GPU Computing with CUDA (and beyond)
Part 2: Heterogeneous Computations

Johannes Langguth
Simula Research Laboratory

Heterogeneous Computing

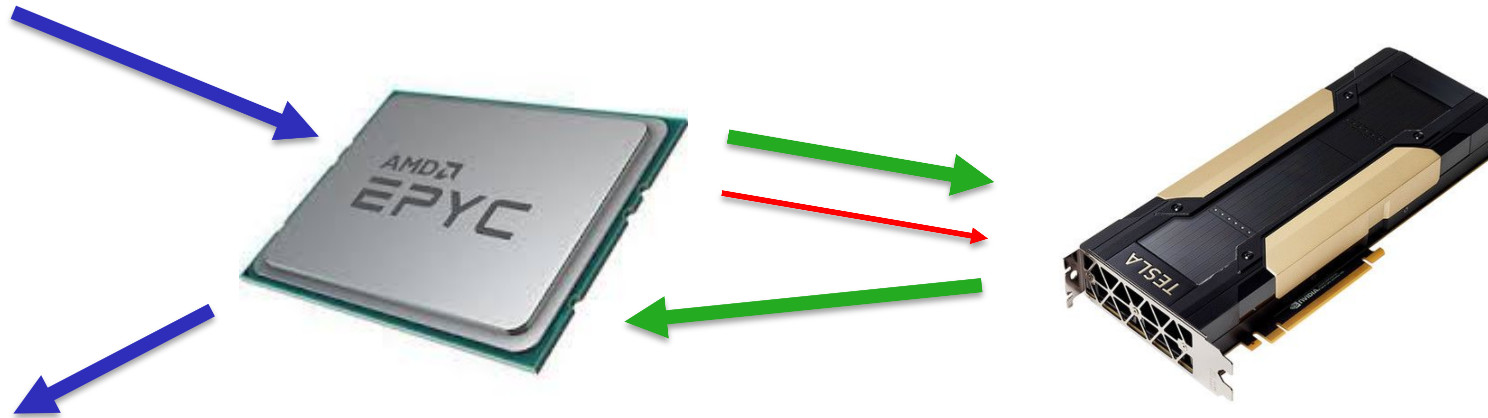
	AMD EPYC 7742	NVIDIA Volta V100
Cores	64	80
Base frequency (Ghz)	2.25	1.6
TFLOPS	2.3	7.5
Memory Bandwidth (GB/s)	204.8	900
Cache (last level in MB)	256	6
Memory	lots	32

Heterogeneous Computing

- Despite rumors to the contrary, GPUs do not provide 100x speedups
- Recently, CPUs have become much more powerful
- We may want to use CPUs and GPUs together

- There are 3 typical cases
  1) CPU only acts as a helper, GPU computes
  2) CPU and GPU perform different parts of the computation
  3) CPU and GPU share work

Case 1: CPU Host as a Helper for GPU



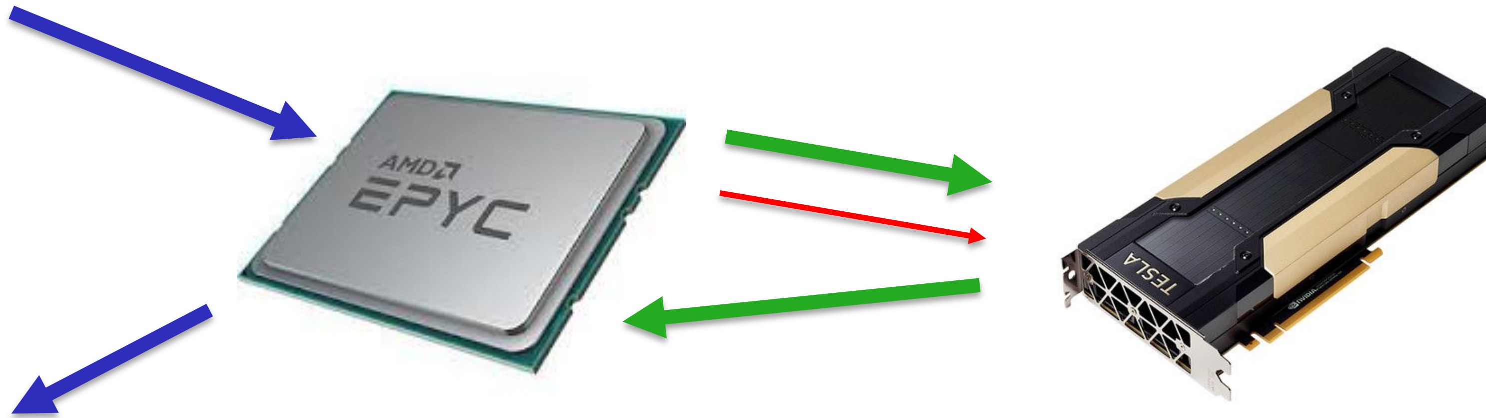
CPU / Host

- I/O
- Memory allocation
- Data preprocessing
- Data transfers
- Kernel launch
- Expand GPU memory

GPU / Device

Computation

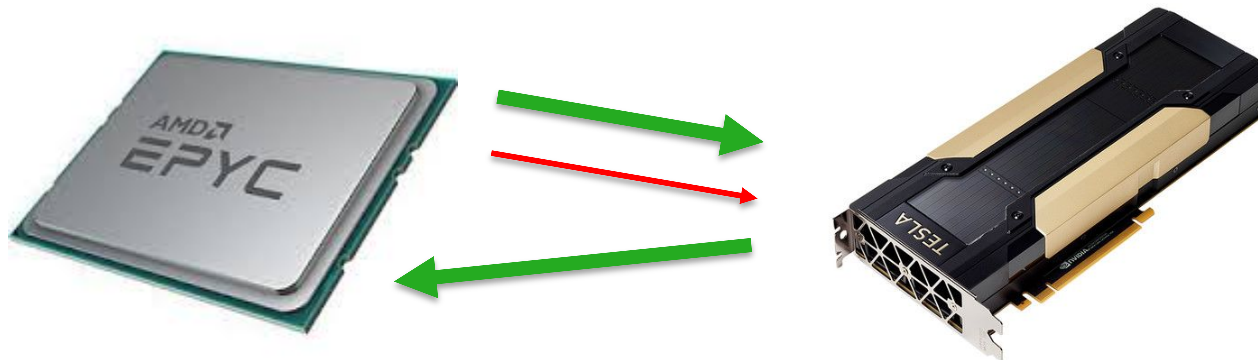
Case 1: CPU Host as a helper for GPU



Advantages
- Simple implementation
- Simple execution
- Works with any CPU (and Python)

Disadvantages
- Powerful CPU is idle
- Some tasks don't fit the GPU
- Streaming from CPU memory may be pointless

Streaming from CPU memory may be pointless




```
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
add<<<ceil(n/128),128>>>(d_a, d_b, d_c);
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
```

```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

Remember the add Kernel (now with doubles)

```
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);
add<<<ceil(n/128),128>>>(d_a, d_b, d_c);
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
```

```
__global__ void add(double *a, double *b, double *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

We need to move 2 values to and one value from the GPU

CPU-GPU Communication: 24 **byte per element (and FLOP)**

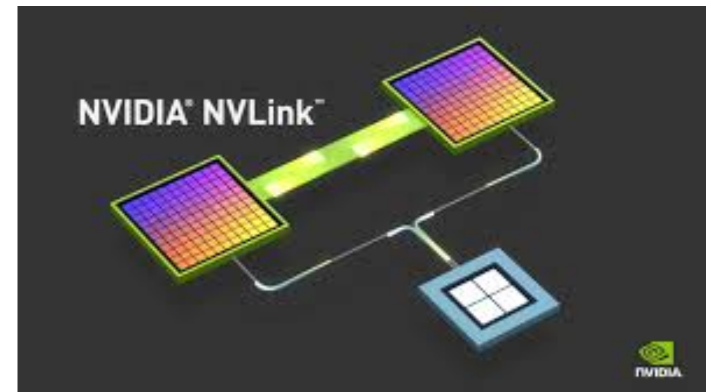
We need to amortize transfer cost

Fundamental Problem of offloading

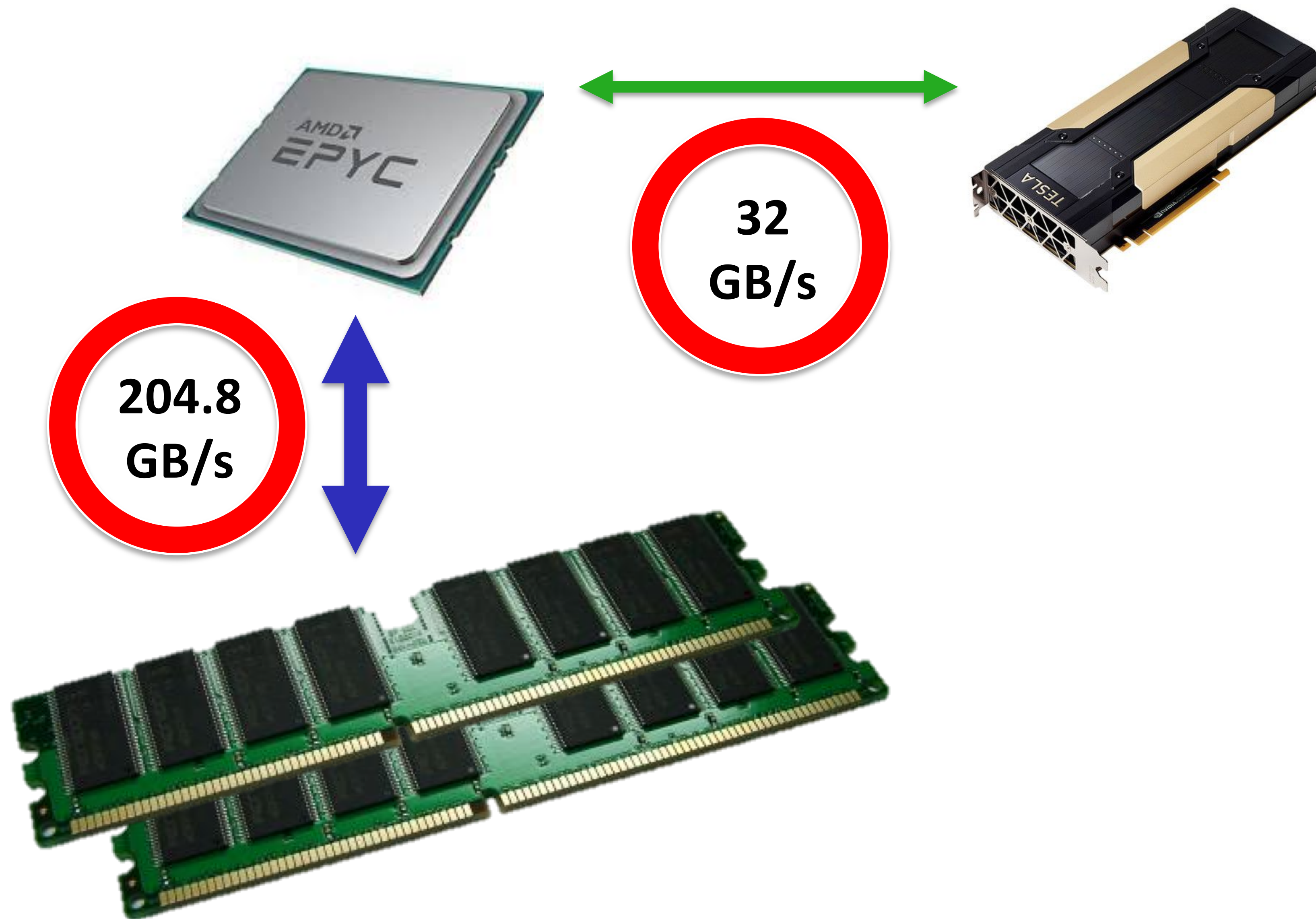
Data has to be reused to benefit from offloading

Offloading computations are limited by:

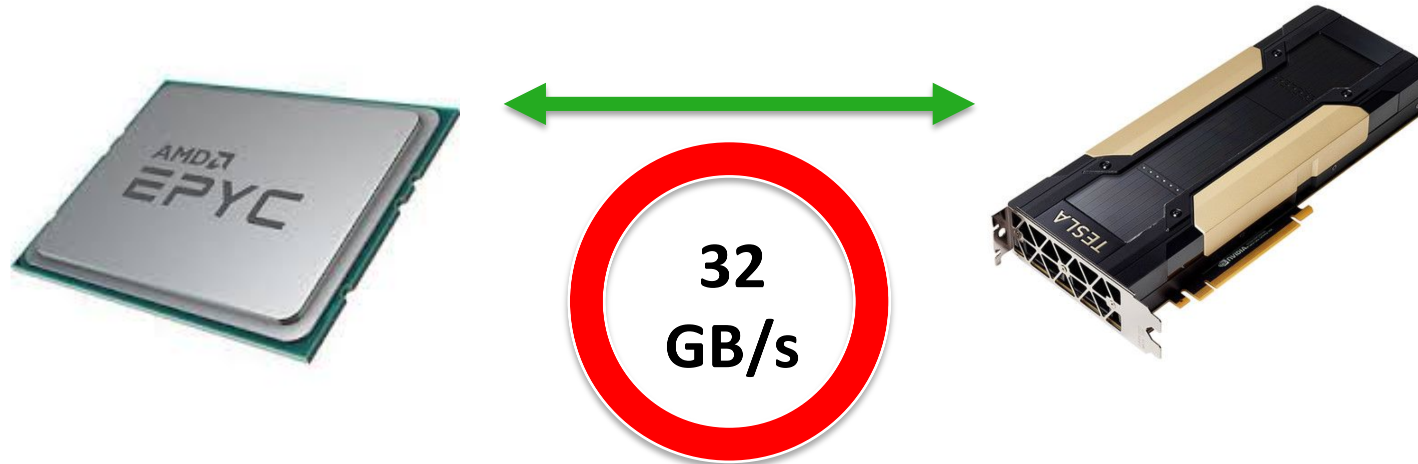
- CPU memory bandwidth
- Transfer latency
- Synchronization and launch overheads
- Host to Device bandwidth (32 – 300 GB/s)



CPU memory and CPU-GPU Bottlenecks



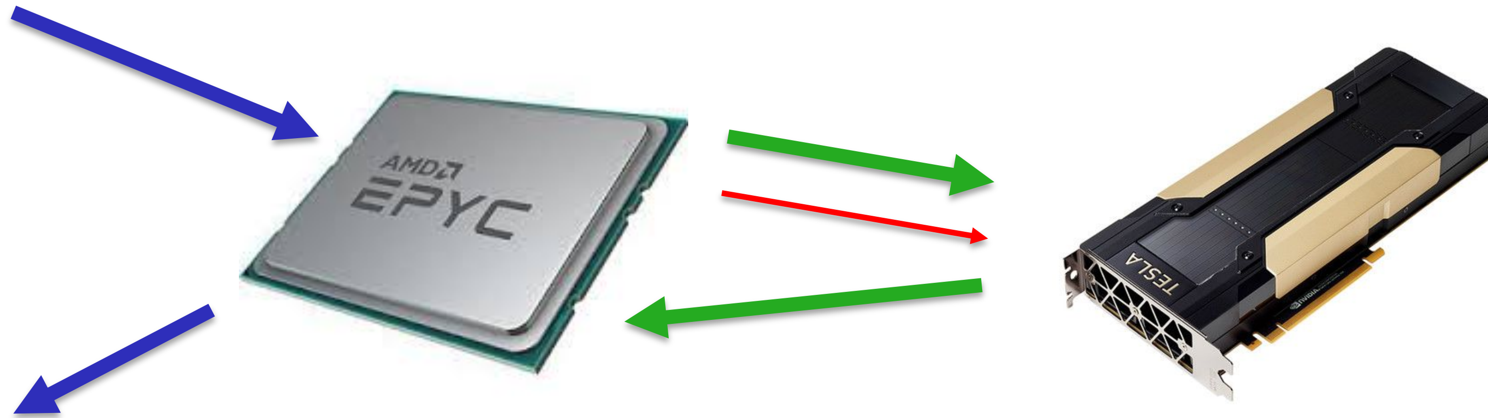
CPU-GPU Bottleneck Example



- Data size: 32 GB = 1 sec.
- Time on CPU: 32/204.8 = 0.15 sec.
- Time on GPU: 32/900 = 0.035 sec.

0.15 – 0.035 = 0.115 -> run kernel 9 times to amortize

Case 2: CPU and GPU perform different parts of the computation

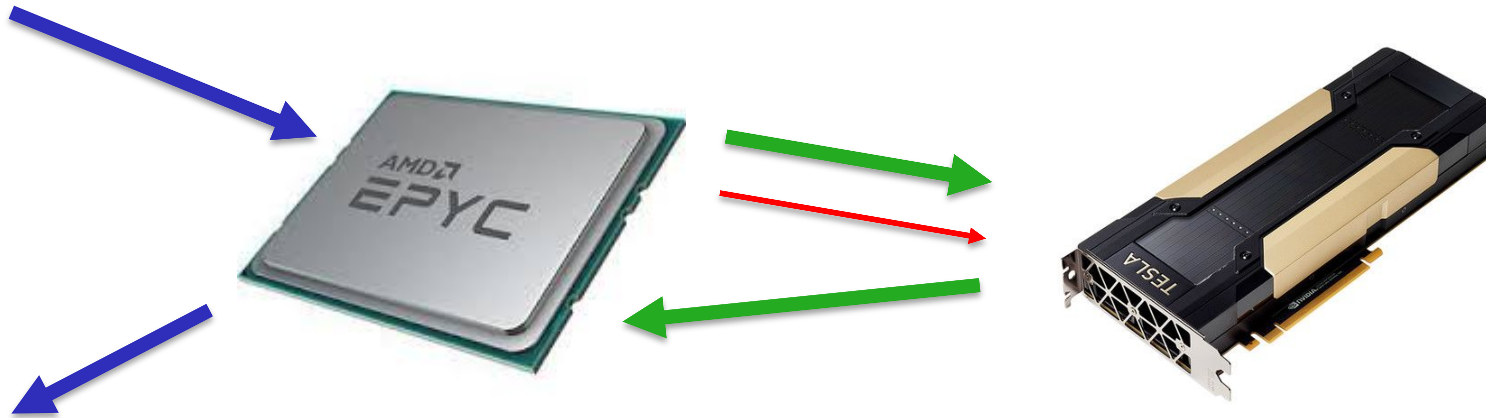


CPU / Host
- I/O
- Data transfers
- Kernel launch
- Low data reuse computation

GPU / Device

High data reuse
Computation

Case 2: CPU and GPU perform different parts of the computation



Example: matrix computation

$$E = \begin{pmatrix} 3 & 2 & 6 \\ 2 & 4 & 8 \\ 5 & 6 & 9 \end{pmatrix} * \begin{pmatrix} 1 & 5 & 6 \\ 0 & 7 & 7 \\ 5 & 0 & 4 \end{pmatrix} + \begin{pmatrix} 2 & 9 & 5 \\ 0 & 9 & 4 \\ 1 & 9 & 4 \end{pmatrix} + \begin{pmatrix} 4 & 5 & 1 \\ 3 & 9 & 1 \\ 7 & 3 & 3 \end{pmatrix}$$

A **B** **C** **D**

Case 2: CPU and GPU perform different parts of the computation

Example: matrix computation

E = A * B + C + D

$$E = \begin{pmatrix} 3 & 2 & 6 \\ 2 & 4 & 8 \\ 5 & 6 & 9 \end{pmatrix} * \begin{pmatrix} 1 & 5 & 6 \\ 0 & 7 & 7 \\ 5 & 0 & 4 \end{pmatrix} + \begin{pmatrix} 2 & 9 & 5 \\ 0 & 9 & 4 \\ 1 & 9 & 4 \end{pmatrix} + \begin{pmatrix} 4 & 5 & 1 \\ 3 & 9 & 1 \\ 7 & 3 & 3 \end{pmatrix}$$

C + D has no data reuse. Can be performed on the CPU.

A * B should move to the GPU

Example of 2D Grid: Matrix Addition

```
dim3 grid, block;
block.x = 16;
block.y = 16;
grid.x = n/16;     //n times n matrix
if(n%16) grid.x++;
grid.y = grid.x;
madd<<<grid,block>>>(d_a, d_b, d_c);

__global__ void madd(int *a, int *b, int *c, int n) {
      int X = threadIdx.x + blockIdx.x * blockDim.x;
      int Y = threadIdx.y + blockIdx.y * blockDim.y;
      if(X < n && Y < n) {
            index = X*n+Y;
            c[index] = a[index] + b[index];
      }
}
```

DGEMM: Dense GEneral Matrix Multiplication

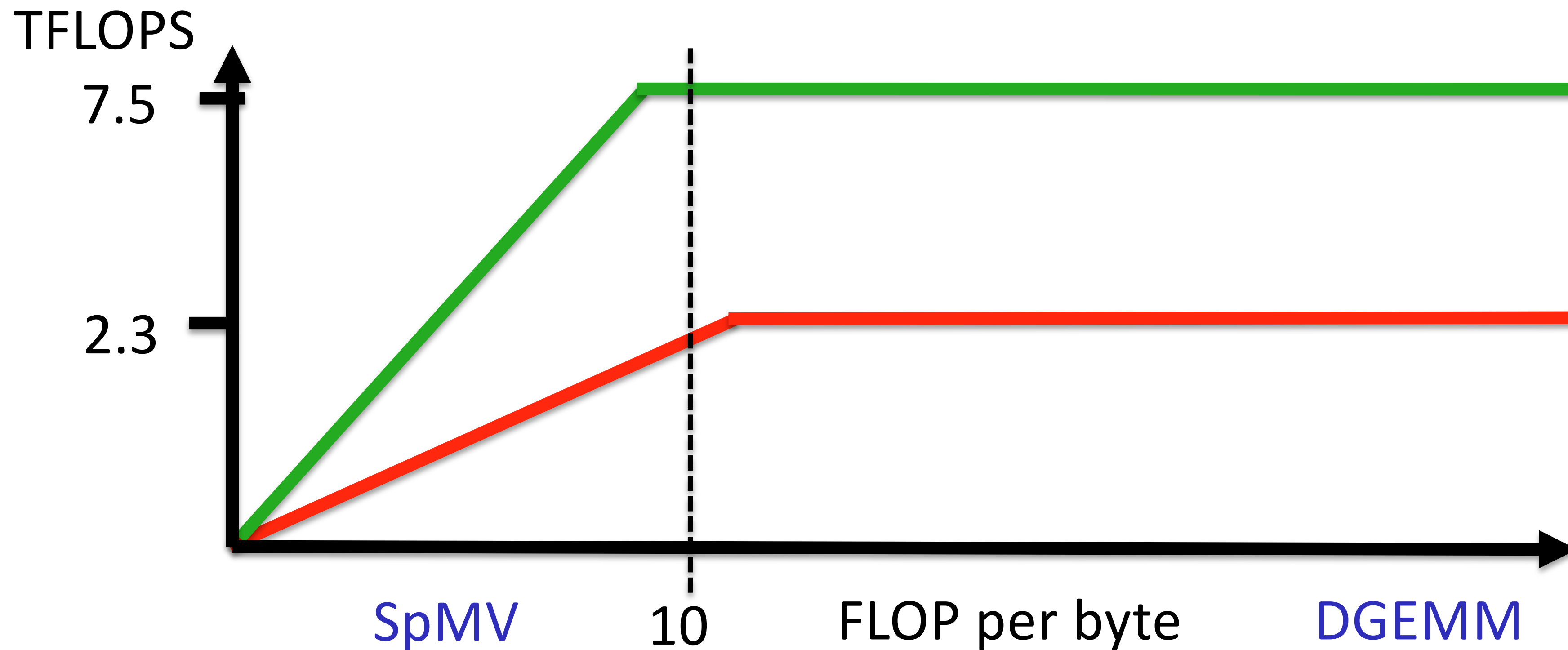
$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

Data: $3 * n^2 *$ sizeof(double)

Compute: $n^2 * 2n$ FLOPS

Much more compute than data. Good candidate to move to GPU

Roofline Model



CPU-GPU Communication: 24 byte per flop = 0.0416 FLOP per byte

AMD EPYC 7742	2300 / 204.8	=	11.23
NVIDIA Volta V100	7500 / 900	=	8.33

Balanced Systems

Compute: **E = A * B + C + D**

Problem: What if CPU is too weak to keep up with GPU ?
Solution: Buy a faster CPU (balance system)
(if that is not possible, keep entire computation on the GPU)

Balanced Systems

Problem 1: What if CPU is too weak to keep up with GPU ?
Solution 1: Buy a faster CPU (balance system)

Problem 2: Now the expensive CPU is idling
Solution 2: Share computation between CPU and GPU

Case 3: CPU and GPU share work

AMD EPYC 7742	2300 GFLOPS	204.8	GB/s
NVIDIA Volta V100	7500 GFLOPS	900	GB/s

Need to assign the right amount of work to each processor...

2 ways of splitting the work:

- Static
- Dynamic

Static CPU / GPU load balancing

AMD EPYC 7742	2300 GFLOPS	204.8	GB/s
NVIDIA Volta V100	7500 GFLOPS	900	GB/s

Static load balancing in a memory bound computation:

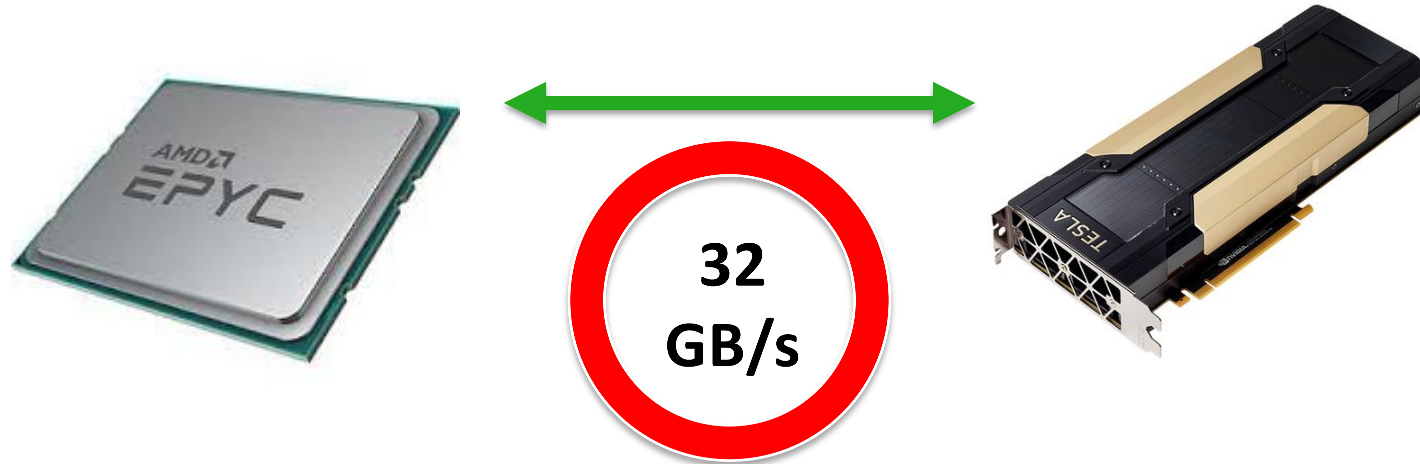
GPU work = 900 / (204.8+900) = 0.814 = 81.4%

Only works if computational performance is predictable

In case of wrong prediction, one device will be idling

Sounds scary, lets try dynamic

Dynamic CPU / GPU load balancing

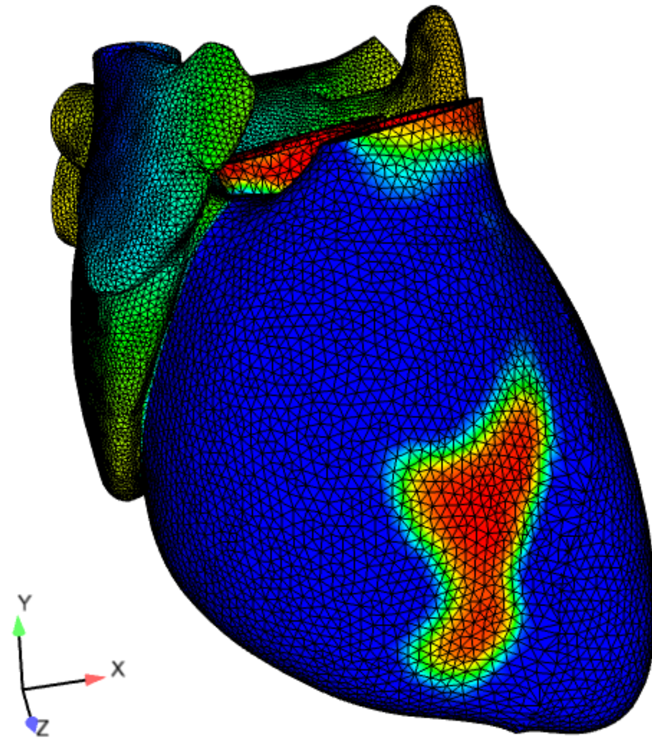


Remember the speed limit...

Moving data between CPU and GPU is often too slow.
In addition, complexities of dynamic load balancing have high overhead.

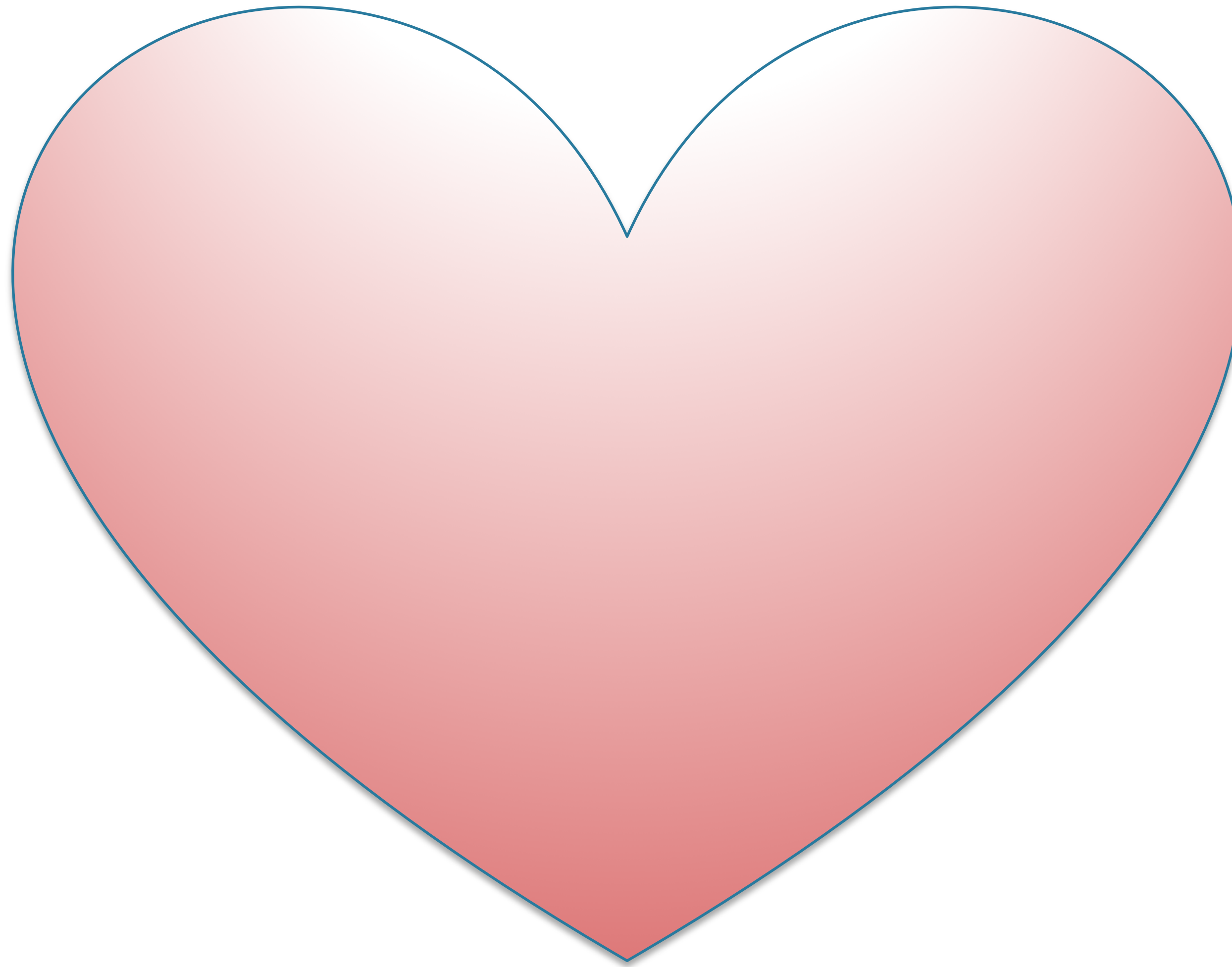
Sounds even worse, lets stick with static

Example Application: Cardiac Electrophysiology

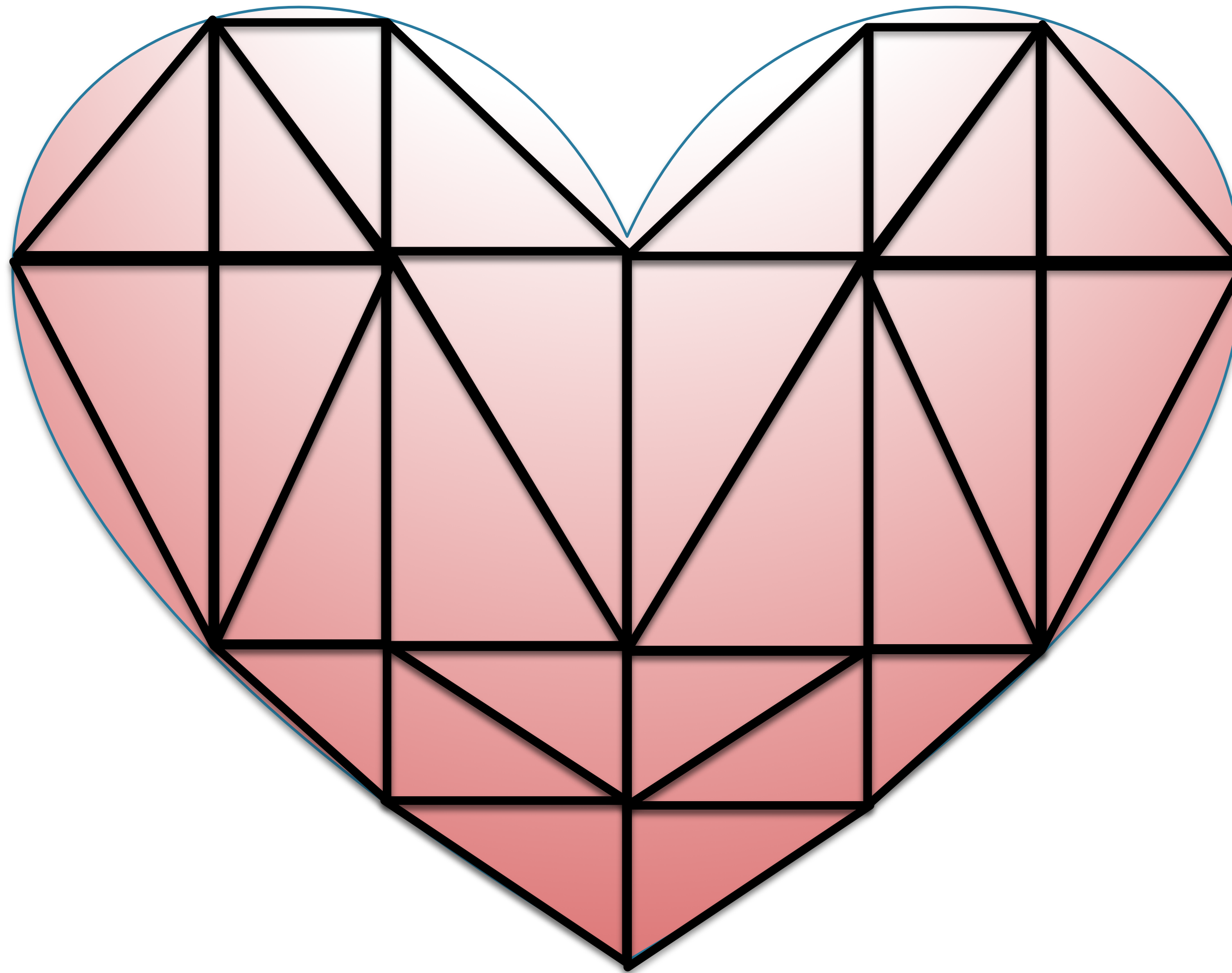


- We want to study electrical activity in the heart
- Activity is governed by PDEs
- Dissolve heart into mesh cells
- Simulate diffusion of voltage over time
- Discretize time and space

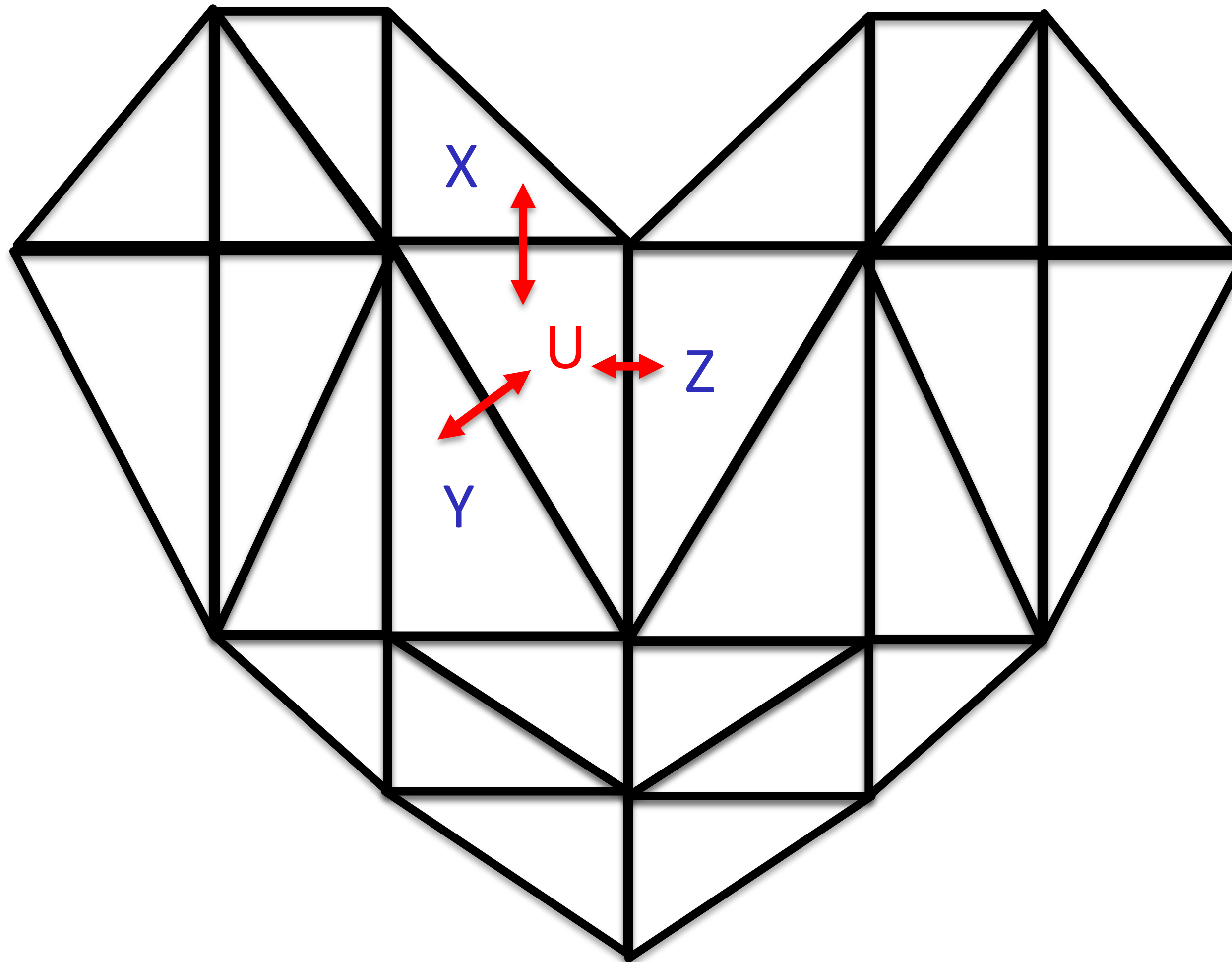
Irregular Shape in 2D



Irregular Mesh in 2D



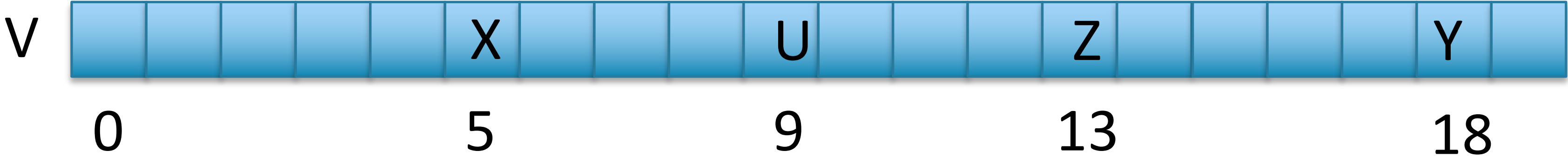
Cell Centric Finite Volume Method in 2D



U has three neighbors it interacts with: X, Y, and Z

From Mesh to Matrix

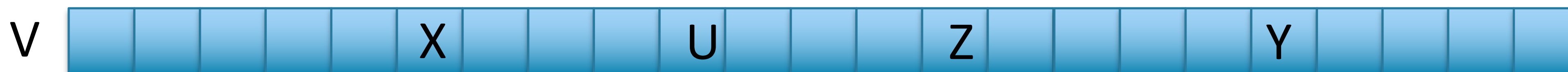
U has three neighbours it interacts with: X, Y, and Z

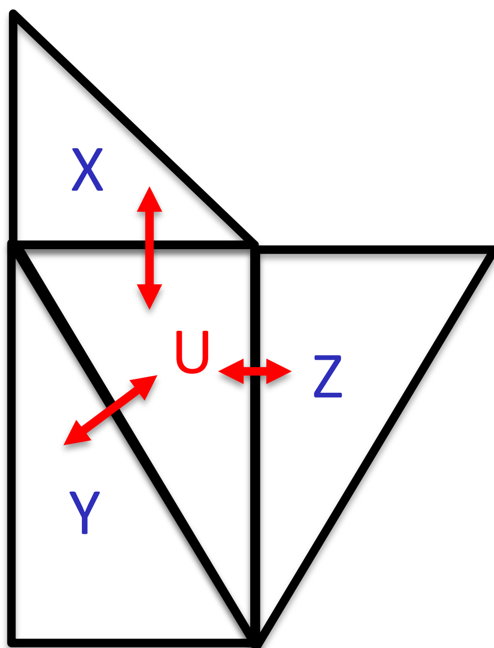


U has three neighbours it interacts with: X, Y, and Z

From Mesh to Matrix

U has three neighbors it interacts with: X, Y, and Z

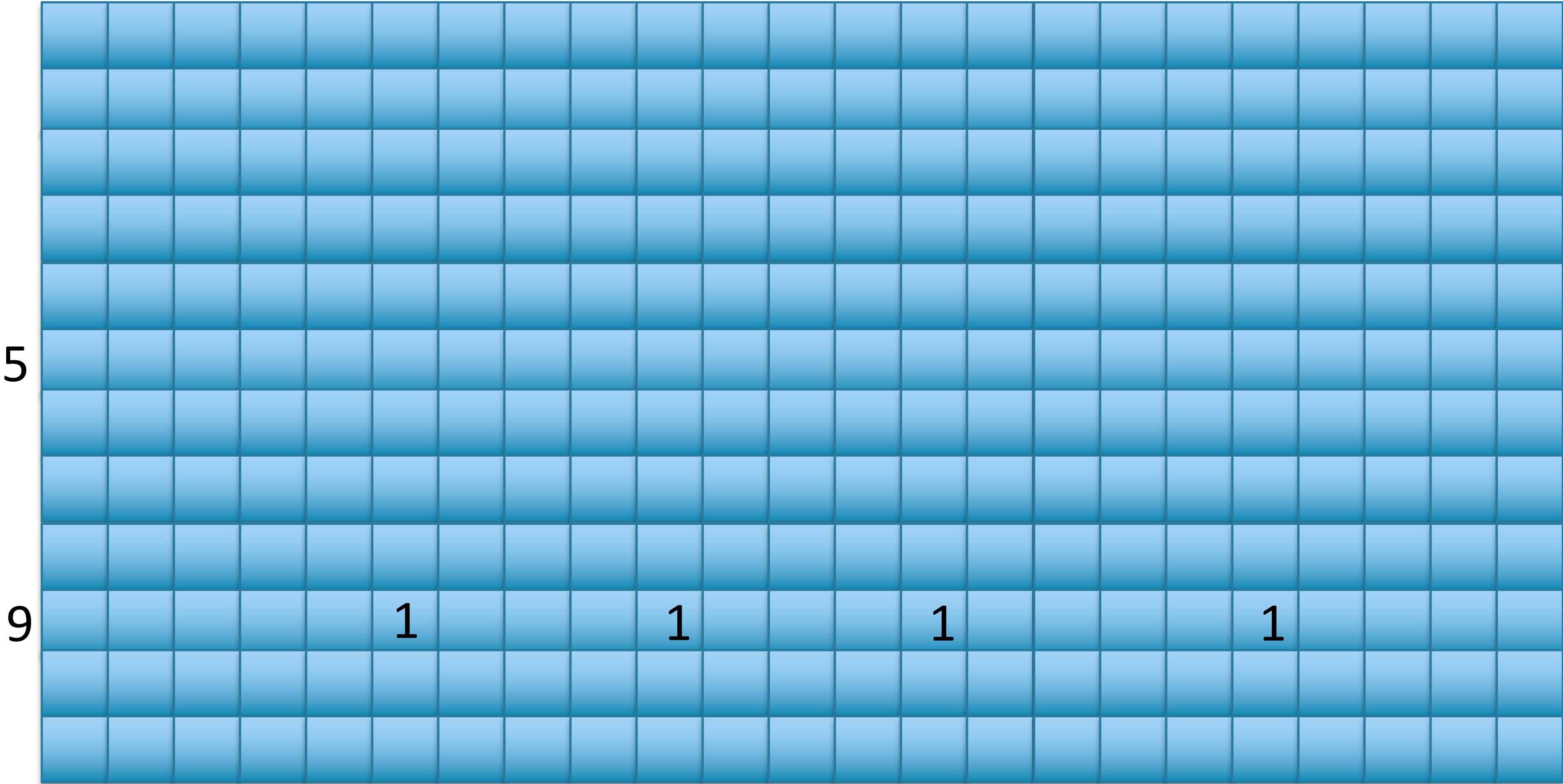




- Use a vector V to store the voltage for each cell
- Need data structure for mesh

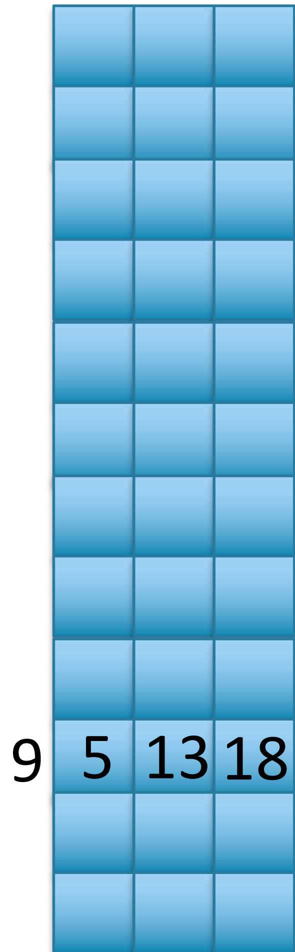
Adjacency matrix

From Mesh to Matrix



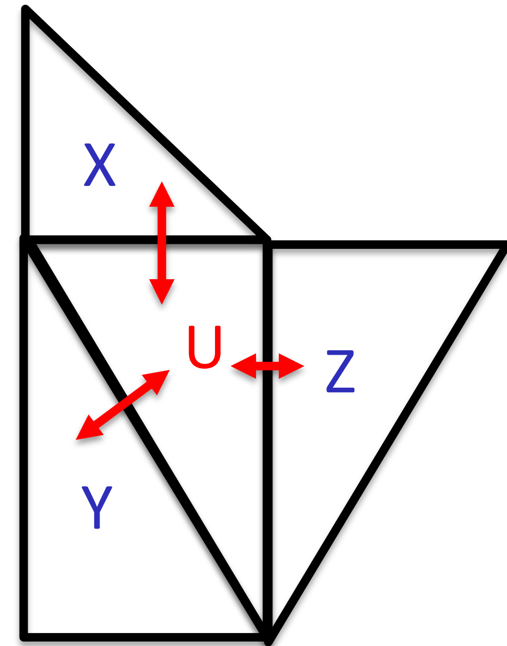
Dense matrix is mostly 0. We need a sparse matrix.

From Mesh to Sparse Matrix

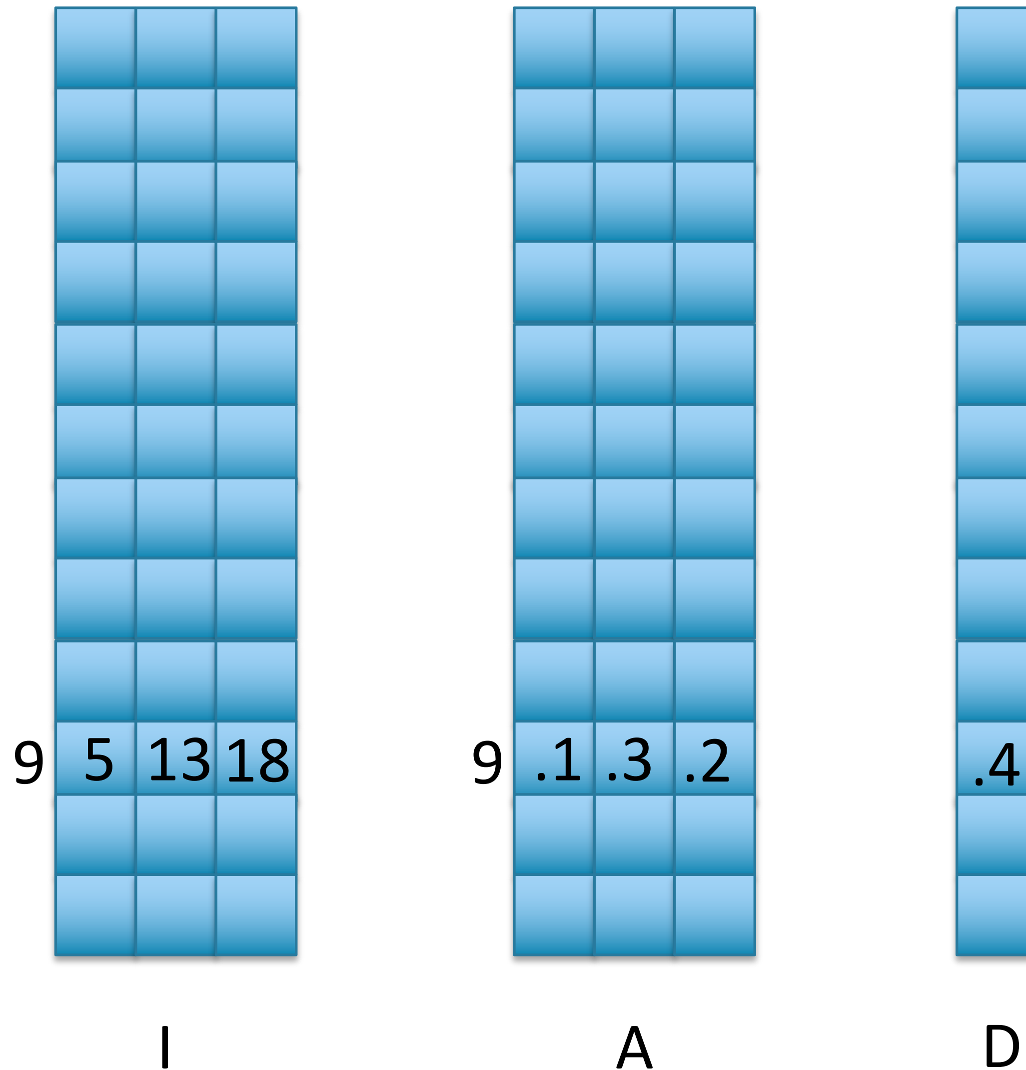


Sparse matrix stores the indices of nonzeroes.
Much more space efficient.

But we also need to store strength of the interaction, and voltage conservation.



From Mesh to Sparse Matrix: ELLpack



Coefficients are stored in A and D

There is no need for indices for D, as these values are on the diagonal.

Now we can compute V[U] for the next time step.

From Mesh to Sparse Matrix: ELLpack

Compute V[U] for the next time step.

$$
\begin{aligned}
V[9]_{t+1} = \; & A[9,0] * V[I[9,0]]_t + \\
& A[9,1] * V[I[9,1]]_t + \\
& A[9,2] * V[I[9,2]]_t + D[9] * V[9]_t
\end{aligned}
$$

I

	0	1	2
9	5	13	18

A

	0	1	2
9	.1	.3	.2

D

.4

V

SpMV Computation for Diffusion

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

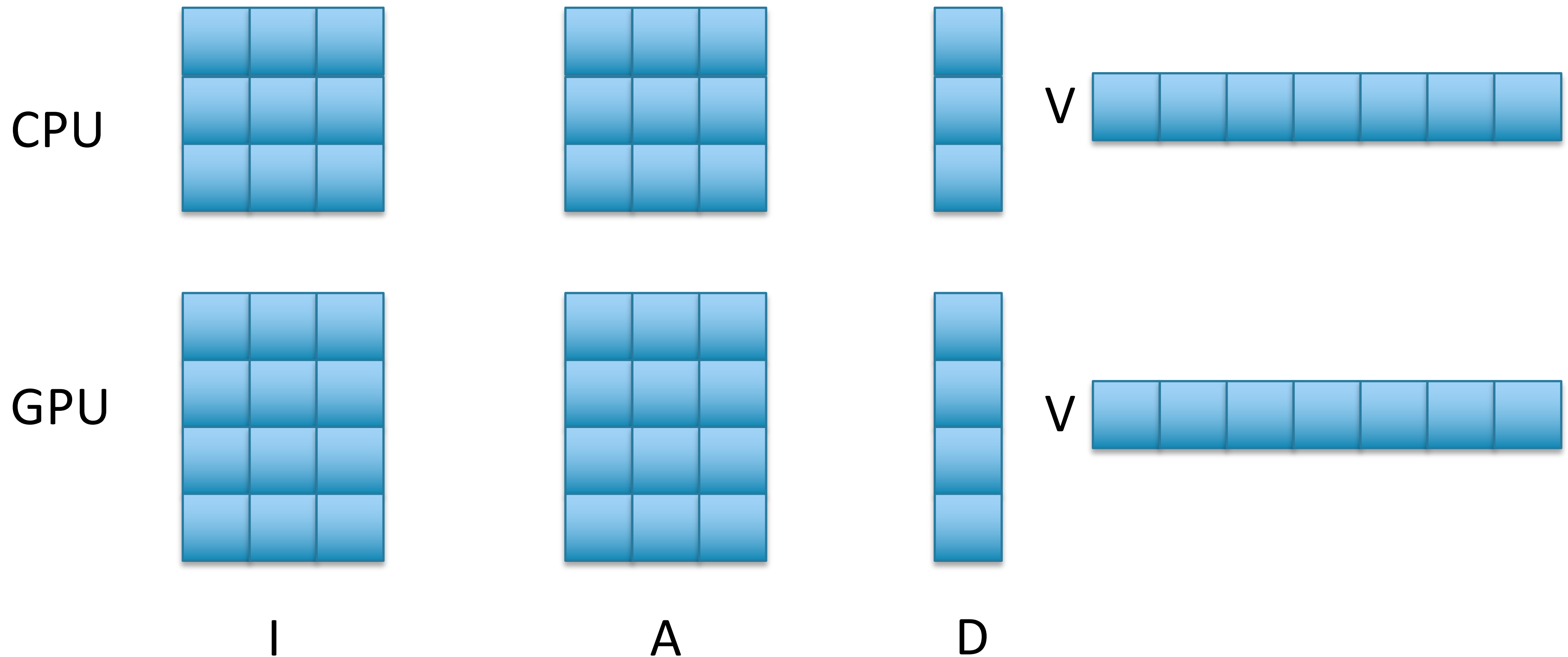
As a matrix operation: $V_{t+1} = AV_t$

We can update each vector element separately.
Thus, we could split the rows between CPU and GPU.

CPU – GPU SpMV Computation for Diffusion

2 possibilities for distributing SpMV computation:

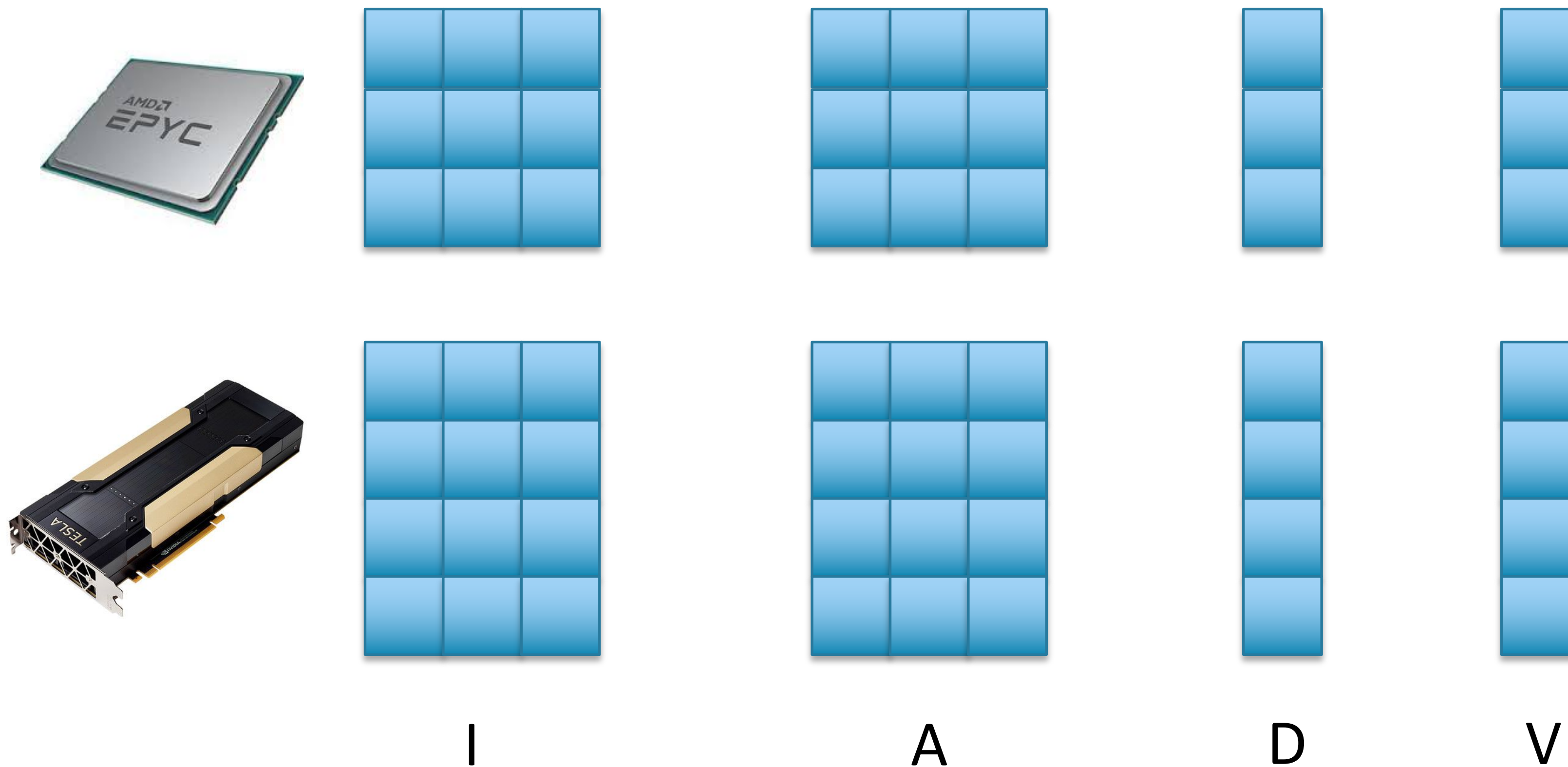
Split I, A, D, replicate V



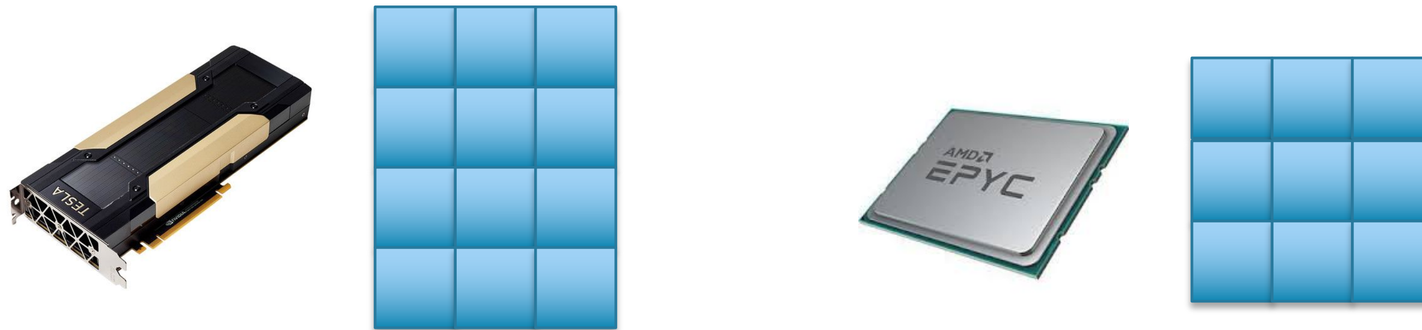
CPU – GPU SpMV Computation for Diffusion

2 possibilities for distributing SpMV computation:

Split I, A, D, and V. Then renumber I (difficult)



CPU – GPU SpMV Computation for Diffusion

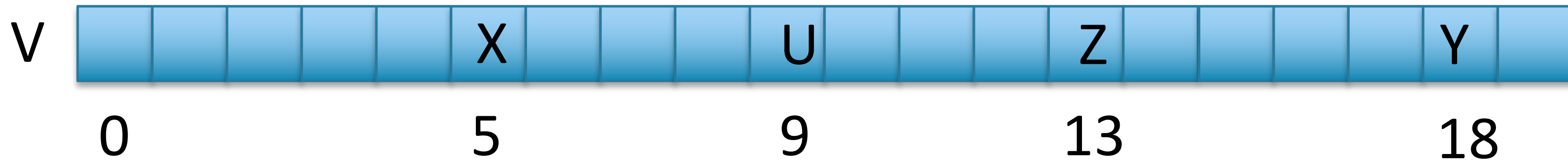


Now we have split the data. Are we done ? No.

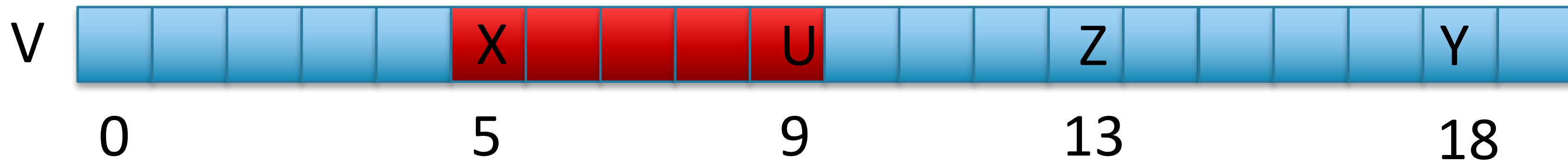
Static partitioning only works if computational performance is predictable

Remember that accesses to V are irregular. Depending on the distribution, they may or may not be cached.

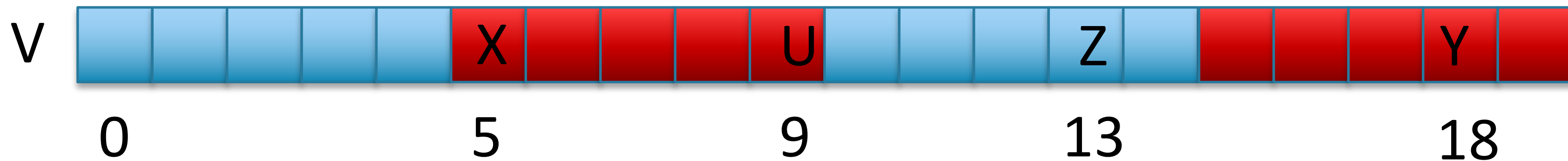
Caching in Irregular Computations



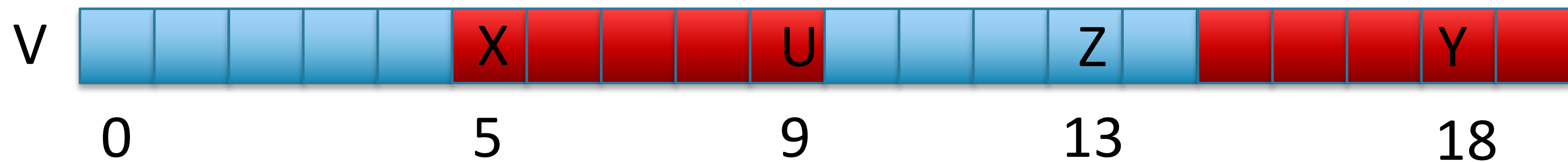
Access X. We load a full cache line. (length 5)



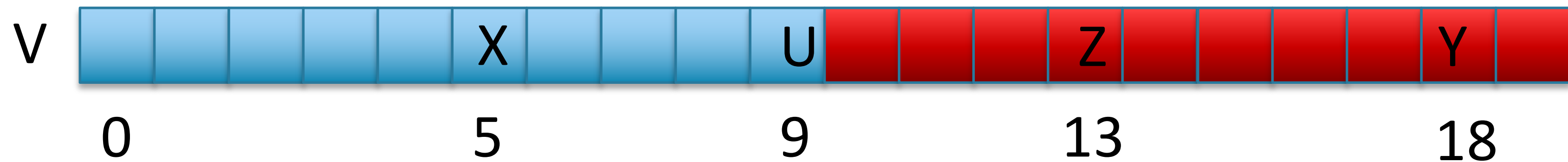
Now access to U is much faster than Z and Y. We load Y.



Caching in Irregular Computations

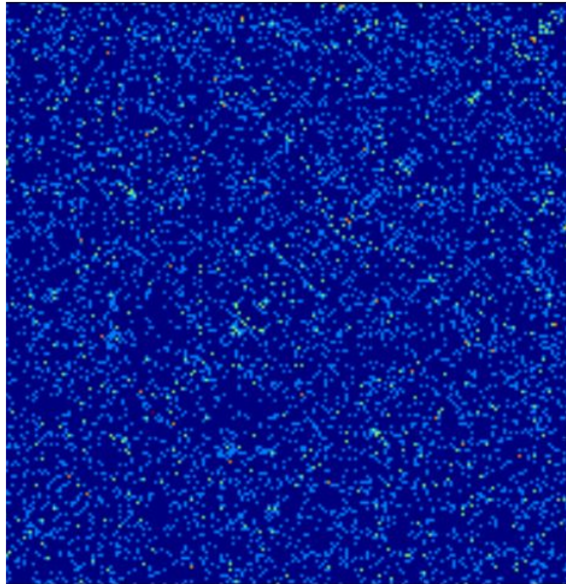


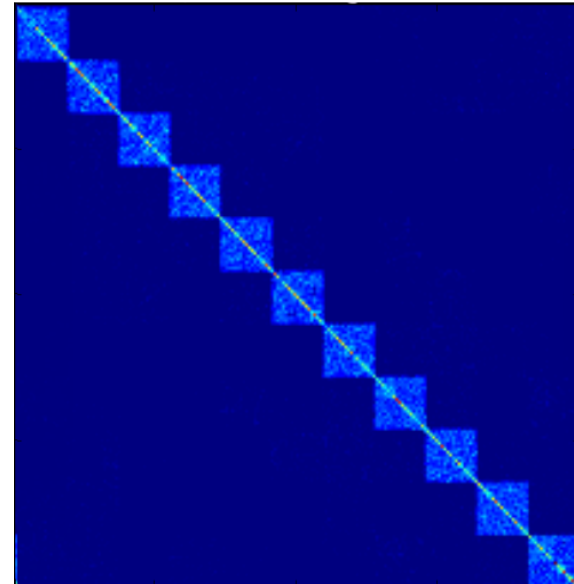
Next we load Z, but we have to evict cells 5-9 first.



Access to U is slow again, since it is no longer in cache.

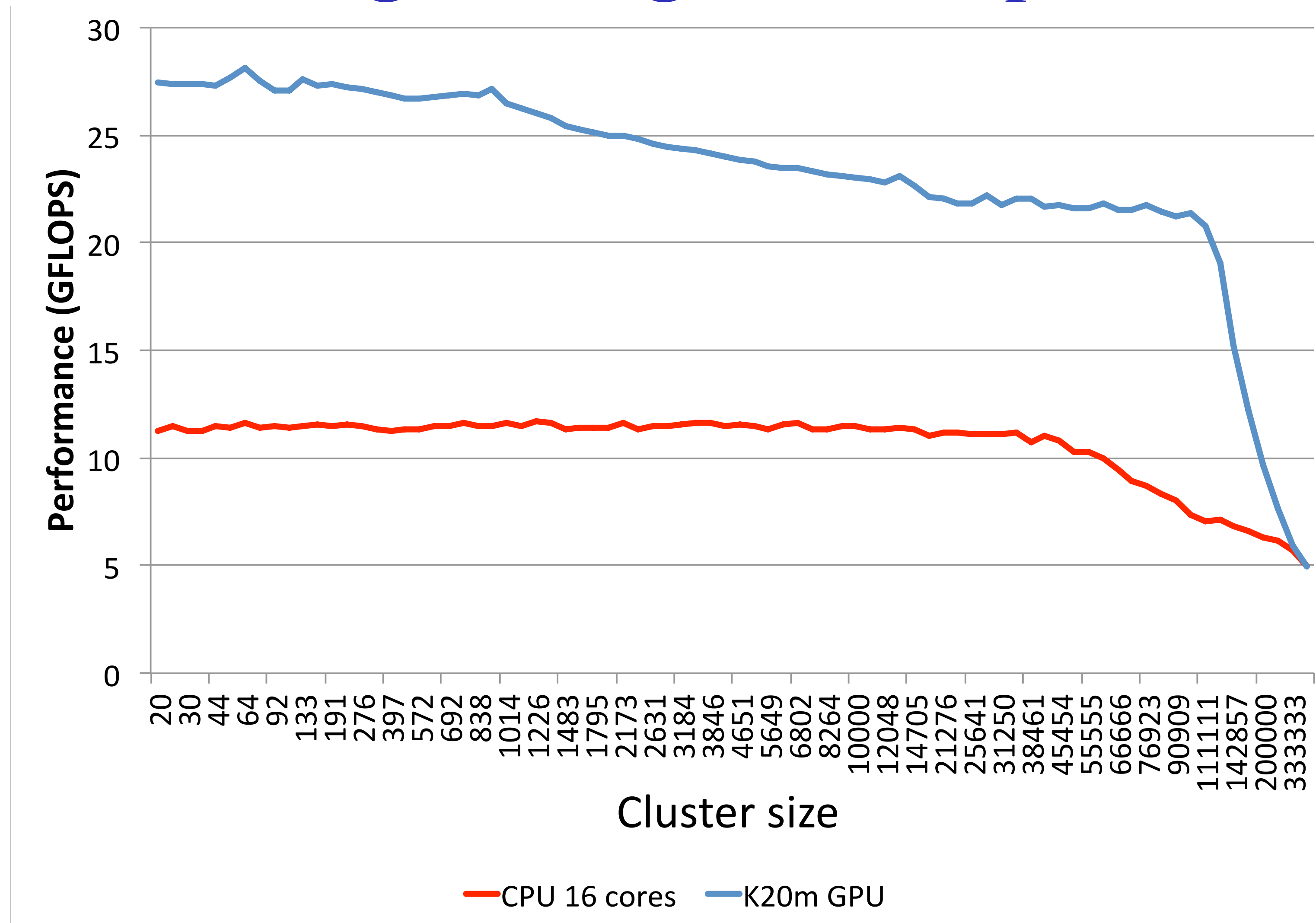
Caching in Irregular Computations





When computing rows consecutively, block diagonal/ low bandwidth shape improves cache performance.
(not the whole story, but close enough)

Caching in Irregular Computations

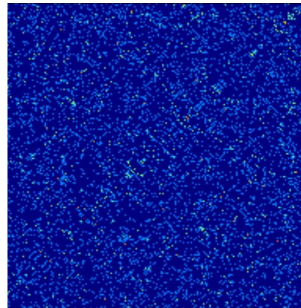


We need to reorder the matrix to get consistent (and high) Performance.

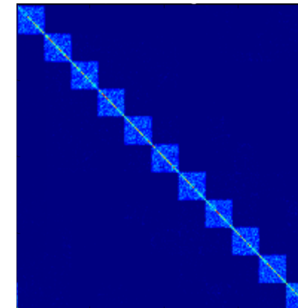
Reordering for Cache Performance

How to get from

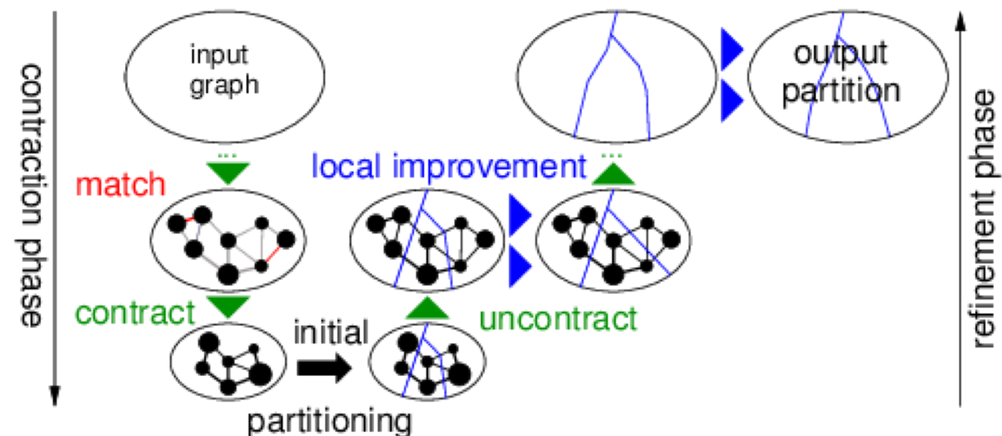
here



to here

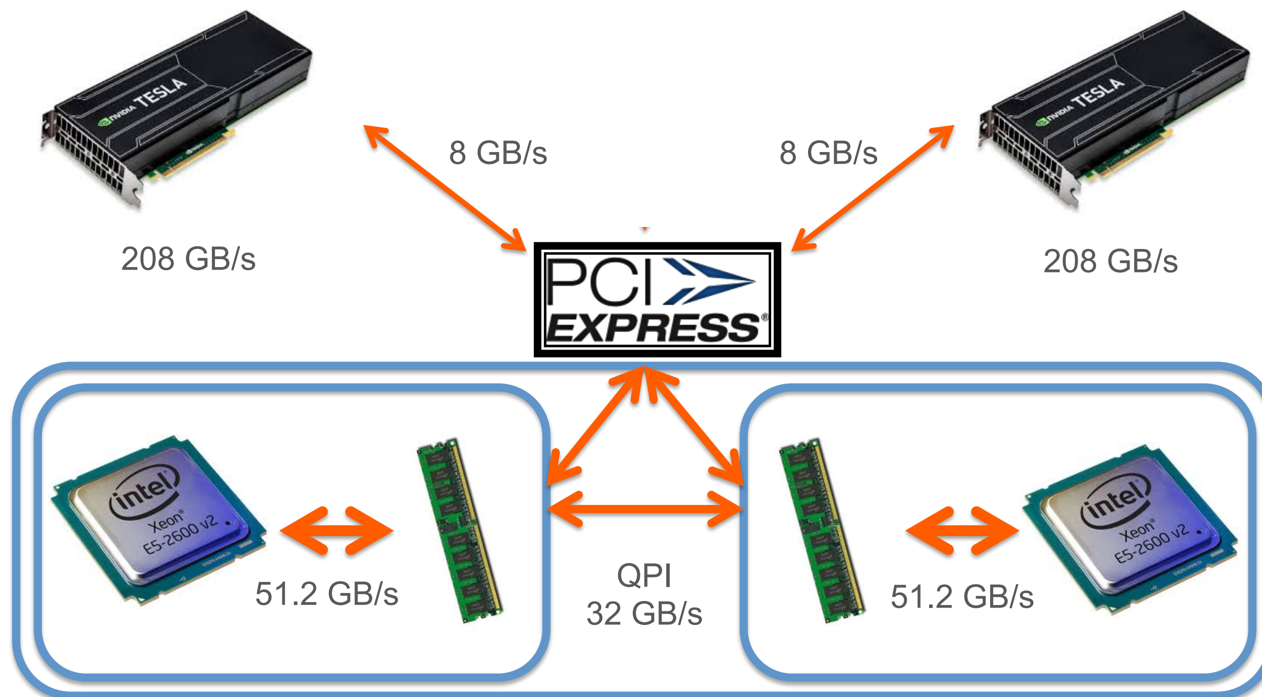


- Reverse Cuthill-McKee algorithm
- Graph partitioner
  1. Metis
  2. KaHip
  3. Patoh
  4. Etc…

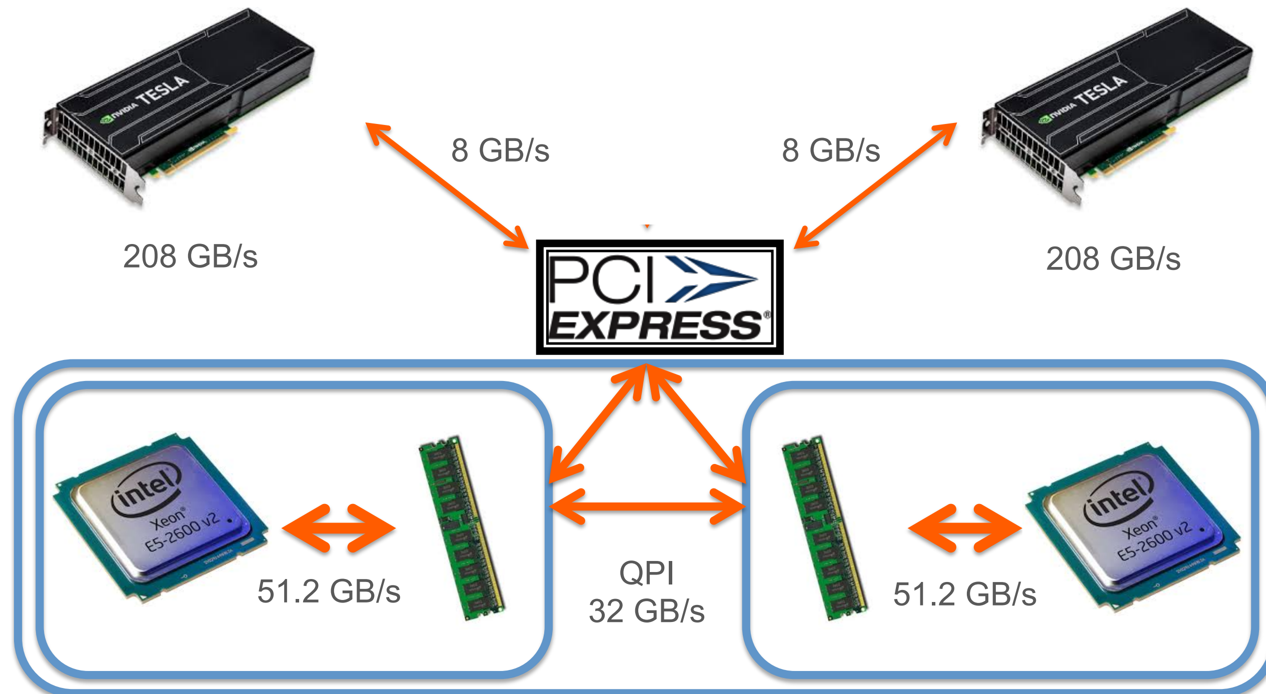


LYNX: SpMV Computation for Diffusion

The LYNX cardiac electrophysiology simulator performs all these steps. We test it on the following (slightly older) system:



LYNX: CPU – GPU Heterogeneous Computation



Memory bandwidth: 80 GB/s CPU and 2 times 208 GB/s GPU

2*208 / 2*208 + 80 = 0.83 GPU workload ratio

LYNX: CPU – GPU Heterogeneous Computation

80
70
60
50
40
30
20
10
0

Performance (GFLOPS)

0.05 0.1 0.15 0.2 0.25 0.3 0.35 0.4 0.45 0.5 0.55 0.6 0.65 0.7 0.75 0.8 0.85 0.9 0.95

Accelerator workload ratio *r*

Synthetic instance   Real Mesh   GPU speedup limit

CPU – GPU Heterogeneous Computation
Was it worth the trouble ?

What did we get ?

- 5x performance for using the GPU
- 5x performance for reordering the matrix
- 20% more performance for using the CPU as well

For tasks that work well on the GPU, adding the CPU has high complexity and little benefit. But not everything works so well on the GPU…

References

Williams, S., Waterman, A., & Patterson, D. (2009). Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, *52*(4), 65-76.

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.

Langguth, J., Wu, N., Chai, J., & Cai, X. (2013, November). On the GPU performance of cell-centered finite volume method over unstructured tetrahedral meshes. In *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms* (pp. 1-8).

Credit: Lecture contains NVIDIA material available at https://developer.nvidia.com/cuda-zone
AMD information from https://www.amd.com/en/products/epyc
Image source: wikipedia.org