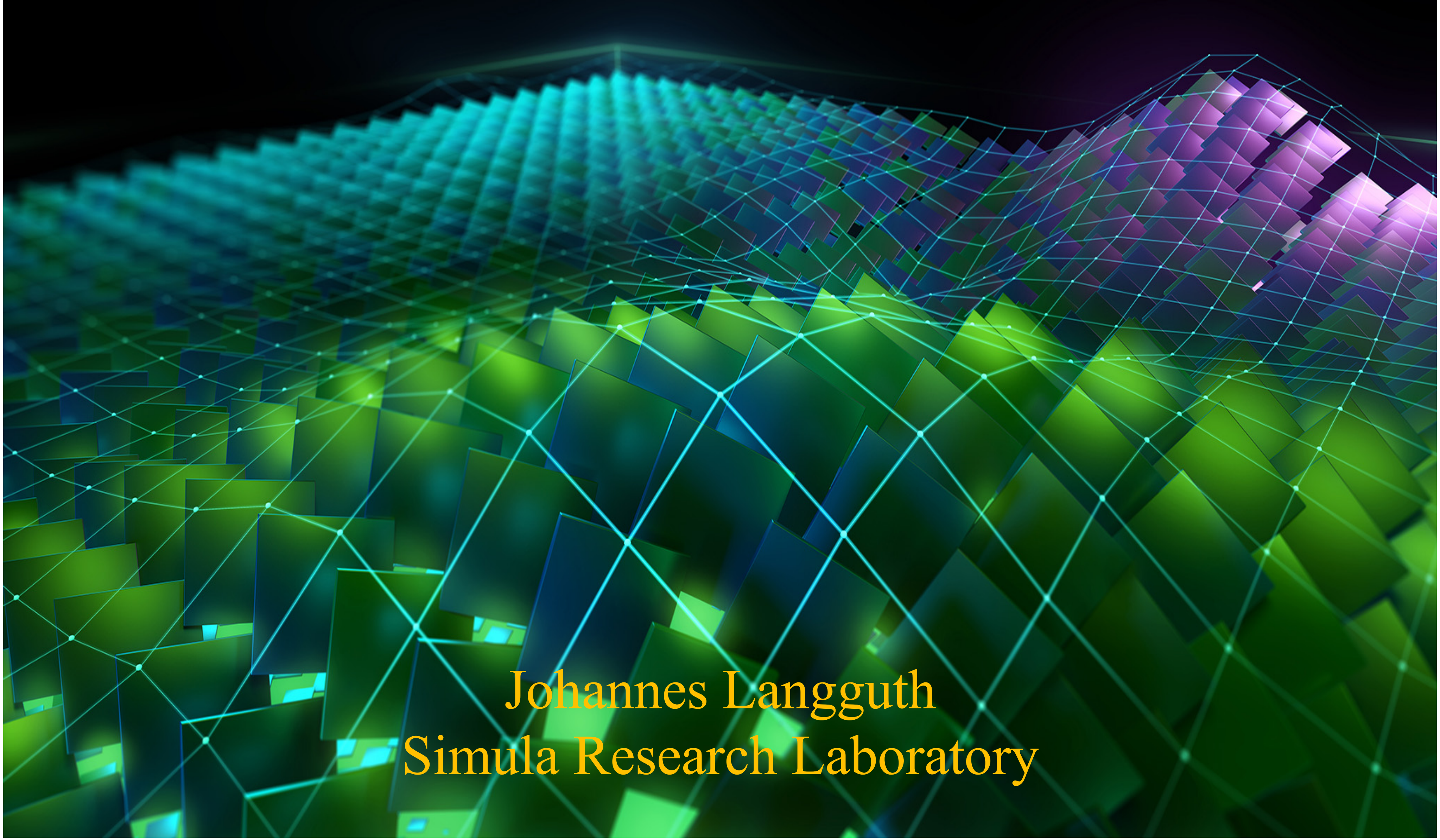
GPU Computing with CUDA (and beyond)
Part 5: Advanced Data Exchange



Johannes Langguth
Simula Research Laboratory

SpMV: more than ELLpack

- SpMV is one of the most important computational kernels
- Performance depends on the storage format

$$A = \begin{bmatrix} 1 & 0 & 0 & 5 \\ 3 & 2 & 0 & 4 \\ 9 & 0 & 0 & 0 \\ 0 & 4 & 0 & 9 \end{bmatrix}$$

$$\underset{val}{\begin{bmatrix} 1 & 5 & * \\ 3 & 2 & 4 \\ 9 & * & * \\ 4 & 9 & * \end{bmatrix}} \underset{Indices}{\begin{bmatrix} 0 & 3 & * \\ 0 & 1 & 3 \\ 0 & * & * \\ 1 & 3 & * \end{bmatrix}}$$

ELL format

$row =$	0	0	1	1	1	2	3	3
$col =$	0	3	0	1	3	0	1	3
$val =$	1	5	3	2	4	9	4	9

COO format

Hybrid Format

- Combination of ELL and COO formats

$$A = \begin{bmatrix} 1 & 0 & 0 & 5 \\ 3 & 2 & 0 & 4 \\ 9 & 0 & 0 & 0 \\ 0 & 4 & 0 & 9 \end{bmatrix}$$

$$\underset{val}{\begin{bmatrix} 1 & 5 \\ 3 & 2 \\ 9 & * \\ 4 & 9 \end{bmatrix}} \underset{Indices}{\begin{bmatrix} 0 & 3 \\ 0 & 1 \\ 0 & * \\ 1 & 3 \end{bmatrix}} \quad \underset{row}{\boxed{1}} \quad \underset{col}{\boxed{3}} \quad \underset{val}{\boxed{4}}$$

HYB format

CSR/CSC Format

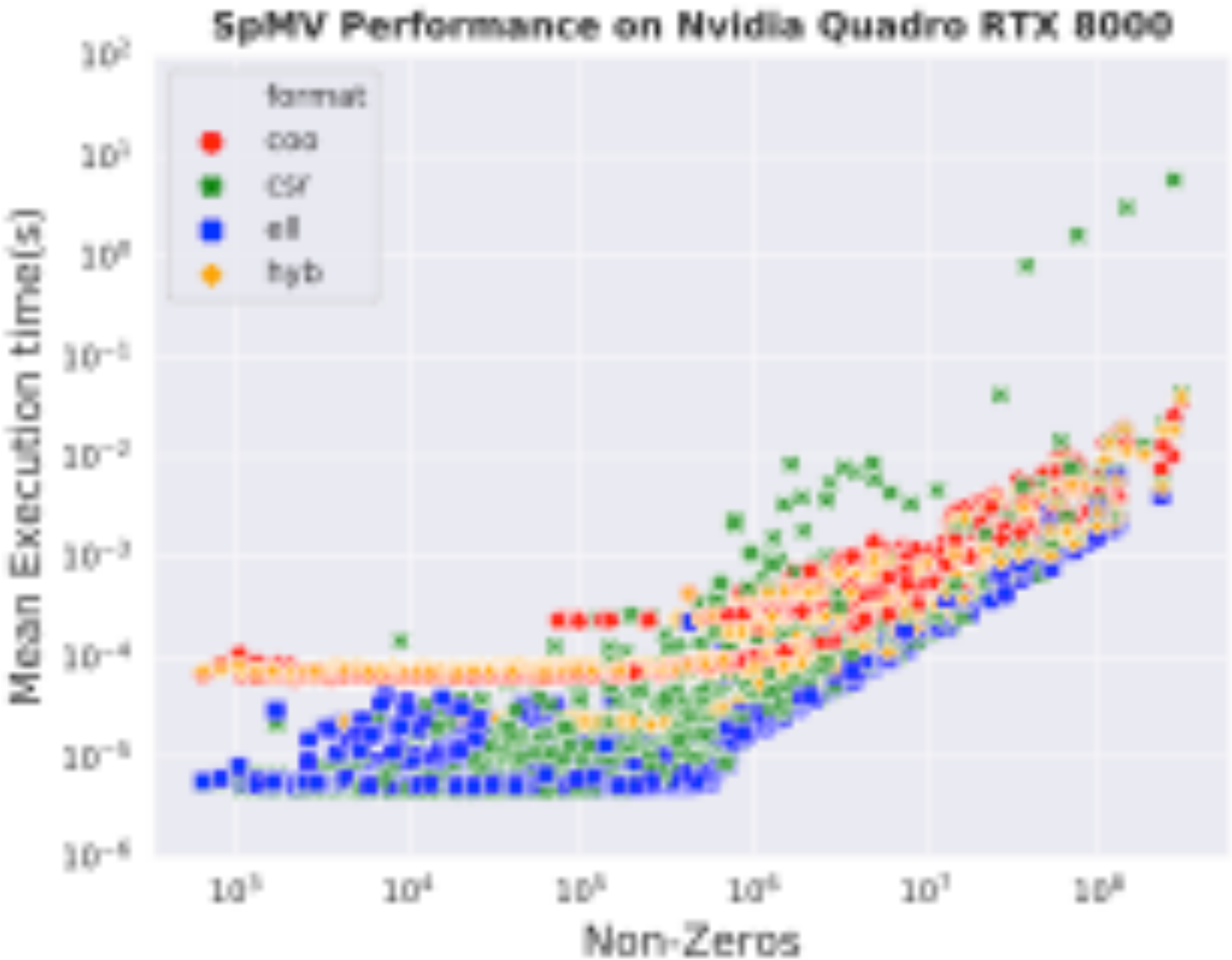
- The most common storage format for graphs and matrices

$$A = \begin{bmatrix} 1 & 0 & 0 & 5 \\ 3 & 2 & 0 & 4 \\ 9 & 0 & 0 & 0 \\ 0 & 4 & 0 & 9 \end{bmatrix}$$

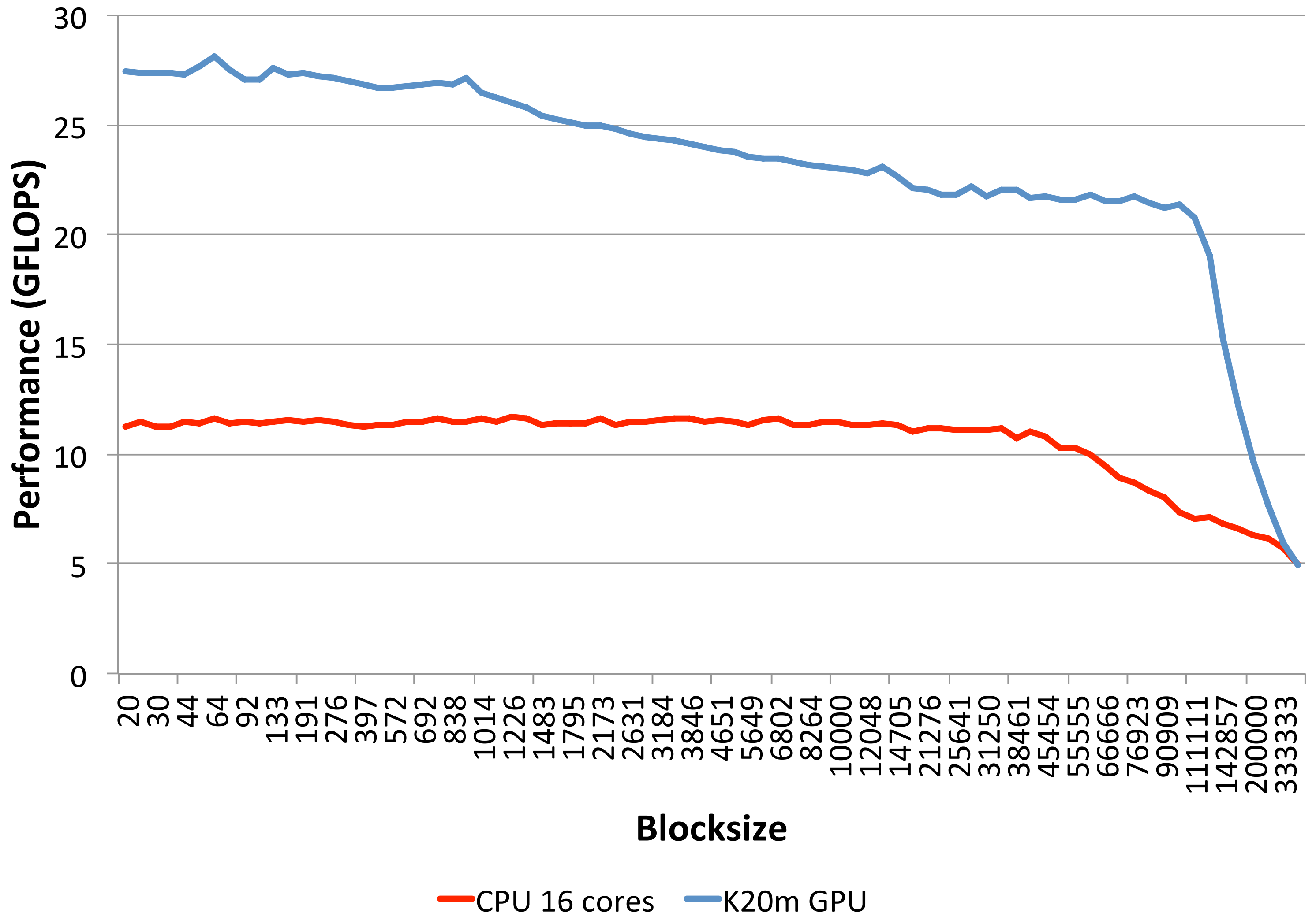
$rptr =$	0	2	5	6	8			
$col =$	0	3	0	1	3	0	1	3
$val =$	1	5	3	2	4	9	4	9

CSR format

Performance of the Different Formats

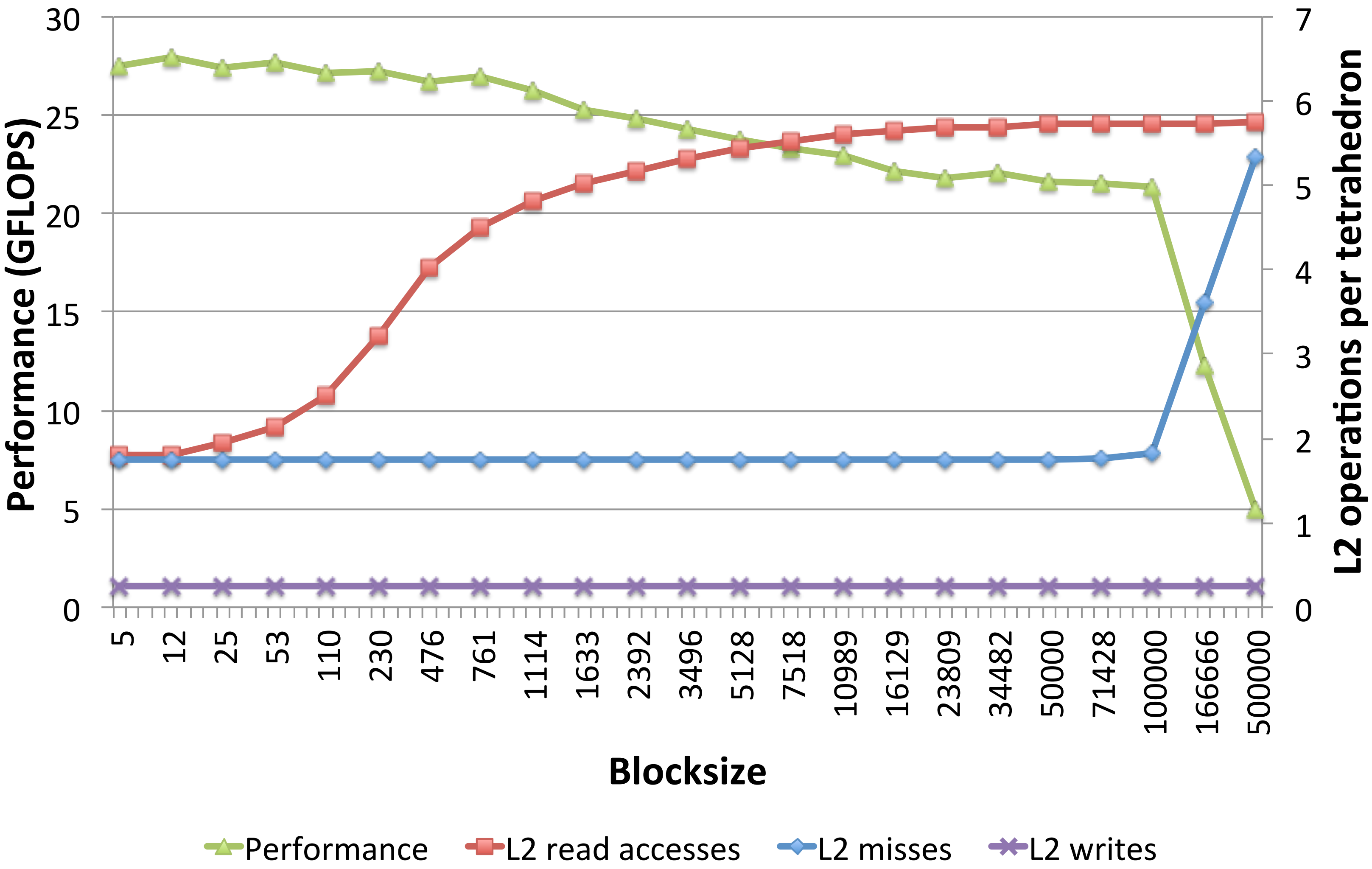


Can we Predict Performance ?



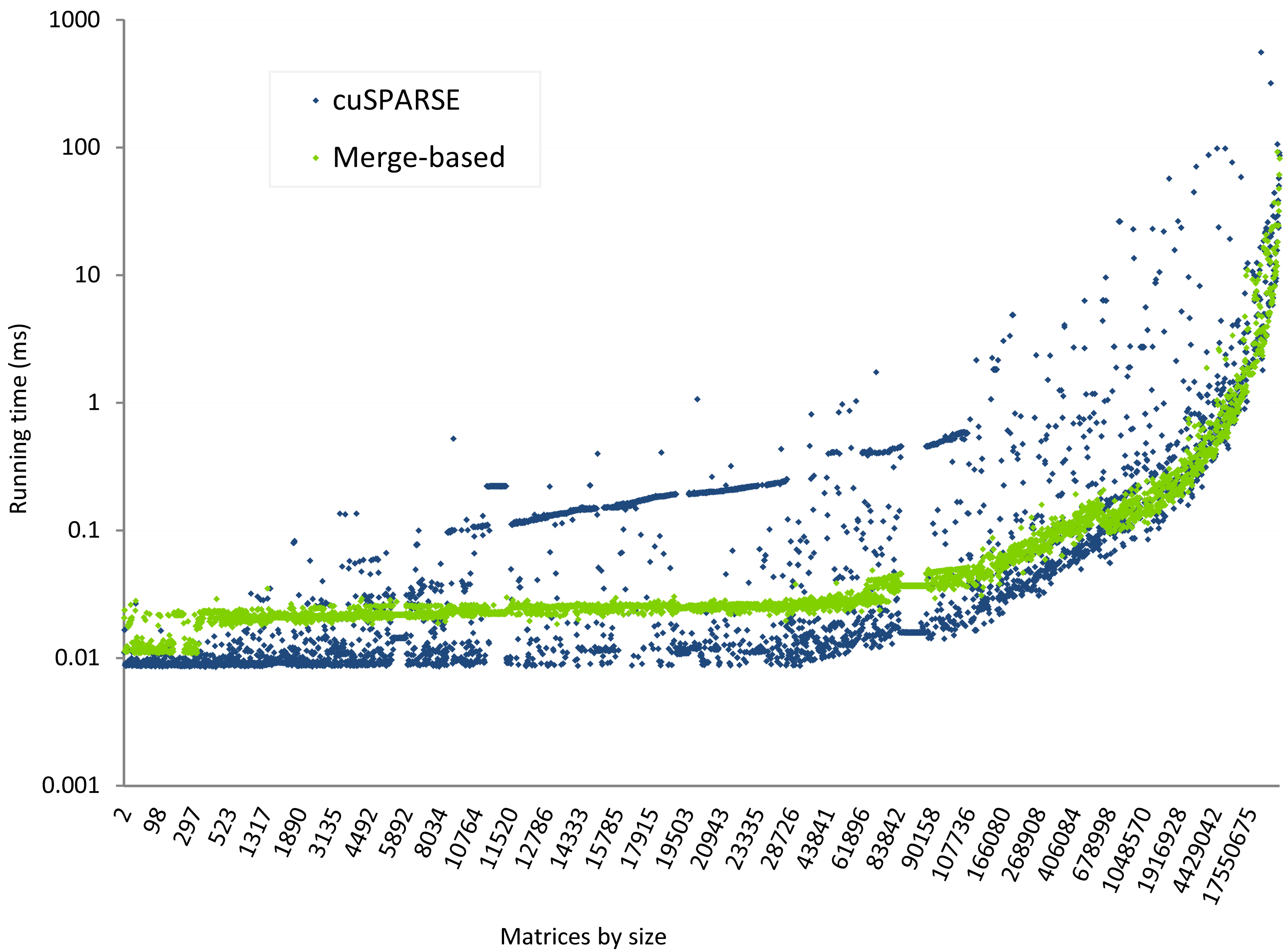
Performance is connected to irregularity. But how exactly ?

Profiling L2 Cache with NVProf



L2 misses are the most costly and most unpredictable factor

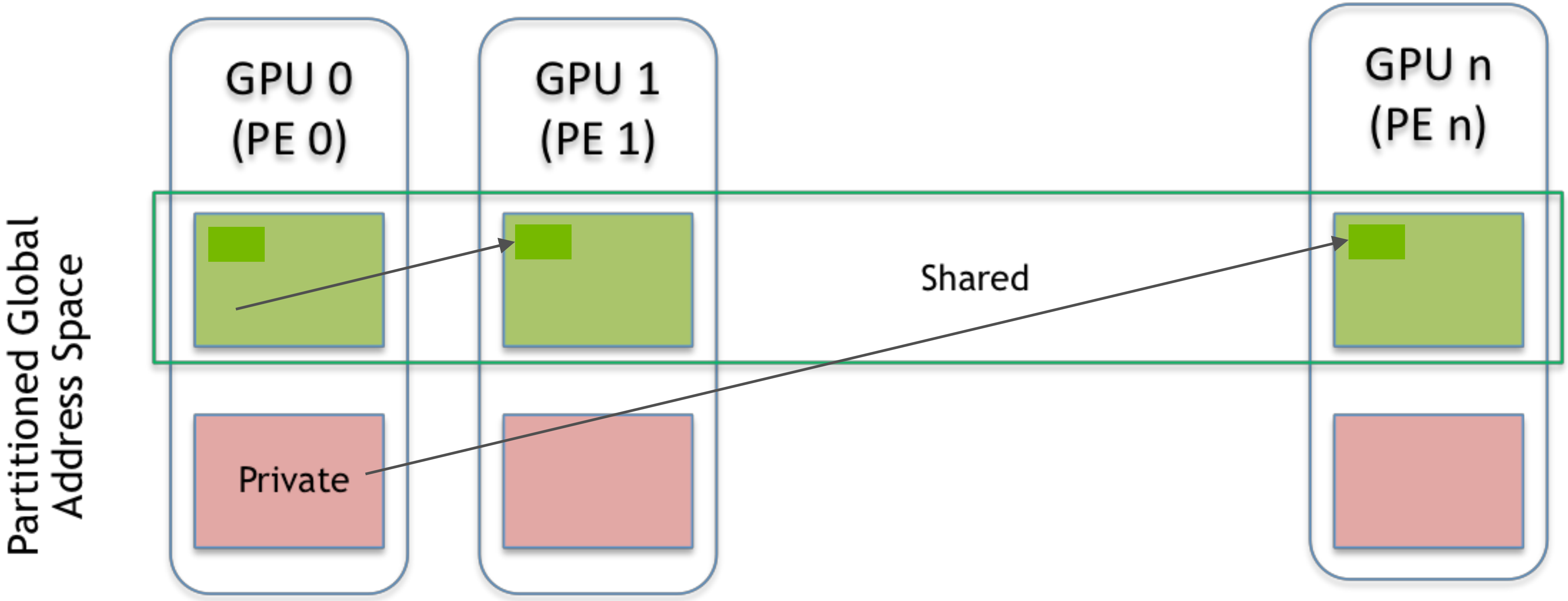
Merge-based SPMV: stable CSR performance



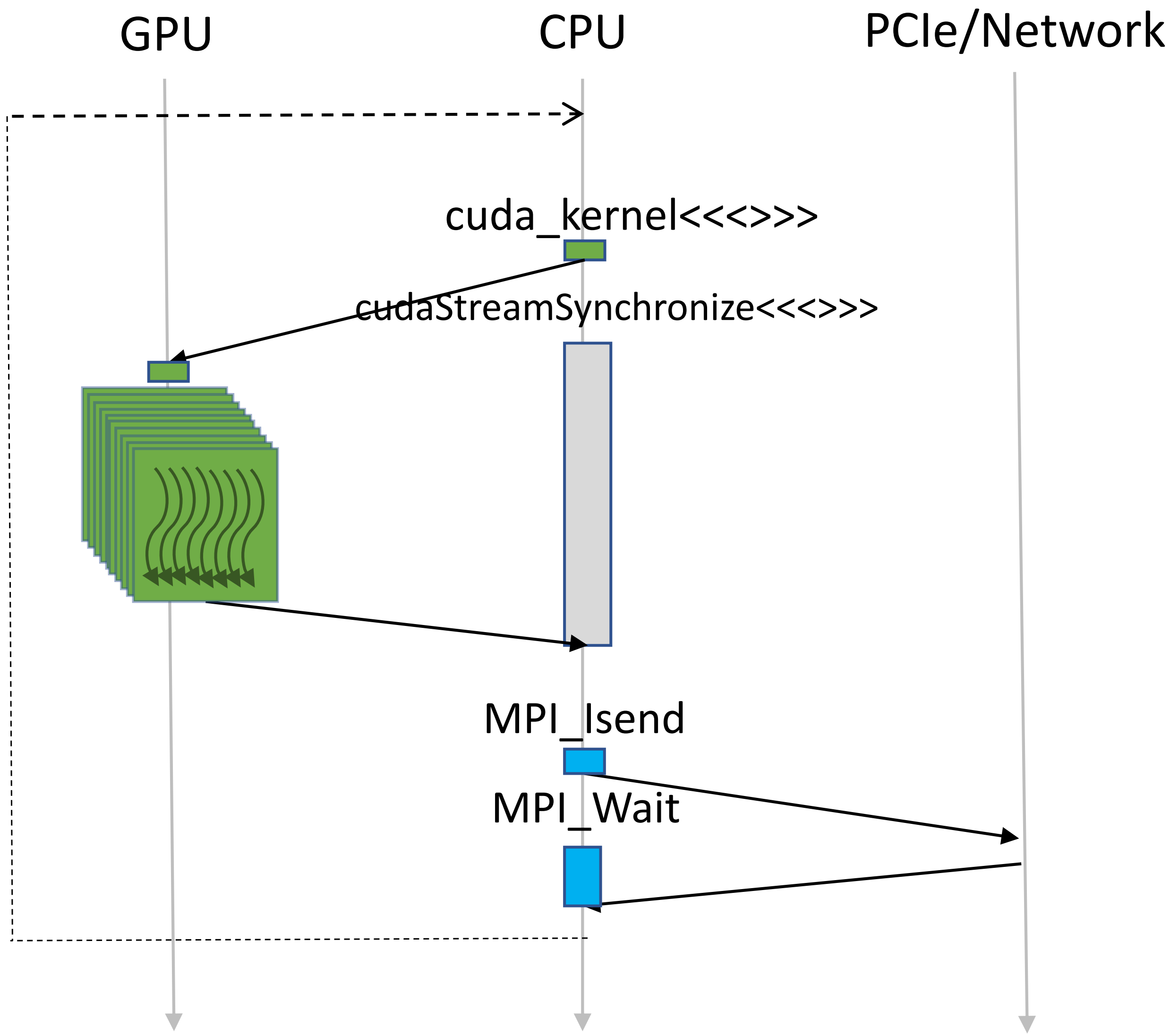
Parallel Programing is Hard

Can we simplify communication instructions ?

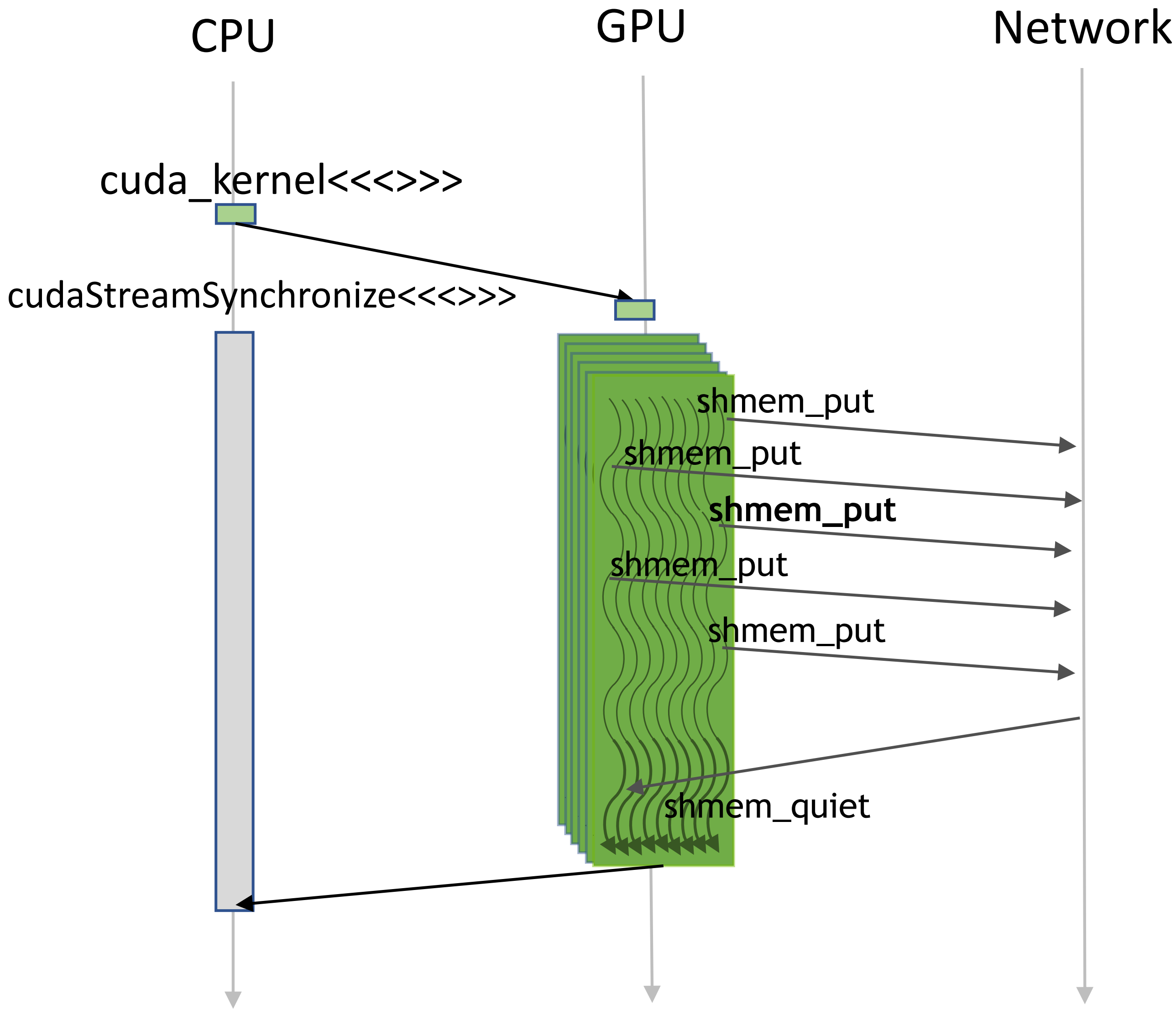
NVSHMEM: Upcoming PGAS system by NVIDIA



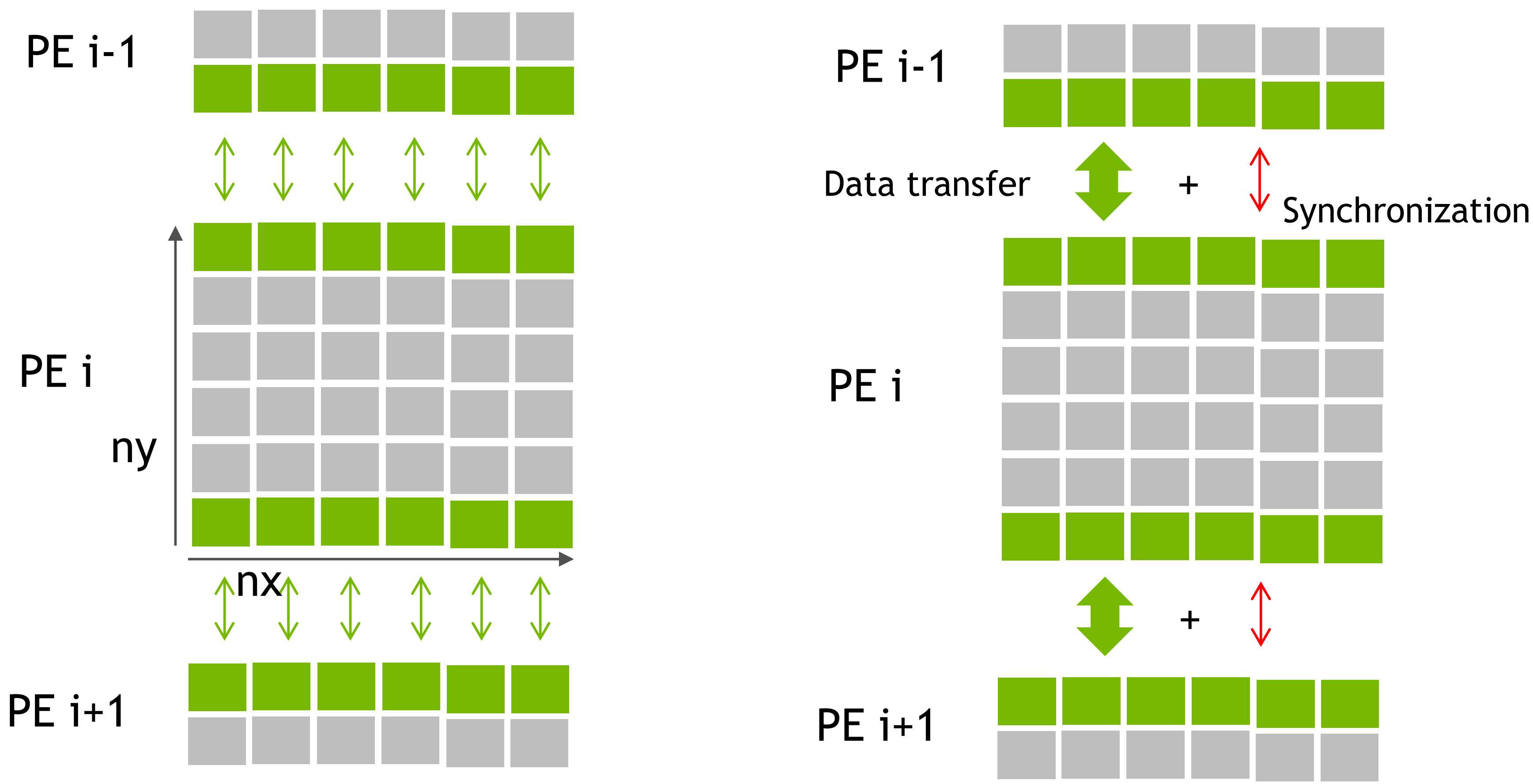
NVSHMEM: CPU based Communication



NVSHMEM : CPU based Communication



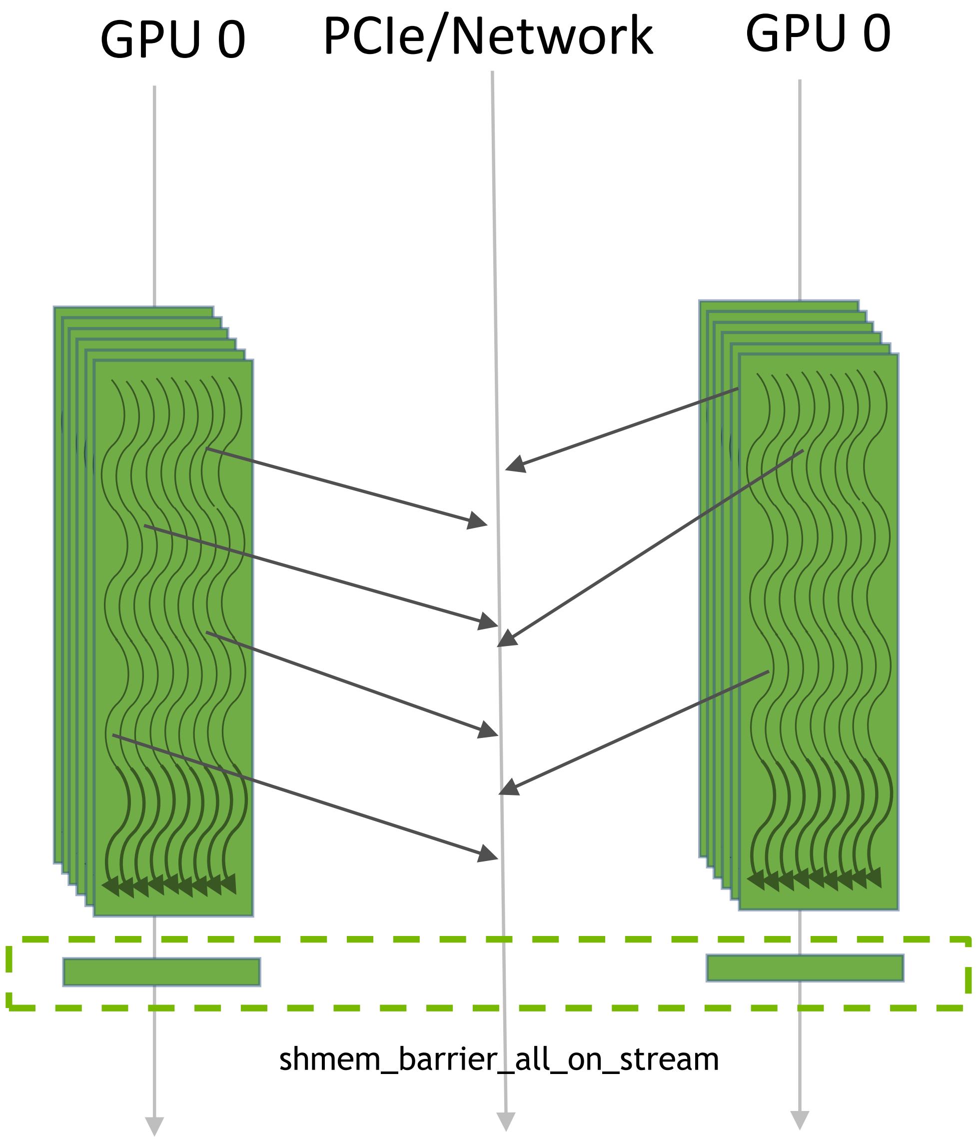
Device Initiated Communication/Synchronization



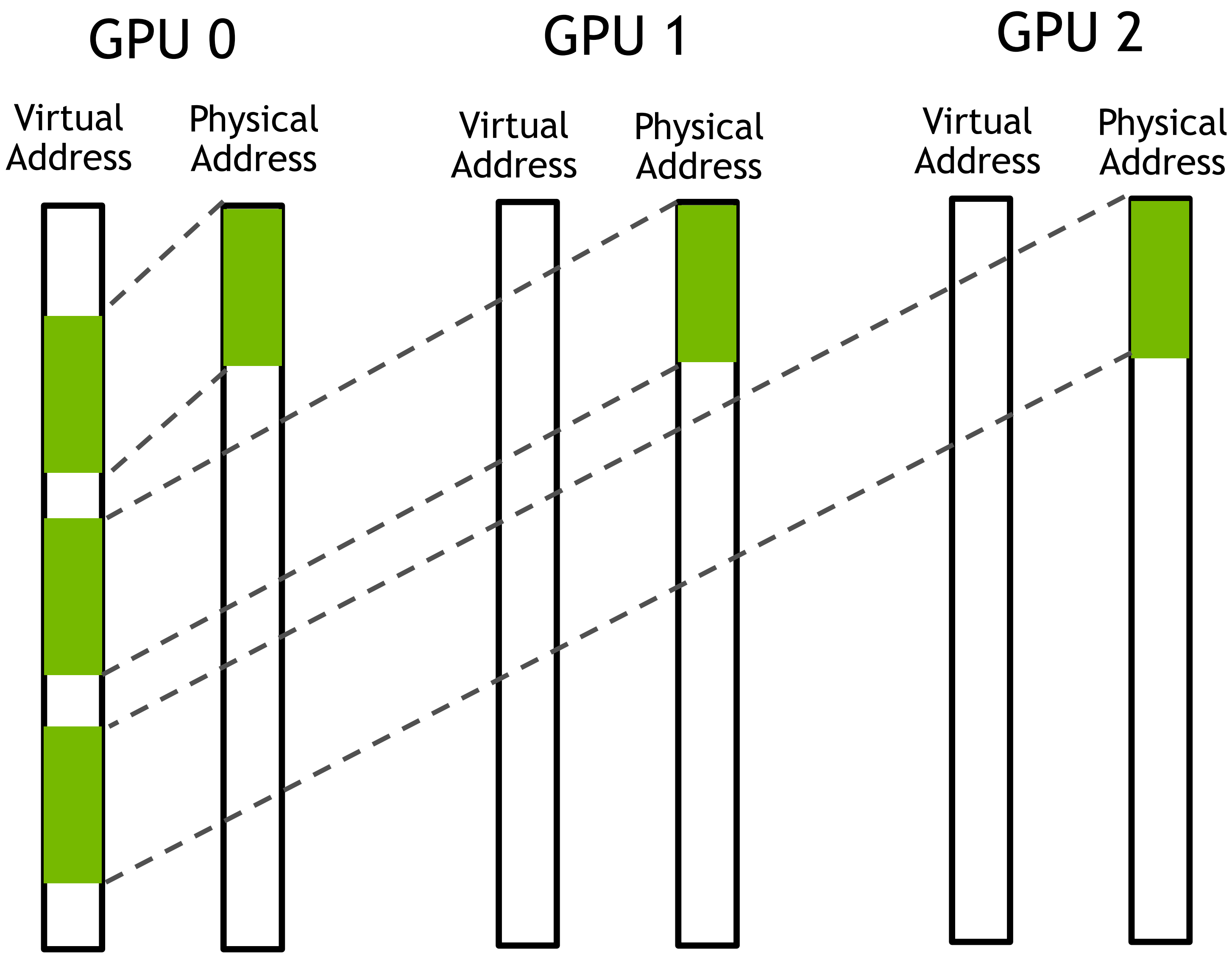
Device Initiated Communication

```
__global__ void stencil_single_step (float *u, ...) {
int ix = threadIdx.x, iy = threadIdx.y;
//compute
//data exchange
if (iy == ny) {
  shmem_float_p (u + ny*nx + ix, u + ix, top_pe);
}
if (iy == 1) {
  shmem_float_p (u + nx + ix, u+(ny+1)*nx + ix,bottom_pe);
}

shmem_barrier_all();
}
```

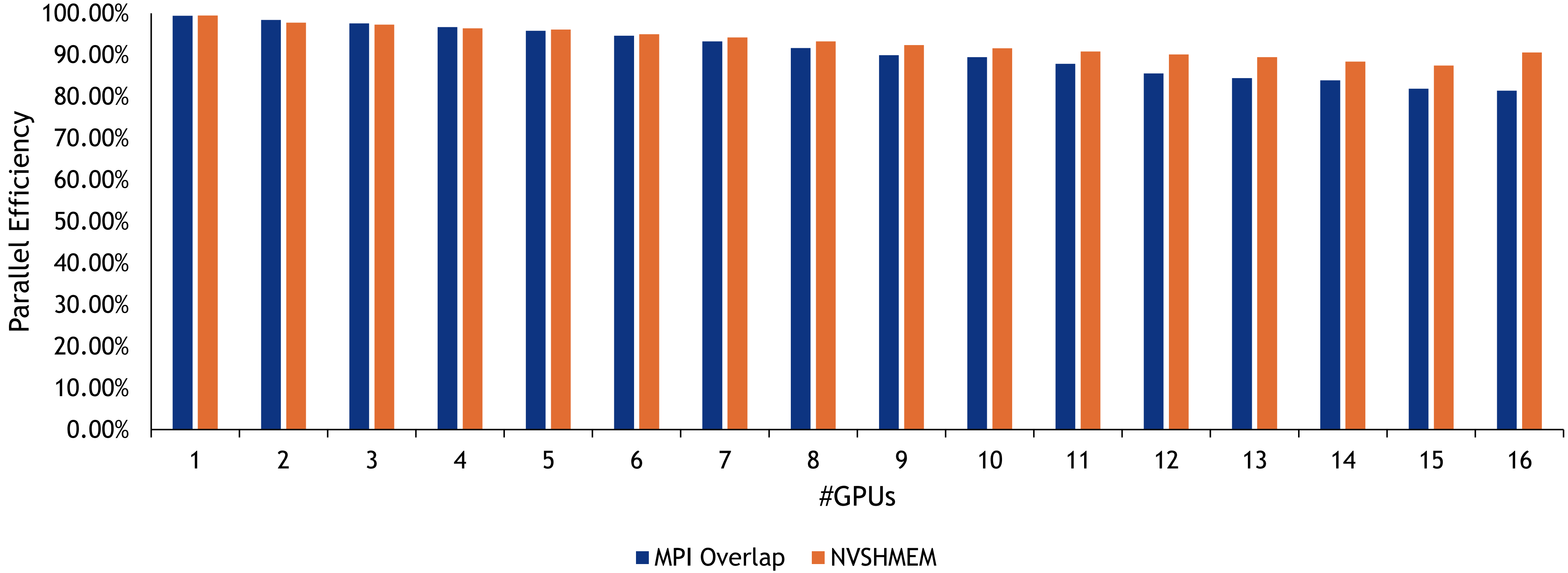
Stream Ordered Operations: avoid costly Sync



Virtual Intra-Node Address Space



NVSHMEM : Performance (NVIDIA numbers)



Benchmarksetup: DGX-2 with OS 4.0.5, GCC 7.3.0, CUDA 10.0 with 410.104 Driver, CUB 1.8.0, CUDA-aware OpenMPI 4.0.0, NVSHMEM EA2 (0.2.3), GPUs@1597Mhz AC, Reported Runtime is the minimum of 5 repetitions

UPC: a PGAS Language

PGAS programing model avoids the complexity of message passing
Many possible performance traps in PGAS implementation

<u>We have some experience with Unified Parallel C (UPC)</u>

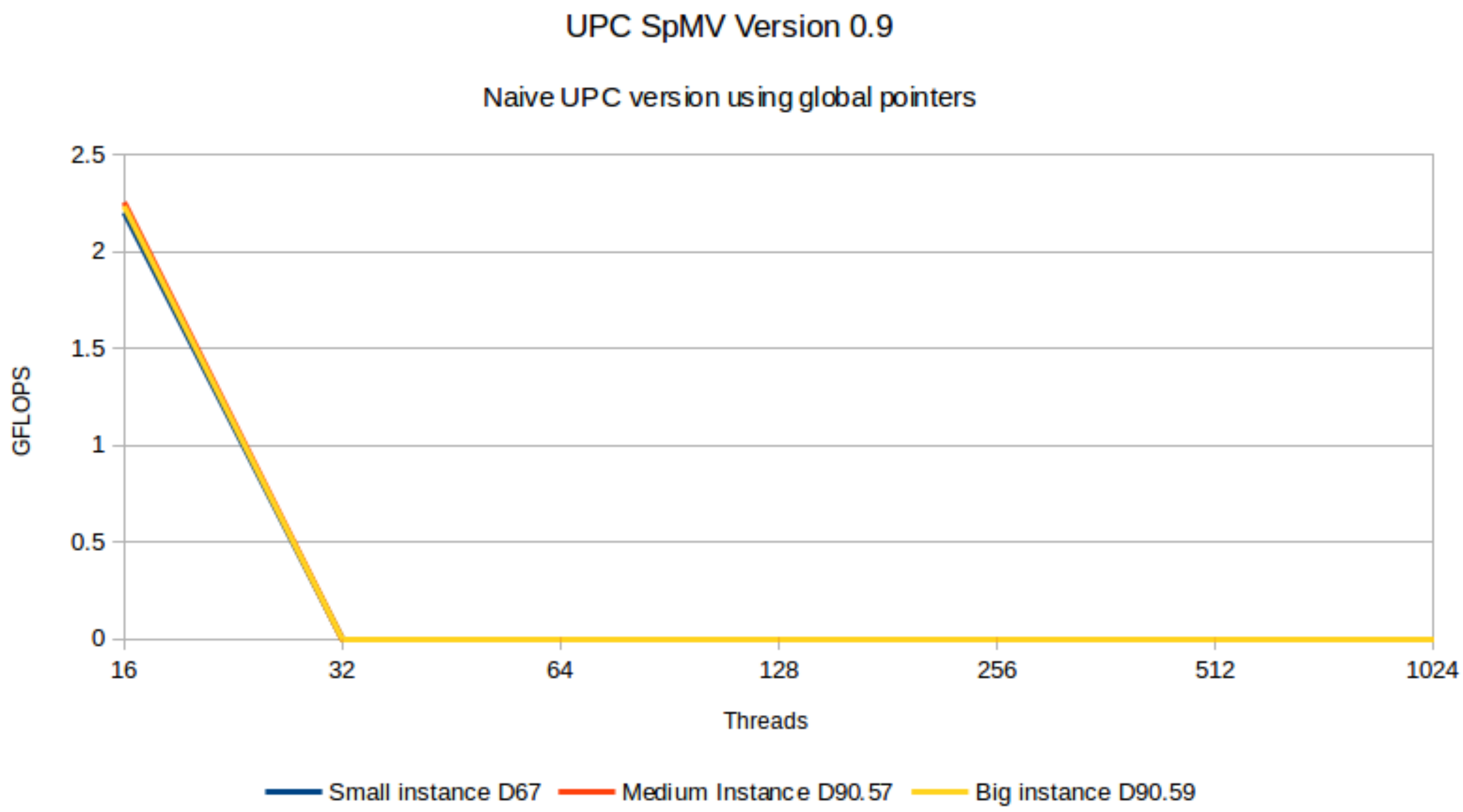
<u>UPC Basics:</u>

- Declare shared arrays
- Communication is generated automatically
- Test problem: Sparse Matrix-Vector from last lecture

ELL SpMV Implementation in UPC

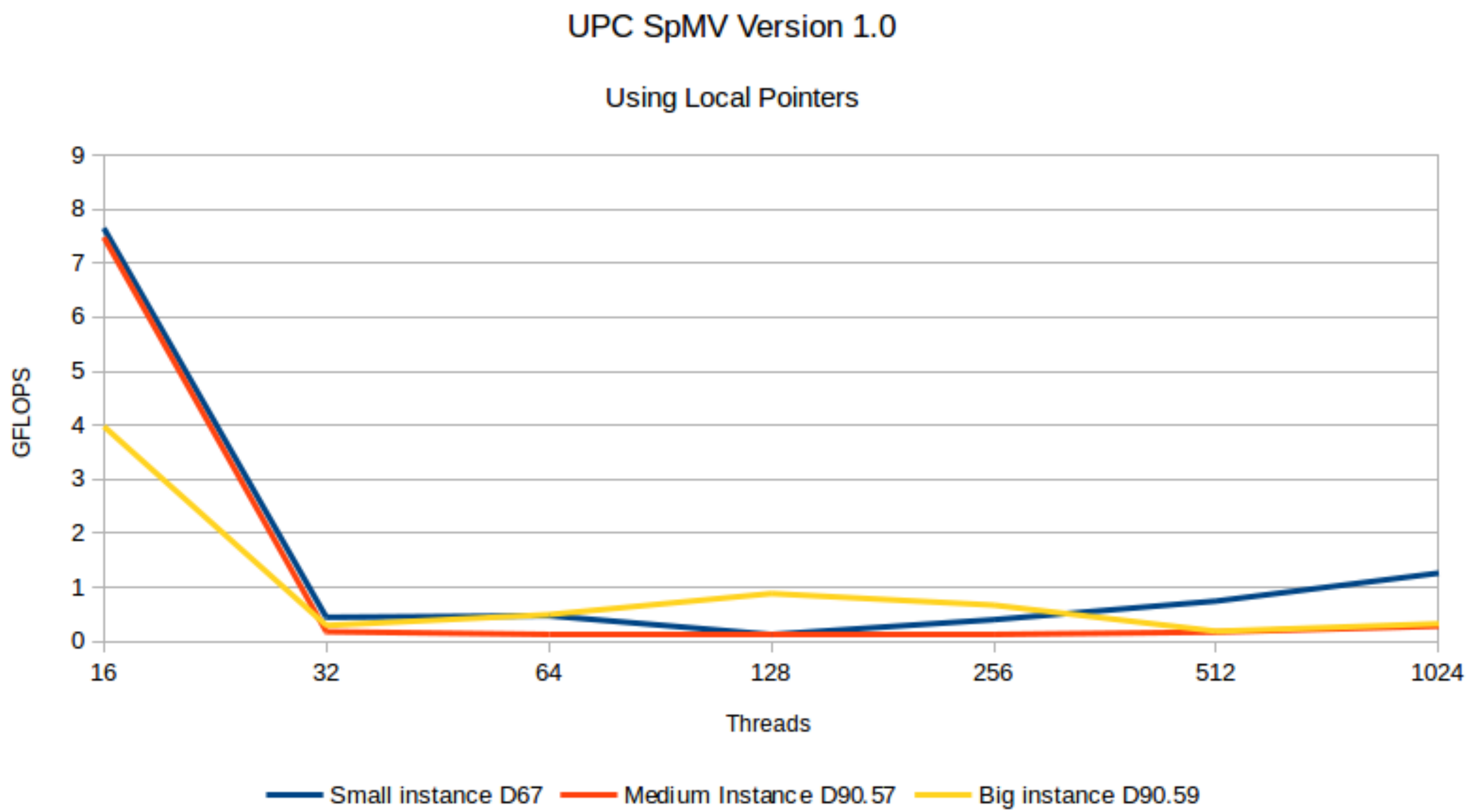
```
shared [BLOCKSIZE] double* Voltage;
shared [BLOCKSIZE] double* NewVoltage;

Voltage = (shared [BLOCKSIZE] double*)
NewVoltage = (shared [BLOCKSIZE] double*)

upc_all_alloc(n_blocks[0],sizeof(double)*BLOCKSIZE);
upc_all_alloc(n_blocks[0],sizeof(double)*BLOCKSIZE);

//Update cell k
NewVoltage[k] = D[k]* Voltage[k]
for(i=0;i<16;i++)
{
       NewVoltage[k] += A[k*16+i]*Voltage[I[k*16+i]];
}
```

UPC: Straightforward Implementation



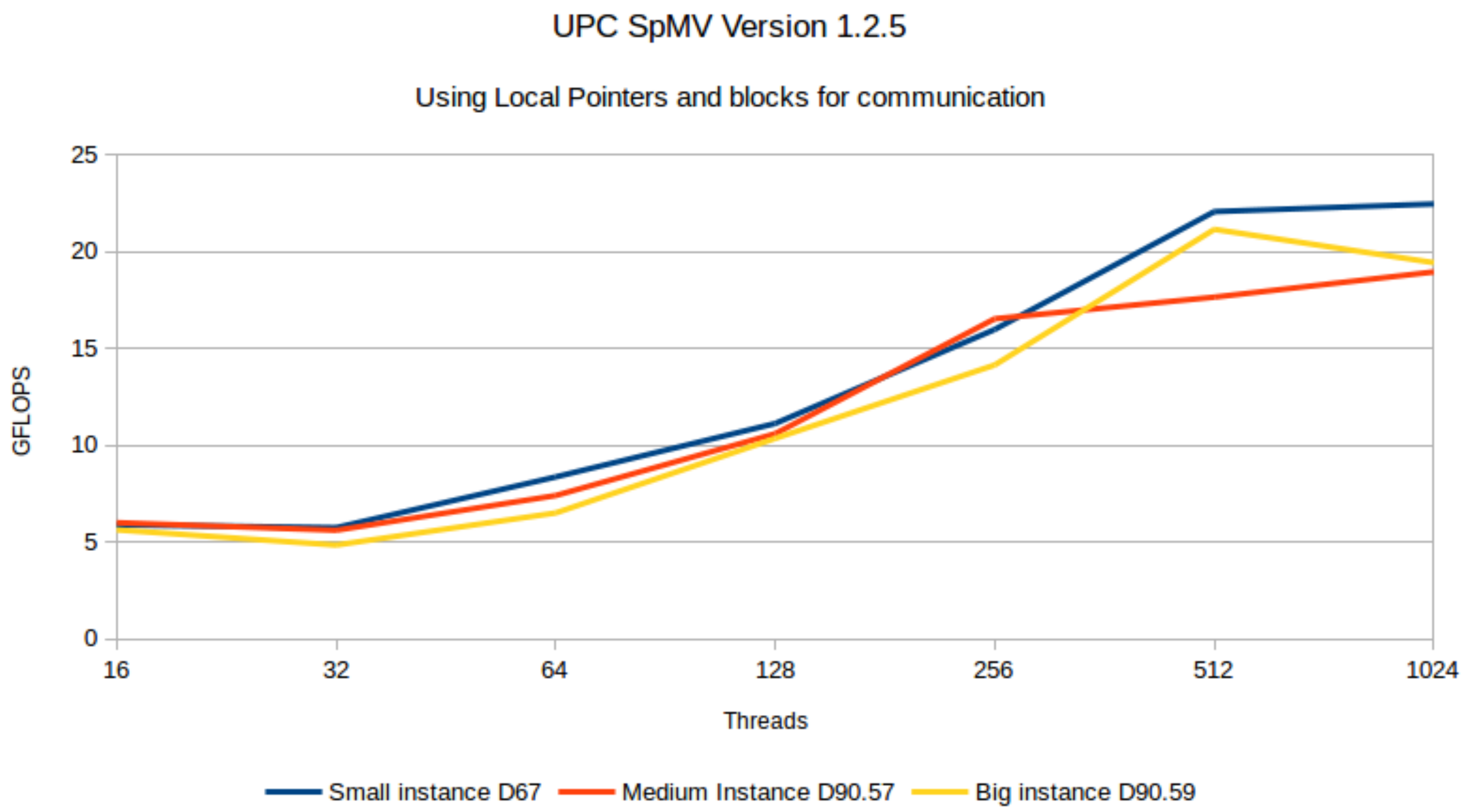
Every single memory access must be checked

UPC: Local Pointers



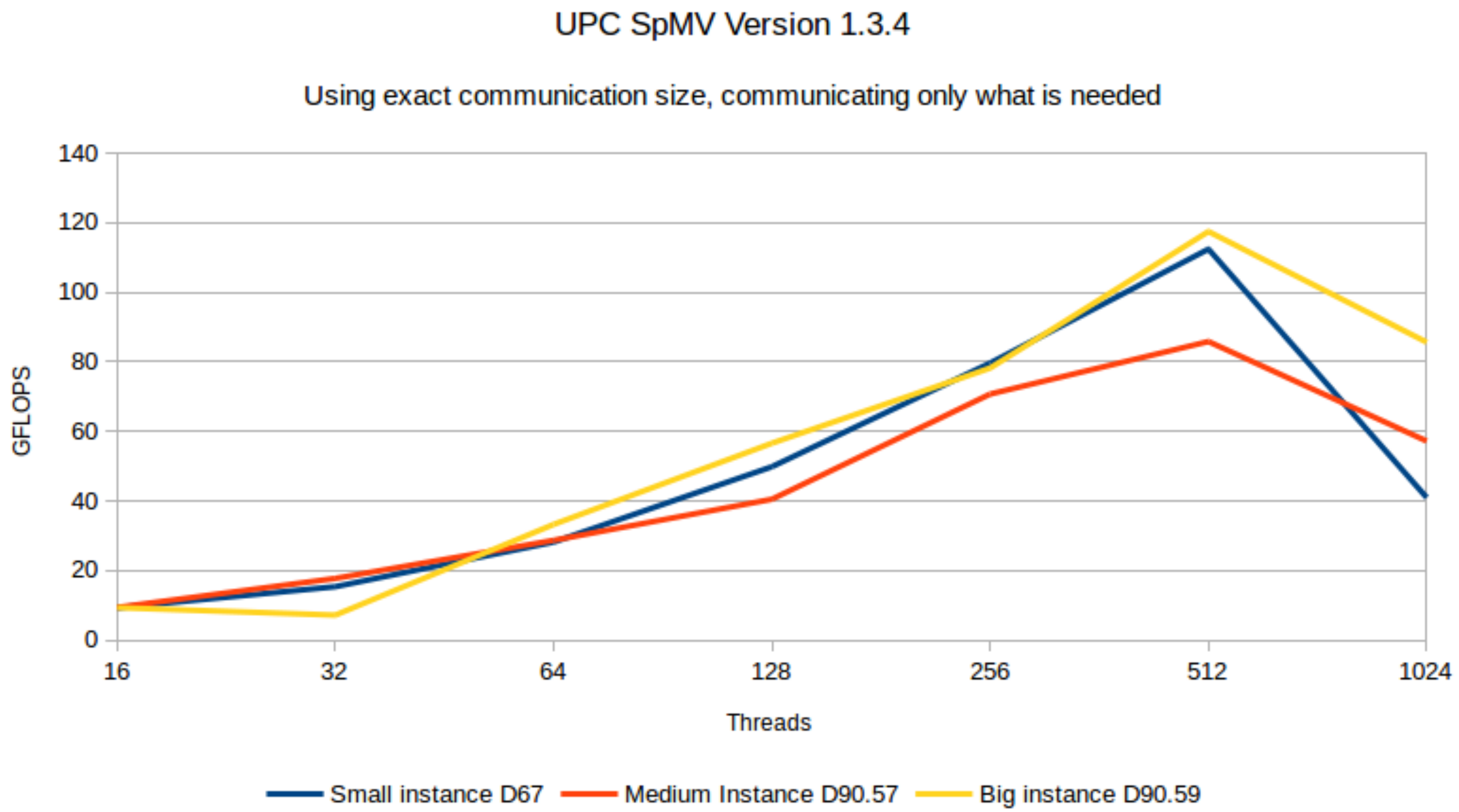
<u>Communicates with very small messages</u>

UPC: Block Transfers



<u>Better, but still way too much communication</u>

UPC: Optimized Communication



Back to the MPI version, but with vector replication

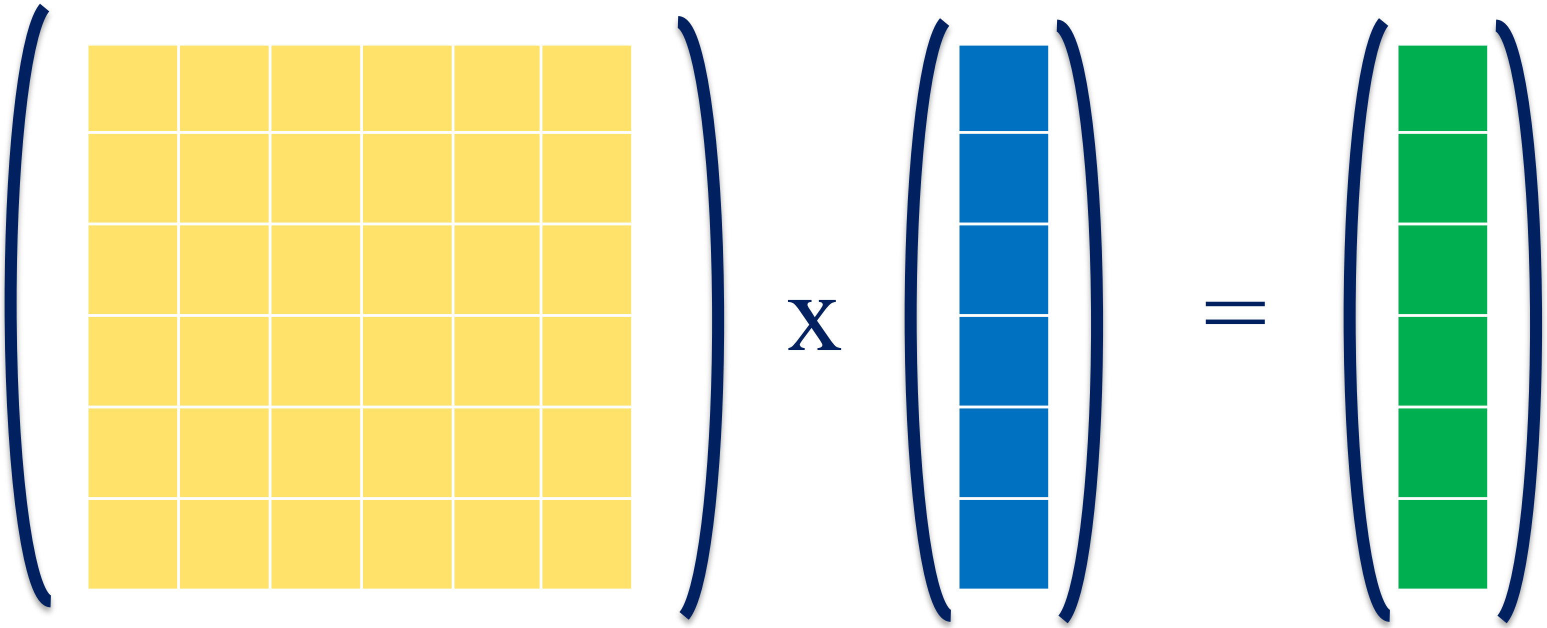
UPC: Lessons Learned

- Implicit communication does not work well
- Compiler does not aggregate transfers
- Program not knowing if a variable is local carries an additional cost
- Maintaining a language is costly
- Lesson learned: shift focus to one-sided messaging, RMA, RPC

<u>What to do with such a system ?</u>

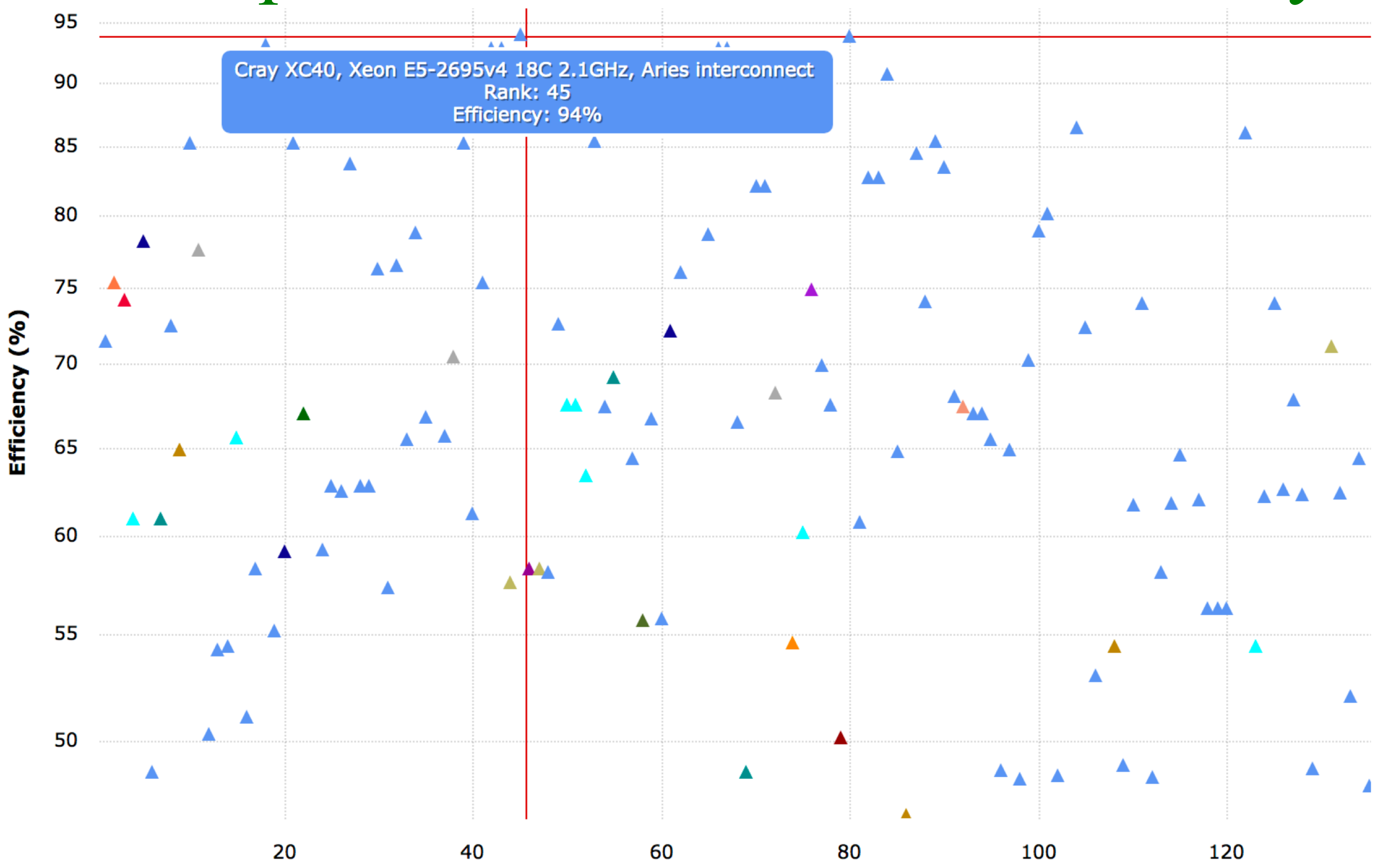
<u>Lets look at the communication patterns!</u>

Easy Case: Dense Matrix Dense Vector



- Common pattern in scientific computing, deep learning, ML
- Data access pattern completely regular
- Batches, tiling, blocking, etc. to run from cache
- Often compute bound
- Balanced communication pattern for distributed memory

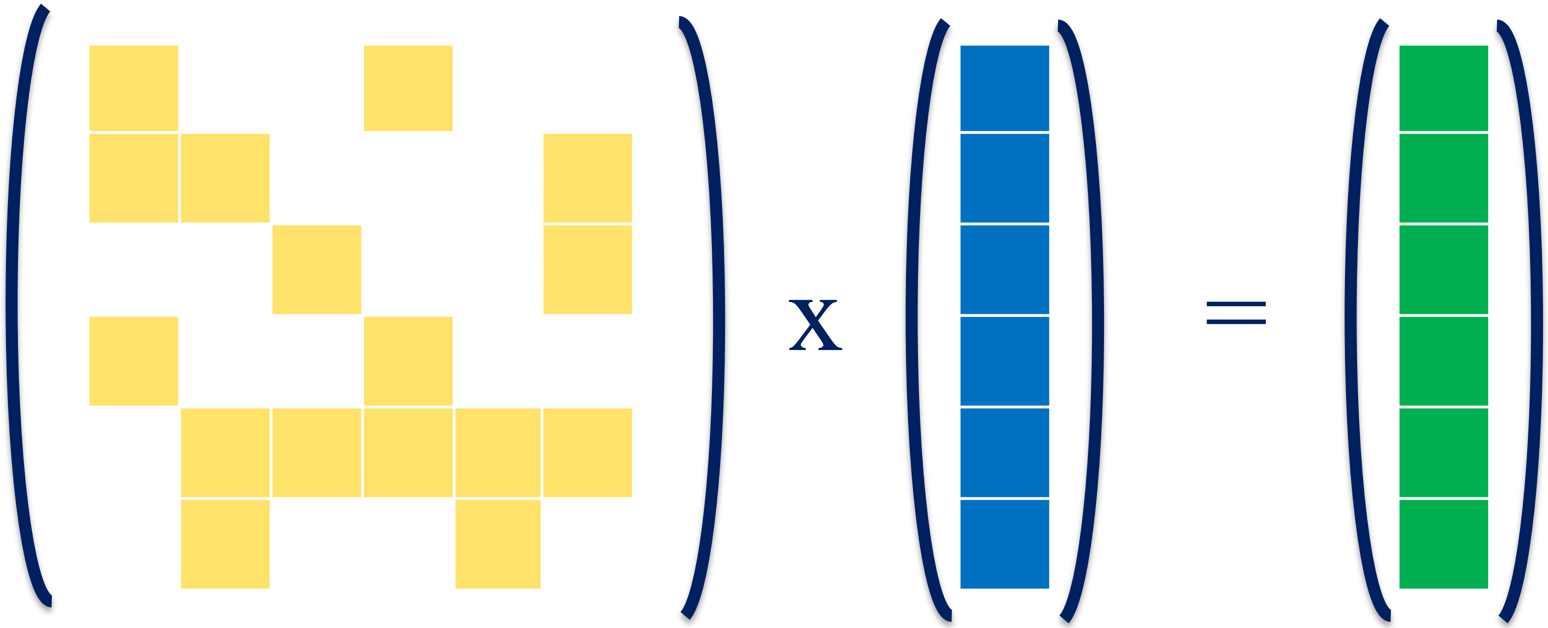
Top 500: Dense Problems are “Easy”



NVIDIA Tesla
Intel Xeon

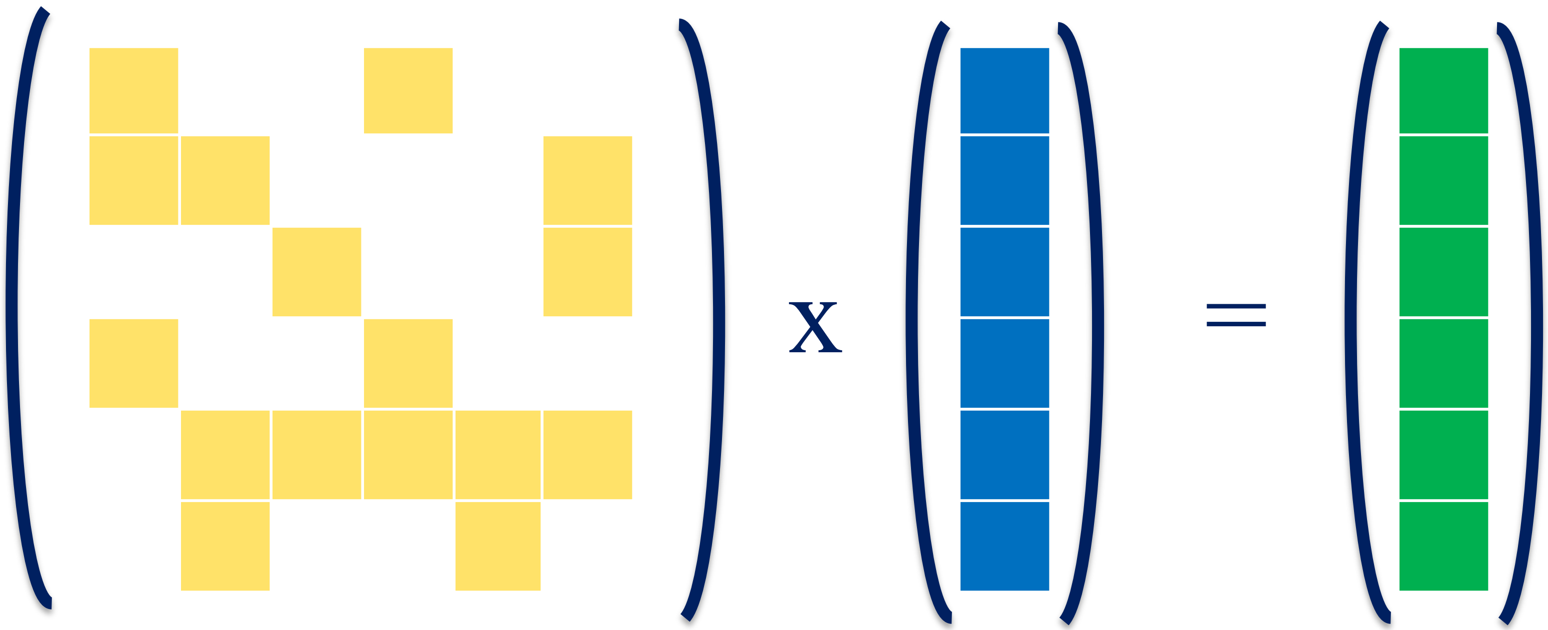
Xeon processors with fast networks reach > 90% efficiency
GPUs have lower efficiencies but higher FLOPS/Watt

Intermediate Case: Sparse Matrix Dense Vector



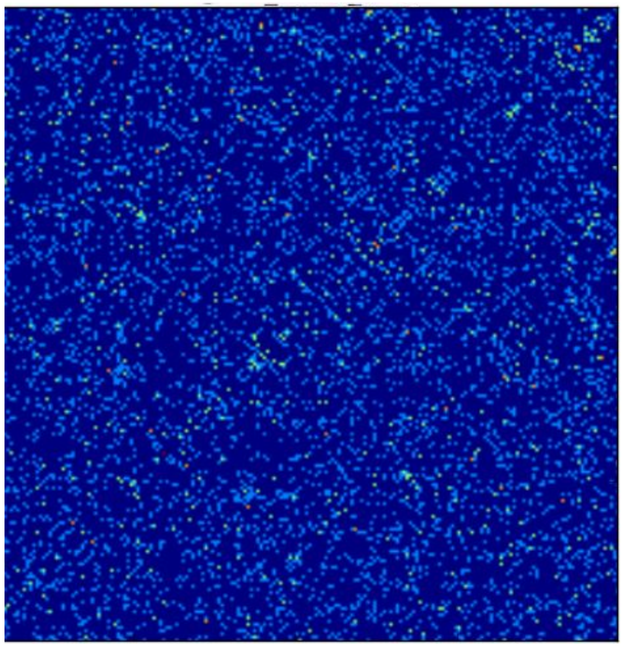
- Unstructured meshes in scientific computing, PageRank
- Data access pattern irregular but static
- Reordering techniques improve cache usage
- Typically memory bandwidth bound
- Unbalanced communication pattern in distributed memory

Intermediate Case: Sparse Matrix Dense Vector

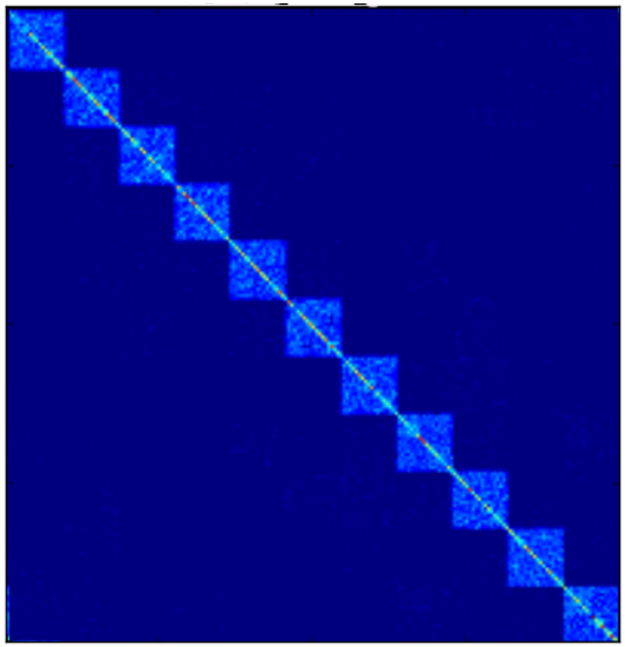


- Unstructured meshes in scientific computing, PageRank
- Data access pattern irregular but static
- Reordering techniques improve cache usage
- Typically memory bandwidth bound
- Unbalanced communication pattern in distributed memory

Sparse Matrix Dense Vector: Problem Structure Matters



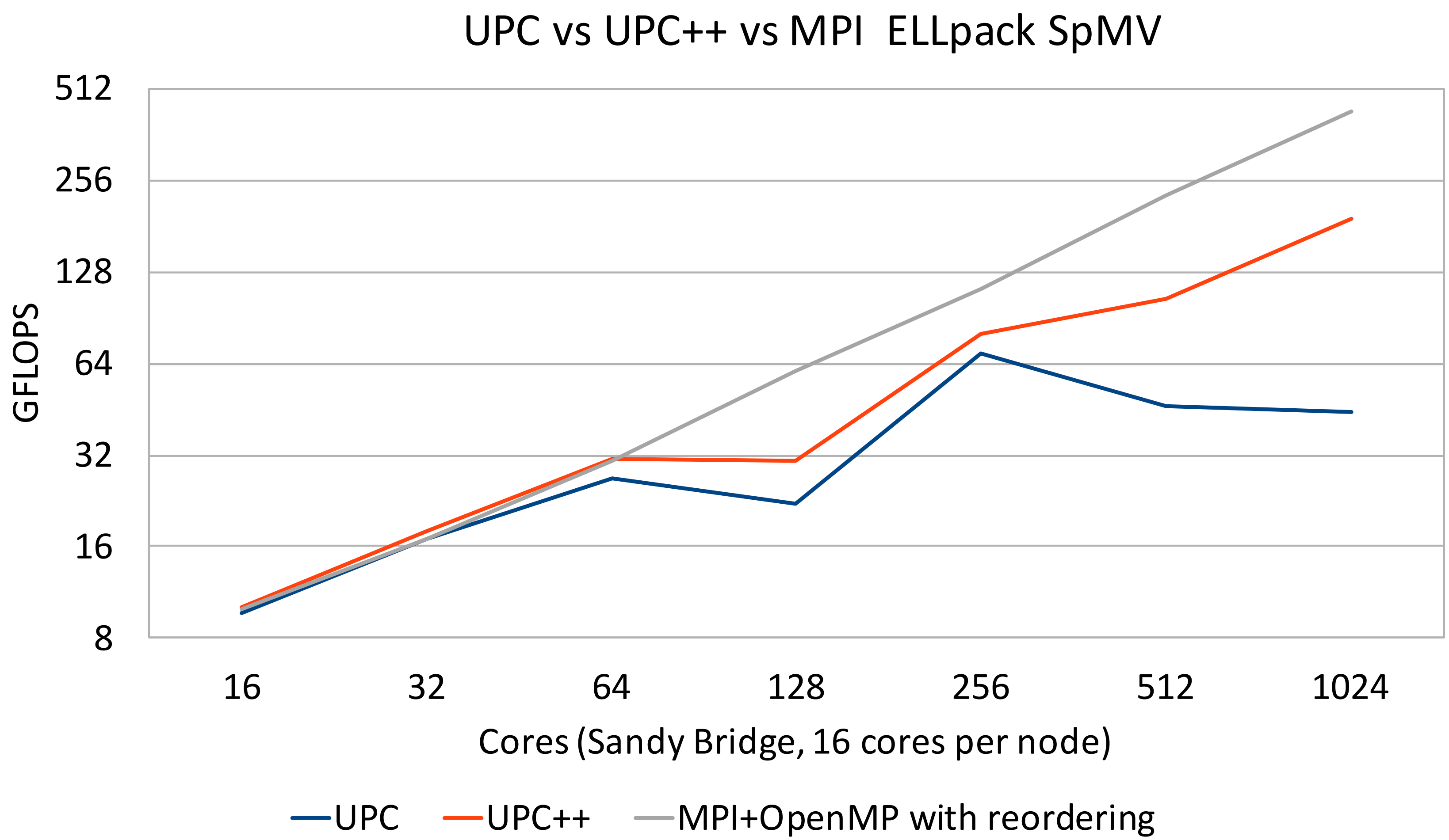
Weak ordering –
Low performance



Strong ordering –
High performance

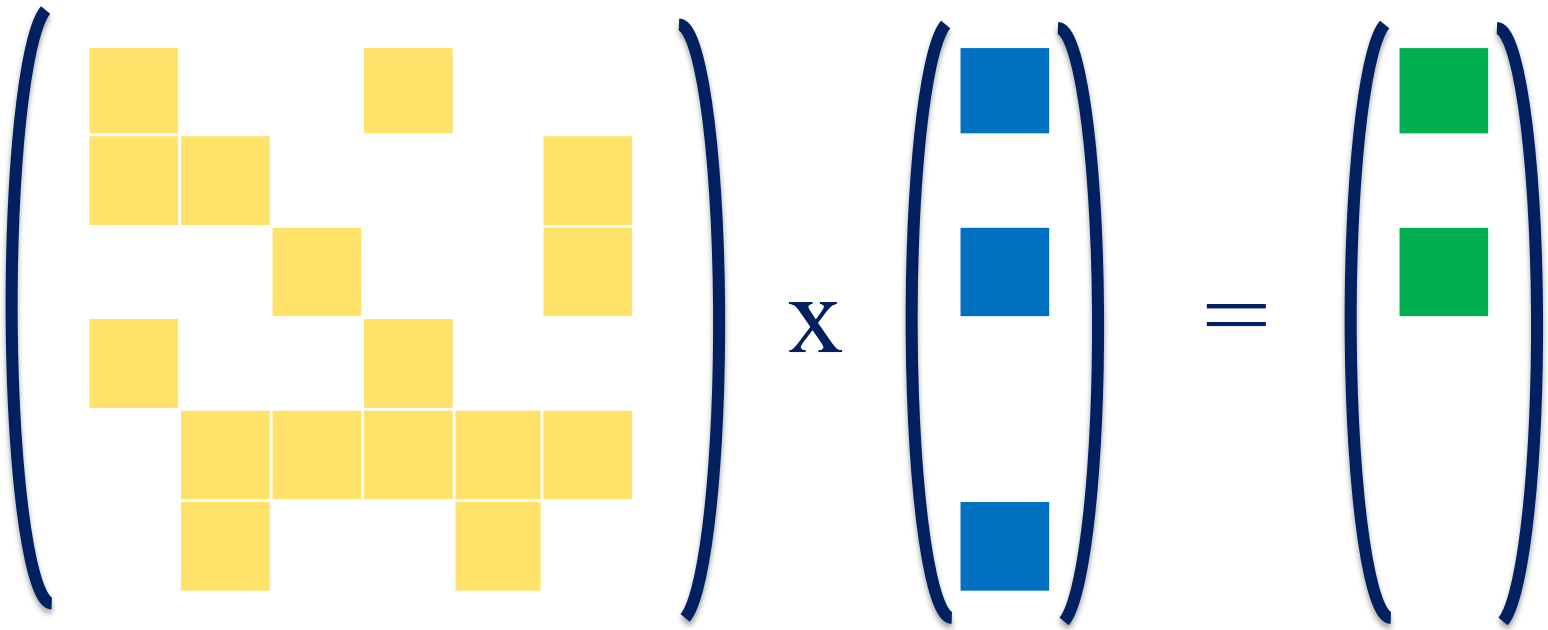
Strong ordering complicates programing
Can we increase productivity ?

Sparse Matrix Dense Vector: Problem Structure Matters



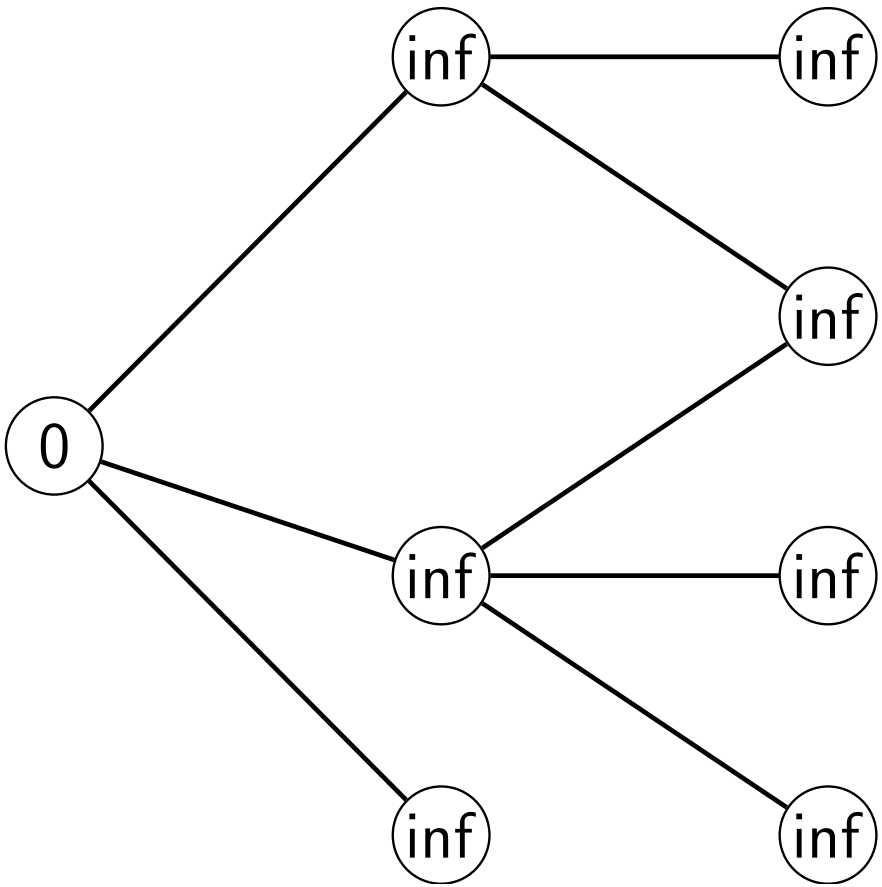
- Large messages, message size matters
- Algorithm helps, at the cost of productivity
- Communication system does not (at bandwidth baseline)

Challenging Case: Sparse Matrix Sparse Vector



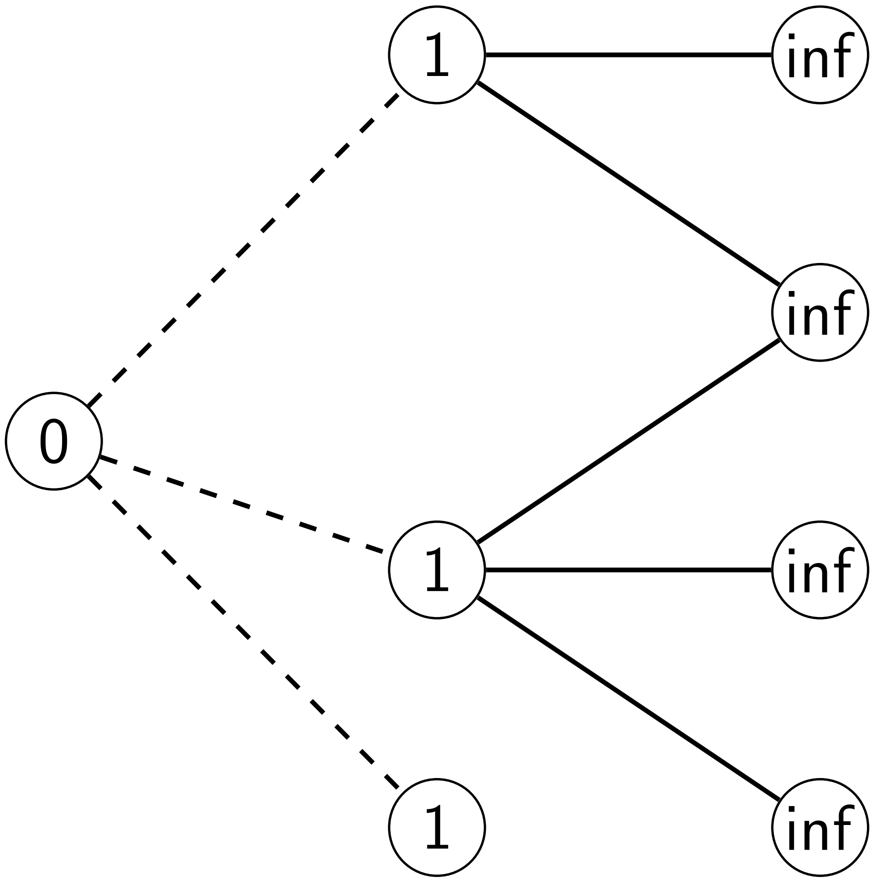
- Graph algorithms, GNNs, data dependent computation paths
- Data access pattern irregular and dynamic
- Possibility of cache reuse is questionable
- Often latency bound
- Unbalanced communication pattern in distributed memory

The Most Basic Graph Algorithm: BFS

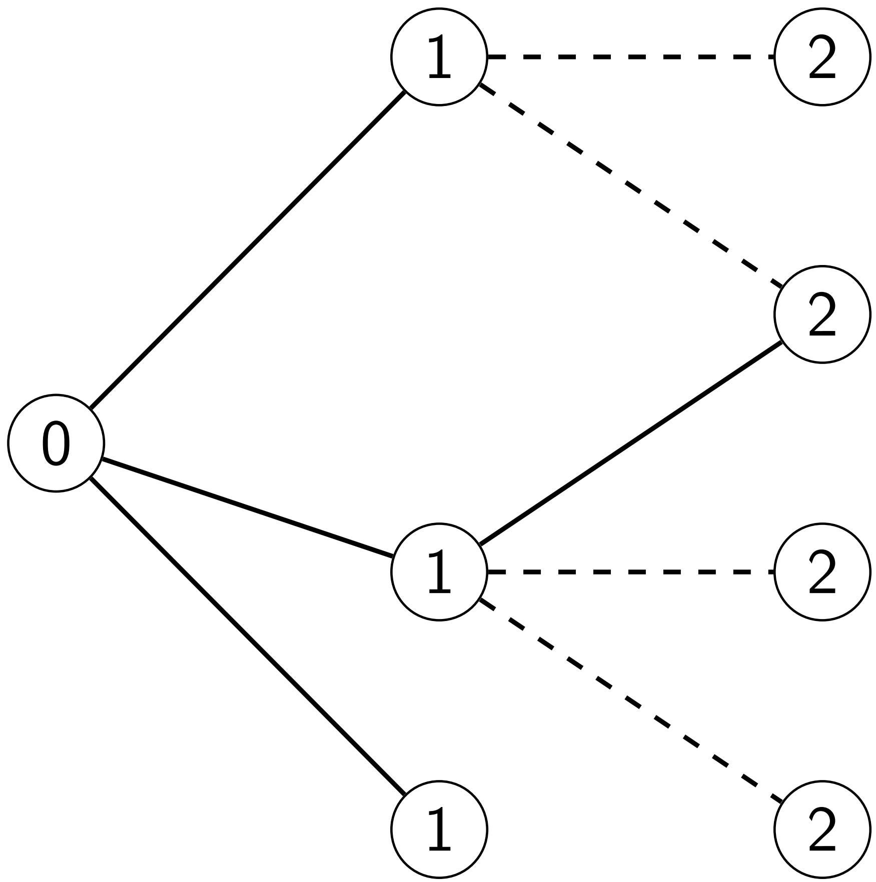


- Basic kernel of the Graph500
- Sequential algorithm trivial

Challenges of Parallel BFS

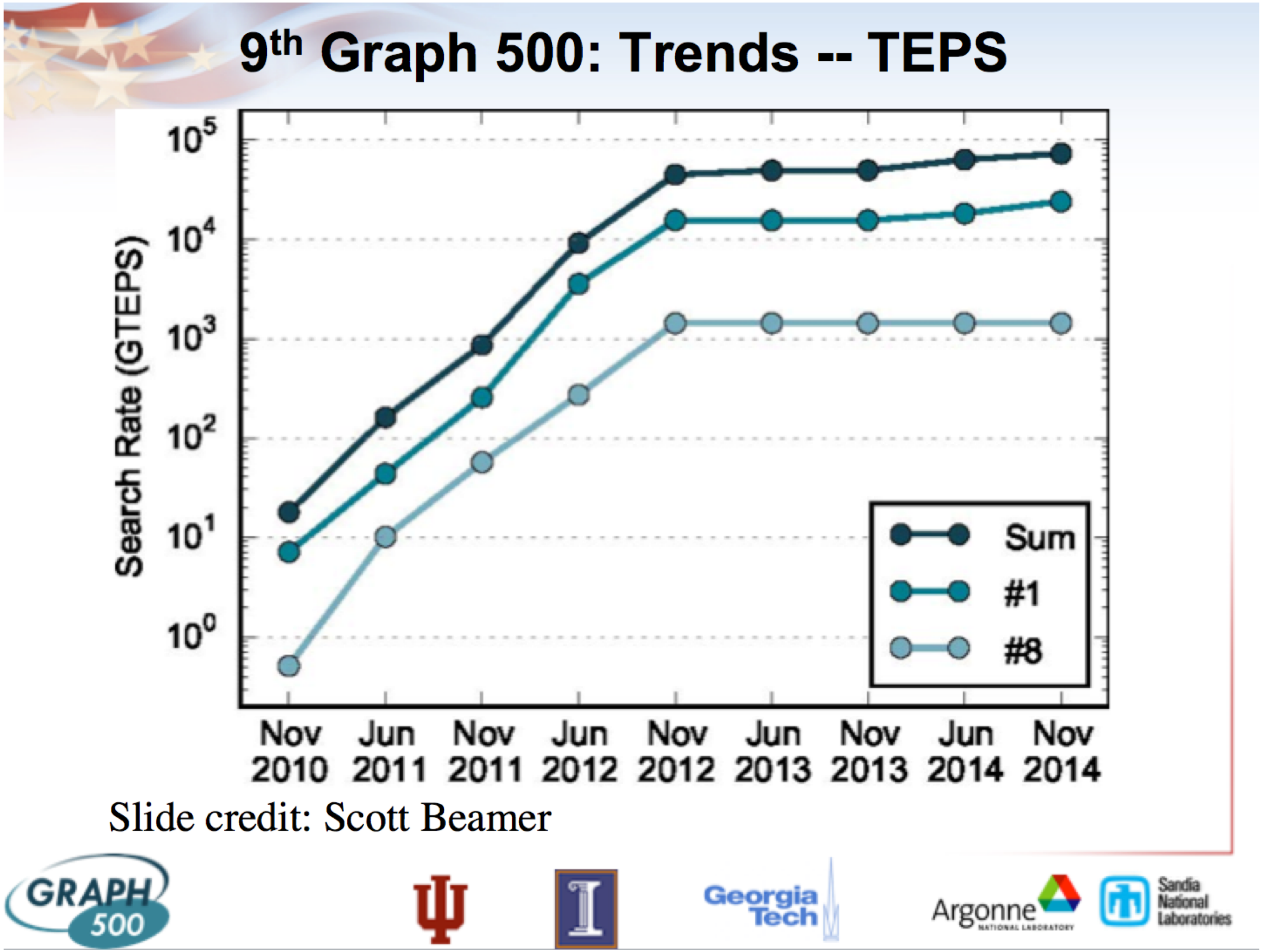


Challenges of Parallel BFS can be Overcome



- Communication pattern, volume changes every round
- Rounds impose clear structure on algorithm
- Heavy all-to-all for small diameter graphs
- Number of rounds bounded by graph diameter
- Ultimately, BFS is a simple graph problem

BFS: Successful Parallelization



Slide credit: Scott Beamer

Why are Parallel Graph Algorithms still Difficult ?

Scalability! But at what COST?

Frank McSherry Unaffiliated
Michael Isard Unaffiliated*
Derek G. Murray Unaffiliated†

Abstract

We offer a new metric for big data platforms, COST, or the Configuration that Outperforms a Single Thread. The COST of a given platform for a given problem is the hardware configuration required before the platform outperforms a competent single-threaded implementation. COST weighs a system's scalability against the overheads introduced by the system, and indicates the actual performance gains of the system, without rewarding systems that bring substantial but parallelizable overheads.

We survey measurements of data-parallel systems recently reported in SOSP and OSDI, and find that many systems have either a surprisingly large COST, often hundreds of cores, or simply underperform one thread for all of their reported configurations.

1 Introduction

> "You can have a second computer once you've shown you know how to use the first one."
>
> –Paul Barham

The published work on big data systems has fetishized scalability as the most important feature of a distributed data processing platform. While nearly all such publications detail their system's impressive scalability, few directly evaluate their absolute performance against reasonable benchmarks. To what degree are these systems truly improving performance, as opposed to parallelizing overheads that they themselves introduce?

Contrary to the common wisdom that effective scal-

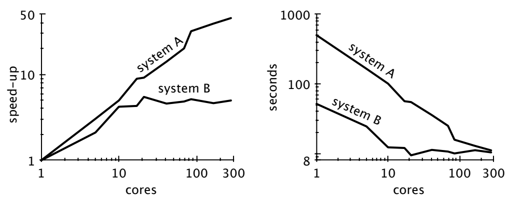


Figure 1: Scaling and performance measurements for a data-parallel algorithm, before (system A) and after (system B) a simple performance optimization. The unoptimized implementation "scales" far better, despite (or rather, because of) its poor performance.

While this may appear to be a contrived example, we will argue that many published big data systems more closely resemble system A than they resemble system B.

1.1 Methodology

In this paper we take several recent graph processing papers from the systems literature and compare their reported performance against simple, single-threaded implementations on the same datasets using a high-end 2014 laptop. Perhaps surprisingly, many published systems have *unbounded* COST—i.e., no configuration outperforms the best single-threaded implementation—for all of the problems to which they have been applied.

The comparisons are neither perfect nor always fair, but the conclusions are sufficiently dramatic that some concern must be raised. In some cases the single-threaded implementations are more than an order of magnitude faster than published results for systems using

Scalability at what COST ?

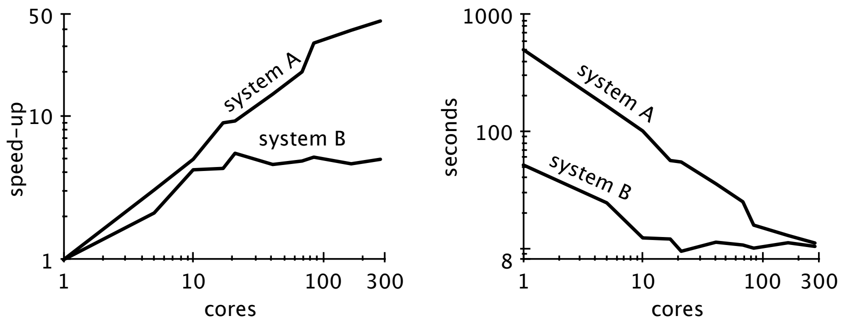


Figure 1: Scaling and performance measurements for a data-parallel algorithm, before (system A) and after (system B) a simple performance optimization. The unoptimized implementation "scales" far better, despite (or rather, because of) its poor performance.

- Most graph algorithms are more difficult than BFS
- Programming models are not designed for graph applications
- Latency cannot be overcome inside the application

Contrary to the common wisdom that effective scaling is evidence of solid systems building, any system can scale arbitrarily well with a sufficient *lack* of care in its implementation.

References

Bell, N., & Garland, M. (2009, November). Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis* (pp. 1-11).

Merrill, D., & Garland, M. (2016). Merge-based sparse matrix-vector multiplication (SpMV) using the CSR storage format. *ACM SIGPLAN Notices*, *51*(8), 1-2.

Langguth, J., Wu, N., Chai, J., & Cai, X. (2015). Parallel performance modeling of irregular applications in cell-centered finite volume methods over unstructured tetrahedral meshes. *Journal of Parallel and Distributed Computing*, *76*, 120-131.

McSherry, F., Isard, M., & Murray, D. G. (2015). Scalability! But at what {COST}?. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*.

Awan, A. A., Hamidouche, K., Venkatesh, A., & Panda, D. K. (2016, September). Efficient large message broadcast using NCCL and CUDA-aware MPI for deep learning. In *Proceedings of the 23rd European MPI Users' Group Meeting* (pp. 15-22).

Credit: Lecture contains NVIDIA material available at https://developer.nvidia.com/cuda-zone
Image source: wikipedia.org, graph500.org, ornl.gov, mvapich.cse.ohio-state.edu
Contains material from ACACES 2018 summer school, originally designed by Scott Baden