

GPU Accelerated Computing on Summit

Tom Papatheodore
Oak Ridge Leadership Computing Facility

Graham Lopez
Oak Ridge Leadership Computing Facility

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



Outline of Hands-On Session

- Logging Into Ascent and Setting Up Environment
- Heterogeneous (CPU+GPU) Programming Basics
- A Simple Example: Vector Addition
- Jacobi Iteration
 - Serial
 - Single GPU
 - Single GPU (explicit data movement)
 - Multiple GPU (OpenMP + OpenACC)

Logging Into Ascent and Setting Up Environment



Log Into Ascent and Change Directory

- ① From the command line:

```
$ ssh USERNAME@login1.ascent.olcf.ornl.gov
```

(This will drop you into the directory /ccsopen/home/USERNAME)

- ② Change to the following directory:

```
$ cd /gpfs/wolf/gen117/scratch/USERNAME
```

Clone Repository and Set up Programming Environment

- Once in /gpfs/wolf/gen117/scratch/USERNAME, clone git repository:

```
$ git clone https://github.com/tom-papatheodore/2019_SIAM_CSE.git
```

- cd into directory:

```
$ cd 2019_SIAM_CSE
```

- Run script to set up environment for the tutorial:

```
$ source environment.sh
```

At this point, your prompt should look like this:

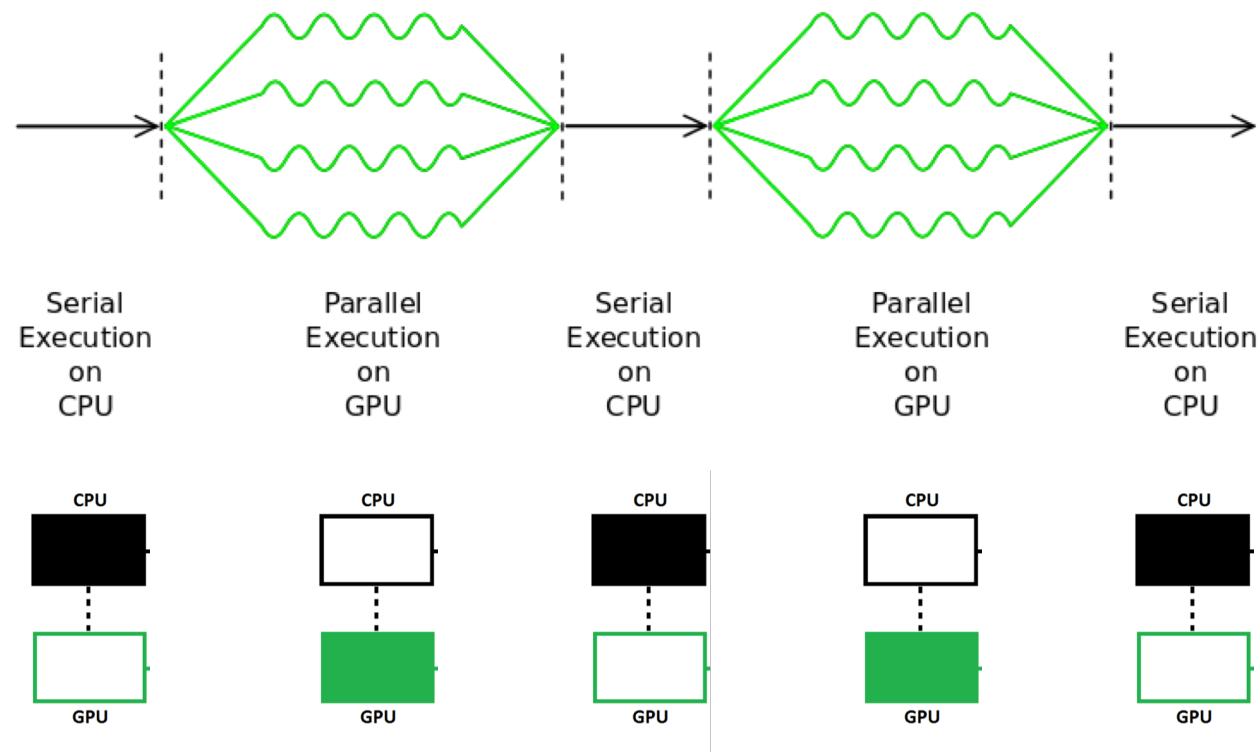
```
[USERNAME@login1: /gpfs/wolf/gen117/scratch/USERNAME/2019_SIAM_CSE]$
```

Heterogeneous Programming Model



Heterogeneous Programming Model

- Program separated into serial regions (run on CPU) and parallel regions (run on GPU)



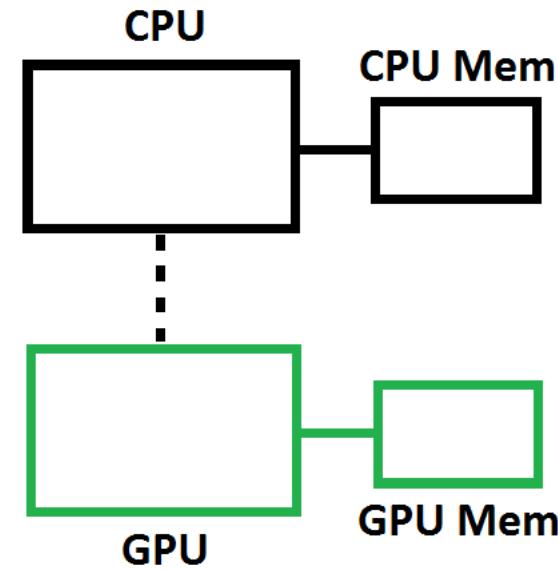
Heterogeneous Programming Model

- Parallel regions consist of many calculations that can be executed independently
 - Data Parallelism (e.g. vector addition)



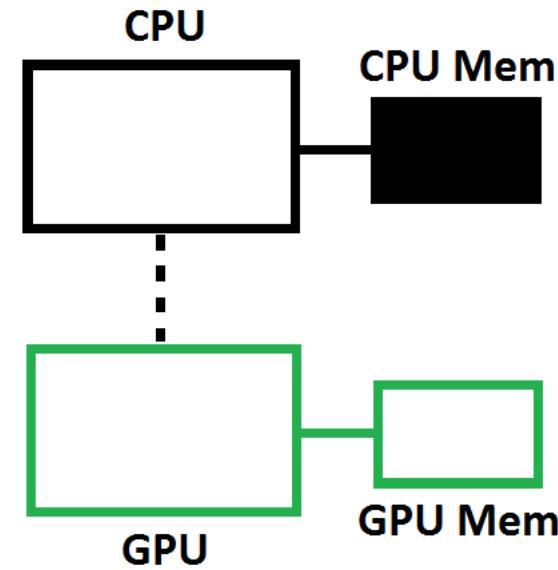
A Basic GPU Program Outline

```
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 Host Memory  
    // Free Device Memory  
}
```



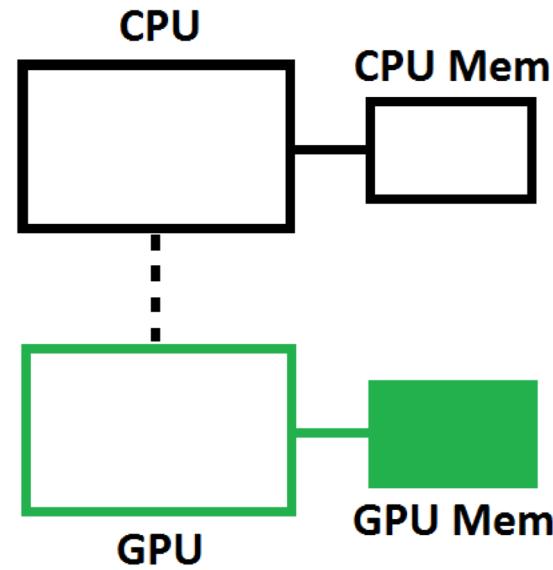
A Basic GPU Program Outline

```
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 Host Memory  
    // Free Device Memory  
}
```



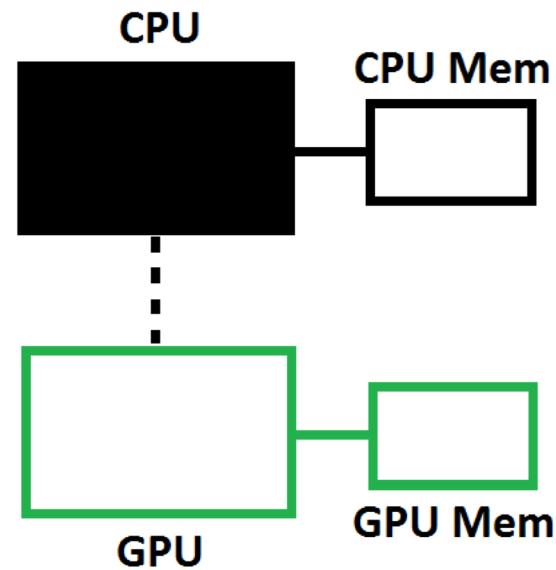
A Basic GPU Program Outline

```
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 Host Memory  
    // Free Device Memory  
}
```



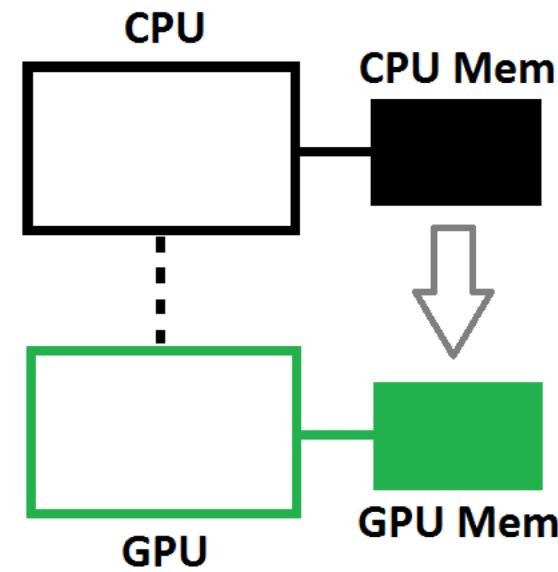
A Basic GPU Program Outline

```
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 Host Memory  
  
    // Free Device Memory  
  
}
```



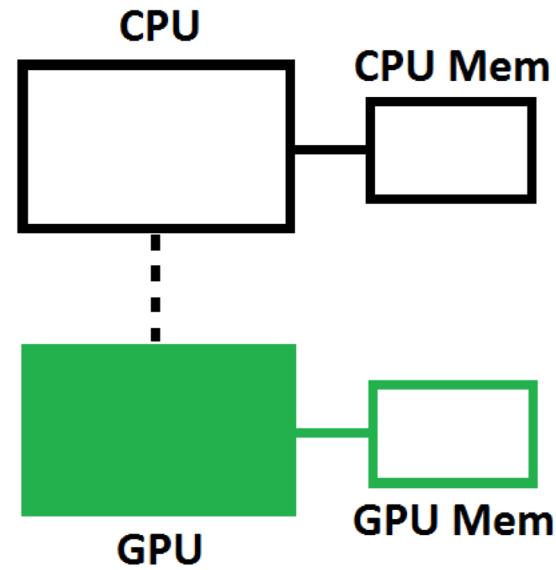
A Basic GPU Program Outline

```
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 Host Memory  
    // Free Device Memory  
}
```



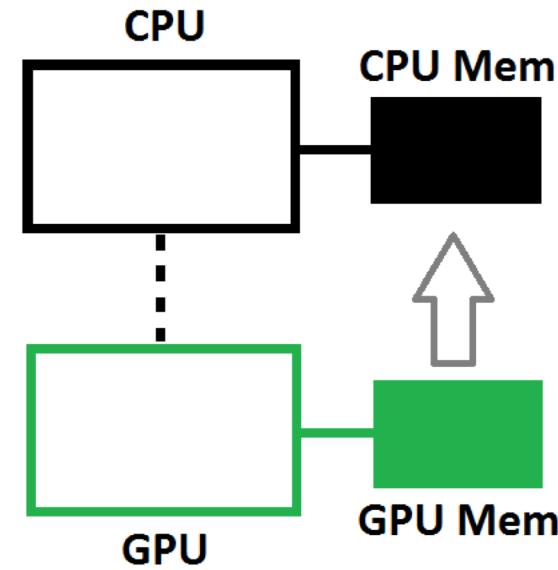
A Basic GPU Program Outline

```
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 Host Memory  
    // Free Device Memory  
}
```



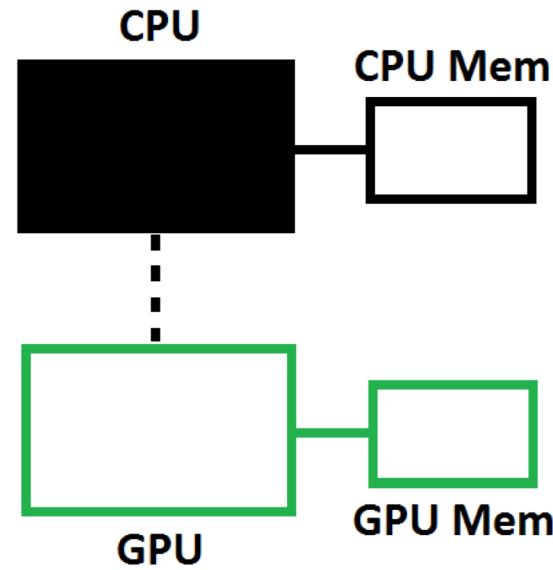
A Basic GPU Program Outline

```
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 Host Memory  
    // Free Device Memory  
}
```



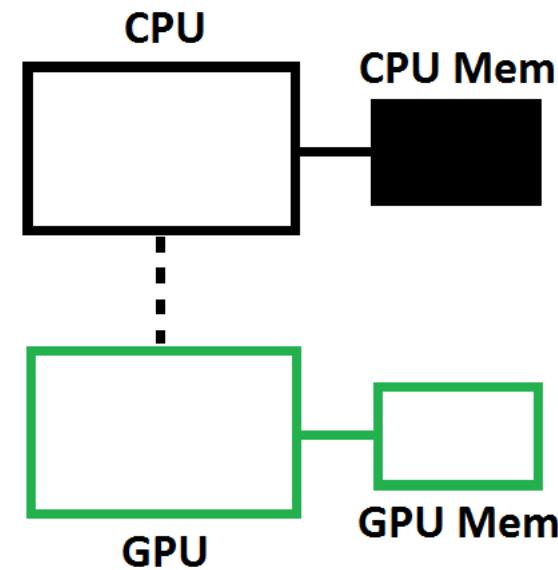
A Basic GPU Program Outline

```
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 Host Memory  
    // Free Device Memory  
}
```



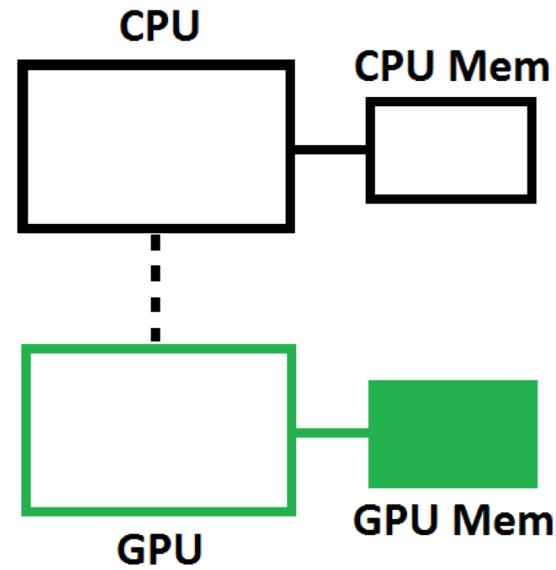
A Basic GPU Program Outline

```
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 Host Memory  
    // Free Device Memory  
  
}
```



A Basic GPU 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 Host Memory  
  
    // Free Device Memory  
}
```



A Simple Example: Vector Addition



CUDA Vector Addition

```
#include <stdio.h>
#define N 1048576

__global__ void add_vectors(int *a, int *b, int *c){
    int id = blockDim.x * blockIdx.x + threadIdx.x;
    if(id < N) c[id] = a[id] + b[id];
}

int main(){
    size_t bytes = N*sizeof(int);

    int *A = (int*)malloc(bytes);
    int *B = (int*)malloc(bytes);
    int *C = (int*)malloc(bytes);

    int *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, bytes);
    cudaMalloc(&d_B, bytes);
    cudaMalloc(&d_C, bytes);

    for(int i=0; i<N; i++){
        A[i] = 1;
        B[i] = 2;
    }

    cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);

    int thr_per_blk = 256;
    int blk_in_grid = ceil( float(N) / thr_per_blk );
    add_vectors<<< blk_in_grid, thr_per_blk >>>(d_A, d_B, d_C);

    cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);

    free(A);
    free(B);
    free(C);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    return 0;
}
```

CUDA Vector Addition

```
#include <stdio.h>
#define N 1048576

__global__ void add_vectors(int *a, int *b, int *c){
    int id = blockDim.x * blockIdx.x + threadIdx.x;
    if(id < N) c[id] = a[id] + b[id];
}

int main(){
    size_t bytes = N*sizeof(int);

    int *A = (int*)malloc(bytes);
    int *B = (int*)malloc(bytes);
    int *C = (int*)malloc(bytes);

    int *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, bytes);
    cudaMalloc(&d_B, bytes);
    cudaMalloc(&d_C, bytes);

    for(int i=0; i<N; i++){
        A[i] = 1;
        B[i] = 2;
    }

    cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);

    int thr_per_blk = 256;
    int blk_in_grid = ceil( float(N) / thr_per_blk );
    add_vectors<<< blk_in_grid, thr_per_blk >>>(d_A, d_B, d_C);

    cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);

    free(A);
    free(B);
    free(C);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#define N 1048576

int main(){
    size_t bytes = N*sizeof(int);

    int *A = (int*)malloc(bytes);
    int *B = (int*)malloc(bytes);
    int *C = (int*)malloc(bytes);

    for(int i=0; i<N; i++){
        A[i] = 1;
        B[i] = 2;
    }

#pragma acc parallel loop
    for(int i=0; i<N; i++){
        C[i] = A[i] + B[i];
    }

    free(A);
    free(B);
    free(C);

    return 0;
}
```

OpenACC Vector Addition

Jacobi Iteration



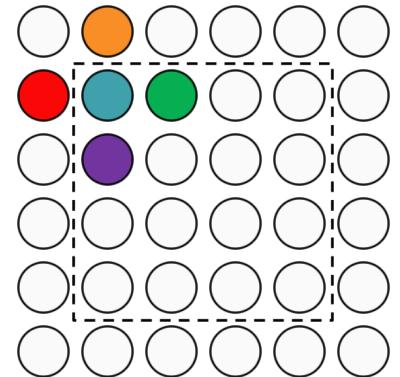
Jacobi Iteration – Problem Description

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

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

Execute a Jacobi Step on the Inner Points

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



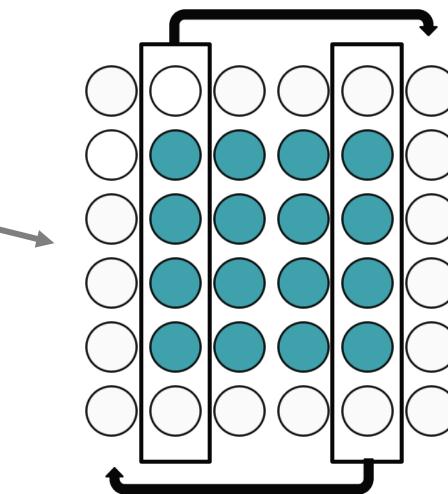
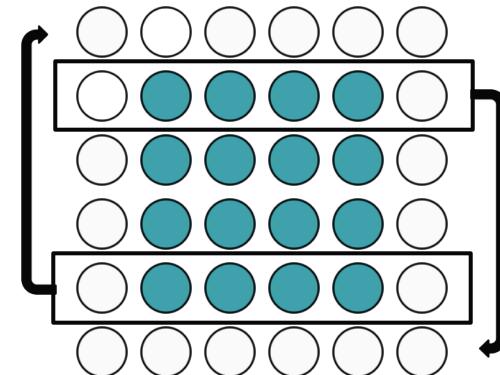
```
for (int iy = 1; iy < NY-1; iy++)
{
    for( int ix = 1; ix < NX-1; ix++ )
    {
        Anew[iy][ix] = -0.25 * (rhs[iy][ix] - ( A[iy][ix+1] + A[iy][ix-1]
                                                + A[iy-1][ix] + A[iy+1][ix] ));
        error = fmax( error, fabs(Anew[iy][ix]-A[iy][ix]));
 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

Copy 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];
    }
}
```

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];
}
```



Serial Version

2019_SIAM_CSE/jacobi/1_serial

Serial Runtime

Compile the code

```
$ make
pgcc -Minfo -fast -c poisson2d.c
main:
  54, Generated vector SIMD code for the loop
    FMA (fused multiply-add) instruction(s) generated
  65, Memory zero idiom, loop replaced by call to __c_mzero8
  84, FMA (fused multiply-add) instruction(s) generated
  90, Generated vector SIMD code for the loop containing reductions
100, Memory copy idiom, loop replaced by call to __c_mcop8
107, Loop not fused: dependence chain to sibling loop
  Generated vector SIMD code for the loop
    Residual loop unrolled 2 times (completely unrolled)
112, Loop not fused: function call before adjacent loop
  Loop unrolled 8 times
pgcc -Minfo -fast poisson2d.o -o run
```

Run the code (on single CPU core)

```
$ bsub submit.lsf
Job <11536> is submitted to default queue <batch>.

$ jobstat
----- Running Jobs: 1 (1 of 16 nodes, 6.25%) -----
JobId Username Project Nodes Remain StartTime JobName
11536 t4p GEN117 1 8:48 02/24 10:03:14 serial
----- Eligible Jobs: 0 -----
----- Blocked Jobs: 0 -----
```

```
$ less output.11536
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
Elapsed Time (s): [REDACTED]
```

(Enter q to quit/exit less)

Single GPU Version

2019_SIAM_CSE/jacobi/2_single_gpu

Difference From Serial Version

- Added OpenACC pragmas to inform compiler where to offload work to GPU

```
#pragma acc kernels
```

- Added (optional) serial version to compare with timing and results of GPU version

```
// Set to 1 to run serial test, otherwise 0
int serial_test = 0;
```

Runtime of Single GPU Version

Compile the code

```
$ make
pgcc -acc -Minfo=acc -ta=tesla:cc70 -fast -c poisson2d.c
main:
 117, Generating implicit copyin(A[:, :], rhs[1:4094][1:4094])
    Generating implicit copyout(Anew[1:4094][1:4094])
 118, Loop is parallelizable
 120, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
 118, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
 120, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
 124, Generating implicit reduction(max:error)
 128, Generating implicit copyin(Anew[1:4094][1:4094])
    Generating implicit copyout(A[1:4094][1:4094])
 129, Loop is parallelizable
 131, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
 129, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
 131, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
 138, Generating implicit copy(A[:, ][1:4094])
 139, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
 139, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
 144, Generating implicit copy(A[1:4094][:])
 145, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
 145, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
pgcc -acc -Minfo=acc -ta=tesla:cc70 -fast poisson2d.o -o run
```

Run the code (on single GPU)

```
$ bsub submit.lsf
$ less output
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
Elapsed Time (s) - Parallel: [REDACTED]
```

Runtime of Single GPU Version

Compile the code

```
$ make
pgcc -acc -Minfo=acc -ta=tesla:cc70 -fast -c poisson2d.c
main:
 117, Generating implicit copyin(A[:, :], rhs[1:4094][1:4094])
    Generating implicit copyout(Anew[1:4094][1:4094])
 118, Loop is parallelizable
 120, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
 118, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
 120, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
 124, Generating implicit reduction(max:error)
 128, Generating implicit copyin(Anew[1:4094][1:4094])
    Generating implicit copyout(A[1:4094][1:4094])
 129, Loop is parallelizable
 131, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
 129, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
 131, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
 138, Generating implicit copy(A[:, ][1:4094])
 139, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
 139, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
 144, Generating implicit copy(A[1:4094][:])
 145, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
 145, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
pgcc -acc -Minfo=acc -ta=tesla:cc70 -fast poisson2d.o -o run
```

Run the code (on single GPU)

```
$ bsub submit.lsf
$ less output
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
Elapsed Time (s) - Parallel: 127.2326
```

Why are we slower than serial version??

How can we answer such questions?

Using NVIDIA's NVProf Profiler, we see...

```
$ bsub submit.lsf (jsrun --smpiargs="none" -n1 -c1 -g1 -a1 nvprof ./run)

$ less output
==56446== NVPORF is profiling process 56446, command: ./run
==56446== Profiling application: ./run

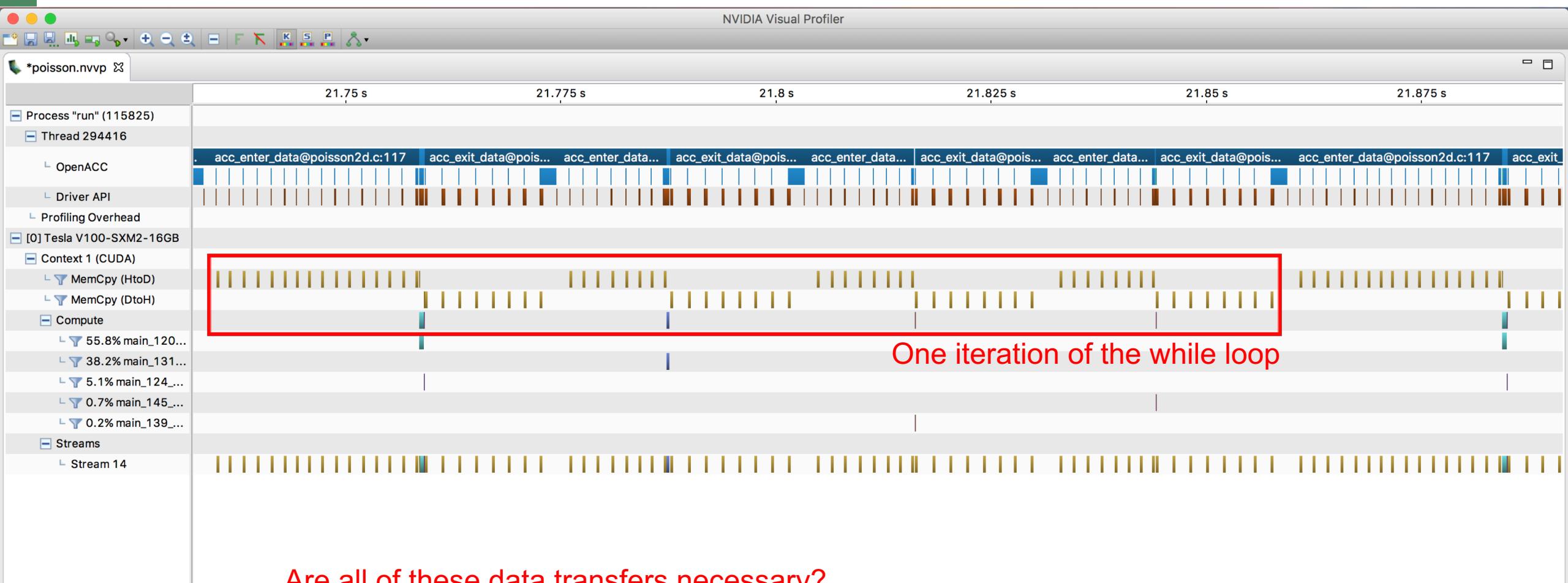
Jacobi relaxation Calculation: 4096 x 4096 mesh
Parallel Execution...
    0, 0.250000
    100, 0.249940
    200, 0.249880
    300, 0.249821
    400, 0.249761
    500, 0.249702
    600, 0.249642
    700, 0.249583
    800, 0.249524
    900, 0.249464
Elapsed Time (s) - Parallel: 130.9012

==56446== Profiling result:
      Type   Time(%)     Time    Calls      Avg       Min       Max     Name
GPU activities:  53.55% 14.4180s  41000  351.66us  1.3110us  382.72us  [CUDA memcpy HtoD]
                  42.84% 11.5335s  33000  349.50us  1.7590us  362.53us  [CUDA memcpy DtoH]
                  2.01% 541.55ms   1000  541.55us  539.61us  546.01us  main_120_gpu
                  1.38% 372.18ms   1000  372.18us  369.47us  376.64us  main_131_gpu
                  0.19% 49.816ms   1000  49.815us  48.448us  51.231us  main_124_gpu_red
                  0.02% 6.1174ms   1000  6.1170us  5.7270us  6.9760us  main_145_gpu
                  0.01% 2.1649ms   1000  2.1640us  1.8880us  2.8480us  main_139_gpu
```

Do we really need all these data transfers?

Let's look at visual output (and compiler output) to see what's going on...

Using NVIDIA's Visual Profiler, we see...



Where are arrays actually needed?

```
// Main iteration loop
while ( error > tol && iter < iter_max )
{
    error = 0.0;

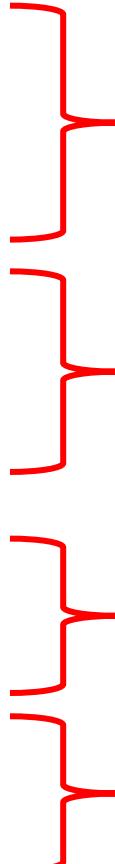
    #pragma acc kernels
    for (int iy = 1; iy < NY-1; iy++)
    {
        for( int ix = 1; ix < NX-1; ix++ )
        {
            Anew[iy][ix] = -0.25 * (rhs[iy][ix] - ( A[iy][ix+1] + A[iy][ix-1]
                                                + A[iy-1][ix] + A[iy+1][ix] ));
            error = fmax( error, fabs(Anew[iy][ix]-A[iy][ix]));
        }
    }

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

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

    if((iter % 100) == 0) printf("%5d, %0.6f\n", iter, error);

    iter++;
}
```



- A_{new} is updated
- A and rhs are not updated
- Reduction performed on $error$

- A is updated
- A_{new} is not updated

- A is updated

- A is updated

But nowhere is this while loop are A_{new} , A , or rhs needed on the CPU!

Single GPU Version with Data Regions

2019_SIAM_CSE/jacobi/3_single_gpu_data

Difference From Initial GPU Version

- Added a data region around while loop

```
#pragma acc data ...
{
    while loop
}
```

- Still have (optional) serial version to compare with timing and results of GPU version

```
// Set to 1 to run serial test, otherwise 0
int serial_test = 0;
```

Runtime of Single GPU Version with Data Directives

Compile the code

```
$ make
pgcc -acc -Minfo=acc -ta=tesla:cc70 -fast -c poisson2d.c
main:
 112, Generating copyin(rhs[:, :])
    Generating create(Anew[:, :])
    Generating copy(A[:, :])
 121, Loop is parallelizable
 123, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
 121, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
 123, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
 127, Generating implicit reduction(max:error)
 132, Loop is parallelizable
 134, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
 132, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
 134, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
 142, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
 142, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
 148, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
 148, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
pgcc -acc -Minfo=acc -ta=tesla:cc70 -fast poisson2d.o -o run
```

Run the code (on single GPU)

```
$ bsub submit.lsf
$ less output
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
Elapsed Time (s) - Parallel:
```

Using NVIDIA's NVProf Profiler, we see...

```
$ bsub submit.lsf (jsrun --smpiargs="none" -n1 -c1 -g1 -a1 nvprof ./run)  
$ less output  
==139388== NVPROF is profiling process 139388, command: ./run  
==139388== Profiling application: ./run
```

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
```

Elapsed Time (s) - Parallel: 1.9883

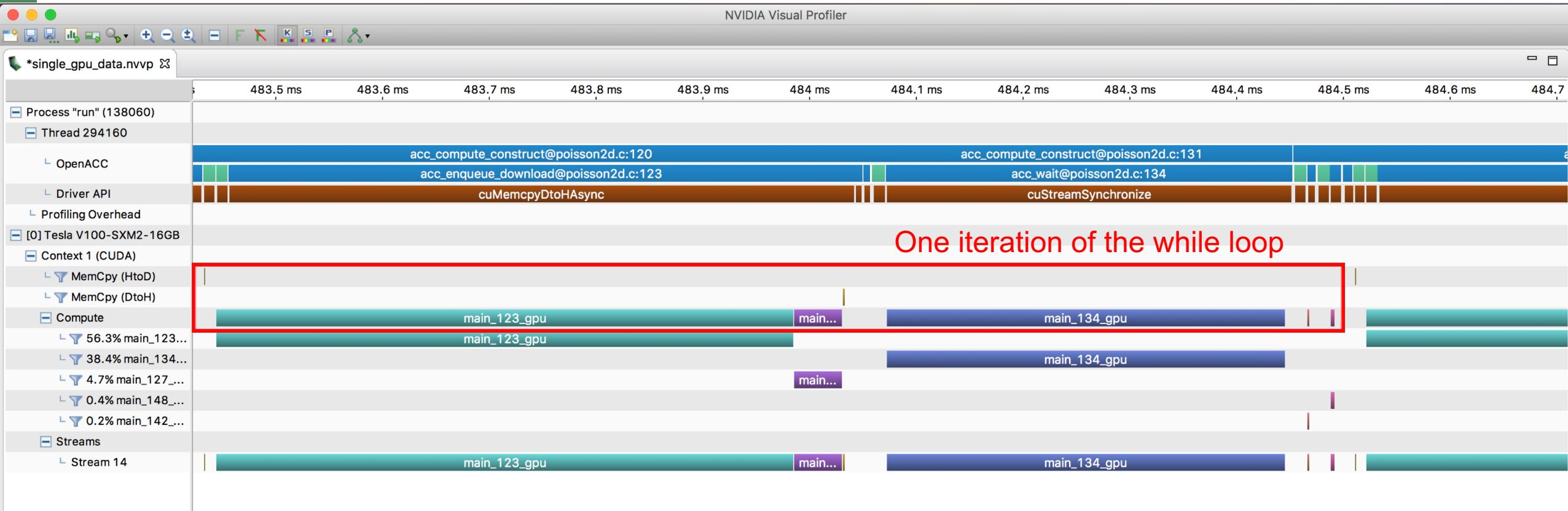
```
==139388== Profiling result:
```

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	55.51%	539.46ms	1000	539.46us	537.53us	542.75us	main_123_gpu
	38.02%	369.46ms	1000	369.46us	366.65us	373.57us	main_134_gpu
	4.76%	46.220ms	1000	46.219us	43.935us	51.040us	main_127_gpu_red
	0.72%	7.0198ms	1016	6.9090us	1.2160us	360.06us	[CUDA memcpy HtoD]
	0.47%	4.5286ms	1009	4.4880us	1.5990us	359.04us	[CUDA memcpy DtoH]
	0.35%	3.4053ms	1000	3.4050us	3.0400us	4.4480us	main_148_gpu
	0.18%	1.7651ms	1000	1.7650us	1.6320us	2.2080us	main_142_gpu

~60X faster with explicit data management

Data transfers are no longer dominating run time.

Using NVIDIA's Visual Profiler, we see...



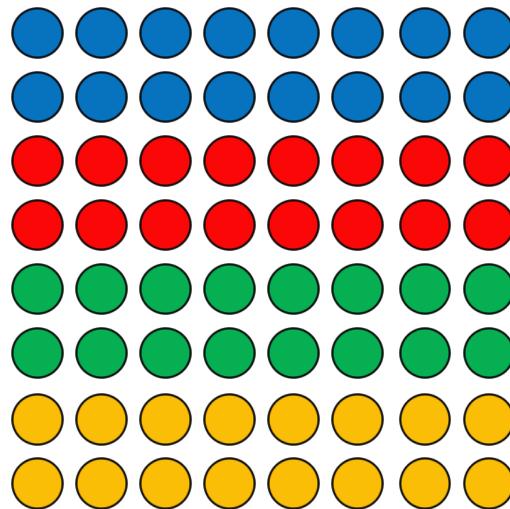
We have eliminated the unnecessary data transfers.

Multiple GPU Version (OpenMP + OpenACC)

2019_SIAM_CSE/jacobi/4_multiple_gpu_openmp

Differences from Single GPU Version

- 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

Differences from Single GPU Version

```
#pragma omp parallel default(shared) firstprivate(num_threads, thread_num) {}
```

```
#ifdef __OPENMP
    num_threads = omp_get_num_threads();
    thread_num = omp_get_thread_num();
#endif /* __OPENMP */
```

```
#ifdef __OPENACC
    int num_devices = acc_get_num_devices(acc_device_nvidia);
    int device_num = thread_num % num_devices;
    acc_set_device_num(device_num, acc_device_nvidia);
#endif /* __OPENACC */
```

Map OpenMP threads
to available GPUs

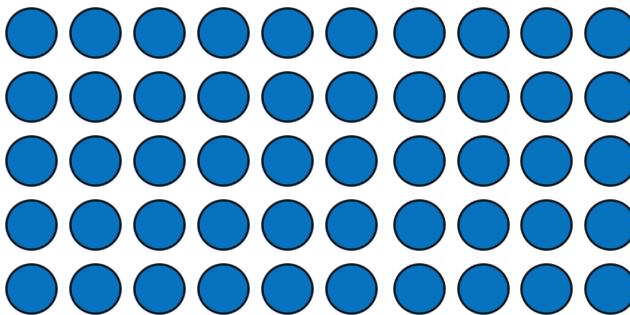
```
#pragma omp master
{
// Set rhs
for (int iy = 1; iy < NY-1; iy++)
{
    for(int ix = 1; ix < NX-1; ix++ )
    {
        const double x = -1.0 + (2.0*ix/(NX-1));
        const double y = -1.0 + (2.0*iy/(NY-1));
        rhs[iy][ix] = exp(-10.0*(x*x + y*y));
    }
}
/* pragma omp master */
```

Only the master thread
needs to set value of rhs

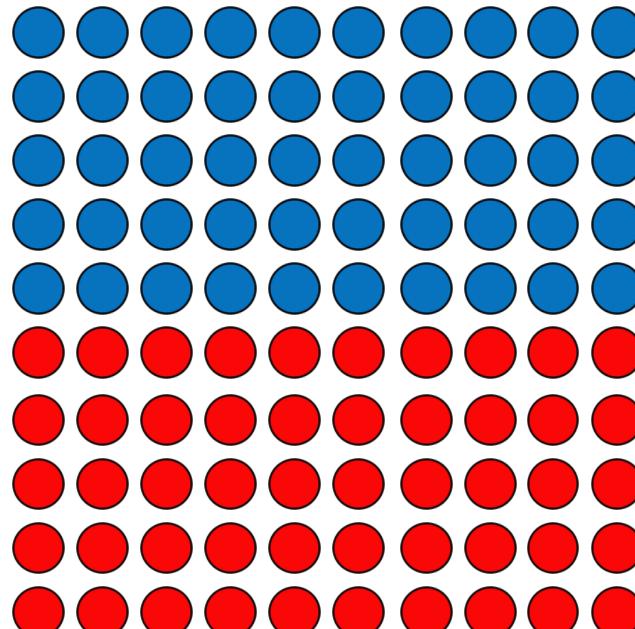
Differences from Single GPU Version

```
#pragma acc data copy(A[(iy_start-1):(iy_end-iy_start)+2][0:NX]) \
copyin(rhs[iy_start:(iy_end-iy_start)][0:NX]) \
create(Anew[iy_start:(iy_end-iy_start)][0:NX])
```

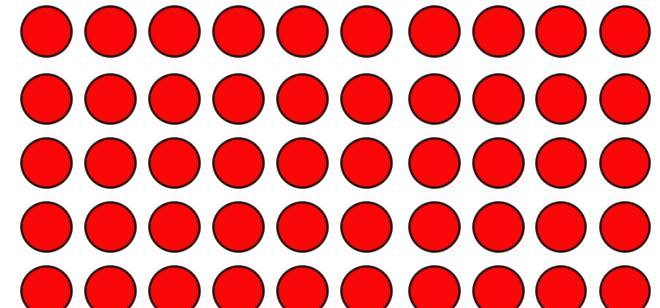
**Thread 0's copy of its rows of A
(on GPU 0)**



CPU copy of A



**Thread 1's copy of its rows of A
(on GPU 1)**

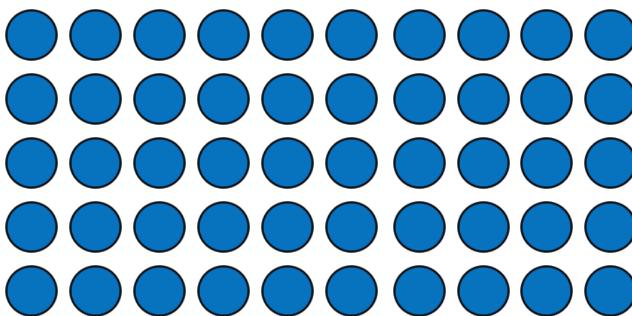


Differences from Single GPU Version

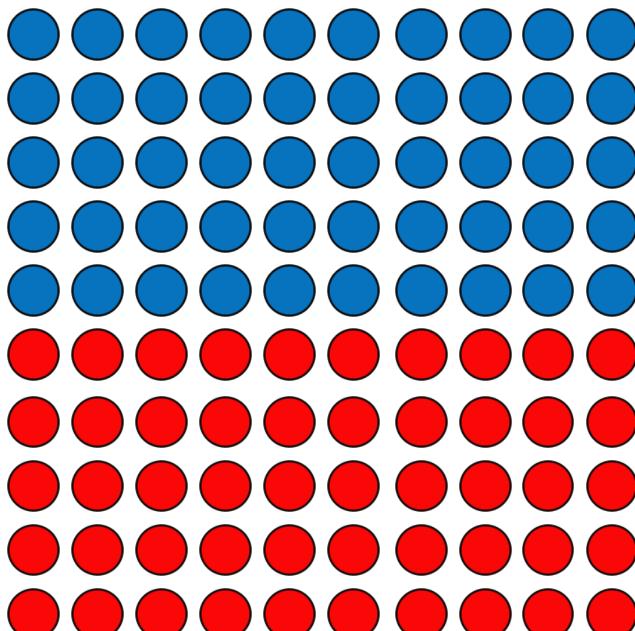
```
#pragma acc kernels
for (int iy = iy_start; iy < iy_end; iy++)
{
    for( int ix = ix_start; ix < ix_end; ix++ )
    {
        Anew[iy][ix] = -0.25 * (rhs[iy][ix] - ( A[iy][ix+1] + A[iy][ix-1]
                                                + A[iy-1][ix] + A[iy+1][ix] ) );
        error = fmax( error, fabs(Anew[iy][ix]-A[iy][ix]) );
    }
}
```

After GPUs update their values of A, the CPU copy is no longer correct

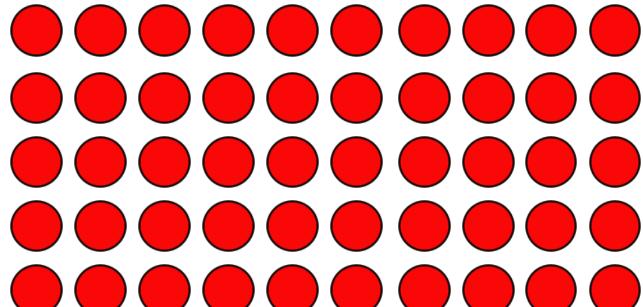
**Thread 0's copy of its rows of A
(on GPU 0)**



CPU copy of A



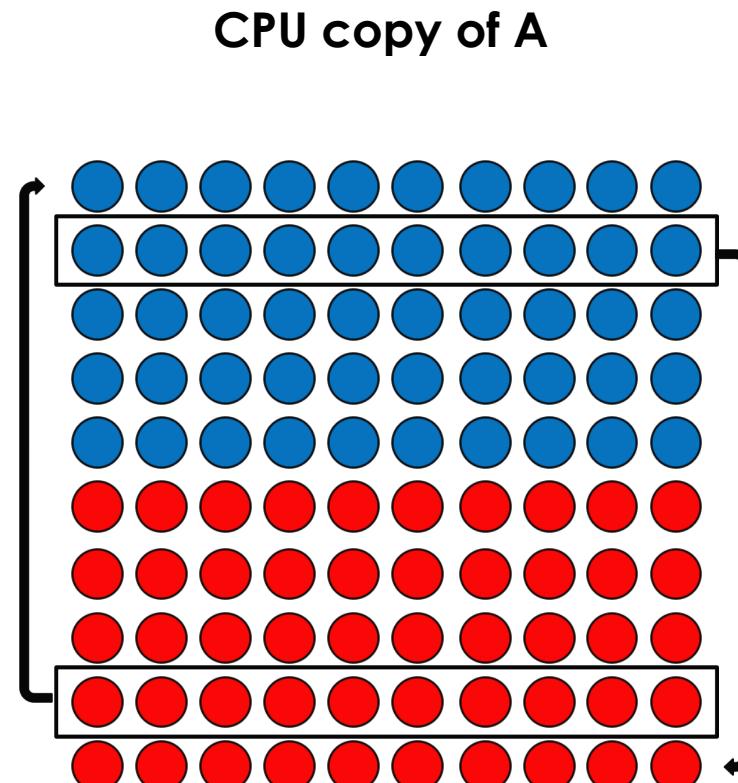
**Thread 1's copy of its rows of A
(on GPU 1)**



Differences from Single GPU Version

Recall that boundary conditions must be updated for A matrix as a whole

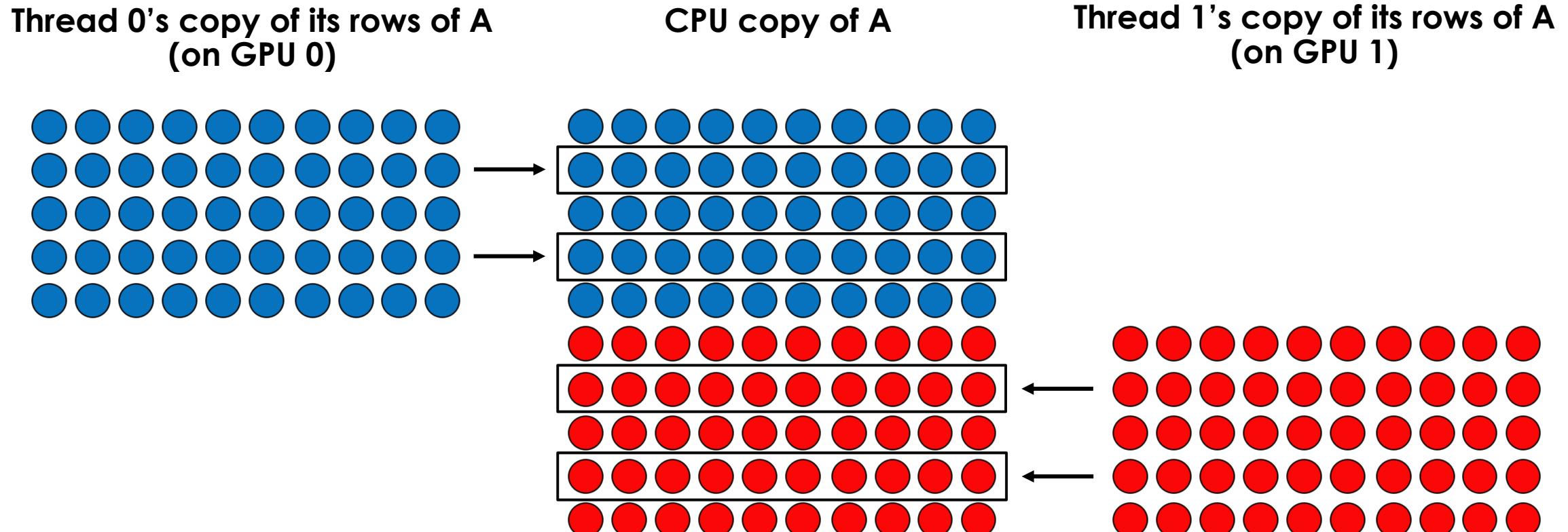
- But each GPU only has its rows of A
- So some data must be passed back to CPU



Differences from Single GPU Version

```
#pragma acc update self(A[iy_start:1][0:NX], A[(iy_end-1):1][0:NX])
```

Each thread updates the “shared” CPU copy of A with its “2nd-to-top” row and “2nd-to-bottom” row



Differences from Single GPU Version

Top/Bottom
Boundaries

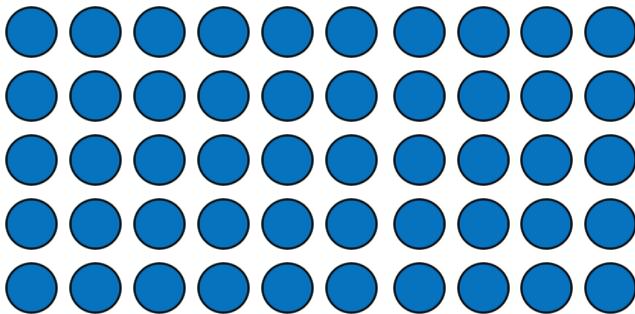
```
if(0 == (iy_start-1))  
{  
    for( int ix = 1; ix < NX-1; ix++ )  
    {  
        A[0][ix] = A[(NY-2)][ix];  
    }  
}
```

Side
Boundaries

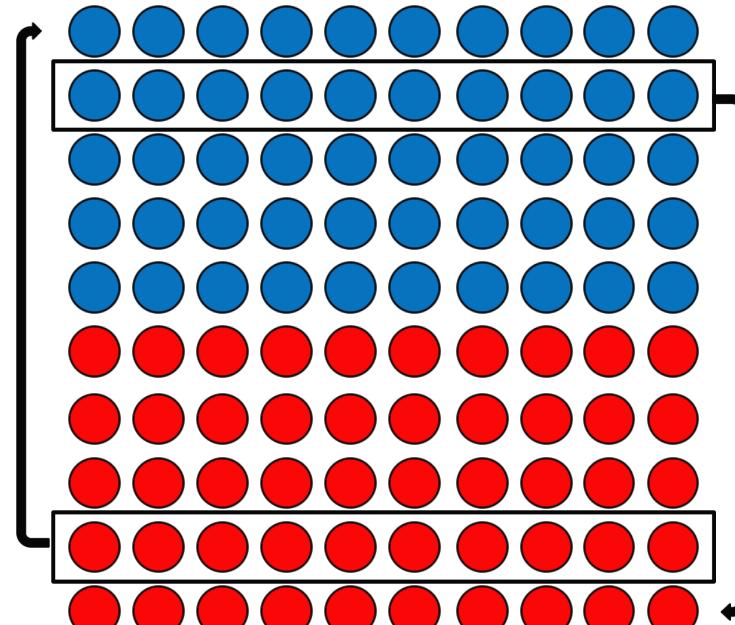
```
if((NY-1) == (iy_end))  
{  
    for( int ix = 1; ix < NX-1; ix++ )  
    {  
        A[(NY-1)][ix] = A[1][ix];  
    }  
}
```

Only the threads with ($0 == (\text{iy_start}-1)$) and ($((\text{NY}-1) == (\text{iy_end}))$) perform the boundary updates

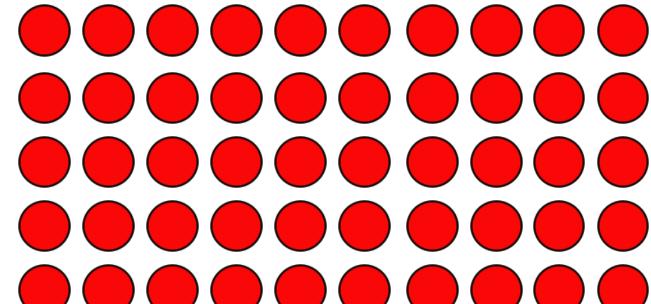
**Thread 0's copy of its rows of A
(on GPU 0)**



CPU copy of A



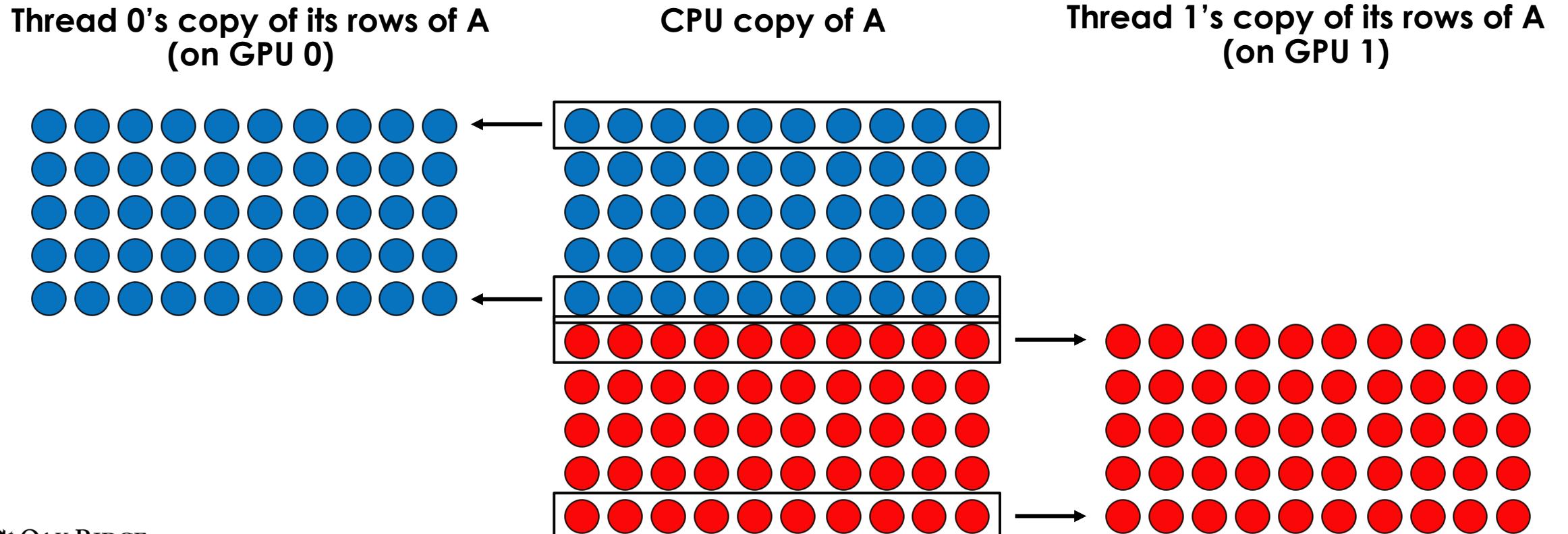
**Thread 1's copy of its rows of A
(on GPU 1)**



Differences from Single GPU Version

```
#pragma acc update device(A[(iy_start-1):1][0:NX], A[iy_end:1][0:NX])
```

Each thread updates its “top” row and “bottom” row from the new values of the CPU copy of A



Runtime of Single GPU Version with Data Directives

Compile the code

```
$ make
pgcc -acc -Minfo=acc -ta=tesla:cc70 -mp -fast -c poisson2d.c
poisson2d_serial: ...
main:
  103, Generating implicit copyout(A[:, :], A_ref[:, :])
  104, Loop is parallelizable
  106, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
  104, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
  106, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
  167, Generating copyin(rhs[iy_start:iy_end-iy_start][:])
    Generating create(Anew[iy_start:iy_end-iy_start][:])
    Generating copy(A[iy_start-1:iy_end-iy_start+2][:])
  181, Loop is parallelizable
  183, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
  181, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
  183, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
  187, Generating implicit reduction(max:error)
  200, Loop is parallelizable
  202, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
  200, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
  202, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
  211, Generating update self(A[iy_start][:], A[iy_end-1][:])
  230, Generating update device(A[iy_start-1][:], A[iy_end][:])
  231, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
  231, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
pgcc -acc -Minfo=acc -ta=tesla:cc70 -mp -fast poisson2d.o -o run
```

Run the code (on 2 GPUs)

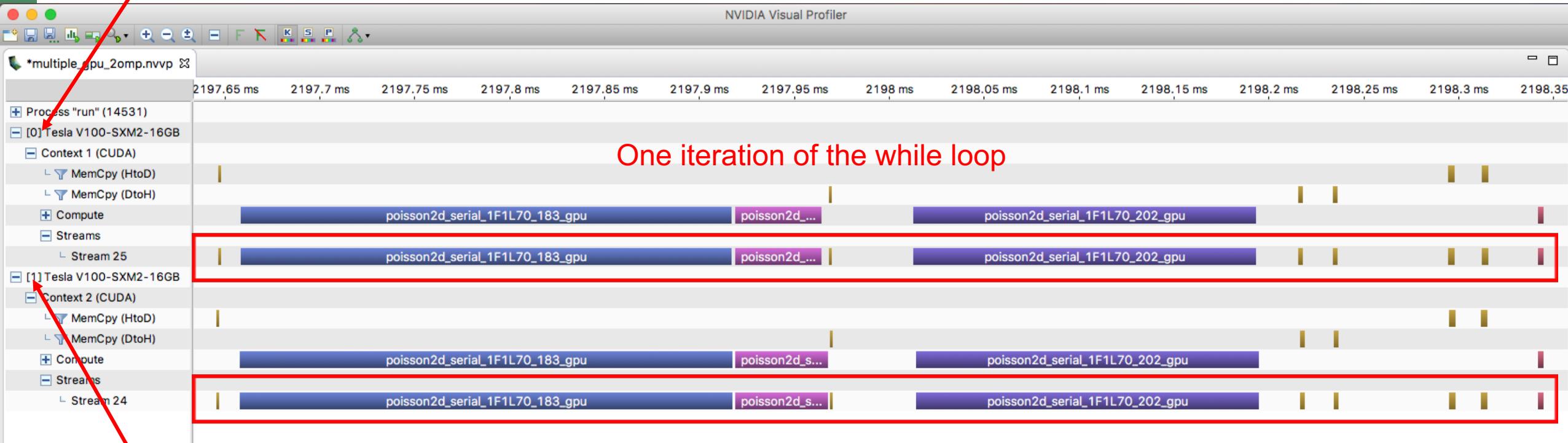
```
$ bsub submit2.ls
$ less output_2omp
Single-GPU Execution...
  0, 0.250000
  100, 0.249940
  200, 0.249880
  300, 0.249821
  400, 0.249761
  500, 0.249702
  600, 0.249642
  700, 0.249583
  800, 0.249524
  900, 0.249464
Parallel Execution...
  0, 0.250000
  100, 0.249940
  200, 0.249880
  300, 0.249821
  400, 0.249761
  500, 0.249702
  600, 0.249642
  700, 0.249583
  800, 0.249524
  900, 0.249464
Elapsed Time (s) - Serial: 1.0990,
                    Parallel: 0.6692,
                    Speedup: 1.6424
```

Using NVIDIA's Visual Profiler, we see...

OpenMP Thread 0 (GPU 0)

One iteration of the while loop

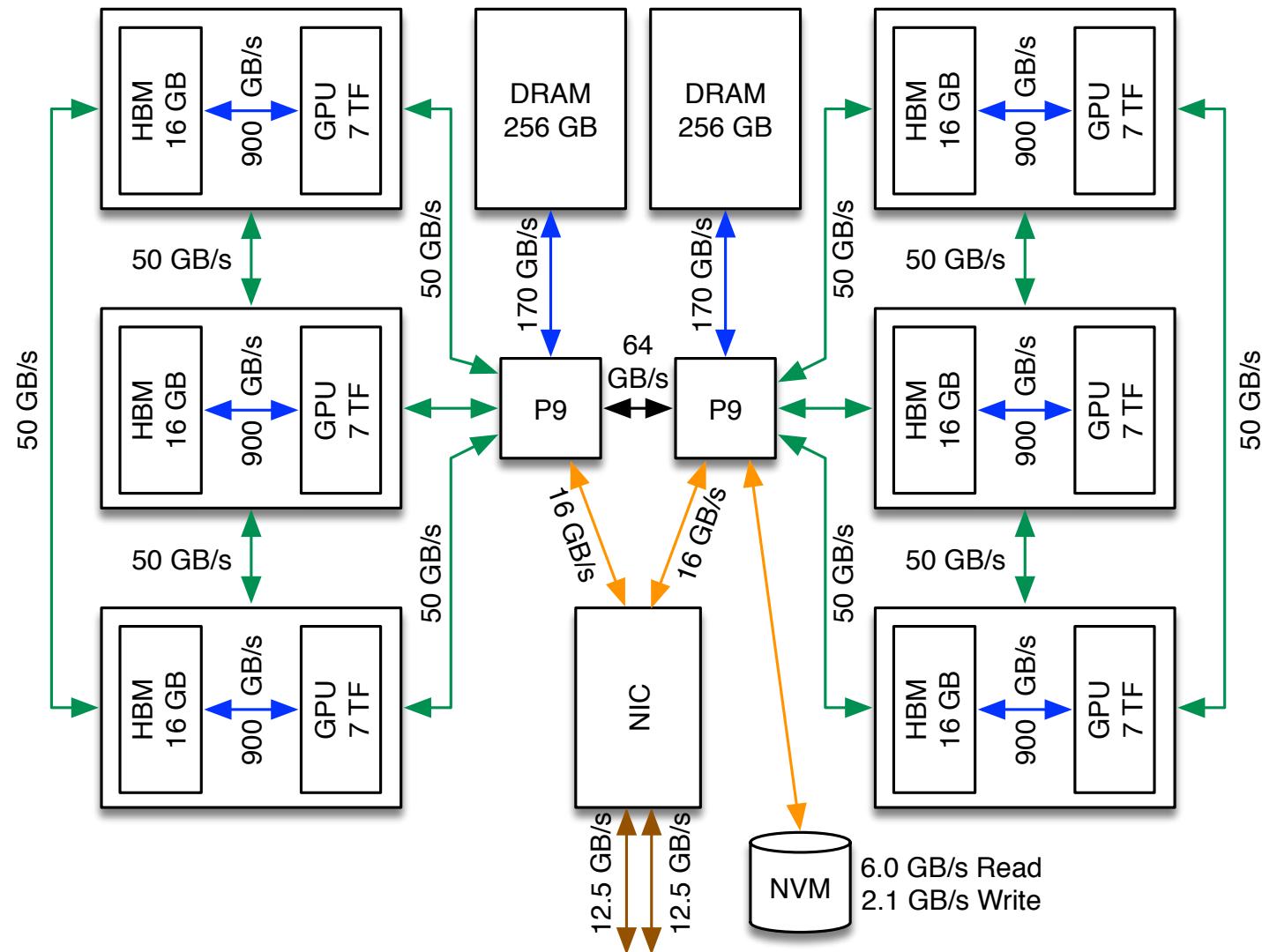
OpenMP Thread 1 (GPU 1)



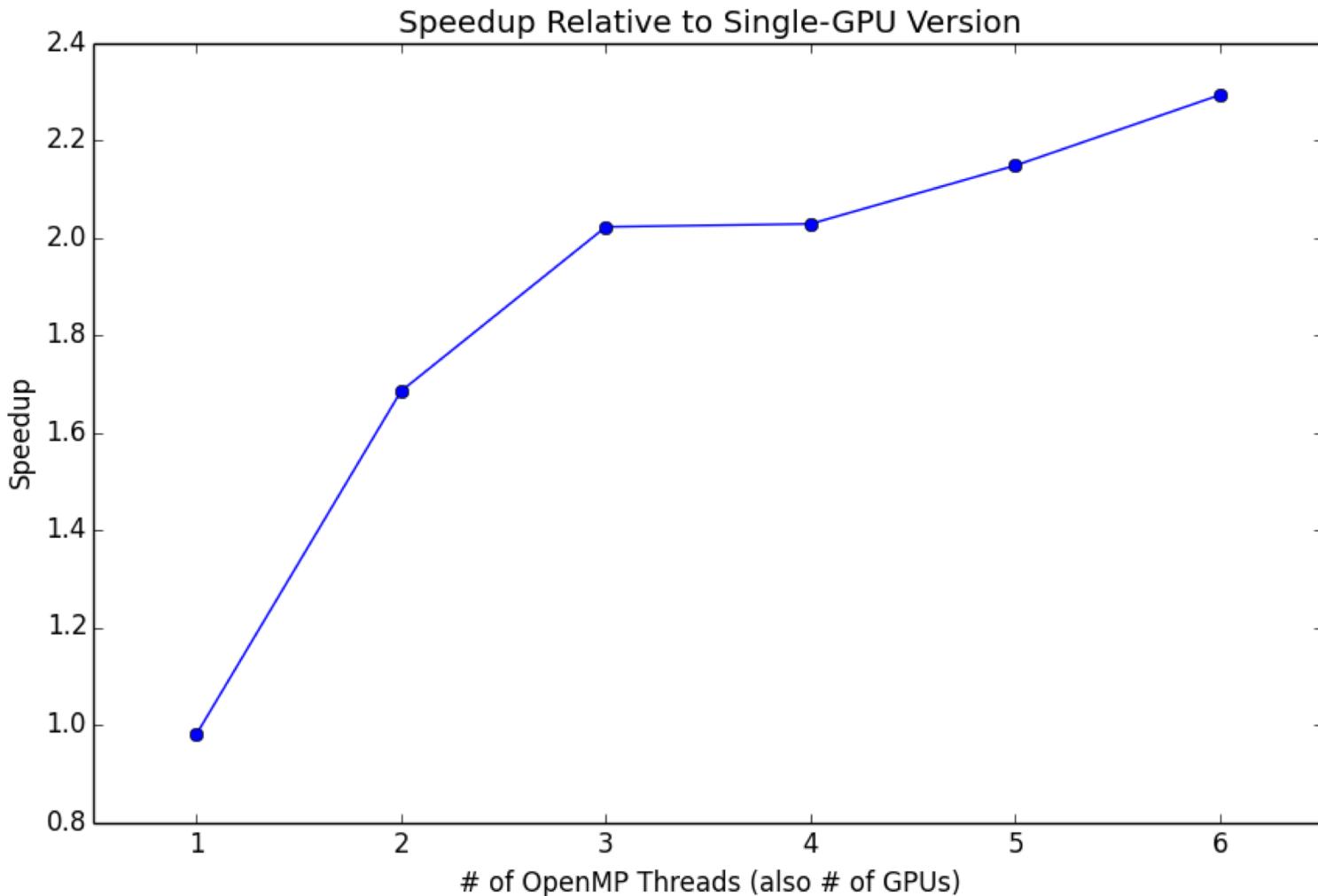
Plot Results of Speedup Versus # of GPUs

Navigate to Google Sheet to enter timing data: <https://goo.gl/G2fNeB>

Summit / Ascent Node Overview



Plot Results of Speedup Versus # of GPUs



Acknowledgements

