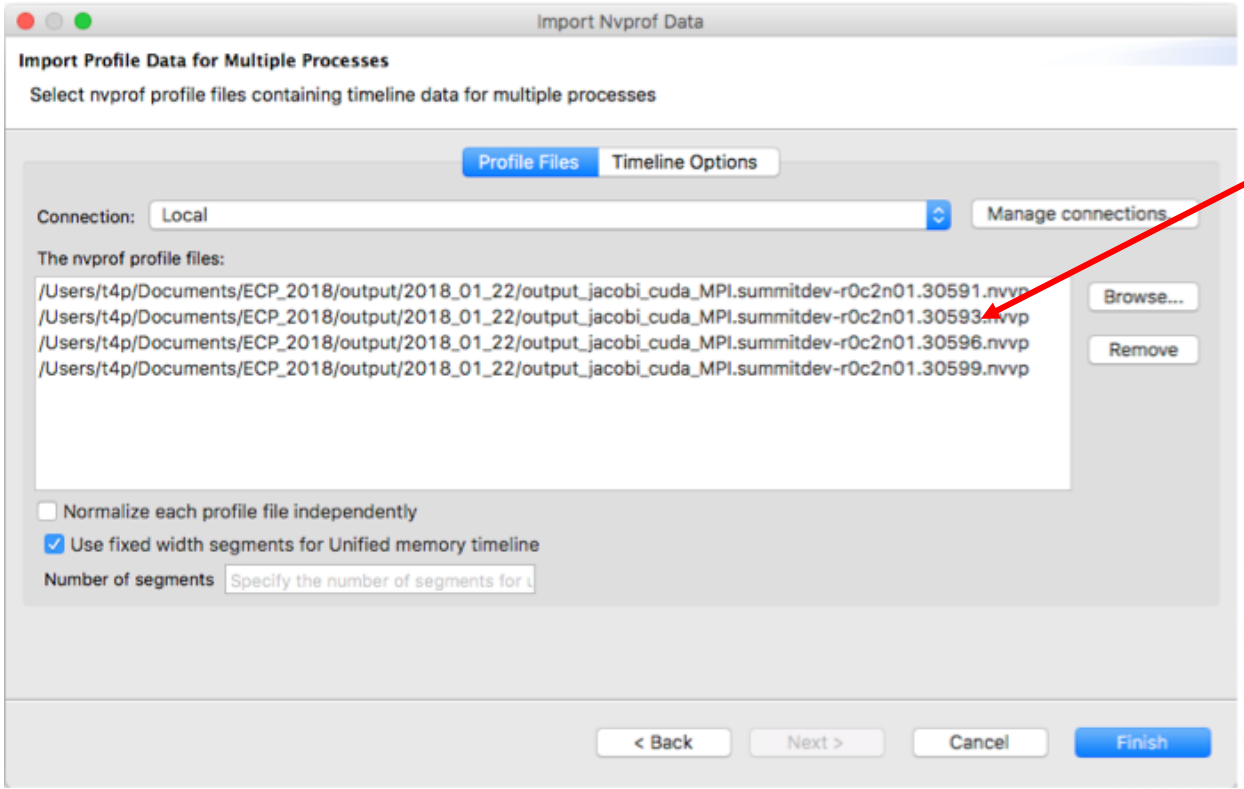
```
4096x4096: GPU 0 or 4:    2.3028 s
4096x4096: GPU 1 or 4:    2.3104 s
4096x4096: GPU 2 or 4:    2.3118 s
4096x4096: GPU 3 or 4:    2.3163 s
```

Notice that individual kernels are named more appropriately

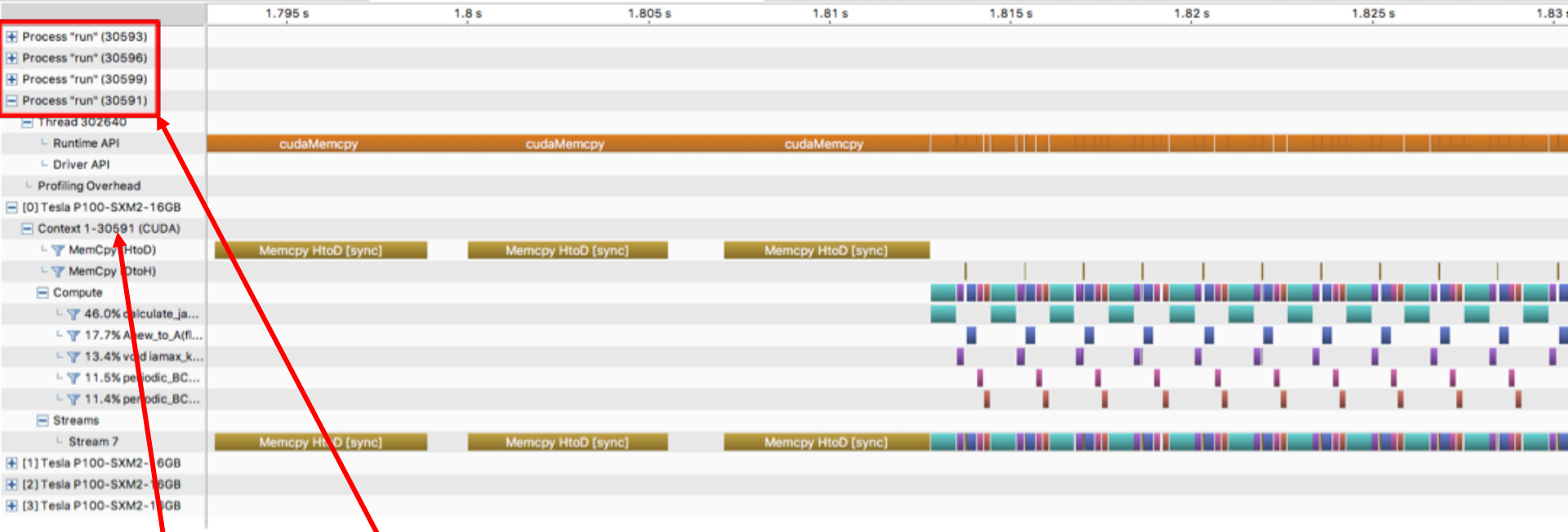
```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   47.23%  1.02219s      1000  1.0222ms  1.0201ms  1.0250ms  calculate_jacobi(double[4096]*, double[4096]*, double[4096]*, int, int, double*)
                   22.64%  489.93ms      1000  489.93us  488.16us  492.07us  Anew_to_A(double[4096]*, double[4096]*, int, int)
                   12.17%  263.29ms      2000  131.64us  3.2960us  269.31us  void iamax_kernel<double, double, int=256, int=0>(cublasIamaxParams<double, double>)
                    7.84%  169.62ms      1000  169.62us  169.09us  172.55us  periodic_BC_rows(double[4096]*, int, int)
                    7.81%  169.06ms      1000  169.06us  168.51us  171.81us  periodic_BC_columns(double[4096]*, int, int)
                    1.65%  35.754ms         4  8.9384ms  1.7280us  15.022ms  [CUDA memcpy HtoD]
                    0.66%  14.329ms      2001  7.1610us     864ns  11.963ms  [CUDA memcpy DtoH]
```

Choose Multiple process





Select the .nvvp files written for each (process) MPI rank



4 Processes (i.e. MPI ranks) – each with its own GPU

Notice how the CUDA context is associated with a process

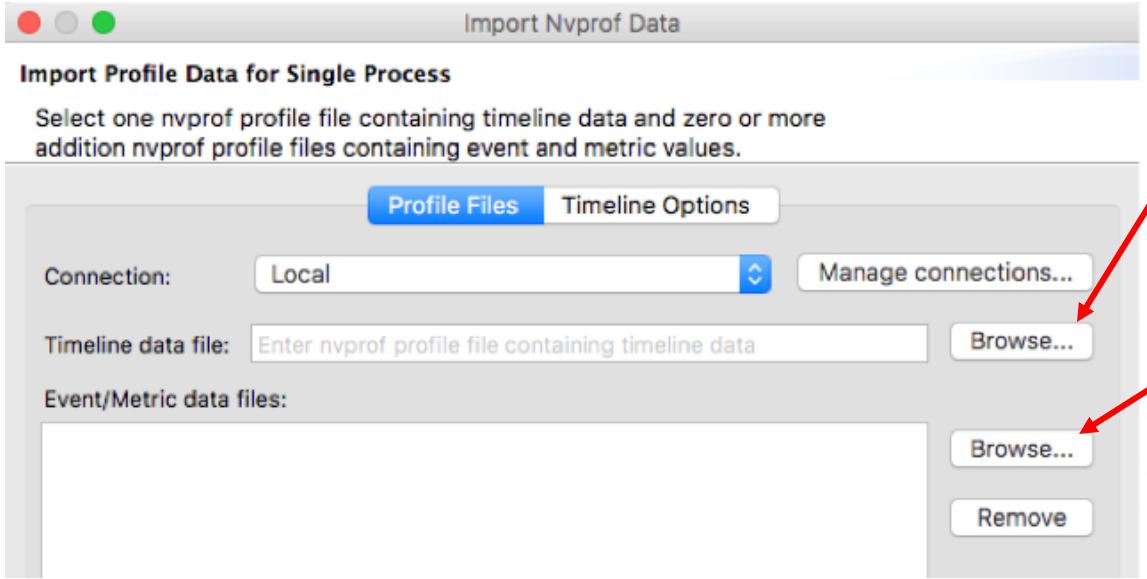
Jacobi Iteration

Guided Analysis with Visual Profiler

Example: Jacobi Iteration

Guided Analysis with NVIDIA Visual Profiler

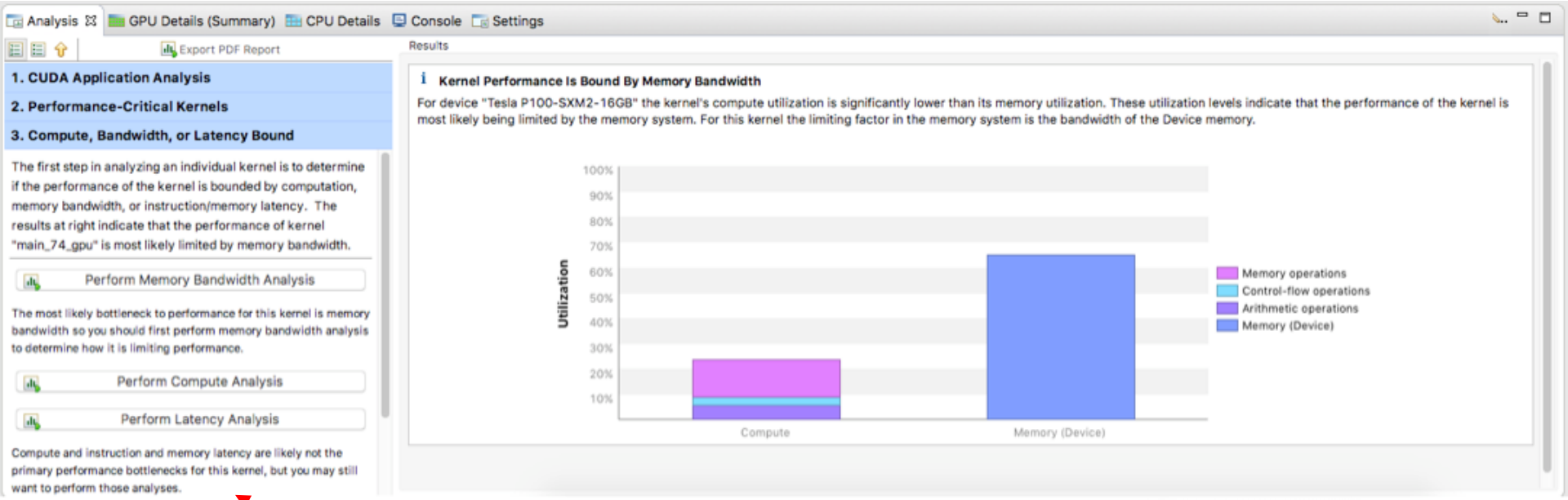
- Can run executable directly from the Visual Profiler
- Can collect metrics on remote machine, then import into Visual Profiler on local machine.
  - `jsrun -n1 nvprof -s -o single_GPU.%h.%p.nvvp ./run`
  - `jsrun -n1 nvprof --analysis-metrics -o single_GPU.%h.%p.prof ./run`



Then import timeline file and metrics file.

Example: Jacobi Iteration

Program Analysis with NVIDIA Visual Profiler



Follow through the steps of the Guided Analysis to look for optimization opportunities.

More Information

- NVIDIA's documentation on the profiler
  - http://docs.nvidia.com/cuda/profiler-users-guide/index.html
- NVIDIA Tools Extension API
  - http://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvtx
  - Allows adding of custom annotations for profiler.

Acknowledgements

- Parallel Programming with OpenACC (by Rob Farber)
  - https://www.amazon.com/Parallel-Programming-OpenACC-Rob-Farber/dp/0124103979
  - The source code in this tutorial is freely available from NVIDIA, but it was obtained from this book, which I highly recommend for learning OpenACC.

- Jeff Larkin, NVIDIA (answering questions, general advice)