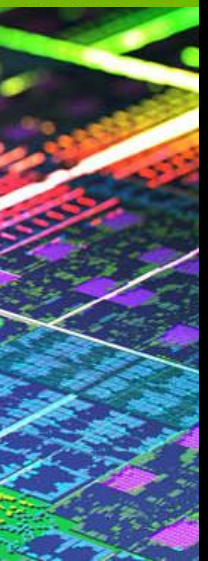
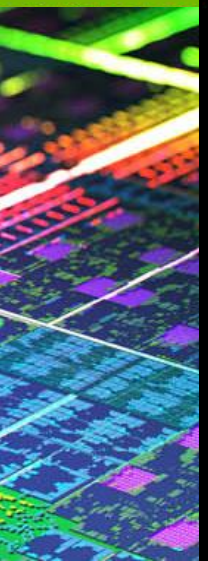
Multi-GPU: A Hands-on Exercise

Justin Luitjens
NVIDIA - Developer Technologies



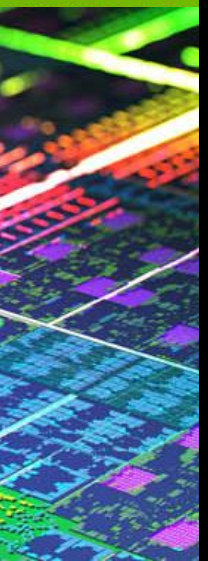
Connection instructions

- Navigate to nvlabs.qwiklab.com
- Login or create a new account
- Select the "Instructor-Led Hands-on Labs" class
- Find the lab called "Scaling to Multiple GPUs" and click Start
- After a short wait, lab instance connection information will be shown
- Please ask Lab Assistants for help!



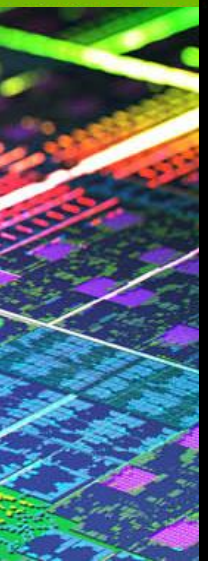
Why Should You Use Multiple GPUs

- Compute Faster
  - More GPU's = Faster time to solution
- Compute Larger
  - More GPU's = More memory for larger problems
- Compute Cheaper
  - More GPU's per node = less overhead in $, power and space



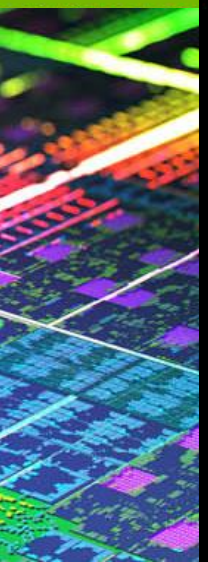
What You Will Learn About Today

- During today's lab
  - Scalability metrics
  - Managing multiple devices
  - Communicating between devices
- Homework
  - Communication Hiding
  - Synchronization



What is not Covered Today

- CUDA Basics
- Kernel Optimization
- MPI Parallelism
- Multi-GPU debugging



Scalability Metrics For Success

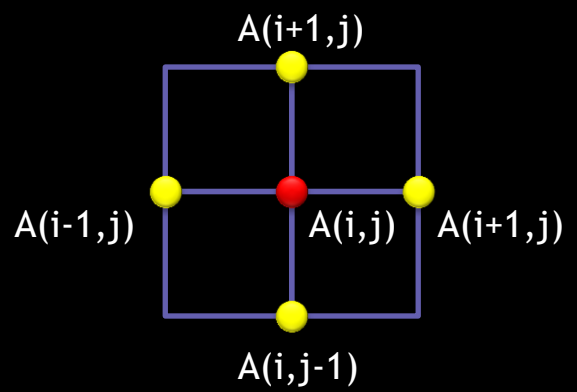
- **Serial Time:** T_s
- Parallel Time: T_p
- # of Processors: P
- Speedup up: **How much faster ?**

$$S = \frac{T_s}{T_p}$$ **Ideal : P**

- **Efficiency: How efficiently are processors used?**

$$E = \frac{S}{P}$$ **Ideal: 1**

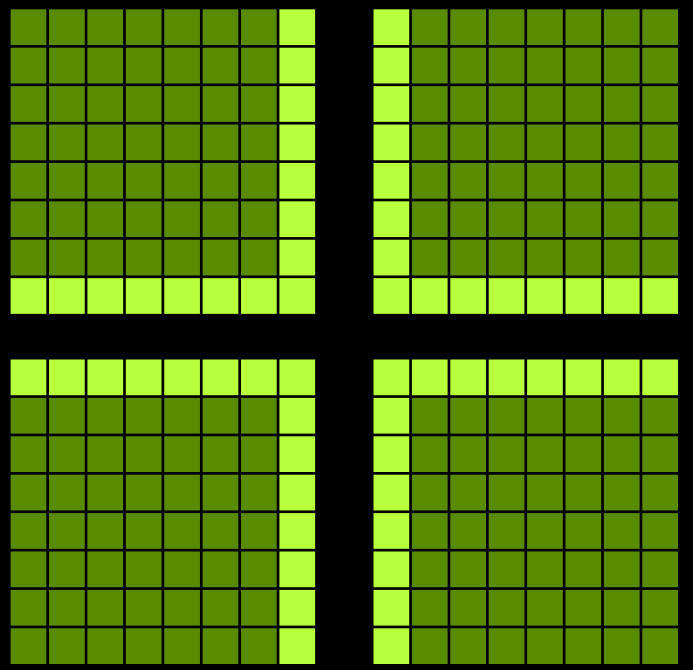
Case Study: 2D Laplace Solver



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

- Given a 2D grid
  - Set every vertex equal to the average of neighboring vertices
    - Repeat until converged
- Common algorithmic pattern
  - Electromagnetism, Astrophysics, Fluid Dynamics, Iterative Solvers

Domain Decomposition: 3 Options

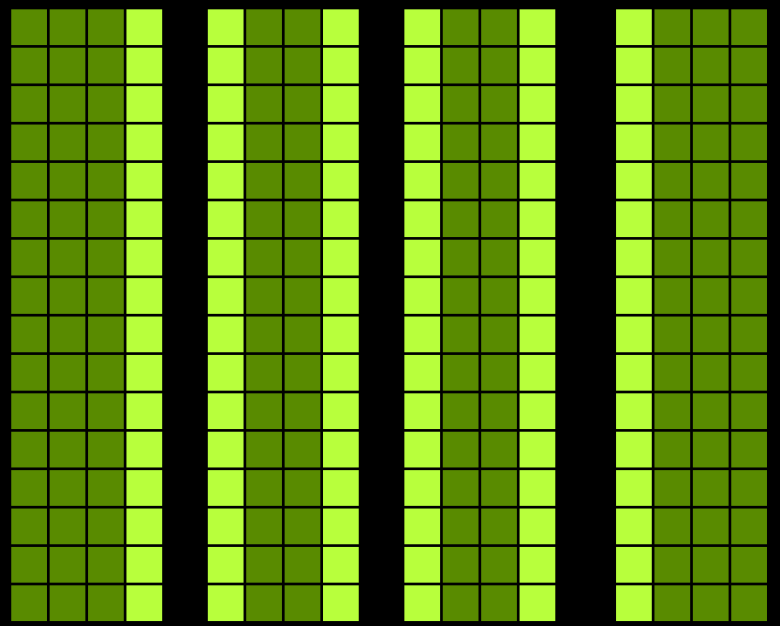


Tiles

Halo Region

- Minimizes surface area/volume ratio
  - Communicate less data
  - Optimal for bandwidth bound communication

Domain Decomposition: 3 Options

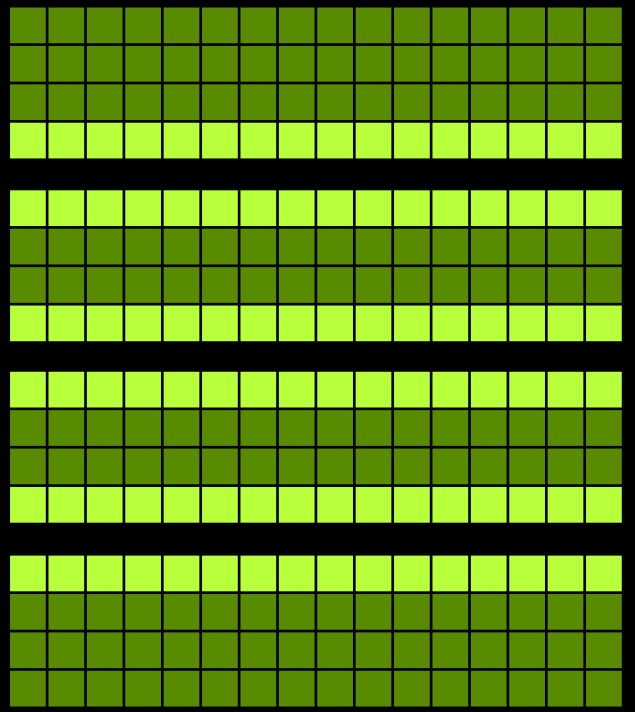


- Minimizes number of neighbors
  - Communicate to less neighbors
  - Optimal for latency bound communication
- Contiguous if data is column-major

Vertical Stripes

- Halo Region

Domain Decomposition: 3 Options



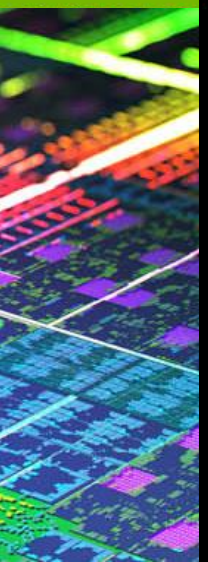
Horizontal Stripes

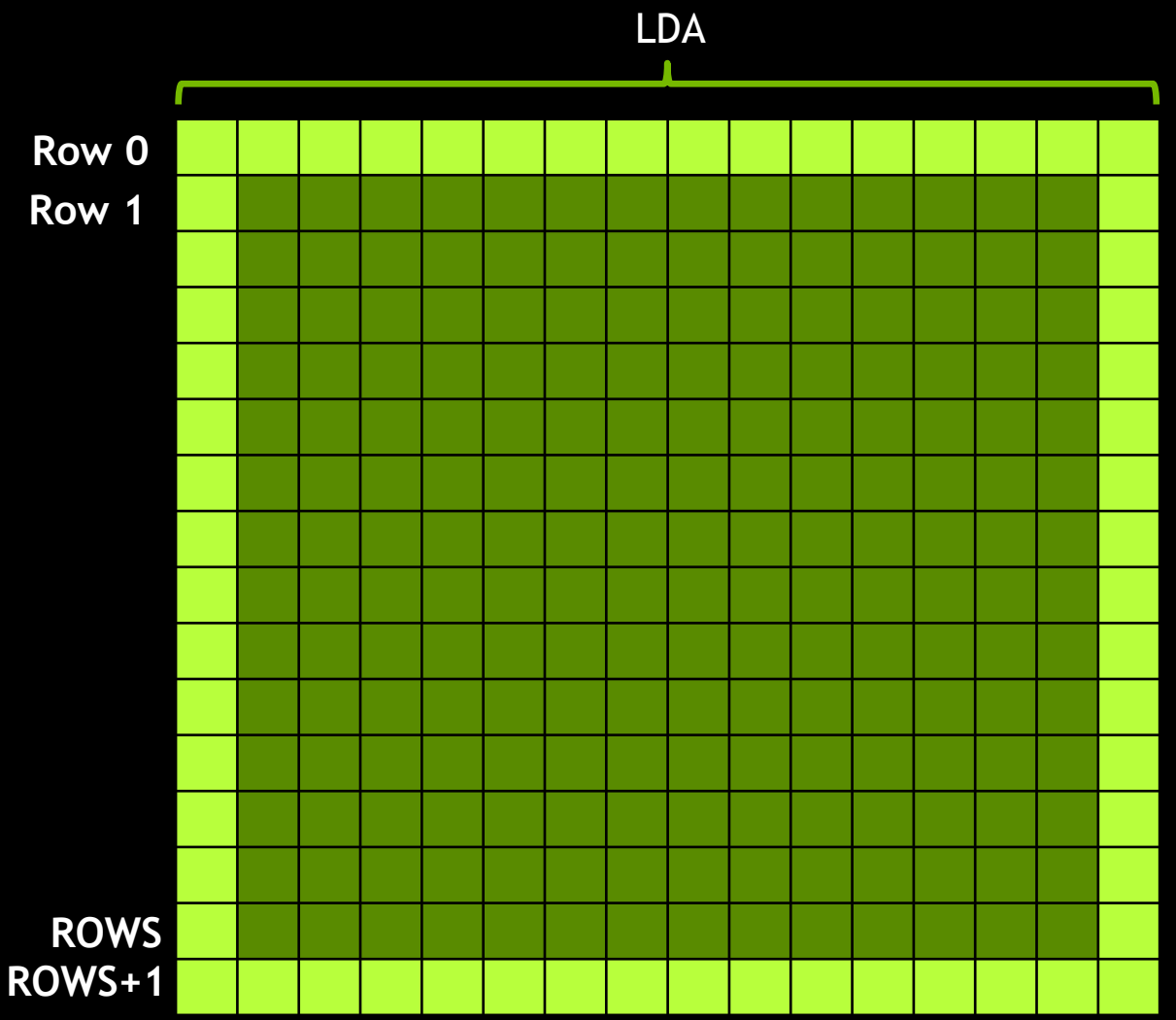
Halo Region

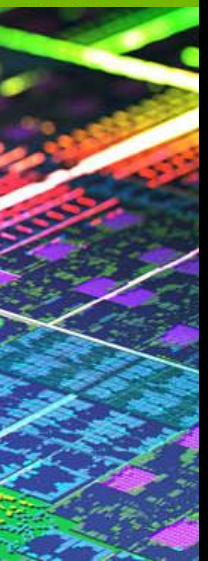
- Minimizes number of neighbors
  - Communicate to less neighbors
  - Optimal for latency bound communication
- Contiguous if data is row-major

Grid Indexing

- Boundary Condition & Halo Region

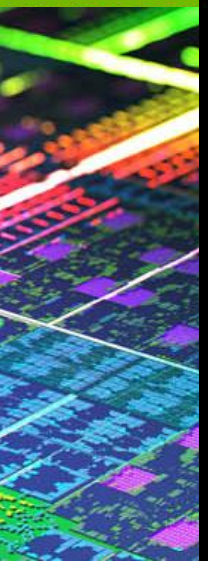






CUDA multi-GPU APIs

- All memory allocation and kernel launches occur on the currently active device
- Simple API's to get and set the device:
  - cudaSetDevice(int d)
  - cudaGetDevice(int *d)
- Memory can be transferred directly between devices
  - cudaMemcpyPeer(…)
  - cudaMemcpyPeerAsync(…)
- For direct memory copies you must enable peer access
  - cudaDeviceCanAccessPeer(&access,i,j)
  - cudaDeviceEnablePeerAccess(…)

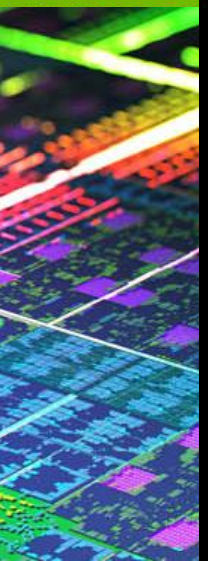


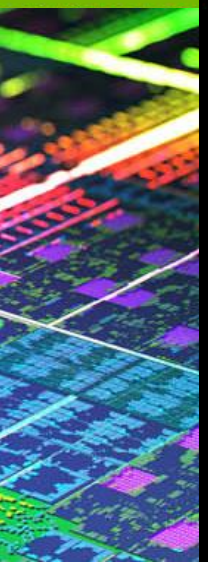
Enabling Peer Access

```
for(int i=0;i<GPUS;i++) {
   cudaSetDevice(i);
   for(int j=0;j<GPUS;j++) {
     if(i!=j) {
       int access;
       cudaDeviceCanAccessPeer(&access,i,j);
       if(access) {
         cudaDeviceEnablePeerAccess(j,0);
       }
   }
 }
```

Hands On Exercise

- Progressive exercise
  - 5 Tasks
  - We will go until we run out of time
  - Each step begins where the last left off
  - Feel free to work ahead





Task 1

- Replicate the computation on all devices
  - Cuda calls apply to the current device

```
for(int d=0;d<numDev;d++) {
  cudaSetDevice(d);
  ...
```

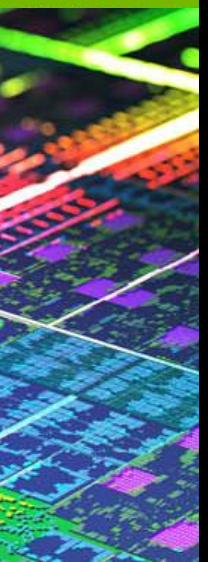
- Use asynchronous memory copies
  - Required for devices to operate in parallel
    - Otherwise host would block until transfers were complete
  - Memory must be pinned
  - For now use stream 0
    - cudaMemcpyAsync(…, 0);

Allocate & Free Memory on Each Device

```
double *d_Ain[                ];
double *d_Aout[               ];
//allocate device memory
[                                        ]
  [                    ]
    cudaMalloc(&d_Ain[ ],bytes);
    cudaMalloc(&d_Aout[ ],bytes);
    cudaCheckError();
[ ]
```

One array per device

Loop over each device

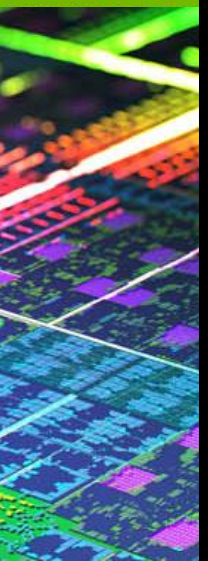


Allocate & Free Memory on Each Device

```
double *d_Ain[MAX_DEVICES];
double *d_Aout[MAX_DEVICES];
//allocate device memory
for(int d=0;d<numDev;d++) {
  cudaSetDevice(d);
  cudaMalloc(&d_Ain[d],bytes);
  cudaMalloc(&d_Aout[d],bytes);
  cudaCheckError();
}
```

One array per device

Loop over each device

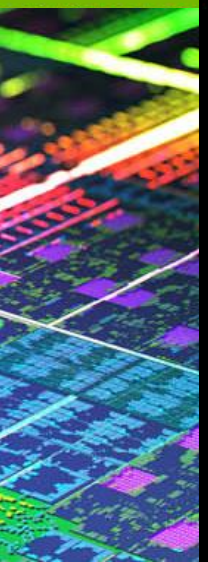


Copy Data To Each Device Asynchronously

```
//copy initial conditions to both buffers
[redacted]

    cudaMemcpy[redacted](d_Ain[redacted],A,bytes,cudaMemcpyHostToDevice,0);
    cudaMemcpy[redacted](d_Aout[redacted],d_Ain[redacted],bytes,
                        cudaMemcpyDeviceToDevice,0);

    cudaCheckError();
[redacted]
```

Loop Over Each Device

Asynchronous Memory Copies

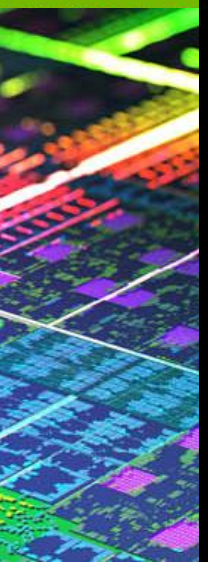


Copy Data To Each Device Asynchronously

```
//copy initial conditions to both buffers
 for(int d=0;d<numDev;d++) {
   cudaSetDevice(d);
   cudaMemcpyAsync(d_Ain[d],A,bytes,cudaMemcpyHostToDevice,0);
   cudaMemcpyAsync(d_Aout[d],d_Ain[d],bytes,
                              cudaMemcpyDeviceToDevice,0);
   cudaCheckError();
 }
```

Loop Over Each Device

Asynchronous Memory Copies

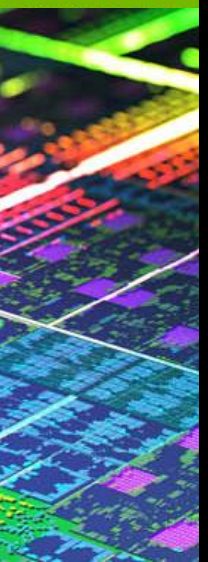


Launch the Kernel on Each Device

```
    simpleLaplaceIter_kernel<<<gridSize,blockSize>>>
                                (ROWS,COLS,d_Ain[d],d_Aout[d]);
    cudaCheckError();
}

for(int d=0;d<numDev;d++) {
    std::swap(d_Ain[d],d_Aout[d]);
}
```

Loop Over Each Device

Swap Input and Output Arrays

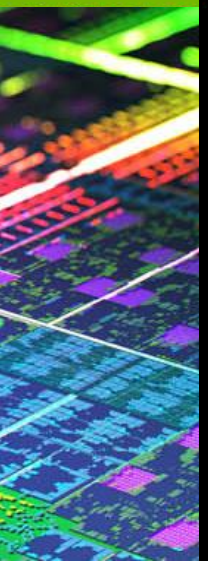


Launch the Kernel on Each Device

```
for(int d=0;d<numDev;d++) {
  cudaSetDevice(d);
  simpleLaplaceIter_kernel<<<gridSize,blockSize>>>
                          (ROWS,COLS,d_Ain[d],d_Aout[d]);
  cudaCheckError();
}

for(int d=0;d<numDev;d++) {
  std::swap(d_Ain[d],d_Aout[d]);
}
```

Loop Over Each Device

Swap Input and Output Arrays

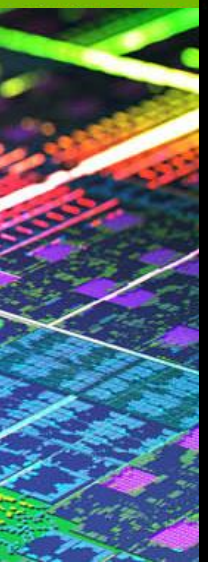


Copy Data From Each Device Asynchronously

```
//copy results back to host
[redacted]
  [redacted]
    cudaMemcpy[redacted](A,d_Ain[redacted],bytes,cudaMemcpyDeviceToHost,0);
    cudaCheckError();
[redacted]
```

Loop Over Each Device

Copy Asynchronously

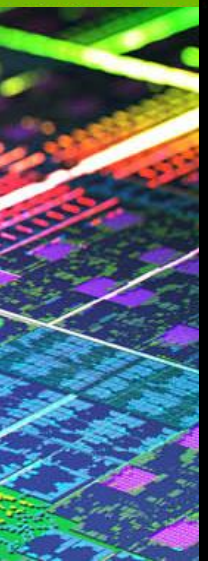


Copy Data From Each Device Asynchronously

```
//copy results back to host
 for(int d=0;d<numDev;d++) {
   cudaSetDevice(d);
   cudaMemcpyAsync(A,d_Ain[d],bytes,cudaMemcpyDeviceToHost,0);
   cudaCheckError();
 }
```

Loop Over Each Device

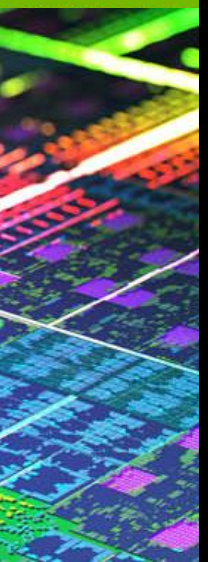
Copy Asynchronously



Free Memory on Each Device

```
//free device memory

    cudaFree(d_Ain  );
    cudaFree(d_Aout  );
    cudaCheckError();
```

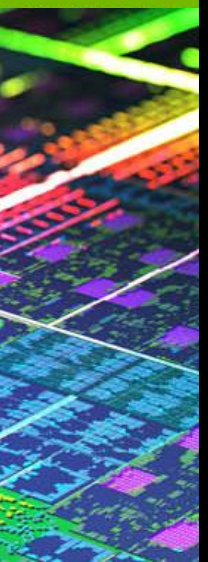
Loop Over Each Device



Free Memory on Each Device

```
//free device memory
 for(int d=0;d<numDev;d++) {
   cudaFree(d_Ain[d]);
   cudaFree(d_Aout[d]);
   cudaCheckError();
 }
```

Loop Over Each Device

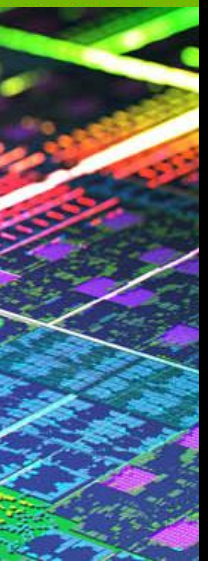


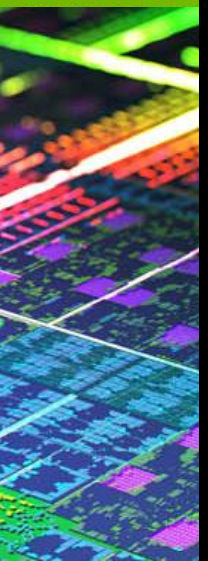
Task 1: Results

- Test machine: Dual Socket Xeon X5675 with 8 M2090s

Size	Speedup	Efficiency
2048x2048	1	12.5%
4096x4096	1	12.5%
8192x8192	1	12.5%

- No slowdown from adding more devices
- All GPUs are solving the same problem
  - Not a very efficient use of resources





Profiler Supports Multi-GPU

- Option 1: Run directly in nsight or NVVP
  - Cuda 5.5 and earlier requires X display
- Option 2:
  - Collect profiles using nvprof:

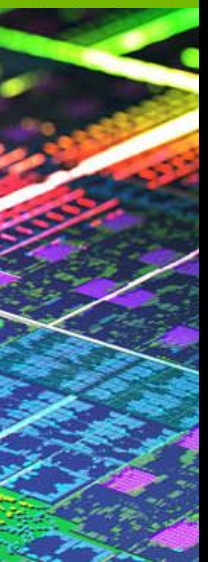
```
nvprof -o profile.nvprof ./laplace
```

  - Import profile into NVVP

- Since we are using IPython we cannot run the profiler now.

Task 2

- Assign unique work to each GPU
  - Horizontal stripes decomposition
    - Assign unique work to each device
    - Copy a subset of the memory to each device
    - Copy data back to the appropriate host location
  - Exchange halo regions at each iteration

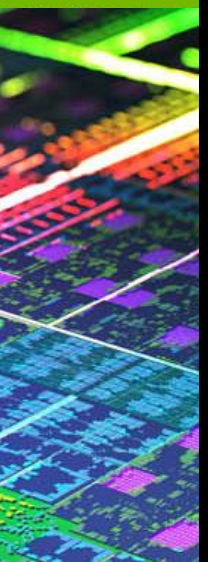


Assign Unique Work To Each Device

```
int rows=
...
dim3 gridSize( ceil((double)(COLS)/blockSize.x),
               ceil((double)(    )/blockSize.y));
...
simpleLaplaceIter_kernel<<<gridSize,blockSize>>>(    ,COLS,
               d_Ain[d],d_Aout[d]);
```

Compute Local Rows Per Device

Adjust Running Dimensions

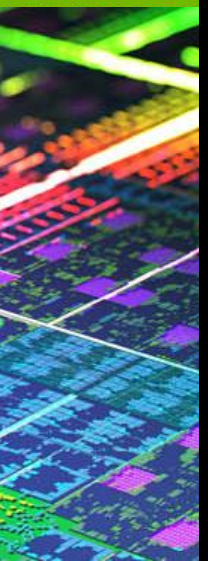


Assign Unique Work To Each Device

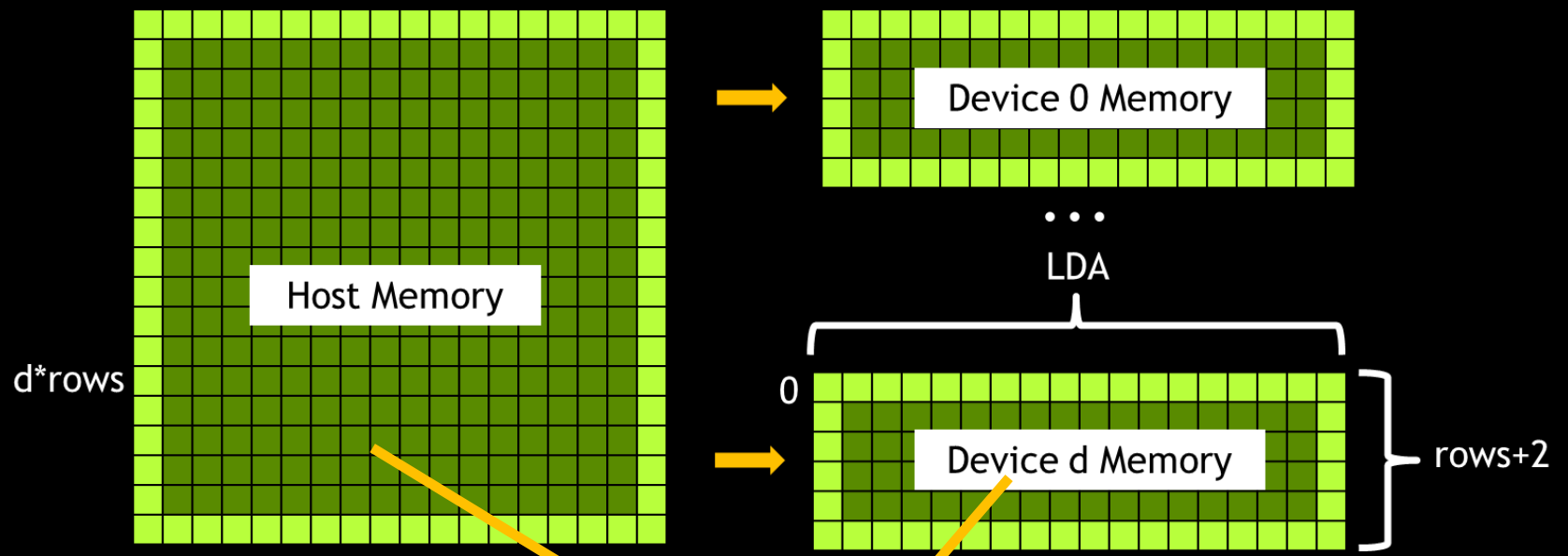
```
int rows=ROWS/numDev;
...
dim3 gridSize( ceil((double)(COLS)/blockSize.x),
               ceil((double)(rows)/blockSize.y));
...
simpleLaplaceIter_kernel<<<gridSize,blockSize>>>(rows,COLS,
               d_Ain[d],d_Aout[d]);
```

Compute Local Rows Per Device

Adjust Running Dimensions



Copy Unique Work to the Devices

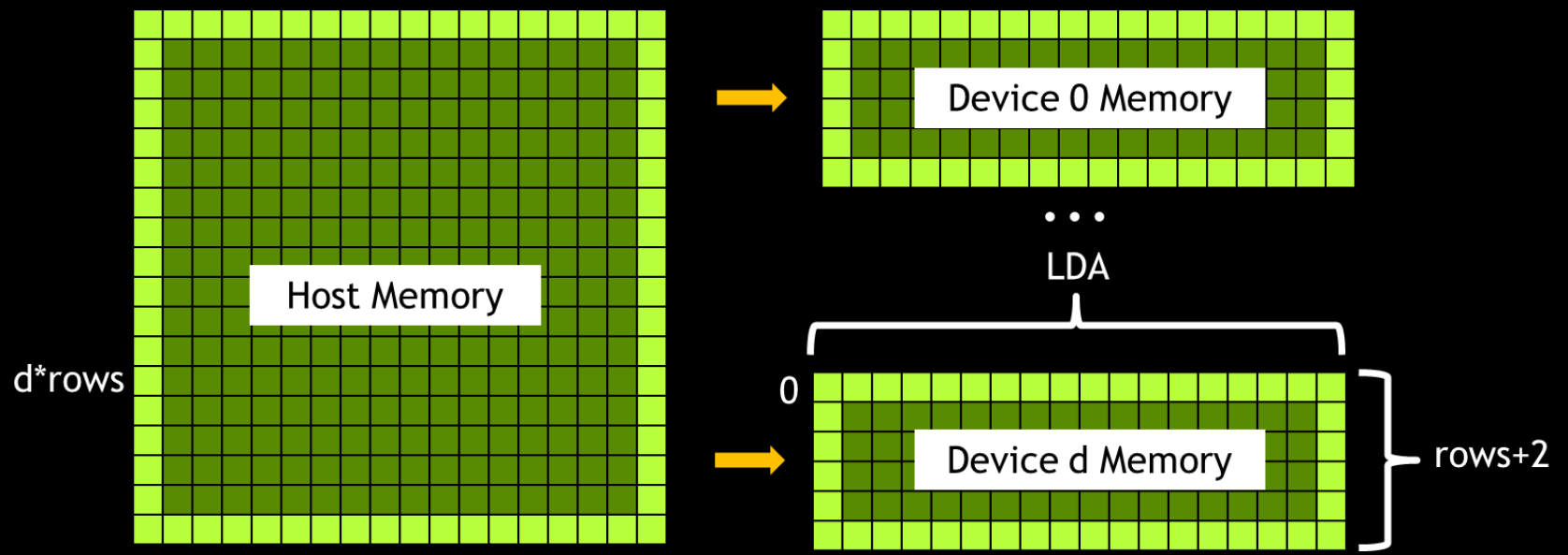



```
size_t bytes=(      +2)*(LDA)*sizeof(double);
...
//copy from host to device
cudaMemcpyAsync(d_Ain[d]+IDX(          ),A+IDX(                    ),
                bytes*LDA*sizeof(double),cudaMemcpyHostToDevice,0);
```

Compute Transfer Size

Compute Indices

Copy Unique Work to the Devices

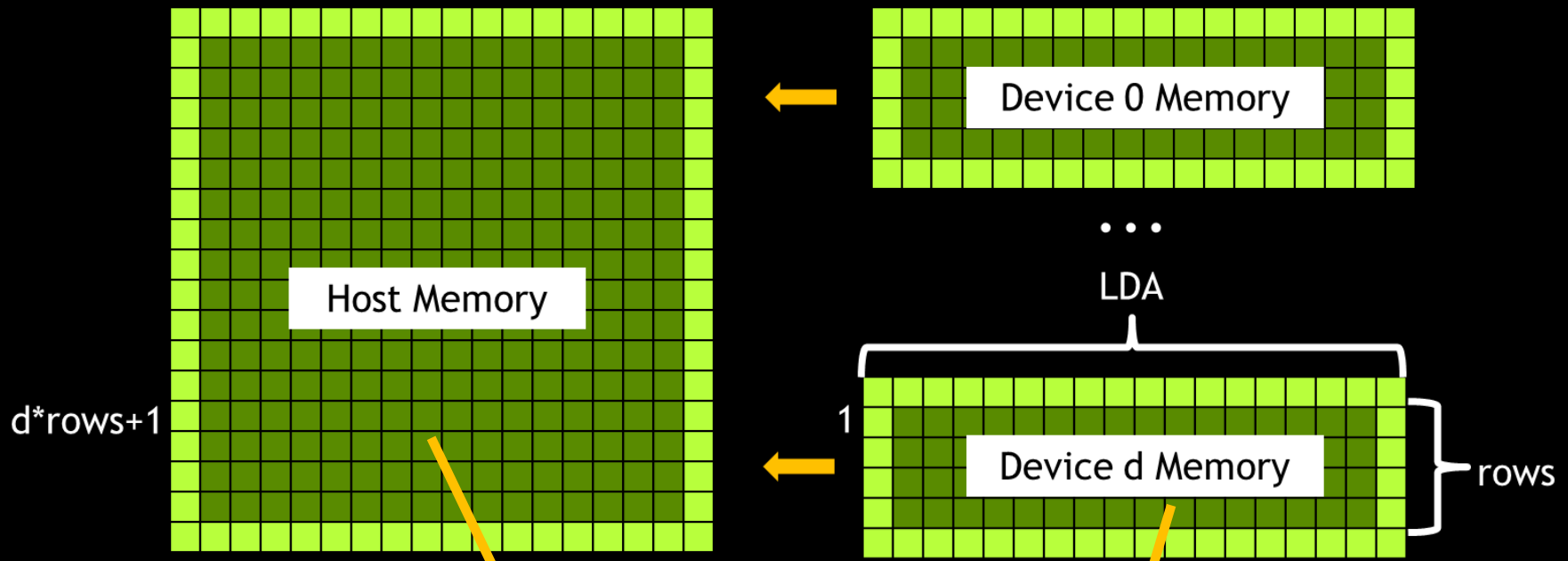



```
size_t bytes=(rows+2)*(LDA)*sizeof(double);
...
//copy from host to device
cudaMemcpyAsync(d_Ain[d]+IDX(0,0,LDA),A+IDX(d*rows,0,LDA),
                bytes*LDA*sizeof(double),cudaMemcpyHostToDevice,0);
```

Compute Transfer Size

Compute Indices

Copy Results to the Host

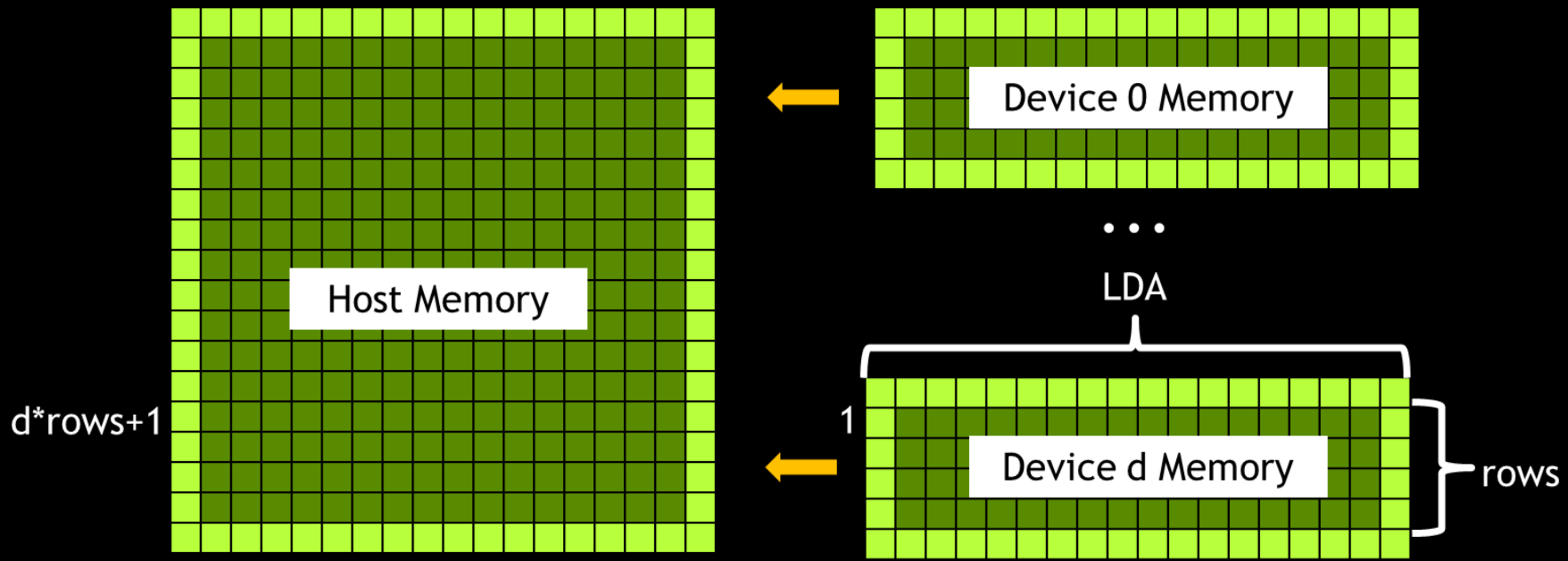



```
int bytes_int=      *LDA*sizeof(double);
...
//copy from device to host
cudaMemcpyAsync(A+IDX(                    ),d_Ain[d]+IDX(      ),
                          ,cudaMemcpyDeviceToHost,0);
```

Compute Transfer Size

Compute Indices

Copy Results to the Host



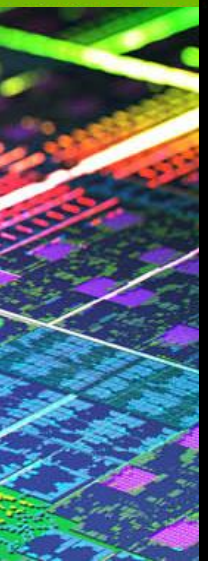

```
int bytes_int=rows*LDA*sizeof(double);
...
//copy from device to host
cudaMemcpyAsync(A+IDX(d*rows+1,0,LDA),d_Ain[d]+IDX(1,0,LDA),
                bytes_int,cudaMemcpyDeviceToHost,0);
```

Compute Transfer Size

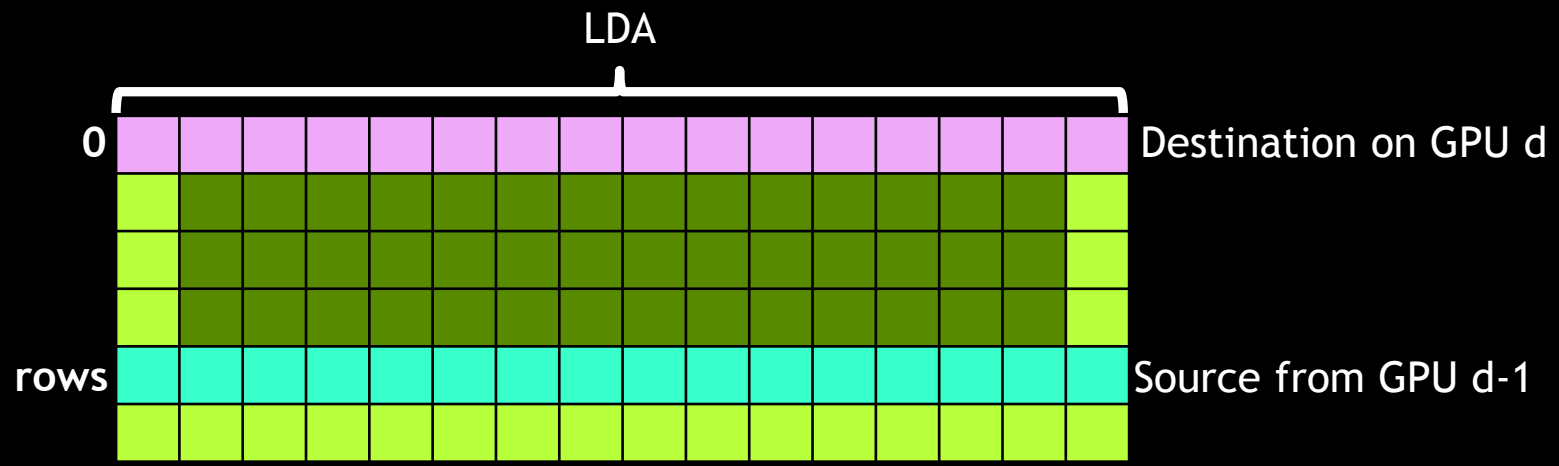
Compute Indices

Exchange Halos After Kernel Launch

```
for(int d=0;d<numDev;d++) {
    simpleLaplaceIter_kernel<<<gridSize,blockSize>>>(...);
}

for(int d=0;d<numDev;d++) {
    //Grab lower boundary


    //Grab upper boundary

  cudaCheckError();
}
```

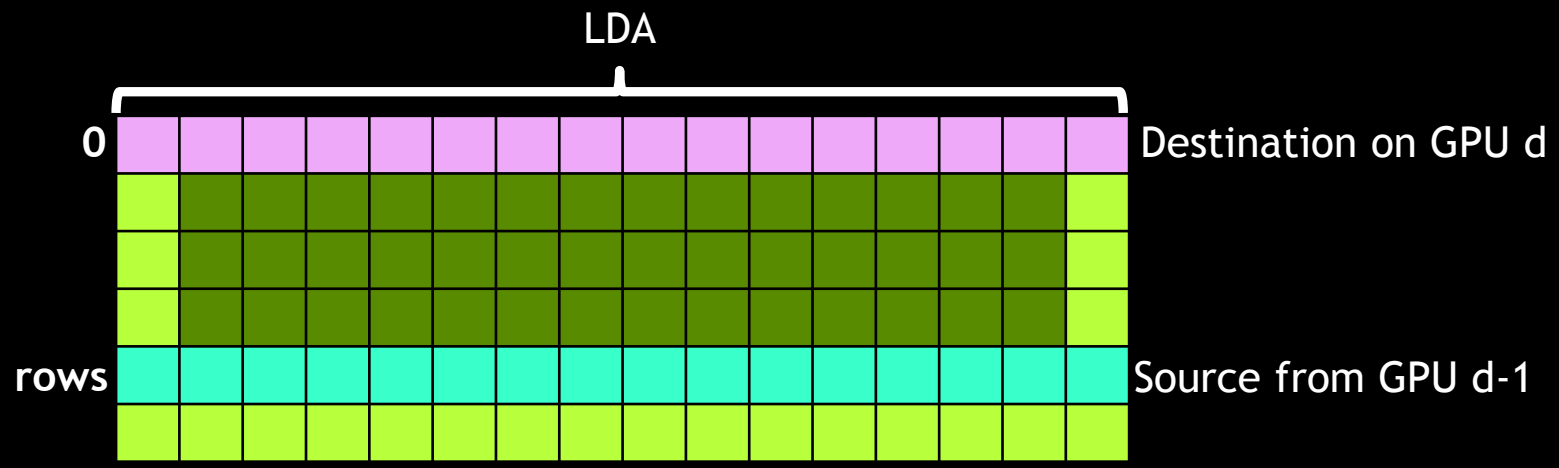


Copy Lower Boundary



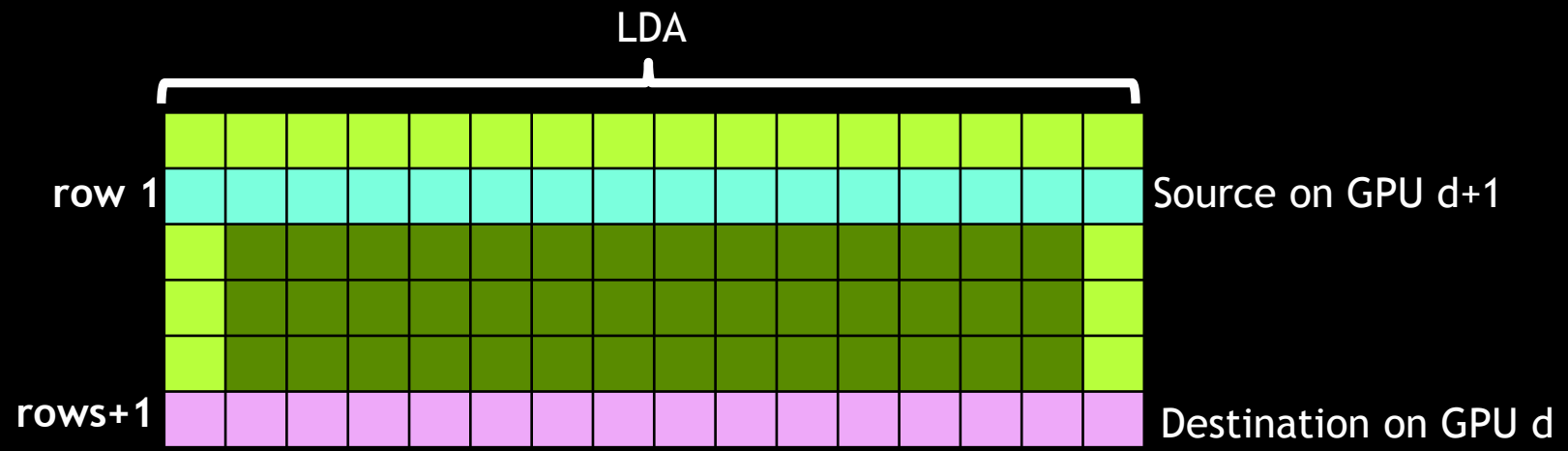

```
//Grab lower boundary
  if(d>0)
    cudaMemcpyPeerAsync(d_Aout[ ]+IDX(          ),  ,
                        d_Aout[    ]+IDX(                  ),    ,
                               *sizeof(double),0);
```

Copy Lower Boundary



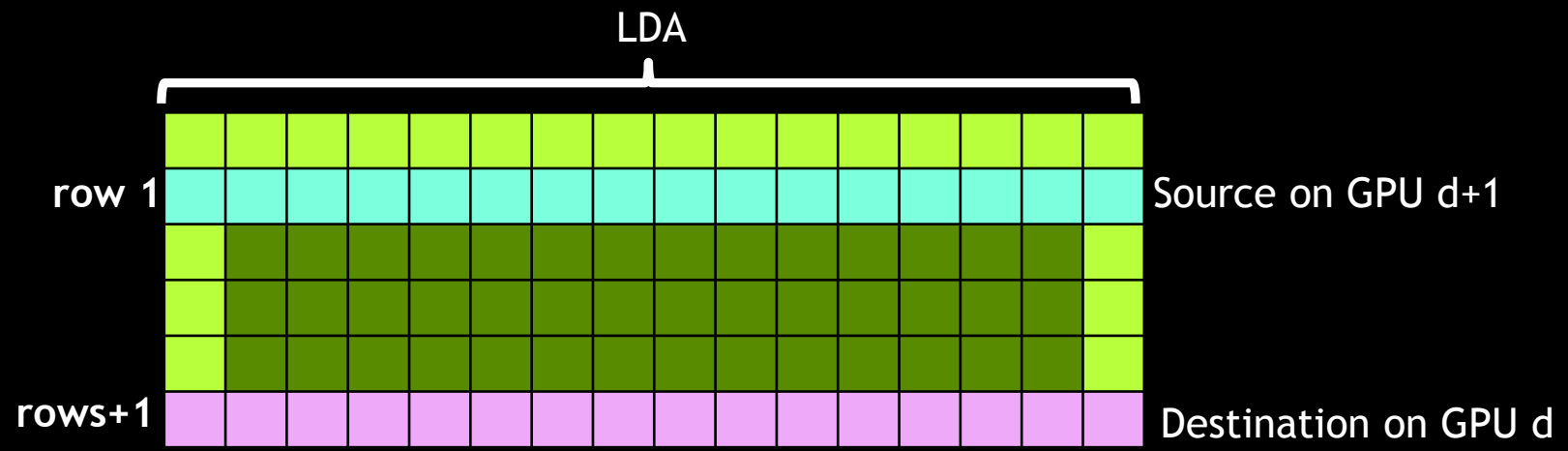

```
//Grab lower boundary
  if(d>0)
    cudaMemcpyPeerAsync(d_Aout[d]+IDX(0,0,LDA),d,
                        d_Aout[d-1]+IDX(rows,0,LDA),d-1,
                        LDA*sizeof(double),0);
```

Copy Upper Boundary




```
//Grab upper boundary
  if(d<numDev-1)
    cudaMemcpyPeerAsync(d_Aout[ ]+IDX(                  ,
                        d_Aout[ ]+IDX(          ),
                        *sizeof(double),0);
```

Copy Upper Boundary



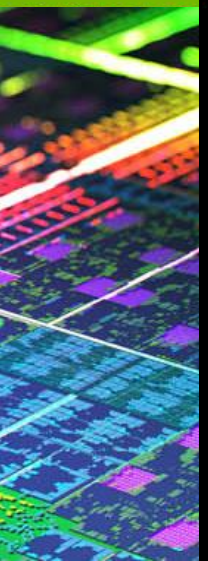

```
//Grab upper boundary
  if(d<numDev-1)
    cudaMemcpyPeerAsync(d_Aout[d]+IDX(rows+1,0,LDA),d,
                        d_Aout[d+1]+IDX(1,0,LDA),d+1,
                        LDA*sizeof(double),0);
```

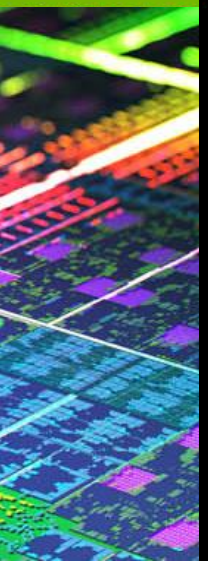
Task 2: Results

- Test machine: Dual Socket Xeon X5675 with 8 M2090s

Size	Speedup	Efficiency
2048x2048	3.30	41%
4096x4096	5.67	71%
8192x8192	7.28	91%

- Decent speedups have been achieved
- Efficiency increases with problem size



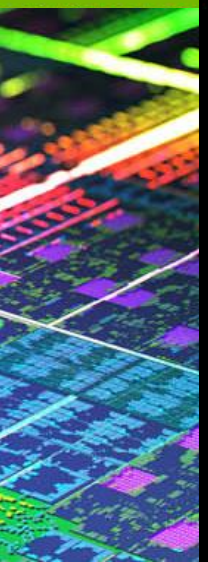


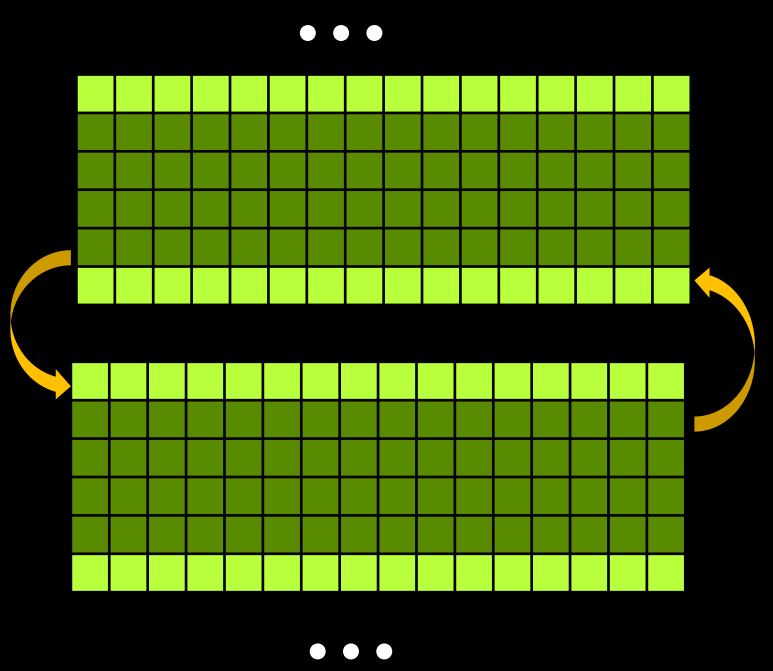
Advanced Multi-GPU

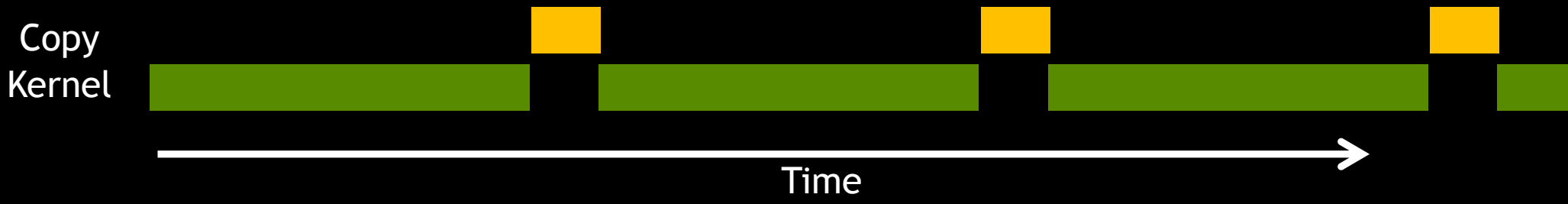
- Communication Hiding
  - Overlap communication & computation
  - Steps 3-5
  - Due to time this part won't be interactive
    - Notebook has steps in it if you want to try keeping up

Current Implementation

- Compute Laplace
- Exchange Halos

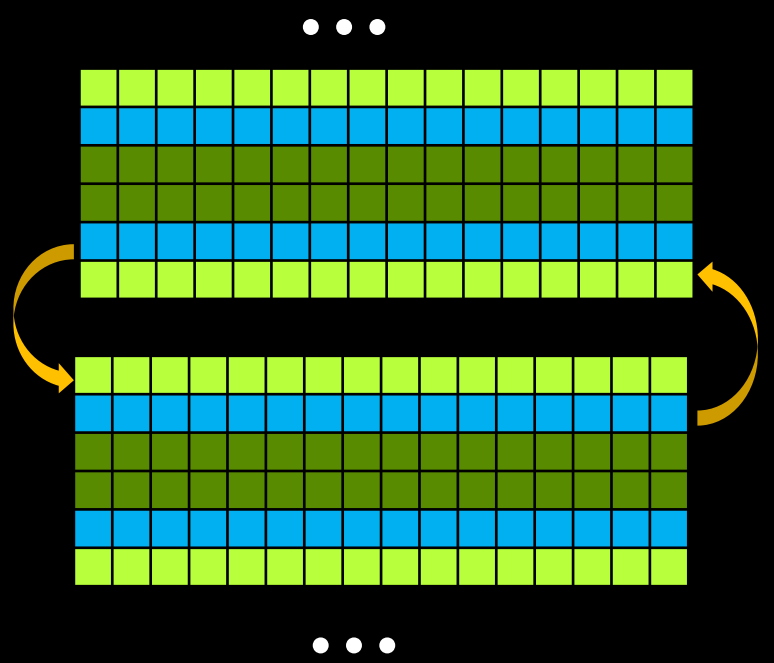


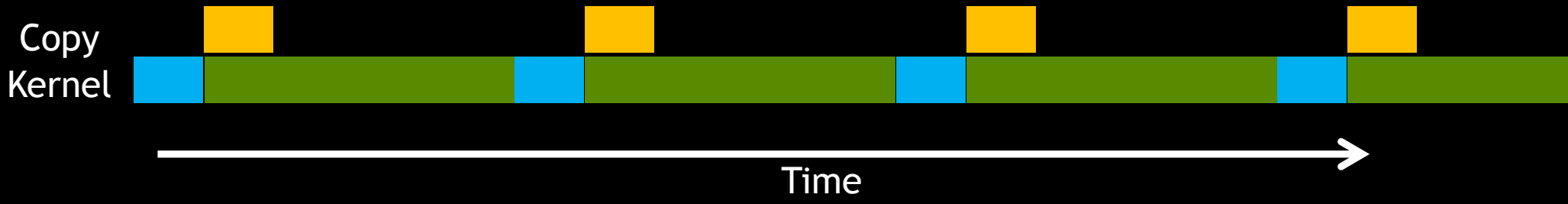


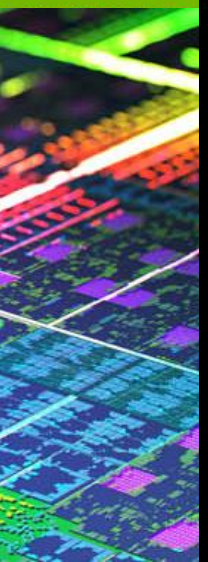


Communication Hiding

- Compute Exterior Async
- Compute Interior Async
- Exchange Halos Async







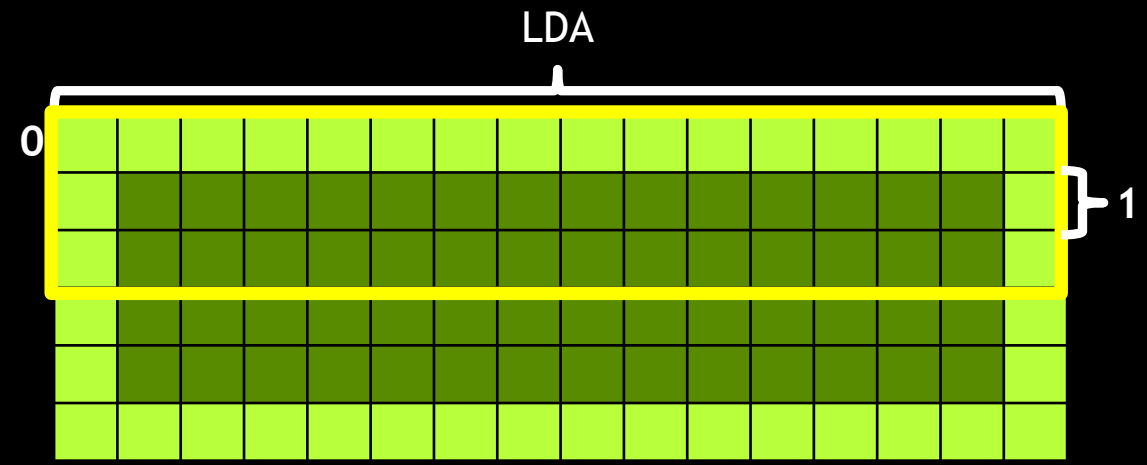
Task 3

- Separate computation into interior & exterior
  - Use 1D kernel for boundary

```
dim3 blockSize_e (128);
dim3 gridSize_e(ceil((double)(COLS)/blockSize_e.x));
```

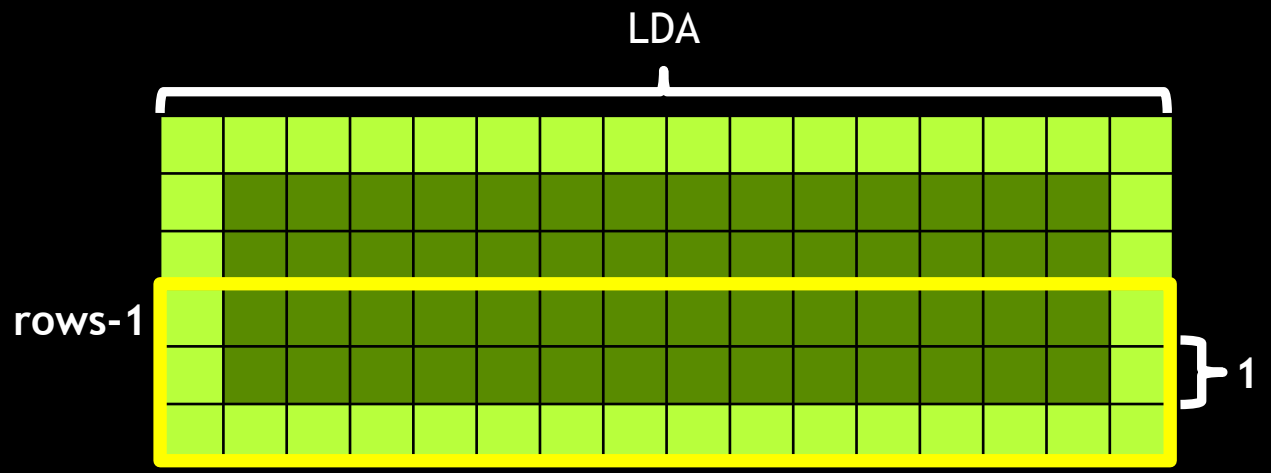
  - Call existing kernel three times
    - Once for lower boundary
    - Once for higher boundary
    - Once for Interior Region

Add Exterior Low Kernel



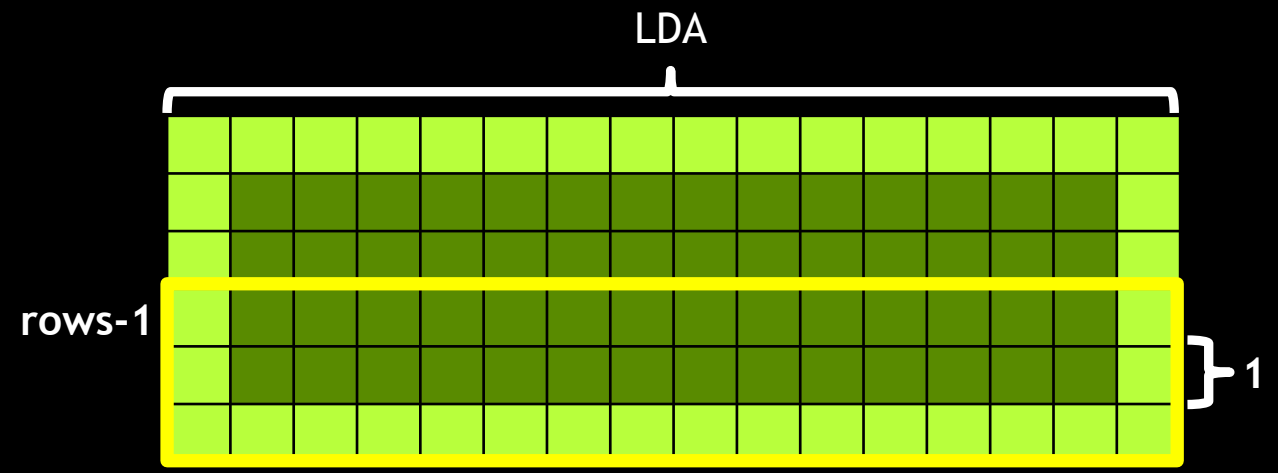

```
//Exterior Low
simpleLaplaceIter_kernel<<<gridSize_e,blockSize_e>>>
    (1,COLS,
     d_Ain[d],
     d_Aout[d]);
```

Add Exterior High Kernel



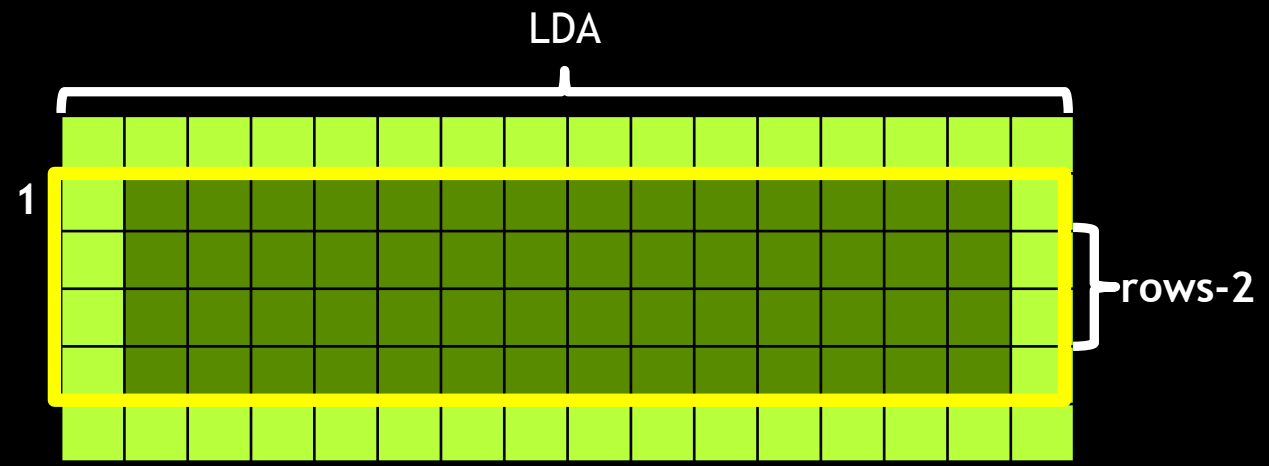

```
//Exterior High
simpleLaplaceIter_kernel<<<gridSize_e,blockSize_e>>>
    (1,COLS,
     d_Ain[d]+IDX(                    ),
     d_Aout[d]+IDX(                    ));
```

Add Exterior High Kernel



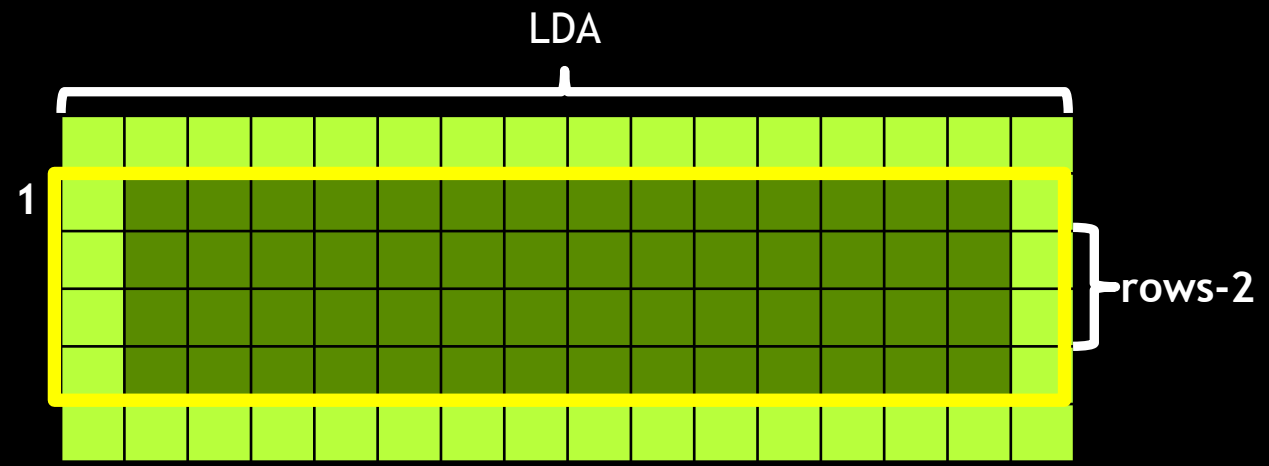

```
//Exterior High
simpleLaplaceIter_kernel<<<gridSize_e,blockSize_e>>>
    (1,COLS,
     d_Ain[d]+IDX(rows-1,0,LDA),
     d_Aout[d]+IDX(rows-1,0,LDA));
```

Add Interior Kernel

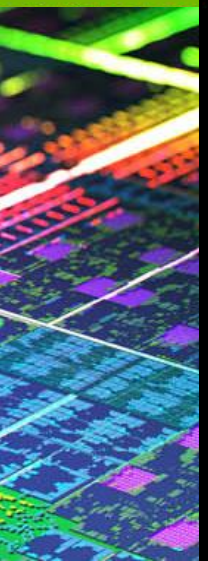



```
//Interior
simpleLaplaceIter_kernel<<<gridSize,blockSize>>>(
      [redacted],
      d_Ain[d]+IDX([redacted]),d_Aout[d]+IDX[redacted]));
```

Add Interior Kernel

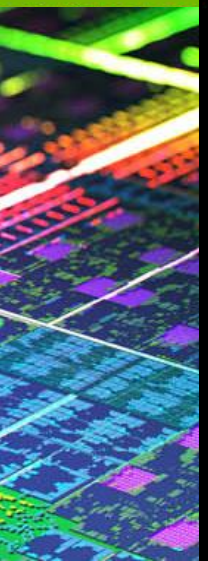



```
//Interior
simpleLaplaceIter_kernel<<<gridSize,blockSize>>>(
     rows-2,COLS,
     d_Ain[d]+IDX(1,0,LDA),d_Aout[d]+IDX(1,0,LDA));
```



Task 3: Results

- Correct results
- About the same speed as step 2

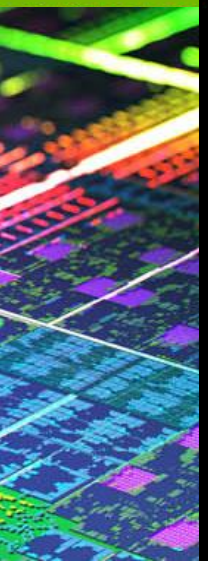


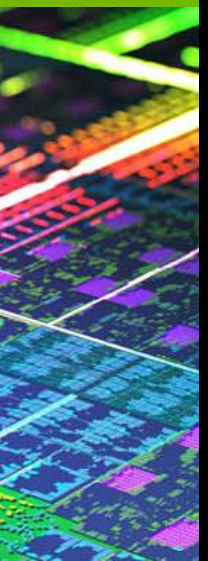
Task 4

- Create 2 streams
  - One for kernels
  - One for transfers
- Place kernels in one stream
- Place transfers the other stream

Create Streams

```
cudaStream_t sTransfer[MAX_DEVICES],
             sKernels[MAX_DEVICES];

...
cudaStreamCreate(&sTransfer[d]);
cudaStreamCreate(&sKernels[d]);
...
```



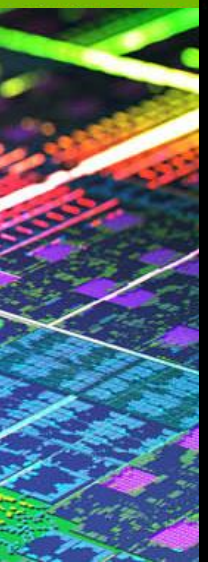


Destroy Streams

```
...
cudaStreamDestroy(sTransfer[d]);
cudaStreamDestroy(sKernels[d]);
...
```

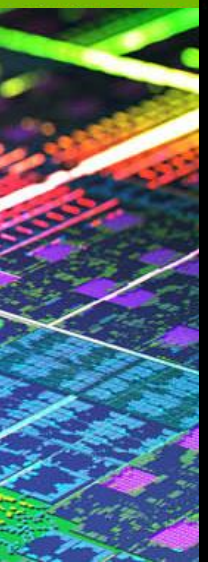
Launch Kernels Into Stream

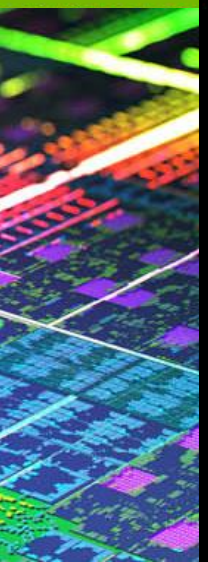
```
//Exterior Low
simpleLaplaceIter_kernel<<<...,                >>>(...);

//Exterior High
simpleLaplaceIter_kernel<<<...,                >>>(...);

//Interior
simpleLaplaceIter_kernel<<<...,                >>>(...);
```



Launch Kernels Into Stream

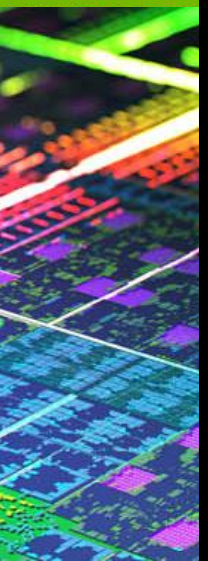
```
//Exterior Low
simpleLaplaceIter_kernel<<<...,0,sKernels[d]>>>(...);

//Exterior High
simpleLaplaceIter_kernel<<<...,0,sKernels[d]>>>(...);

//Interior
simpleLaplaceIter_kernel<<<...,0,sKernels[d]>>>(...);
```





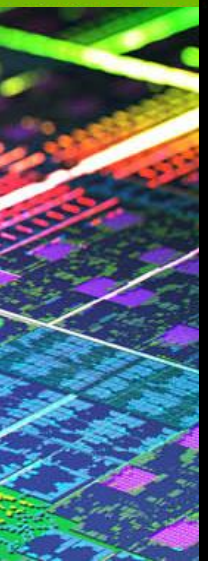
Launch Memory Copies Into Stream

```
for(int d=0;d<numDev;d++) {
  //Grab lower bc
  if(d>0) {
    cudaMemcpyPeerAsync(...,                    );
  }
  //Grab upper bc
  if(d<numDev-1) {
    cudaMemcpyPeerAsync(...,                    );
  }
  cudaCheckError();
}
```



Launch Memory Copies Into Stream

```
for(int d=0;d<numDev;d++) {
  //Grab lower bc
  if(d>0) {
    cudaMemcpyPeerAsync(...,sTransfers[d]);
  }
  //Grab upper bc
  if(d<numDev-1) {
    cudaMemcpyPeerAsync(...,sTransfers[d]);
  }
  cudaCheckError();
}
```



Step 4: Results

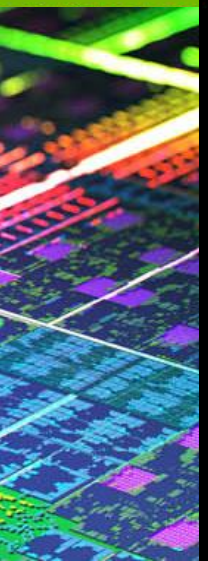
- Slightly faster than before
- Incorrect results

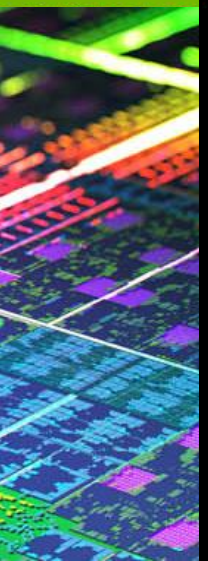
Why?

Race conditions related to memory copies

Task 5

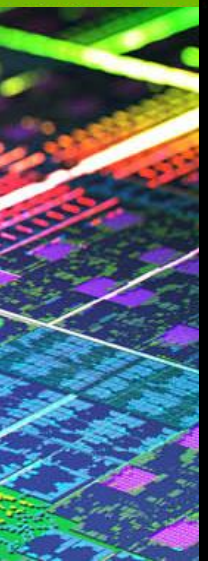
- Add synchronization to fix race conditions
  - Can be done with 1 event
  - 2 race conditions
    - Sending data before exterior region is completed
    - Computing next iteration before communication has completed





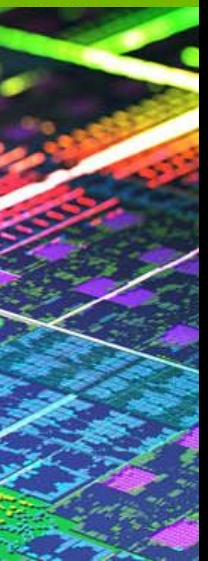
Create and Destroy Event

```
cudaEvent_t event[MAX_DEVICES];

...

cudaEventCreateWithFlags(&event[d],cudaEventDisableTiming);

...

cudaEventDestroy(event[d]);
```



Fix First Race Condition

```
for(int d=0;d<numDev;d++) {
  cudaSetDevice(d);
  //Exterior Low
  simpleLaplaceIter_kernel<<<...,sKernels[d]>>>(...);
  //Exterior High
  simpleLaplaceIter_kernel<<<...,sKernels[d]>>>(...);
  cudaEventRecord(event[d],sKernels[d]);
  //Interior
  simpleLaplaceIter_kernel<<<...,sKernels[d]>>>(...);
  cudaCheckError();
}
for(int d=0;d<numDev;d++) {
  cudaEventSynchronize(event[d]);
}
```



Fix Second Race Condition

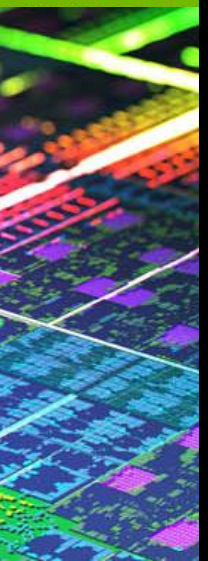
```
//Grab lower bc
      if(d>0) {
        cudaMemcpyPeerAsync(...);
        cudaEventRecord(event[d],sTransfer[d]);
        cudaStreamWaitEvent(sKernels[d],event[d],0);
      }
      //Grab upper bc
      if(d<numDev-1) {
        cudaMemcpyPeerAsync(d_Aout[d]+IDX(rows+1,0,LDA),d,
                            d_Aout[d+1]+IDX(1,0,LDA), d+1,
                            bc_size*sizeof(double),sTransfer[d]);
        cudaEventRecord(event[d],sTransfer[d]);
        cudaStreamWaitEvent(sKernels[d],event[d],0);
      }
```

Step 5: Final Results

- Test machine: Dual Socket Xeon X5675 with 8 M2090s

Size	Speedup	Efficiency
2048x2048	3.78	47%
4096x4096	7.11	96%
8192x8192	8.29	104%

- Efficiency increased
- Perfect scaling at large problem sizes!



Start Using multi-GPU Today

- Use multi-GPU to
  - Compute Faster, Larger, and Cheaper

GPU Test Drive

http://www.nvidia.com/GPUTestDrive

Feel Free to Download IPython notebook