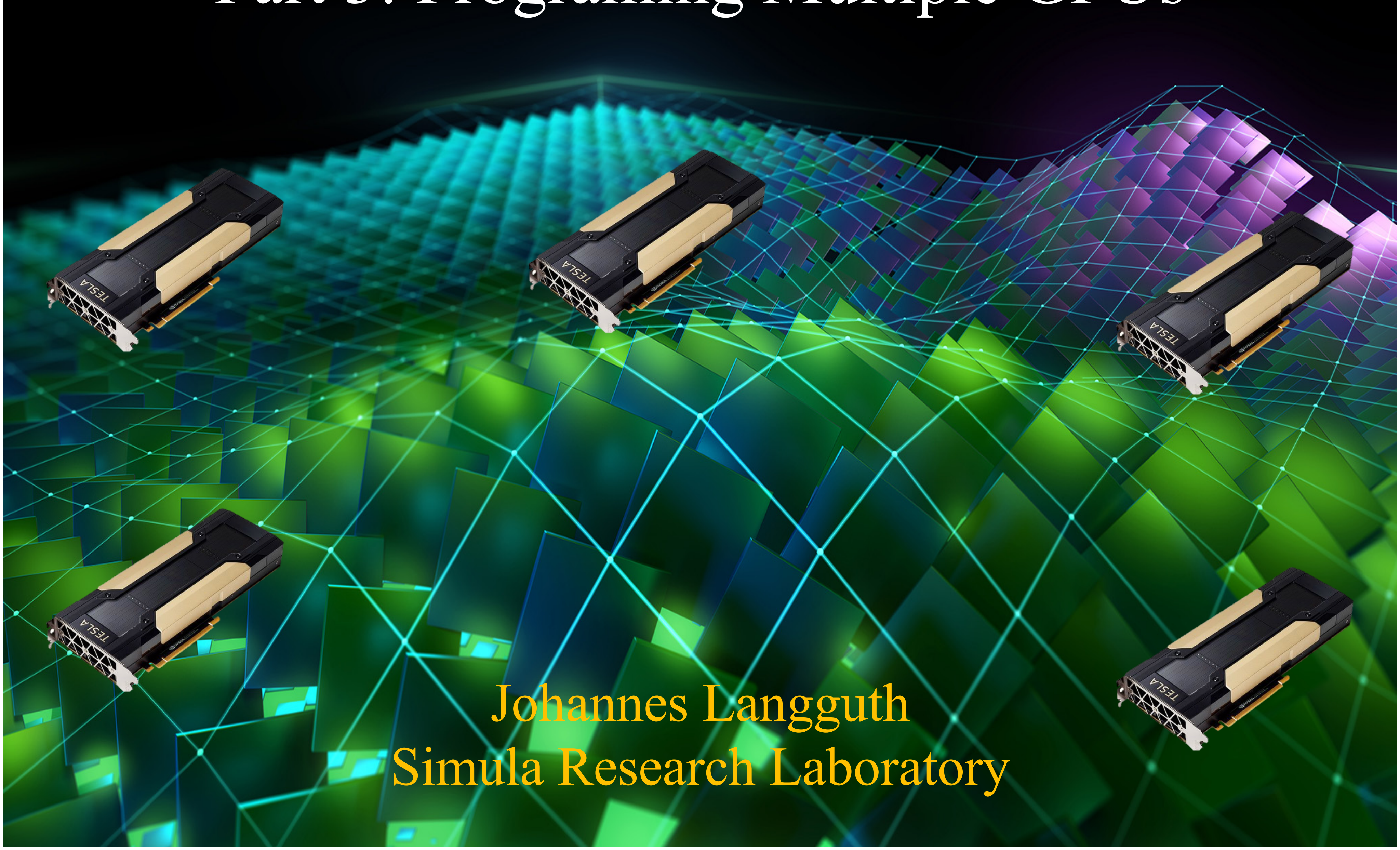
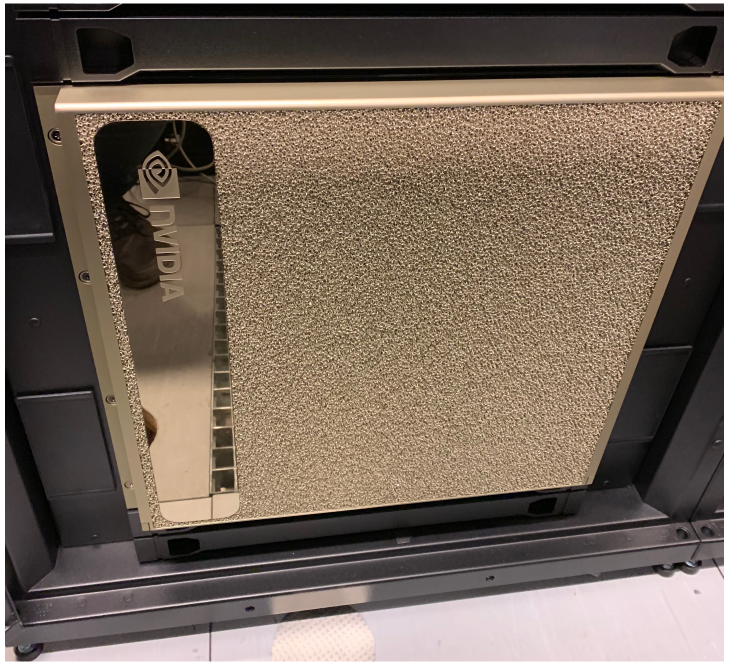
GPU Computing with CUDA (and beyond)
Part 3: Programing Multiple GPUs



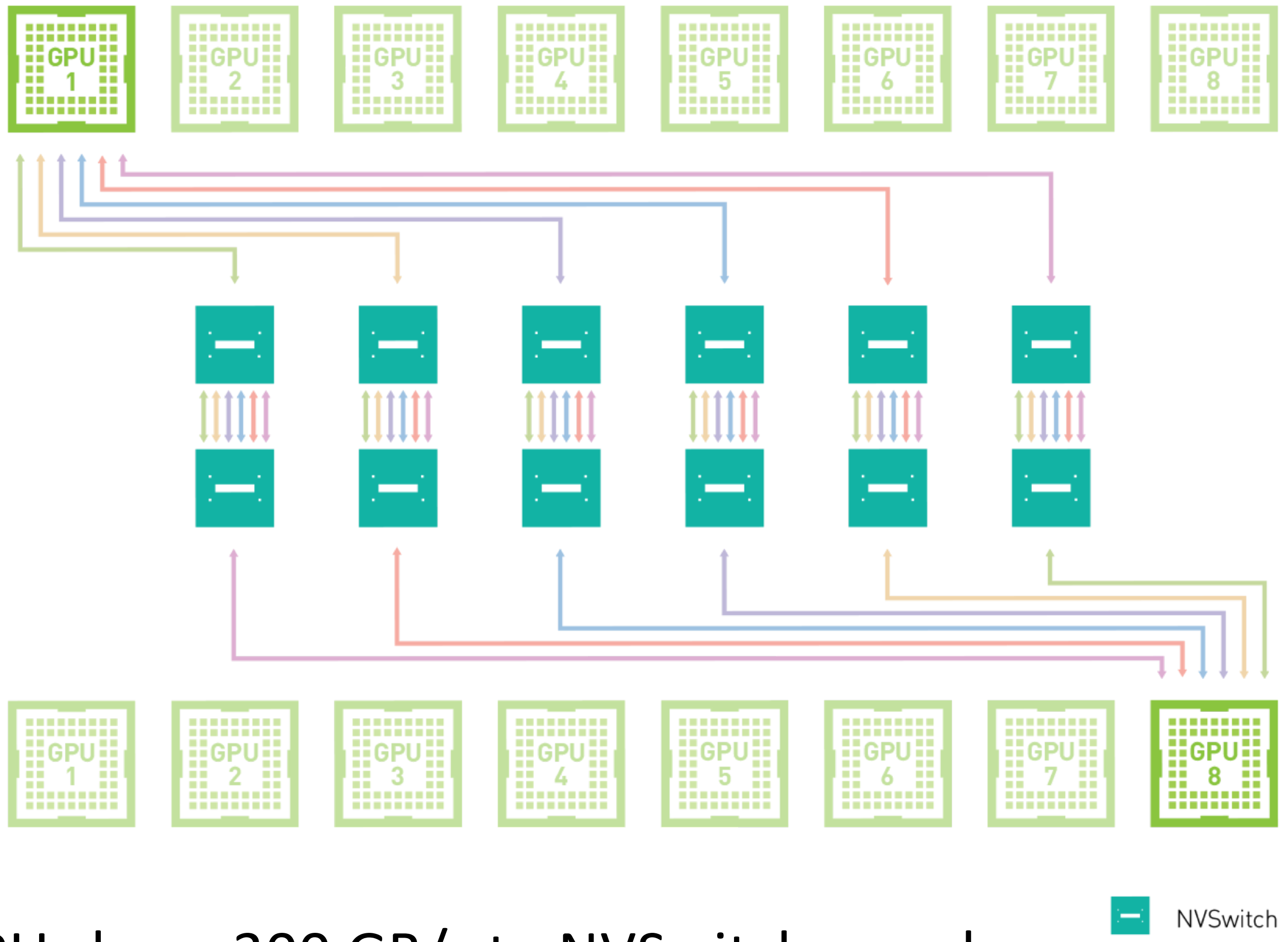
Johannes Langguth
Simula Research Laboratory

DGX–2: Lots of GPUs in one Box



- 16 NVIDIA V100 Volta GPUs
- 16x 32 GB RAM
- 300 GB/s between GPUs
- 2 Intel Skylake Xeon CPUs

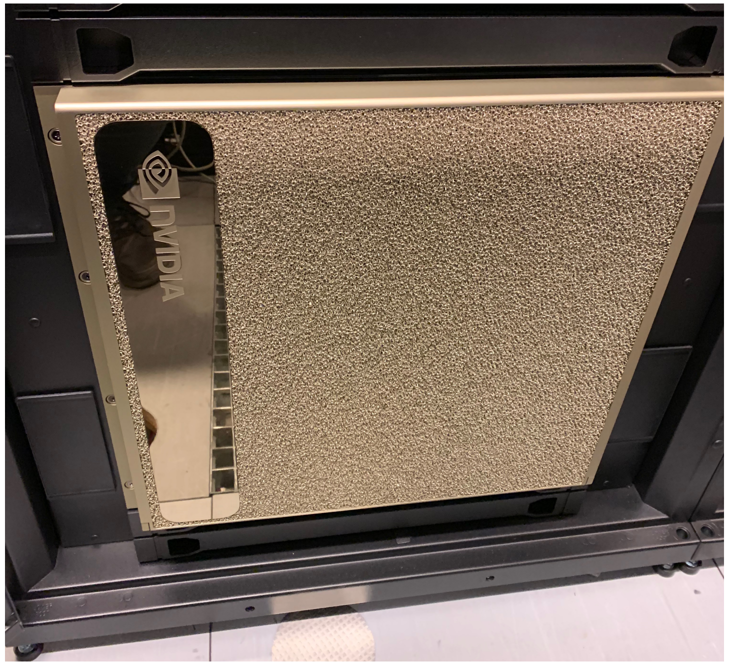
DGX–2: Lots of GPUs in one Box



GPUs have 300 GB/s to NVSwitch crossbar

Bisection bandwidth: 2.4 TB/s

DGX–2: Lots of GPUs in one Box



	FP64 throughput		Memory bandwidth	
Device type	TFLOPS	Ratio of total	GB/s	Ratio of total
CPUs	2.07	0.016	140	0.010
GPUs	130.88	0.984	14192	0.990
Total	132.95	1.000	14332	1.000

A Note on Memory Bandwidth: STREAM

- Attainable memory bandwidth is lower than the max
- A simple benchmark gives a realistic upper bound

```
void tuned_STREAM_Triad(STREAM_TYPE scalar) {
      ssize_t j;
      #pragma omp parallel for
      for (j=0; j<STREAM_ARRAY_SIZE; j++)
            a[j] = b[j]+scalar*c[j];
}
```

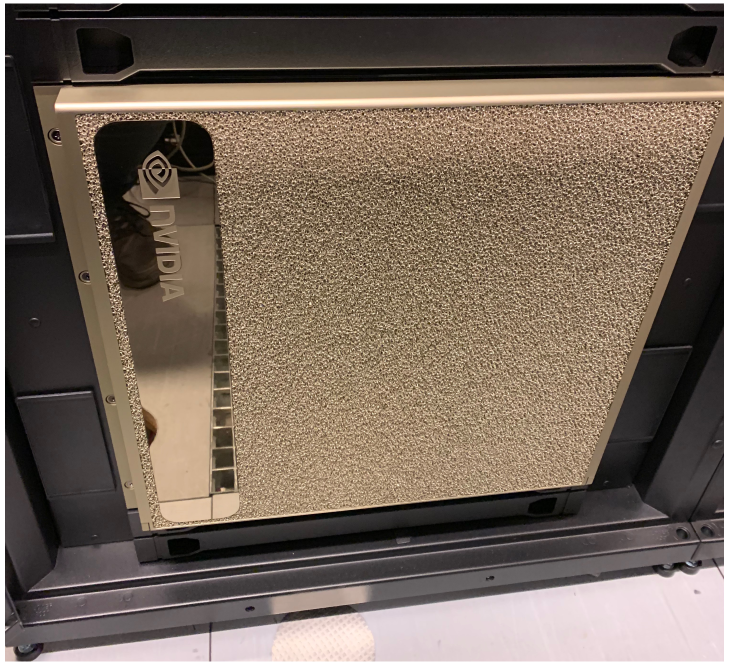
A Note on Memory Bandwidth: STREAM

- Attainable memory bandwidth is lower than the max
- A simple benchmark gives a realistic upper bound

```
void tuned_STREAM_Triad(STREAM_TYPE scalar) {
      ssize_t j;
      #pragma omp parallel for
      for (j=0; j<STREAM_ARRAY_SIZE; j++)
            a[j] = b[j]+scalar*c[j];
}
```

Some numbers:	STREAM	Max
Xeon Platinum 8160	223 GB/s	238 GB/s
Pascal P100	557 GB/s	720 GB/s
Volta V100	855 GB/s	900 GB/s

DGX–2: Lots of GPUs in one Box



- Rather than expanding the role of the CPU, make sure CPU doesn’t become a bottleneck
- 300 GB/s between GPUs is faster than almost all CPUs
- We need to keep data on the GPUs as long as possible

How to program for a DGX–2

Some helpful functions:

```
cudaGetDeviceCount(int *count)
cudaSetDevice(int device)
cudaGetDevice(int *device)
cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
```

We can launch a kernel on each GPU with:

```
int *count;
cudaGetDeviceCount(count);
for(int i=0; i<count; i++) {
      cudaSetDevice(i);
      add<<<n,128>>>(d_a[i], d_b[i], d_c[i]);
}
```

How to program for multiple GPUs

We can launch a kernel on each GPU with:

```
int *count;
cudaGetDeviceCount(count);
for(int i=0; i<count; i++) {
      cudaSetDevice(i);
      add<<<n,128>>>(d_a[i], d_b[i], d_c[i]);
}
```

Launching Kernels is not enough. We also need to communicate with the GPUs.

How to program for multiple GPUs

Remember that we also need to copy data to and from the GPUs:

```
int *count;
cudaGetDeviceCount(count);
for(int i=0; i<count; i++) {
  cudaSetDevice(i);
  cudaMemcpy(d_a[i],a[i], size, H2D);
  cudaMemcpy(d_b[i],b[i], size, H2D);
  add<<<n,128>>>(d_a[i], d_b[i], d_c[i]);
  cudaMemcpy(c[i],d_c[i], size, D2H);
}
```

Now we have a problem….

How to program for multiple GPUs

Maybe we can launch kernels on all GPUs and then collect the results:

```
for(int i=0; i<count; i++) {
  cudaSetDevice(i);
  cudaMemcpy(d_a[i],a[i], size, H2D);
  cudaMemcpy(d_b[i],b[i], size, H2D);
  add<<<n,128>>>(d_a[i], d_b[i], d_c[i]);
}

for(int i=0; i<count; i++) {
  cudaSetDevice(i);
  cudaMemcpy(&c[i],d_c[i], size, D2H);
}
```

How to program for multiple GPUs

Maybe we can launch kernels on all GPUs and then collect the results:

```
for(int i=0; i<count; i++) {
  cudaSetDevice(i);
  cudaMemcpy(d_a[i],a[i], size, H2D);
  cudaMemcpy(d_b[i],b[i], size, H2D);
  add<<<n,128>>>(d_a[i], d_b[i], d_c[i]);
}

for(int i=0; i<count; i++) {
  cudaSetDevice(i);
  cudaMemcpy(&c[i],d_c[i], size, D2H);
}
```

Now the kernels run in parallel, but the Memcpy is still sequential.

Asynchronous Operation in CUDA

We need to take a closer look at things here:

```
cudaMemcpy(d_b[i],b[i], size, H2D);
```

cudaMemcpy is **synchrounous for the host**.
CPU will wait until copy is done.

Asynchronous Operation in CUDA

We need to take a closer look at things here:

```
cudaMemcpy(d_b[i],b[i], size, H2D);
```

cudaMemcpy is **synchrounous for the host**.
CPU will wait until copy is done.

```
add<<<n,128>>>(d_a[i], d_b[i], d_c[i]);
```

Kernel launch is **asynchrounous for the host**. CPU will continue immediately (unless CUDA_LAUNCH_BLOCKING = 1).

Asynchronous Operation in CUDA

```
add<<<n,128>>>(d_a[i], d_b[i], d_c[i]);
```

Kernel launch is **asynchrounous for the host**. CPU will continue immediately.

```
cudaMemcpy(c[i],d_c[i], size, D2H);
```

In our examples, kernel launch is followed by cudaMemcpy, so the CPU waits again.

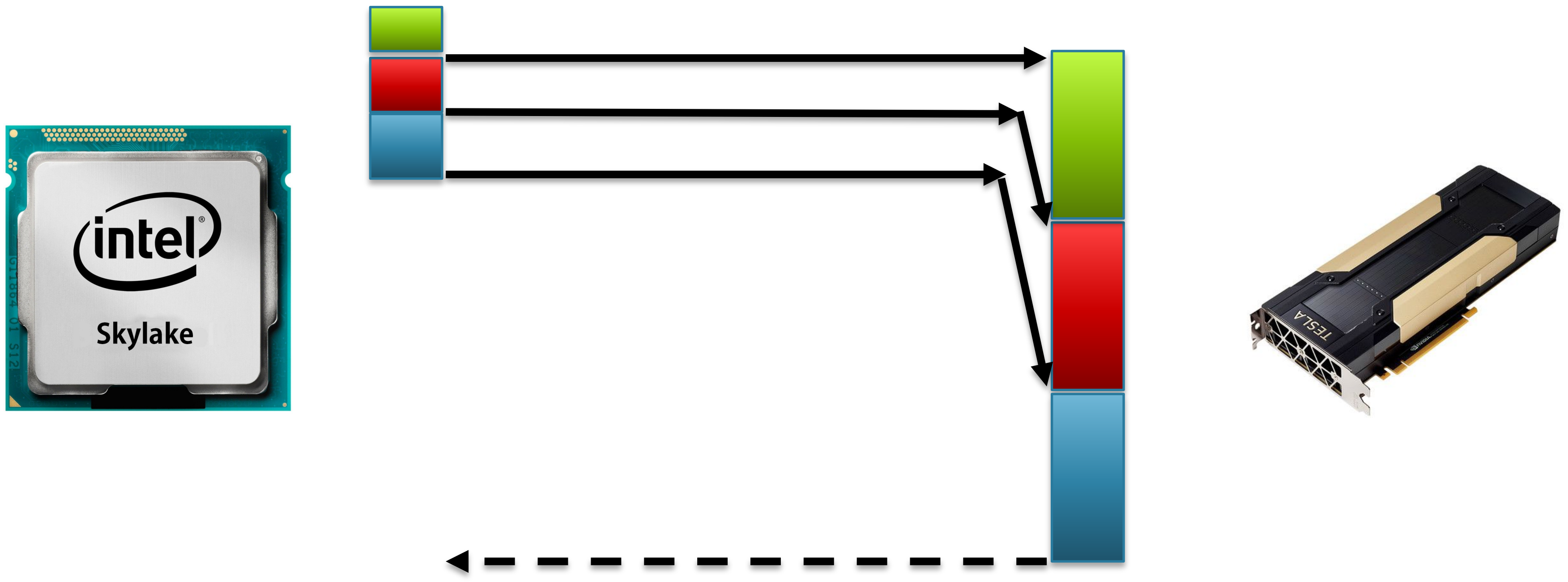
Let’s enable concurrent transfers !

Using cudaMemcpyAsynch

```
for(int i=0; i<count; i++) {
  cudaSetDevice(i);
  cudaMemcpyAsynch(d_a[i],a[i], size, H2D);
  cudaMemcpyAsynch(d_b[i],b[i], size, H2D);
  add<<<n,128>>>(d_a[i], d_b[i], d_c[i]);
  cudaMemcpyAsynch(c[i],d_c[i], size, D2H);
}
```

Does this work ? What happens to the calls on the GPU ?

CUDA Streams



- All calls to the GPU are executed in FIFO queues called **Streams**
- By default, only one default stream, the default stream 0
- Therefore, **`cudaMemcpyAsynch`** happen one after the other on the GPU
- **Problem**: tell CPU that GPU is done

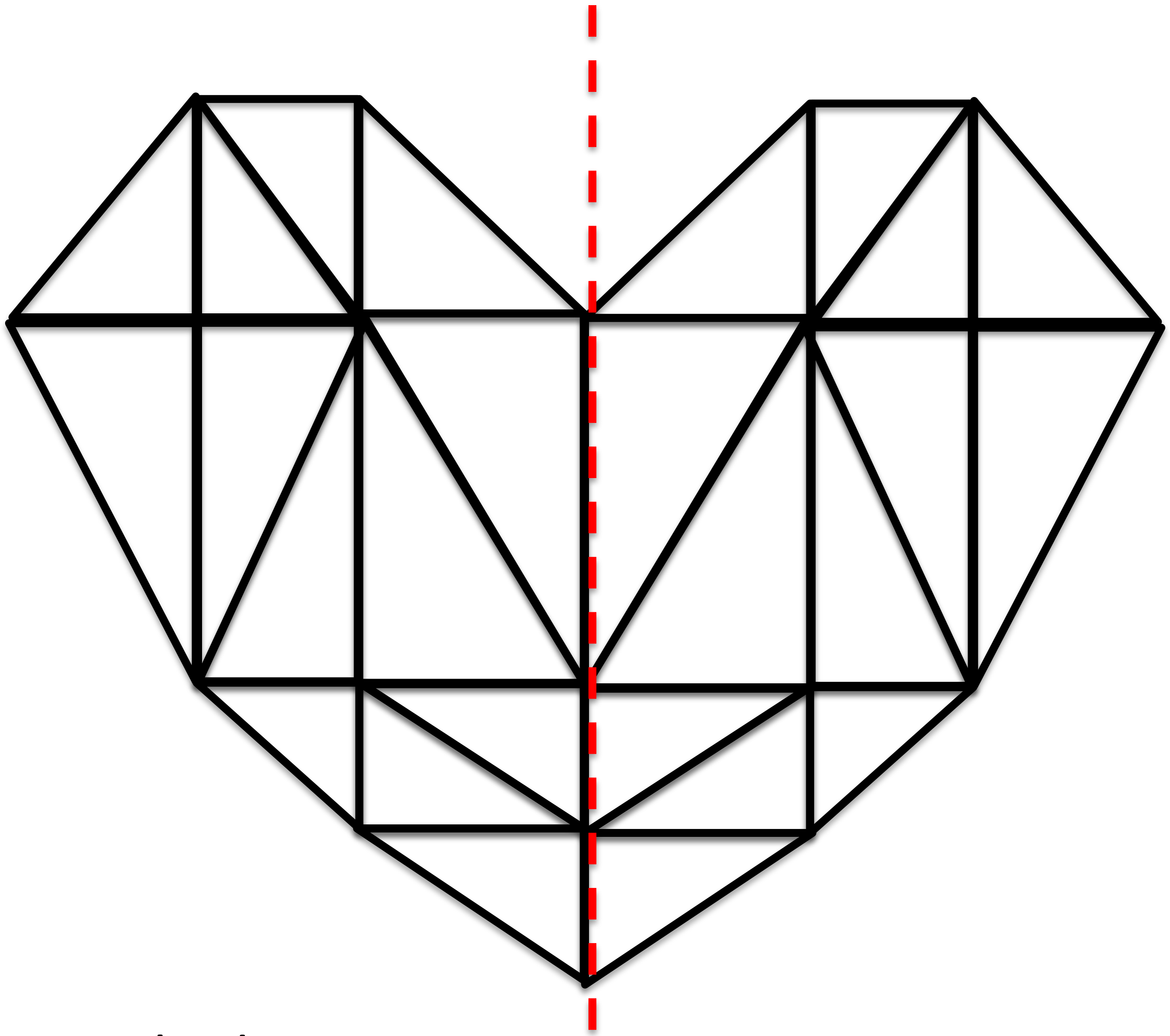
Using cudaMemcpyAsynch

```
for(int i=0; i<count; i++) {
  cudaSetDevice(i);
  cudaMemcpyAsynch(d_a[i],a[i], size, H2D);
  cudaMemcpyAsynch(d_b[i],b[i], size, H2D);
  add<<<n,128>>>(d_a[i], d_b[i], d_c[i]);
  cudaMemcpyAsynch(c[i],d_c[i], size, D2H);
}

for(int i=0; i<count; i++) {
  cudaSetDevice(i);
  cudaStreamSynchronize(0);
}
```

Synchronize each GPU after computation and transfer.
Alternative: `cudaDeviceSynchronize();`

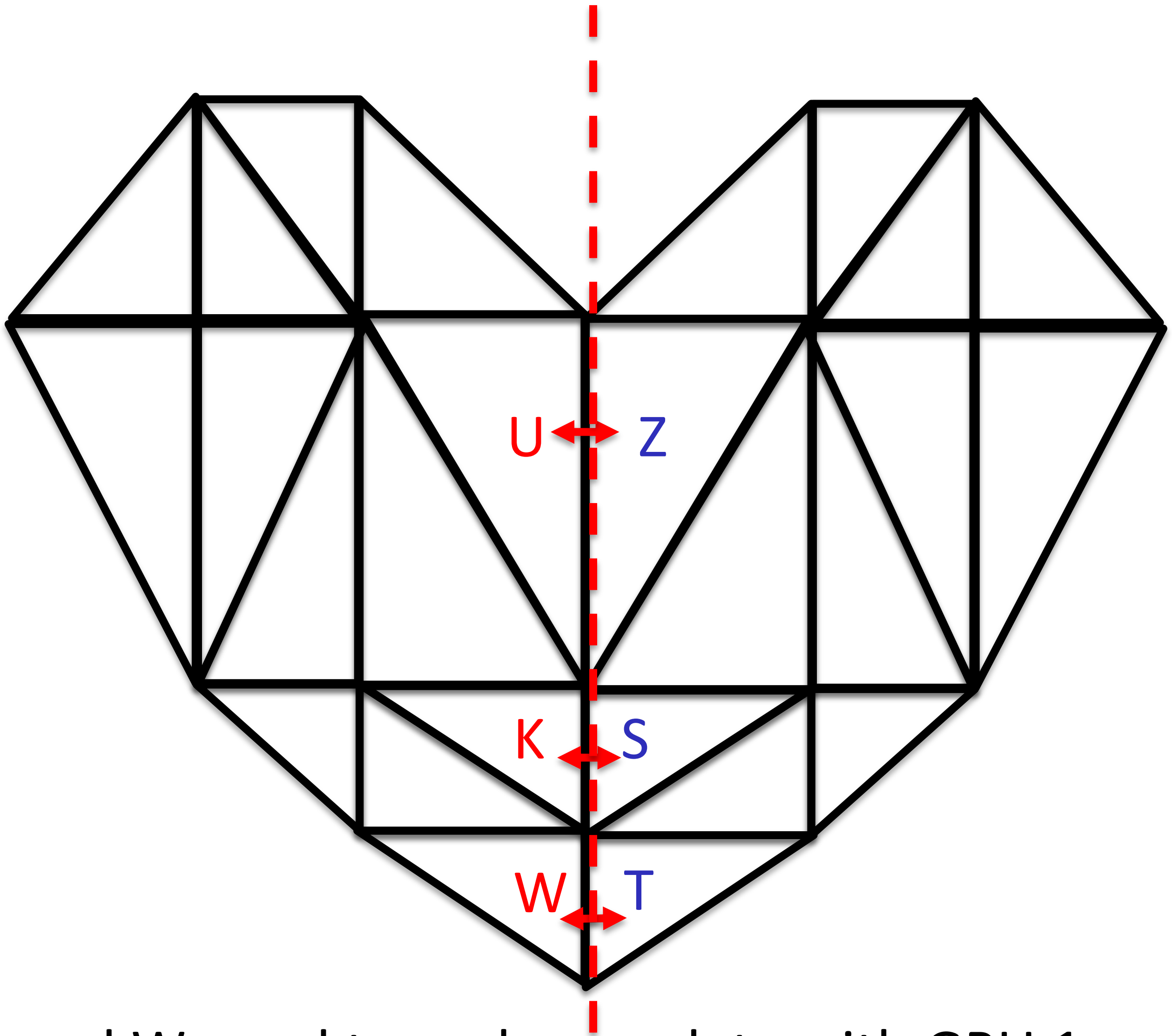
Back to our Application



Split domain into one part per GPU

Data Exchange

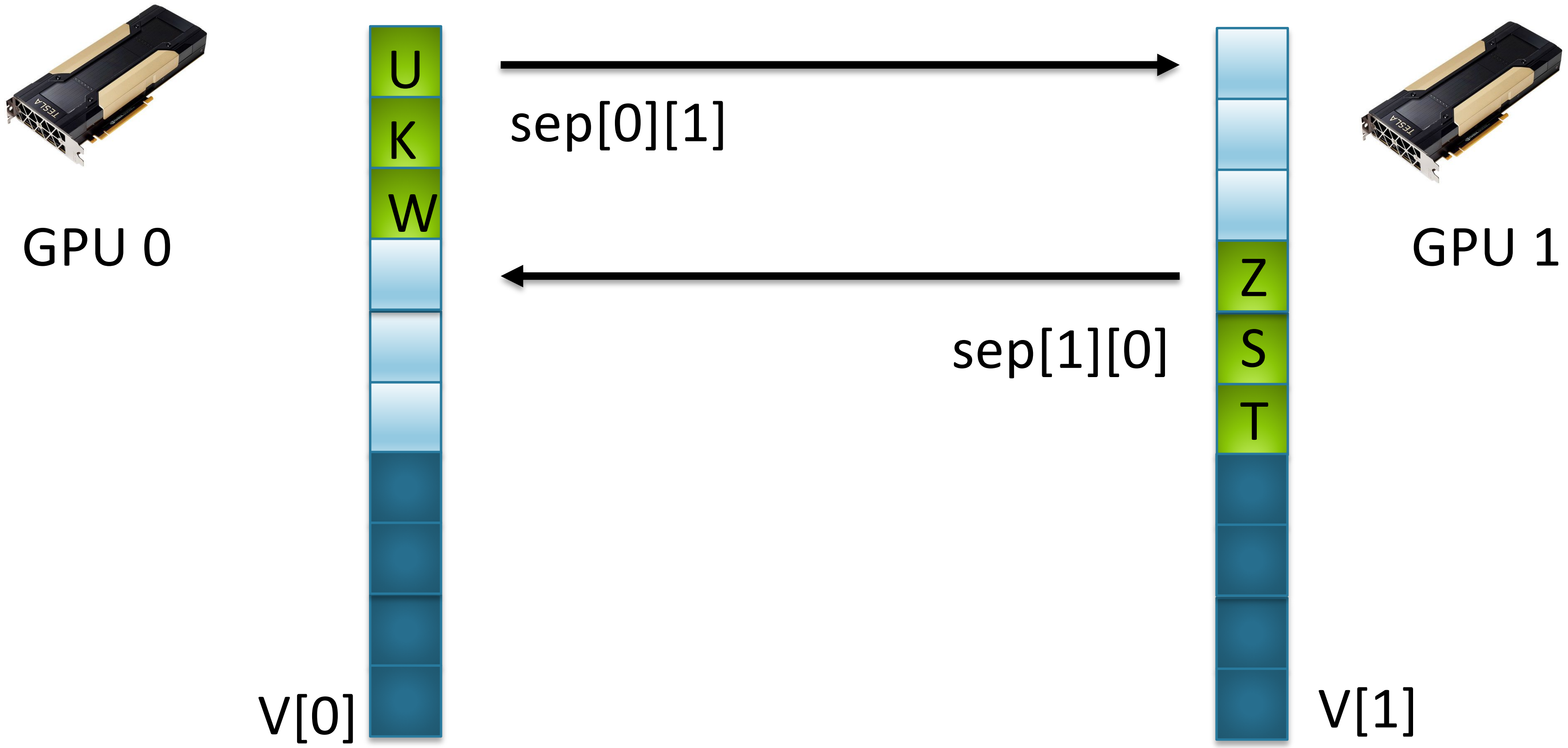
GPU 0



GPU 1

U, K, and W need to exchange data with GPU 1

Data Exchange



Replicated vector means that offsets never change.
We only copy newly computed data to update.
Let sep[0][1] be the offset of the separator between 0 and 1.

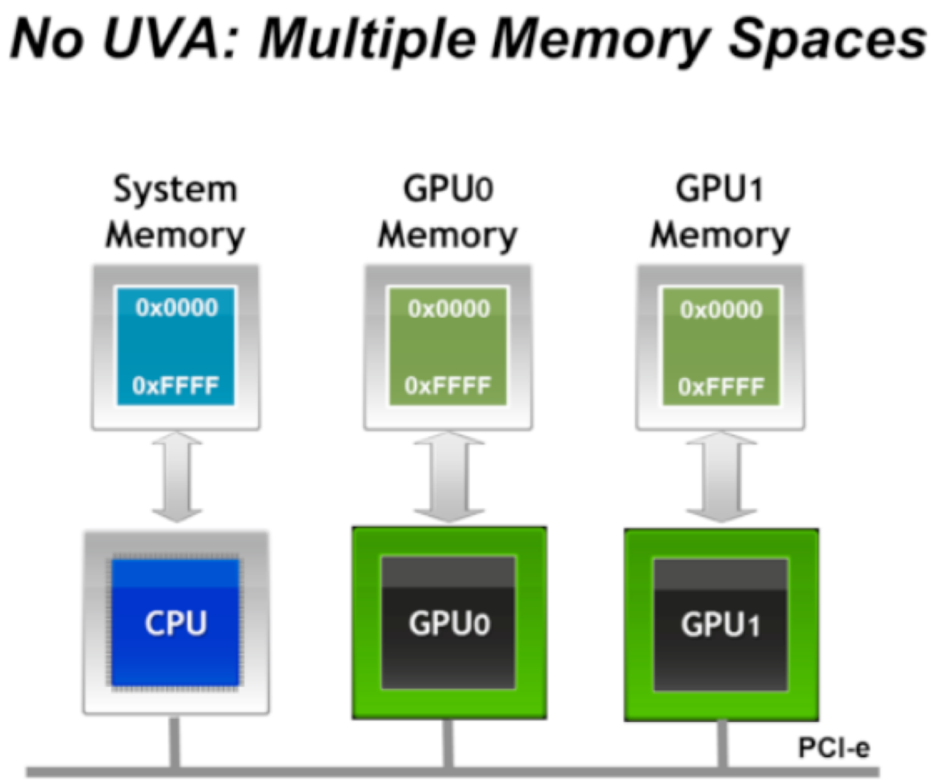
Data Exchange

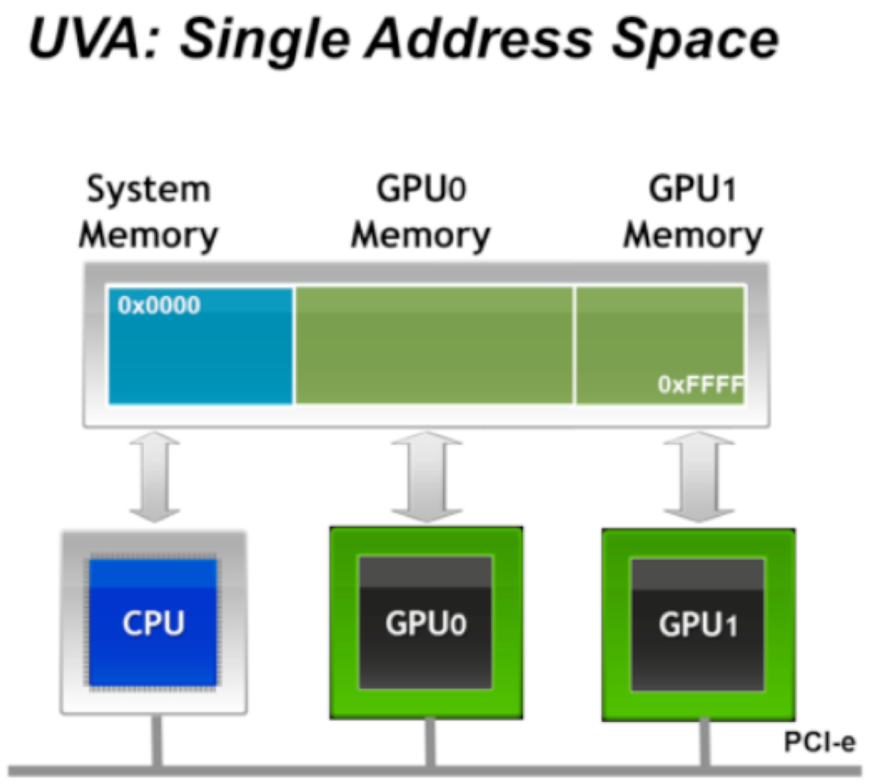
U, K, and W need to exchange data with GPU 1
U, K, and W form a separator (need to reorder in one block)
We can use device to device memcpy

```
for(int i=0; i<count; i++) {
  cudaSetDevice(i);
  computeNewTimeStep<<<n,128>>>(A,I,D,V);
  for(int j=0; j<count; j++)
    if(i != j)
      cudaMemcpyAsynch(V[i][sep[i][j]],V[i][sep[i][j]],
      sepsize, cudaMemcpyDeviceToDevice);
}
```

Target device is found automatically through address of `V[i][sep[i][j]]` in Unified Virtual Adressing

Unified Virtual Adressing





A single adress space among CPU and GPUs.

Data Exchange

Need to make sure transfers are completed:

```
for(int i=0; i<count; i++) {
  cudaSetDevice(i);
  computeNewTimeStep<<<n,128>>>(A,I,D,V);
  for(int j=0; j<count; j++)
    if(i != j)
      cudaMemcpyAsynch(V[i][sep[i][j]],V[i][sep[i][j]],
      sepsize, cudaMemcpyDeviceToDevice);
}

for(int i=0; i<count; i++) {
  cudaSetDevice(i);
  cudaStreamSynchronize(0);
}
```

Improved Data Exchange

Data exchange works, but it has several drawbacks:

- Computation has to wait
- Communication may be very unbalanced
- We have to wait for the slowest
- The more GPUs, the higher the overhead

How can we overlap communication and computation ?

With more streams!

Communication / Computation Overlap

```
for(int i=0; i<count; i++) {
  cudaSetDevice(i);
  cudaStream_t stream1, stream2;
  cudaStreamCreate ( &stream1) ;
  cudaStreamCreate ( &stream2) ;

  computeNewTimeStepSEP<<<k,128,0,stream1>>>(A,I,D,V);
  computeNewTimeStepMAIN<<<n,128,0,stream2>>>(A,I,D,V);
  for(int j=0; j<count; j++)
    if(i != j)
      cudaMemcpyAsynch(V[i][sep[i][j]],V[i][sep[i][j]],
      sepsize, cudaMemcpyDeviceToDevice,stream1);
}
```

This should put the main computation and memcopy in different streams, but….

Communication / Computation Overlap

```
for(int i=0; i<count; i++) {
  cudaSetDevice(i);
  cudaStream_t stream1, stream2;
  cudaStreamCreate ( &stream1) ;
  cudaStreamCreate ( &stream2) ;
  computeNewTimeStepSEP<<<n,128,0,stream1>>>(A,I,D,V);
  computeNewTimeStepMAIN<<<n,128,0,stream2>>>(A,I,D,V);
  for(int j=0; j<count; j++)
    if(i != j)
      cudaMemcpyAsynch(V[i][sep[i][j]],V[i][sep[i][j]],
      sepsize, cudaMemcpyDeviceToDevice, stream1);
}
```

Each CUDA context needs its own streams. Easy to fix but messy to write. Also, each Memcpy can use its own stream.

OpenMP for Multi-GPU Control

- OpenMP is a C/C++/Fortran language extension.
- OpenMP is primarily meant to make multicore programing.
- Using OpenMP to create has multiple advantages
  1. Code becomes easier to read
  2. CPU reacts faster to GPU events
  3. Modern CPUs have little memory bandwidth per core

OpenMP for Multi-GPU Control

- OpenMP forks the master thread to create **parallel regions**.
- An OpenMP thread is an OS thread, not a CUDA thread.
- We should have at least one core per GPU (physical/virtual).
- We can declare private variables inside a parallel region.
- Every OpenMP thread will have a copy of the variable.

OpenMP®
Enabling HPC since 1997

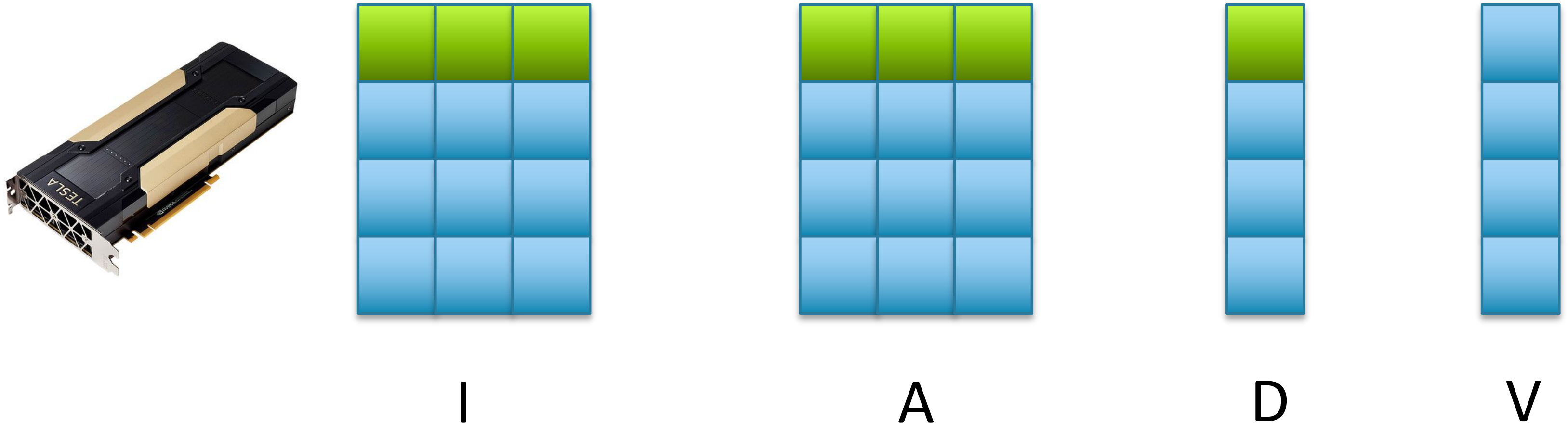
OpenMP for Multi-GPU Control

```
omp_set_num_threads(count);
#pragma omp parallel
{
  int i = omp_get_thread_num();
  cudaSetDevice(i);
  double *V;
  cudaMalloc((void **)&V, n*sizeof(double));
  cudaStream_t stream1, stream2;
  cudaStreamCreate ( &stream1) ;
  cudaStreamCreate ( &stream2) ;
  computeNewTimeStepSEP<<<n,128,0,stream1>>>(A,I,D,V);
  computeNewTimeStepMAIN<<<n,128,0,stream2>>>(A,I,D,V);
  for(int j=0; j<count; j++)
    if(i != j)
      cudaMemcpyAsynch(V[sep[i][j]],V[sep[i][j]],
      sepsize, cudaMemcpyDeviceToDevice, stream1);
  cudaDeviceSynchronize();
}
```

One more Problem: Warps

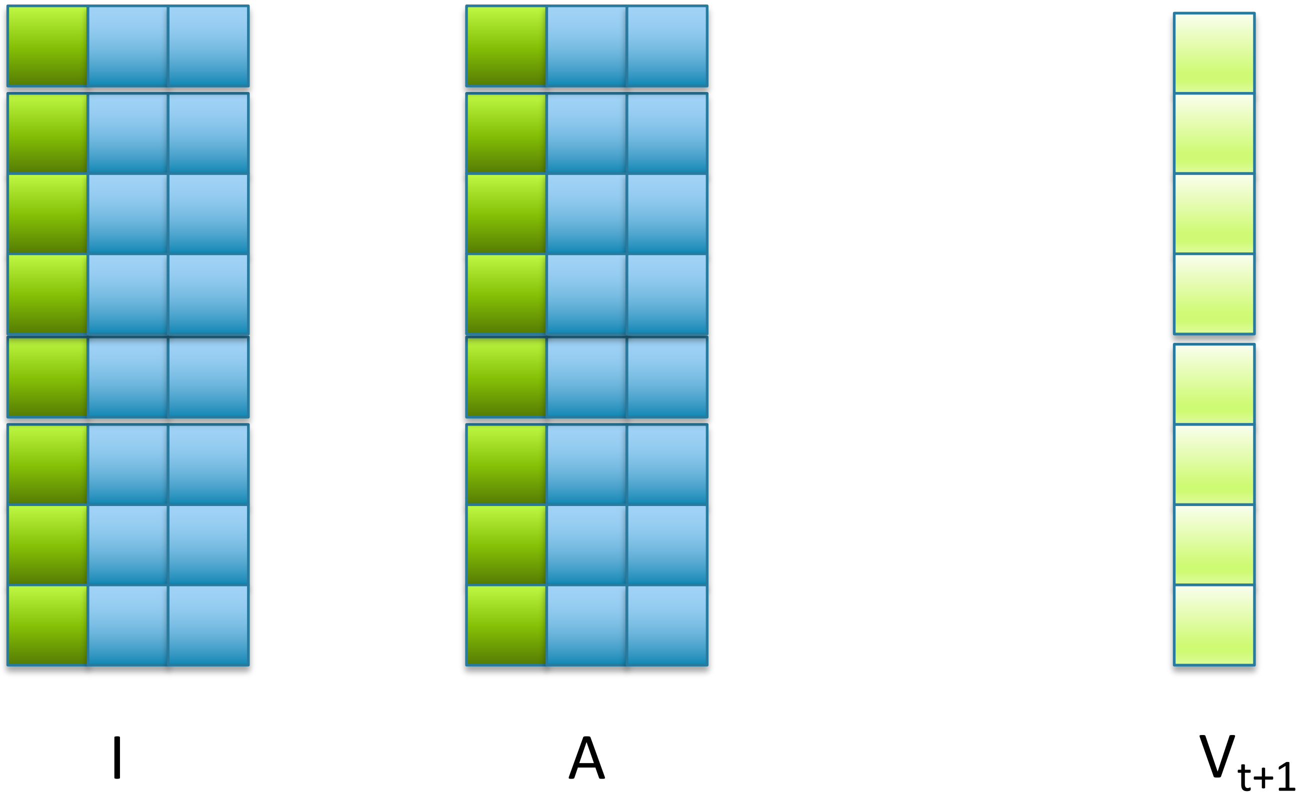
$$V[9]_{t+1} = A[9,0] * V[I[9,0]]_t + A[9,1] * V[I[9,1]]_t + A[9,2] * V[I[9,2]]_t + D[9] * V[9]_t$$

Rowwise computation is possible, but not coalesced.
Columnwise also allows FMA



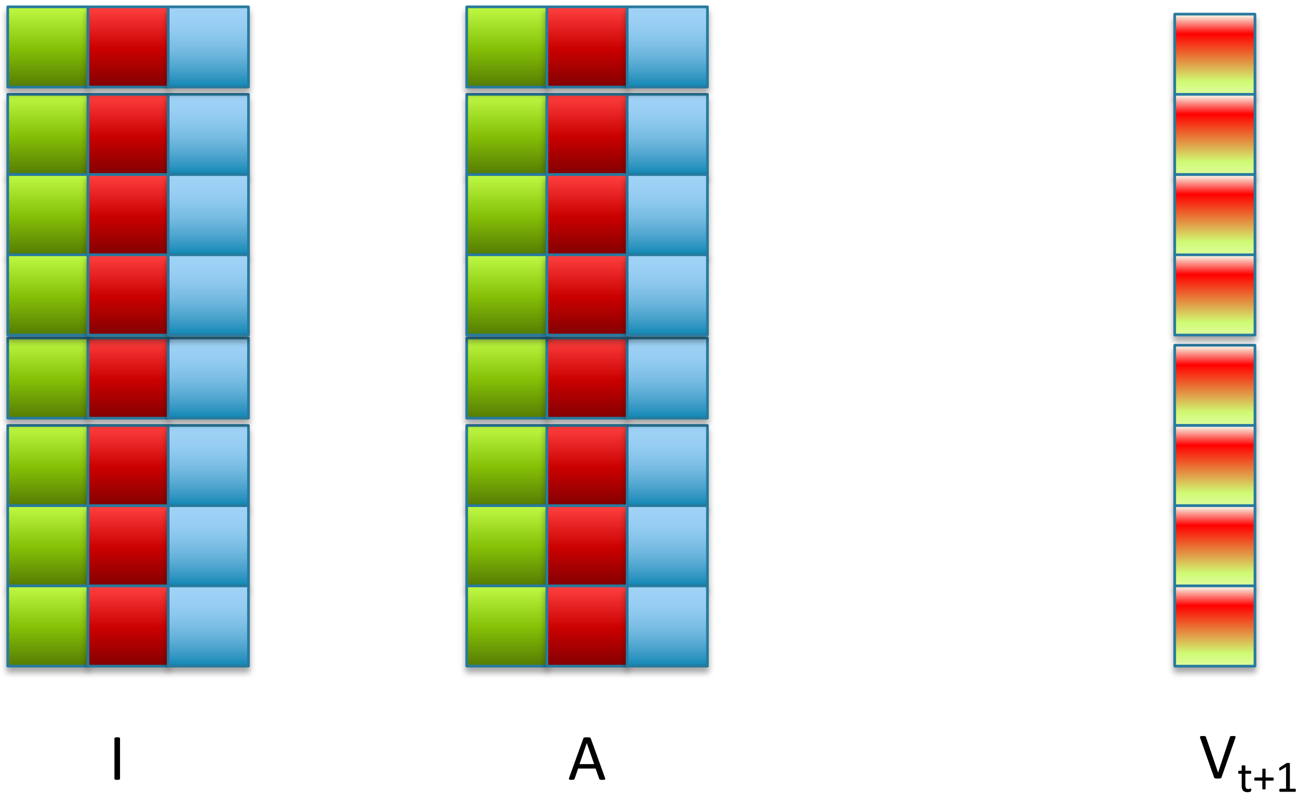
One more Problem: Warps

Compute $V[k]_{t+1} = V[k]_{t+1} + A[k,0] * V[I[k,0]]_t$ for 32 rows at once. Then move to column 1.



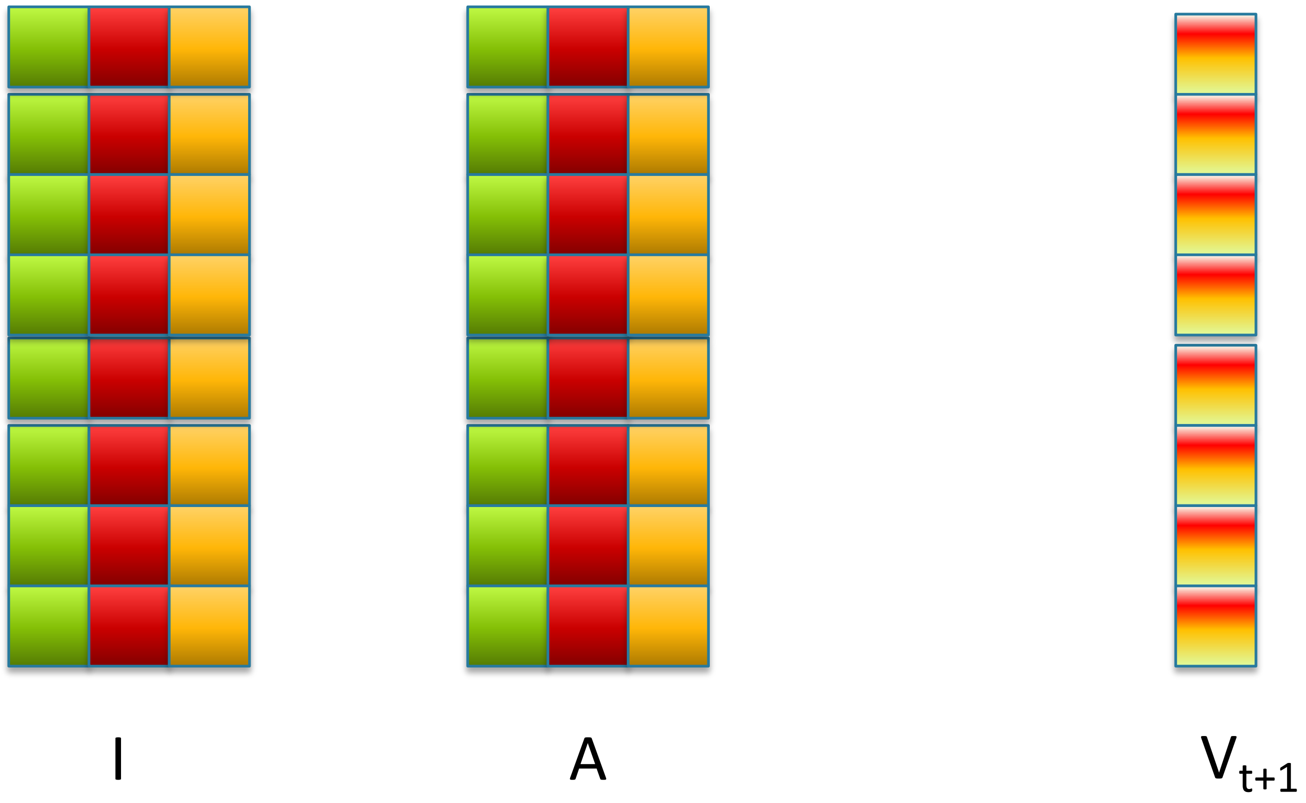
One more Problem: Warps

Compute $V[k]_{t+1} = V[k]_{t+1} + A[k,1] * V[I[k,1]]_t$ for 32 rows at once. Then move to column 2.



One more Problem: Warps

Compute $V[k]_{t+1} = V[k]_{t+1} + A[k,0] * V[I[k,0]]_t$ for 32 rows at once. Then move to column 1.



LYNX Performance Analysis on DGX-2

	P100	V100
T_R (achieved time)	7.60 ms	4.20 ms
$T_R^{\text{memory bound}}$	5.64 ms	3.55 ms
$T_{R\,\text{unoptimised}}^{\text{compute bound}}$	7.45 ms	4.34 ms
$T_{R\,\text{optimised}}^{\text{compute bound}}$	5.22 ms	3.03 ms
$T_R^{\text{memory bound}} / T_R$	0.742	0.845
$T_{R\,\text{optimised}}^{\text{compute bound}} / T_R$	0.686	0.721

- Time steps with 11M tetrahedra
- PDE computation ELLPACK with 17 nonzeroes/row
- ODE computation with 19 variables per cell
- Communication via OpenMP and cudaMemcpy

LYNX Scaling Analysis on DGX-2

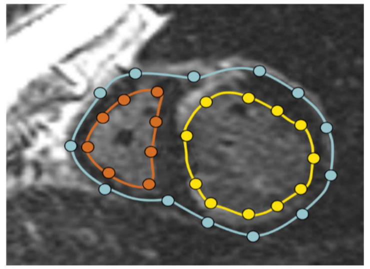
GPUs	Time (s)	Speedup	Scaling efficiency	$\frac{\text{Rel. time}}{\text{GPUs}}$	$\frac{T^{\text{achieved}}}{T^{\text{memory bound}}}$
1	400.098	1.000	1.000	1.000	1.156
2	208.500	1.919	0.959	1.042	1.205
4	105.570	3.790	0.947	1.055	1.220
8	53.800	7.437	0.930	1.076	1.244
16	28.160	14.208	0.888	1.126	1.302

- Strong scaling good but not perfect
- Major impediment: not enough work per GPU

Virtual heart Arrhythmia Risk Predictor (VARP)

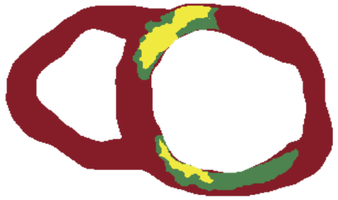
Virtual Heart Model Creation

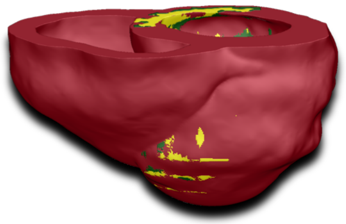
LGE MRI



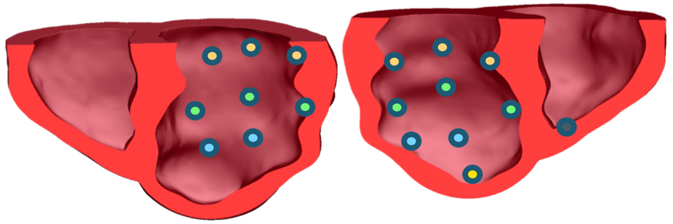
Ventricular Segmentation

Infarct Segmentation





In-Silico Stimulation to induce Arrythmia



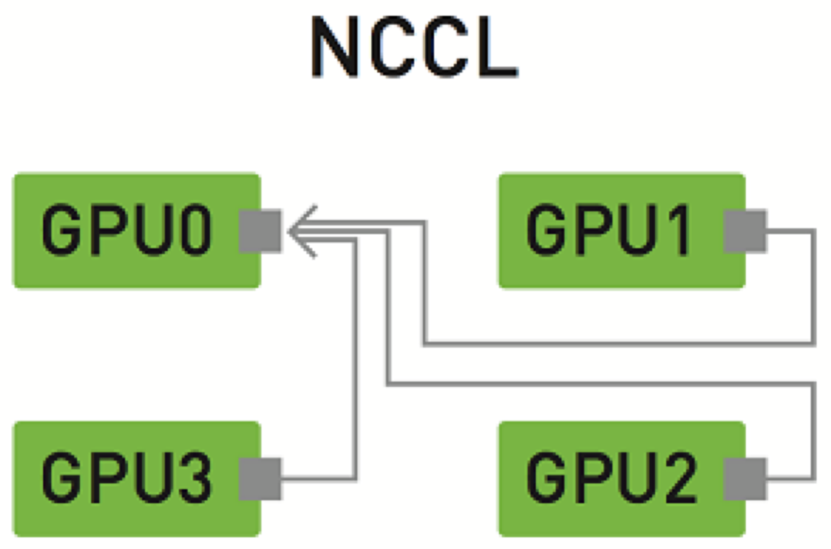
Ultimate Goal: real-time simulation

Communication Loop: can we simplify this ?

```
for(int i=0; i<count; i++) {
  for(int j=0; j<count; j++)
    if(i != j)
      cudaMemcpyAsynch(sep[j][i],&sep[i][j],
      sepsize[i][j], cudaMemcpyDeviceToDevice,stream1);
}
```

- Each GPU sends and receives data from all other GPUs
- Such an All-to-All communication pattern is common
- Can we get a library implementation for this ?

NCCL – The easy way out ?



AllReduce
Broadcast
Reduce
AllGather
ReduceScatter

- NVIDIA NCCL implements collective communication
- Mostly aimed at deep learning
- Has been around for years, but All-to-All is still missing

References

Langguth, J., Sourouri, M., Lines, G. T., Baden, S. B., & Cai, X. (2015). Scalable heterogeneous CPU-GPU computations for unstructured tetrahedral meshes. *IEEE Micro*, *35*(4), 6-15.

Arevalo, H. J., Vadakkumpadan, F., Guallar, E., Jebb, A., Malamas, P., Wu, K. C., & Trayanova, N. A. (2016). Arrhythmia risk stratification of patients after myocardial infarction using personalized heart models. *Nature communications*, *7*(1), 1-8.

Credit: Lecture contains NVIDIA material available at https://developer.nvidia.com/cuda-zone
Image source: wikipedia.org, https://www.openmp.org/