Heterogeneous Programming for Finite Element Methods:

Insights from benchmarks

Igor Baratta, Chris Richardson and Garth N. Wells
Department of Engineering

Outline

Why heterogeneous computing?

Finite Element Overview

Benchmarks

Results

Where to go from here?

Compute variety at scale

TOP 500
The List.

Supercomputer Fugaku	Summit	Sierra	Sunway TaihuLight	Perlmutter
				
A64FX 48C 2.2GHz	IBM POWER9 22C 3.07GHz NVIDIA Volta GV100	IBM POWER9 22C 3.1GHz NVIDIA Volta GV100	Sunway SW26010 260C 1.45GHz, Sunway	AMD EPYC 7763 64C 2.45GHz NVIDIA A100 SXM4 40 GB
Rmax (TFlop/s) 442,010.0	Rmax (TFlop/s) 148,600.0	Rmax (TFlop/s) 94,640.0	Rmax (TFlop/s) 93,014.6	Rmax (TFlop/s) 70,870.0

Future exascale systems

System	Delivery	CPU	Accelerator
Tianhe-3	2021	Phytium	Phytium
Frontier	2021?	AMD	NVIDIA
Aurora	2022	Intel	Intel
El Capitan	2023	AMD	AMD

- CPU + Accelerator
- Different vendors
- Complex software stack

TIER 2 HPC

CSD3 is one Tier 2 National HPC Services hosted by the Research Computing Services at the University of Cambridge.

CPU Nodes

Cascade Lake - 2x Intel(R) Xeon(R) Platinum 8276

Ice Lake - 2x Intel(R) Xeon(R) Platinum 8368Q

GPU Nodes

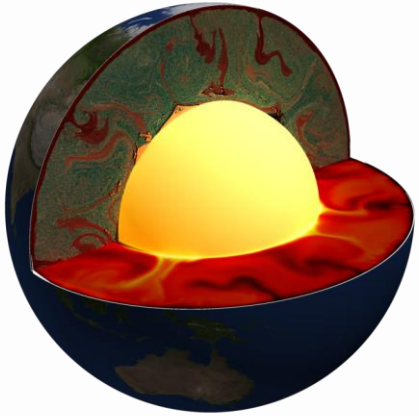
1x Intel Xeon E5-2650 12-Core + 4x NVIDIA P100 16GB

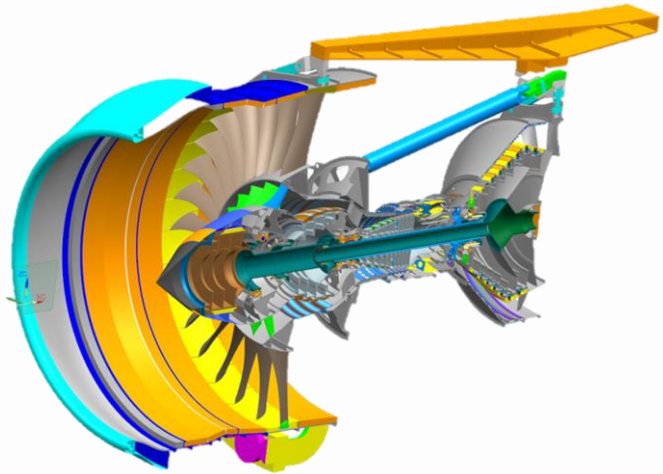
2x AMD EPYC 7763 64-Core + 4x NVIDIA A100-SXM-80GB GPUs

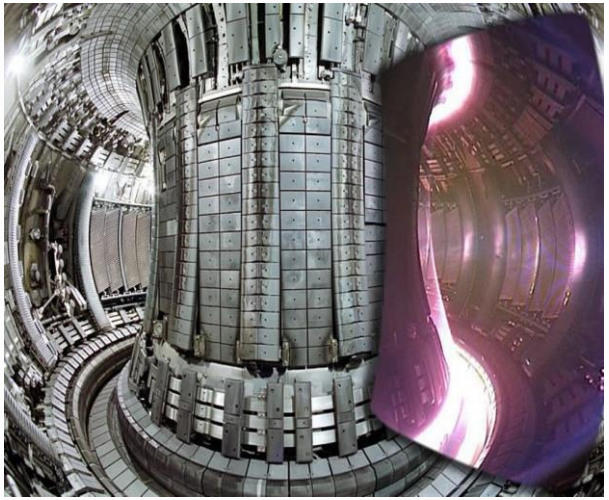
\+ Fujitsu A64FX, AMD MI100, Intel GPUs

Finite element overview

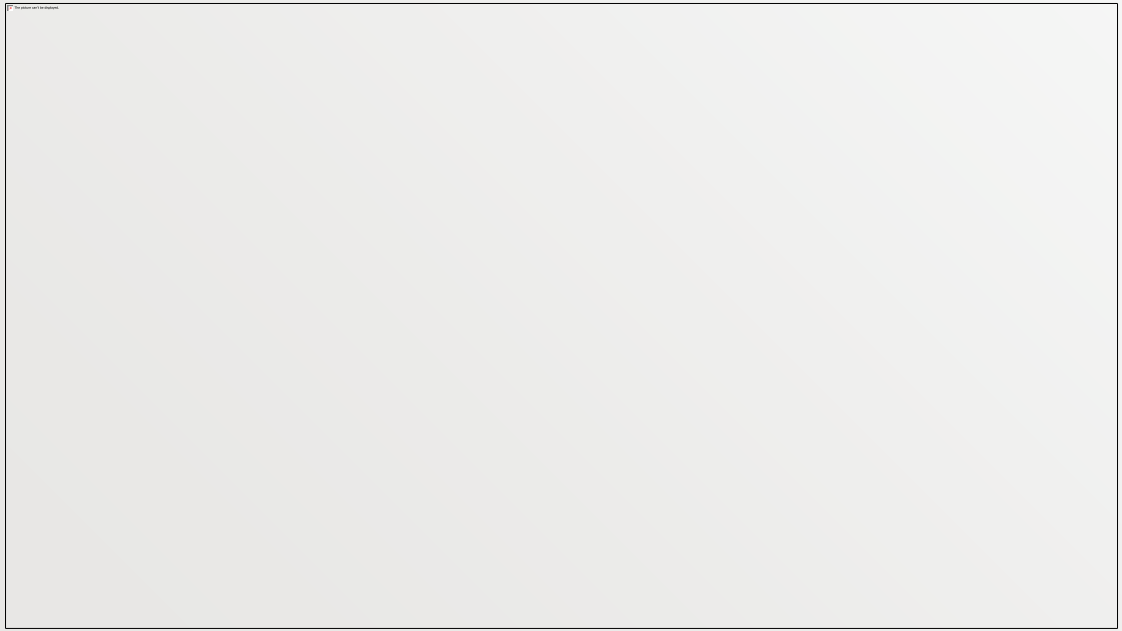
FENICS PROJECT







Finite element overview



Linear Algebra Backend

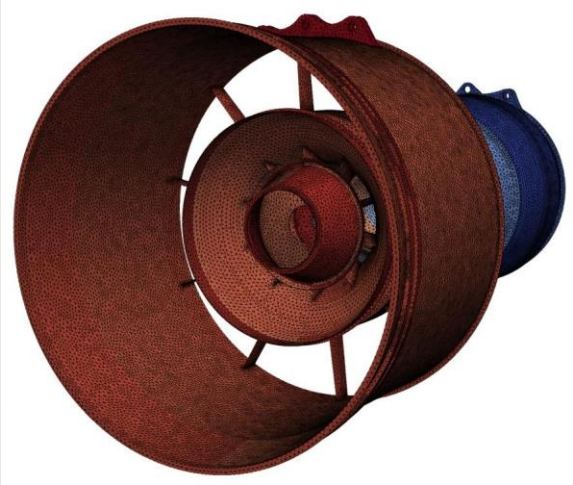
PETSc

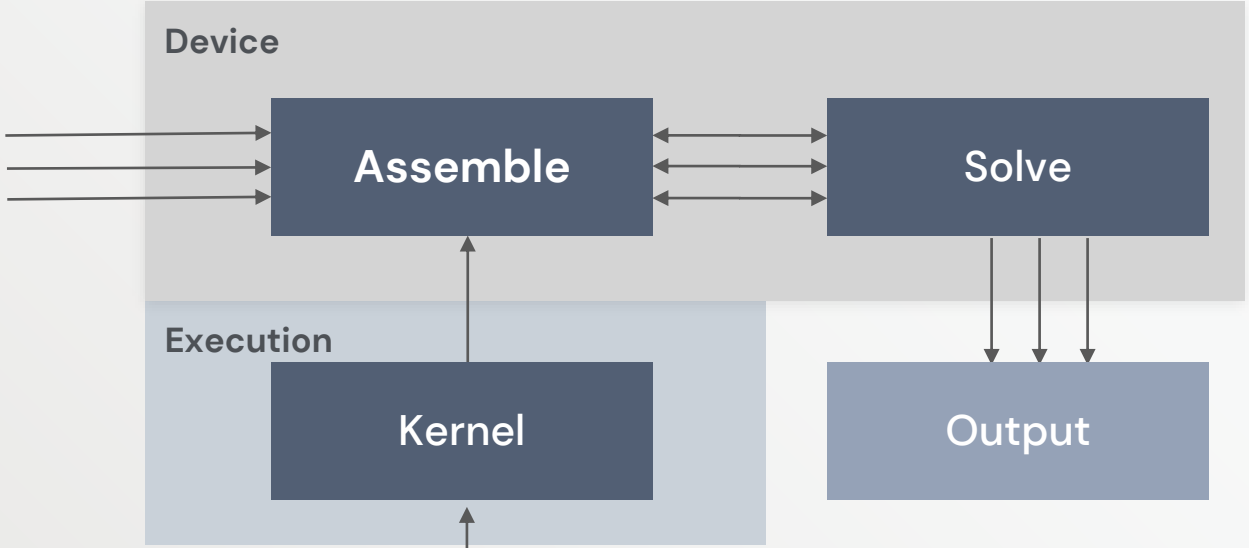
TRILINOS

Ginkgo

AmgX

Modular finite element method





$$-\nabla \cdot (\mu \nabla u) + \gamma u = f$$

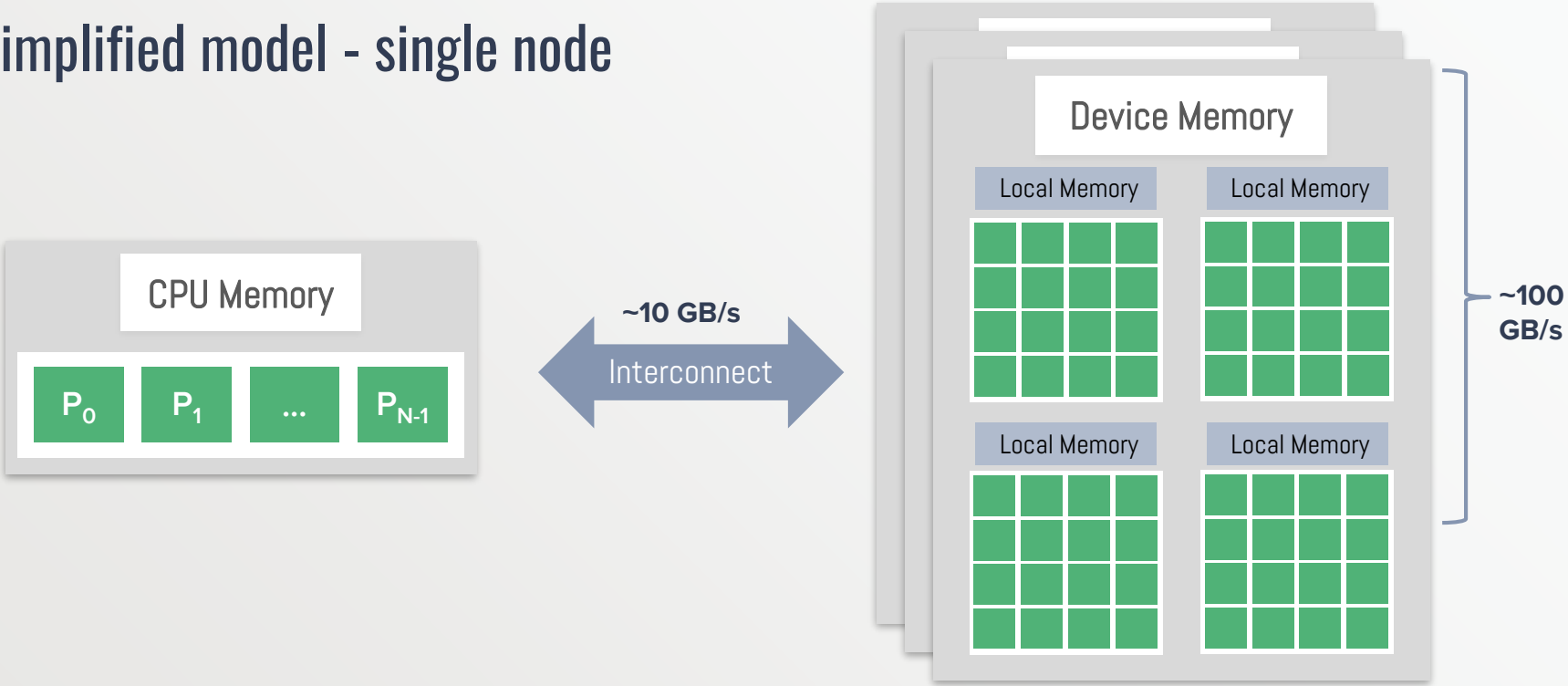
```
element = FiniteElement("Lagrange", tetrahedron, 3)
...
a = mu*inner(grad(u), grad(v)) * dx + gamma*inner(u, v) * dx
L = inner(f, v) * dx
```

Programming model

SYCL is a high-level single source parallel programming model, that can target a range of heterogeneous platforms:

- uses completely standard C++;
- both host CPU and device code can be written in the same C++ source file;
- open standard coordinated by the Khronos group.

Simplified model - single node



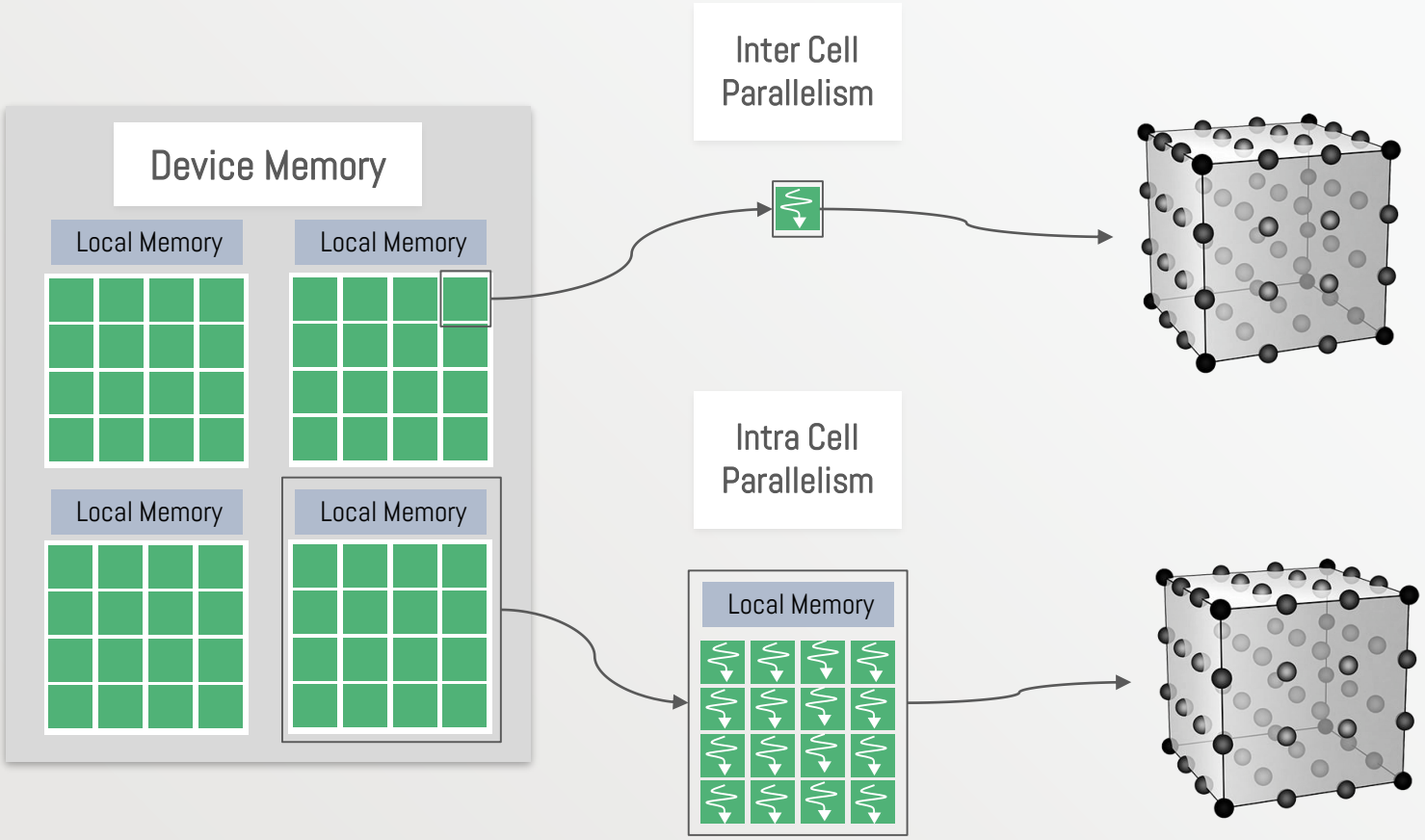
Copy input data from **Host** memory to **Device** memory

Launch kernels for execution on the **Device**

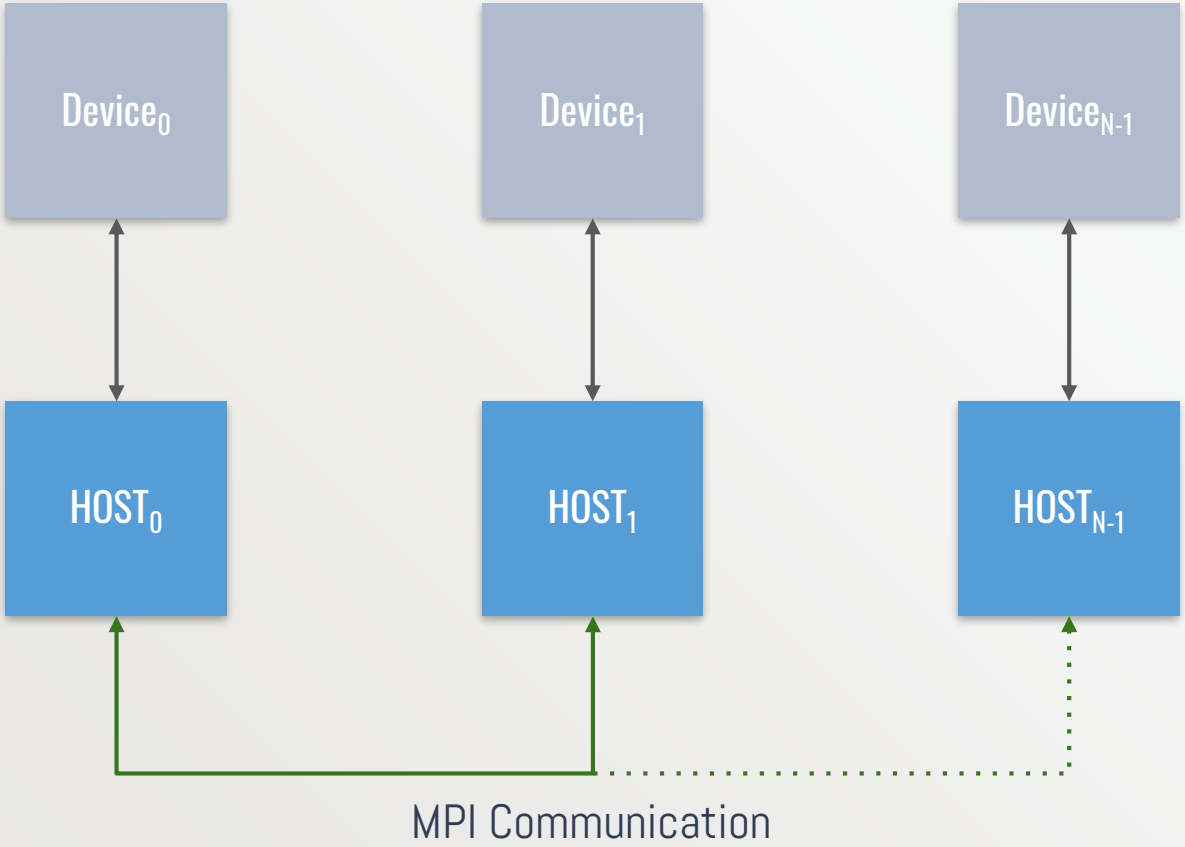
Wait for the execution queue to finish

Copy results back to **Host** from **Device**

Parallelization strategies



Communication among devices



Ceed Benchmarks

https://ceed.exascaleproject.org/bps/
Benchmarks, results, codes

What can we learn by comparison of kernel operations in other codes?

MFEM	libCEED	Nek
https://github.com/mfem	https://github.com/CEED/libCEED	https://github.com/Nek5000

Benchmark description

Bake-off problems: high-order kernels/benchmarks designed to test and compare the performance of high-order codes.

$$-\nabla \cdot (\mu \nabla u) + \gamma u = f$$

$$\mathrm{H}\mathbf{u} = \mathrm{A}\mathbf{u} + \mathrm{B}\mathbf{u} = \mathbf{b}$$

Where $\boldsymbol{\mu}$ and $\boldsymbol{\gamma}$ are non-negative functions of x.

Benchmark	Parameters	Solver
BP1 (Mass)	$\boldsymbol{\mu}$=0, $\boldsymbol{\gamma}$=1	(P)CG
BP3 (Stiffness)	$\boldsymbol{\mu}$=1, $\boldsymbol{\gamma}$=0	(P)CG

CG Solver benchmark: kernels

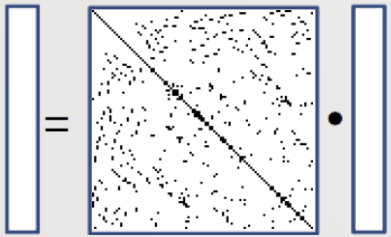
Memory Bound

Memory or Compute Bound

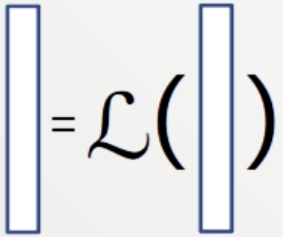
Name	Expression	
Vector Copy	$\mathbf{y} = \mathbf{x}$	BLAS, oneMKL, cuBLAS
Vector axpy	$\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$	BLAS, oneMKL, cuBLAS
Vector Inner Product	$\alpha = \mathbf{x} \cdot \mathbf{y}$	BLAS, oneMKL, cuBLAS
Gather	$\mathbf{x} = Z^T\mathbf{x}^K$	FEM kernel
Scatter	$\mathbf{x}^K = Z\mathbf{x}$	FEM kernel
Matrix Vector Product	$\mathbf{y} = A \cdot \mathbf{y}$	FEM kernel

Matrix-Vector Product Action

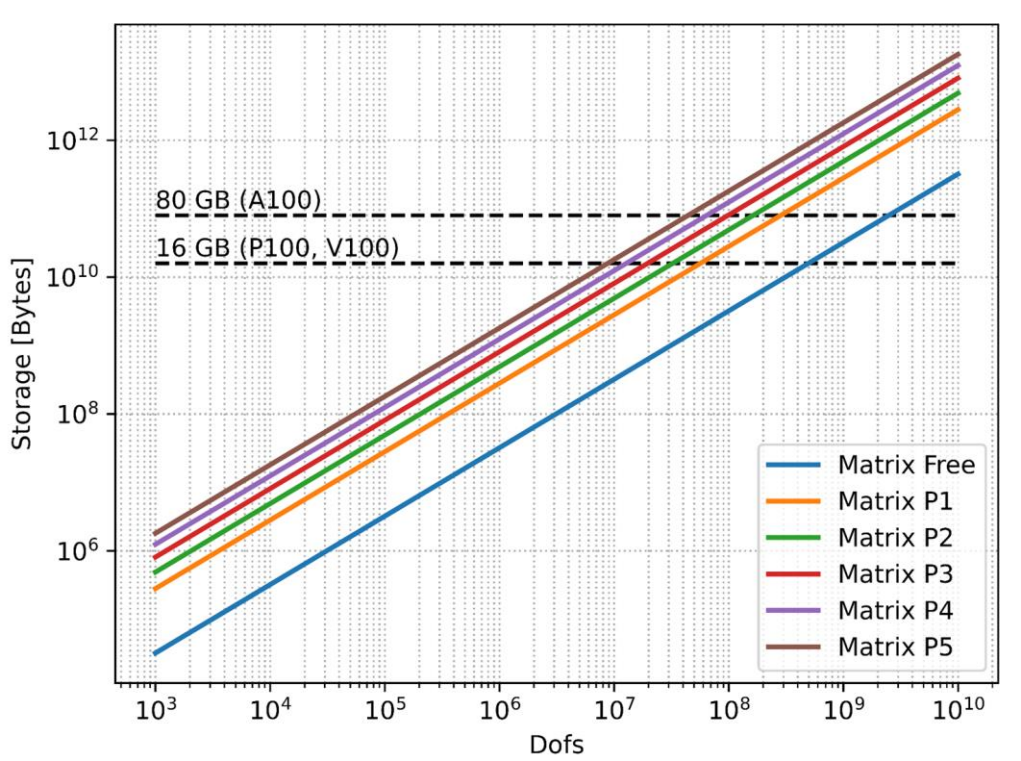
"Full assembly"



"Matrix free"



Matrix-free vs Standard sparse matrices [Local Storage]



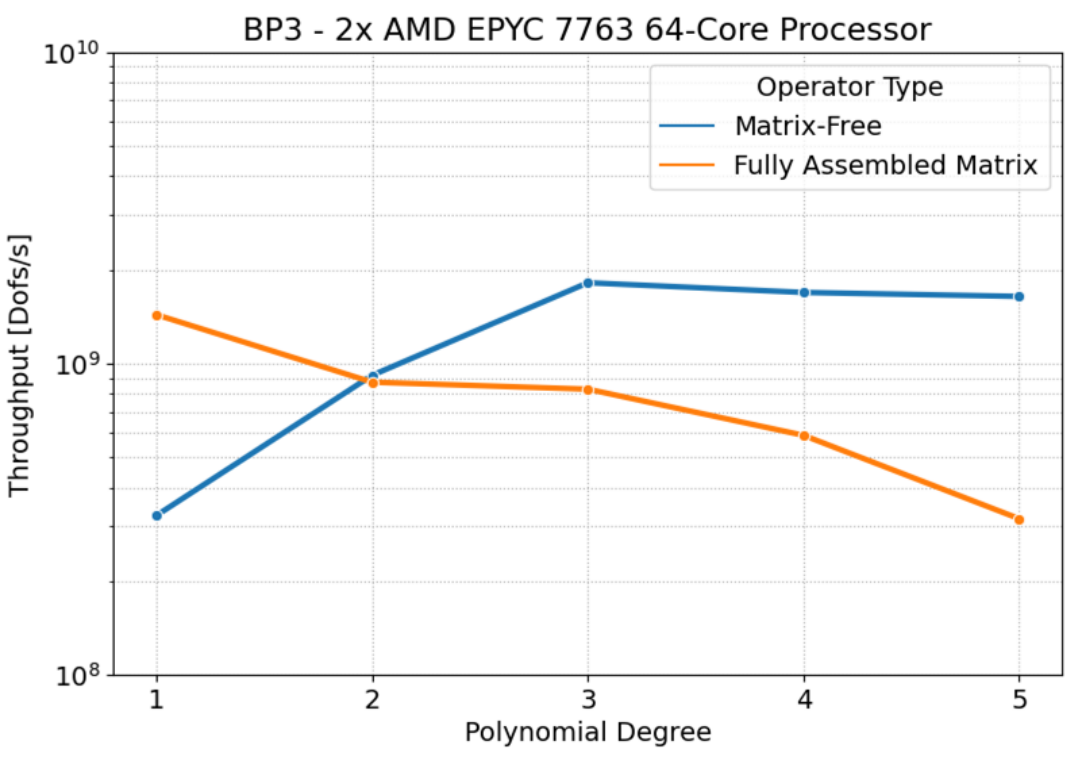
Vectors

ndofs * 8 bytes (double)

CSR Matrix

nnz * 8 bytes (double)
nnz * 4 bytes (int32)
ndofs * 4 bytes (int32)

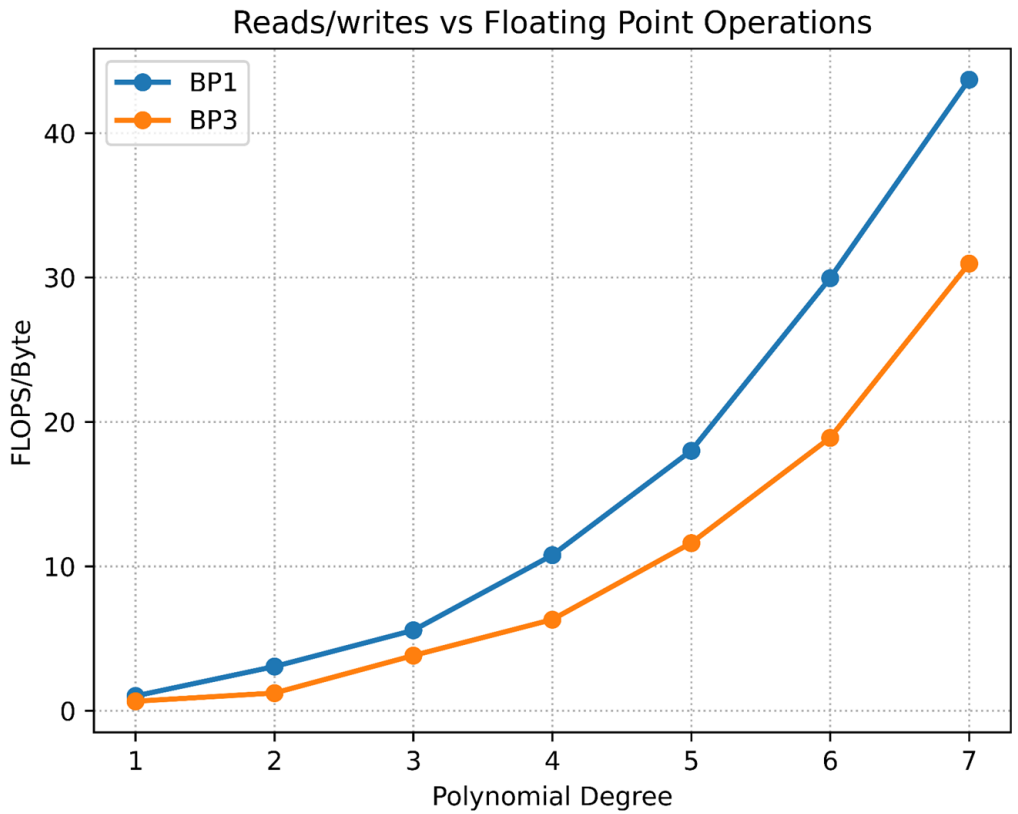
Matrix-free Vs Standard sparse matrices [Throughput]



Matrix Free require only $O(n)$ data movement and storage.

Fully assembled matrix performance falls rapidly at high orders while matrix-free operators scale well.

Floating-point operations (FLOPs) per memory access



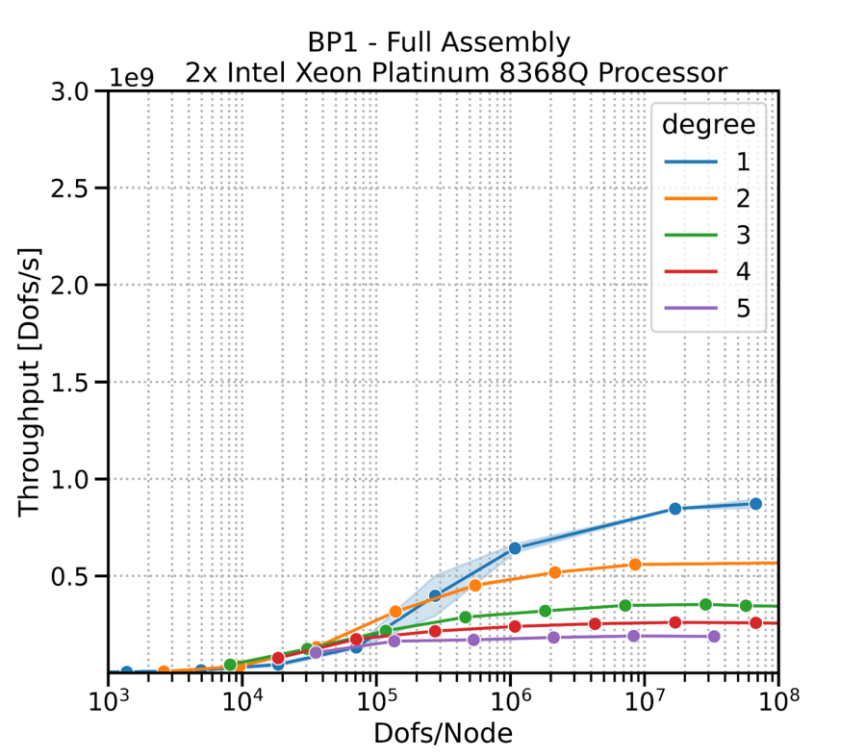
Historically, FLOPs were the performance-limiting factor.

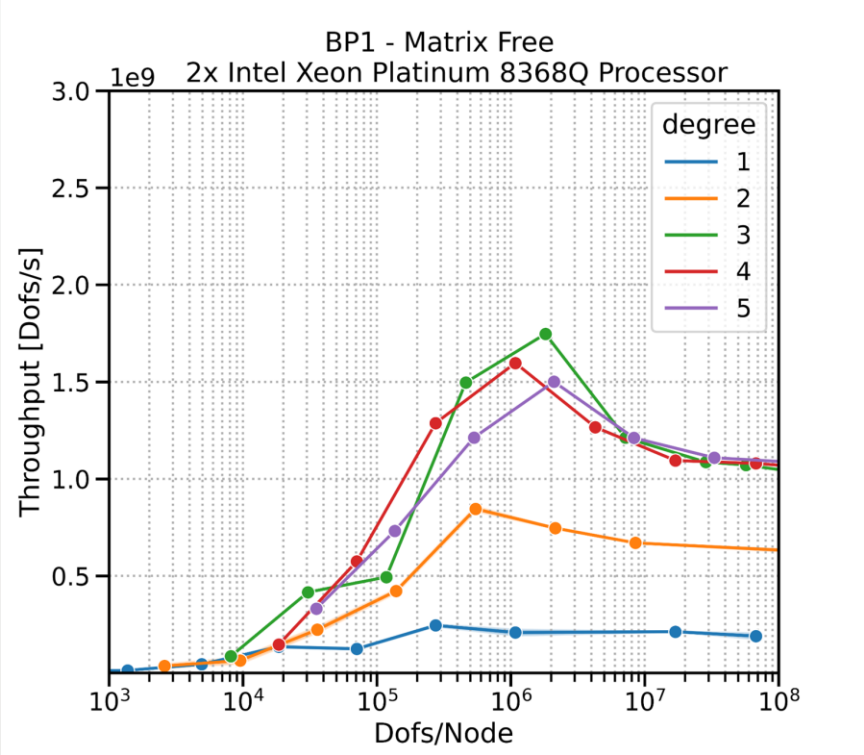
Gap between computer processing speed and memory access.

High order methods:

- Control over arithmetic intensity.

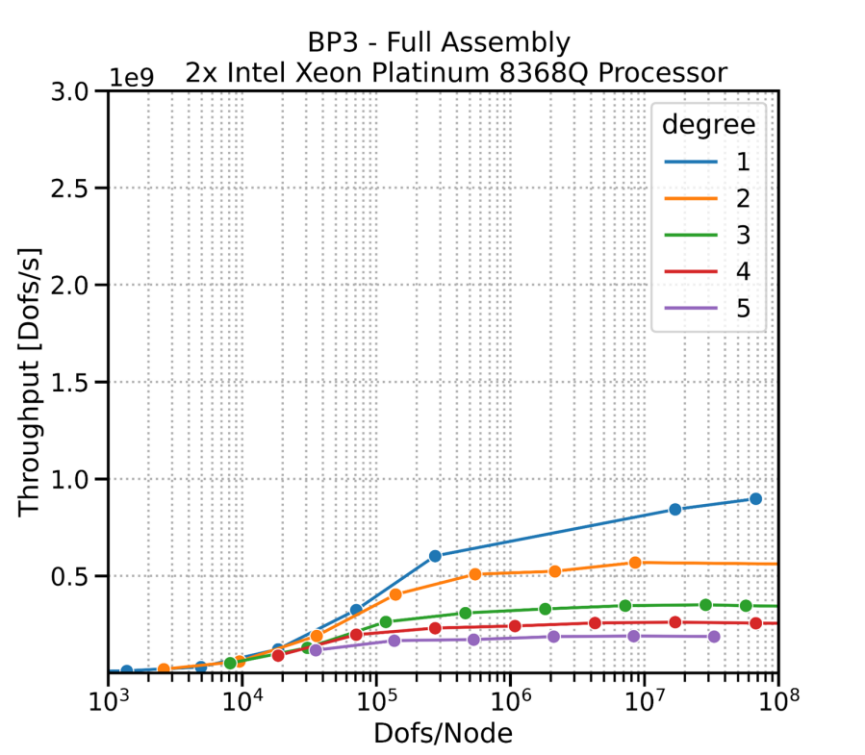
BP1 - Ice Lake node

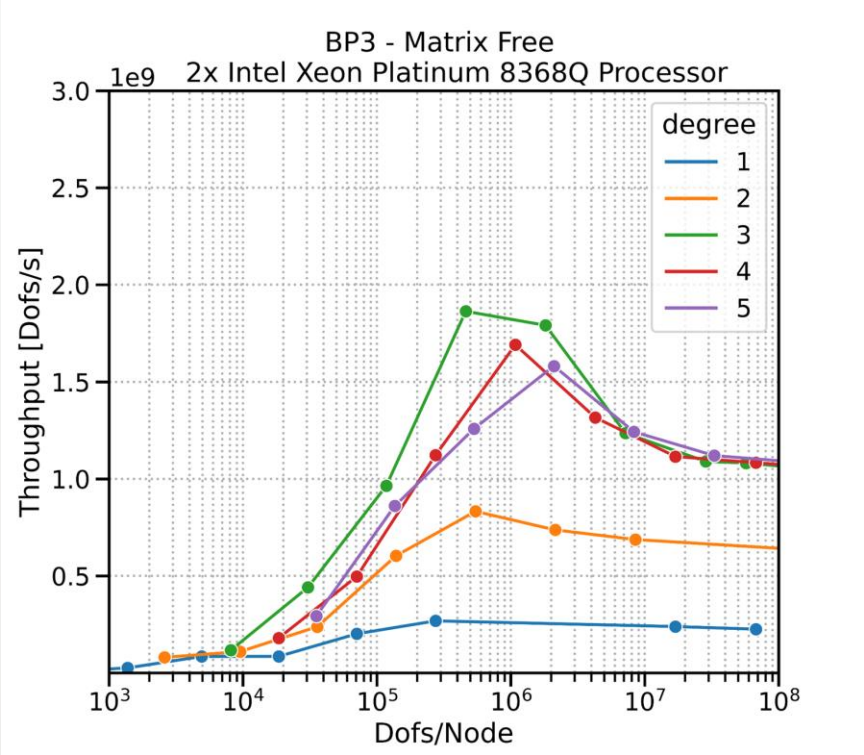




- Local Performance vs Local Problem Size
- Peak* Performance: 1.1 TDOF/s per node

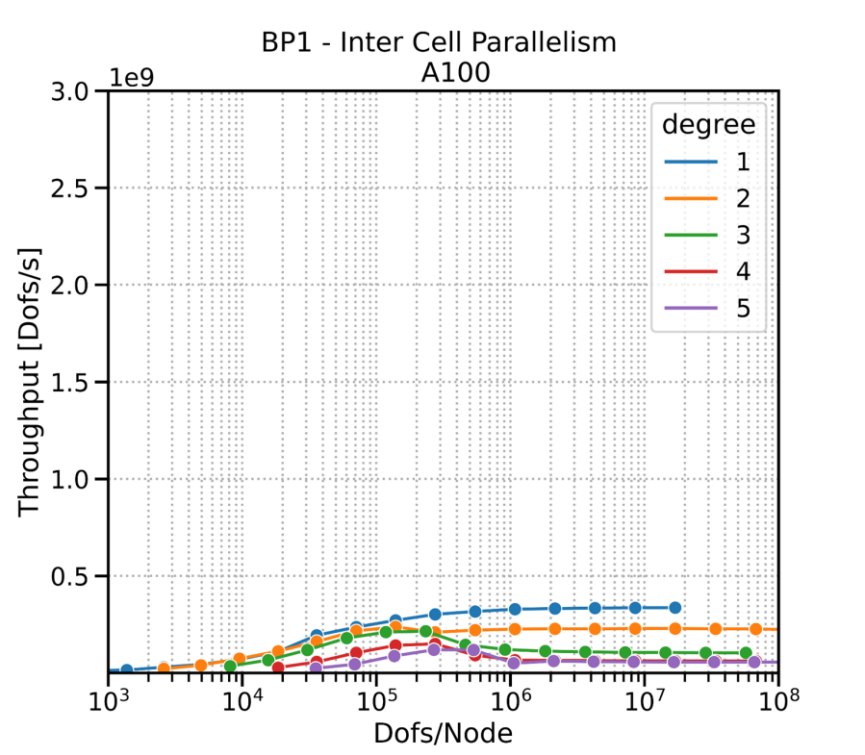
BP3 - Ice Lake node

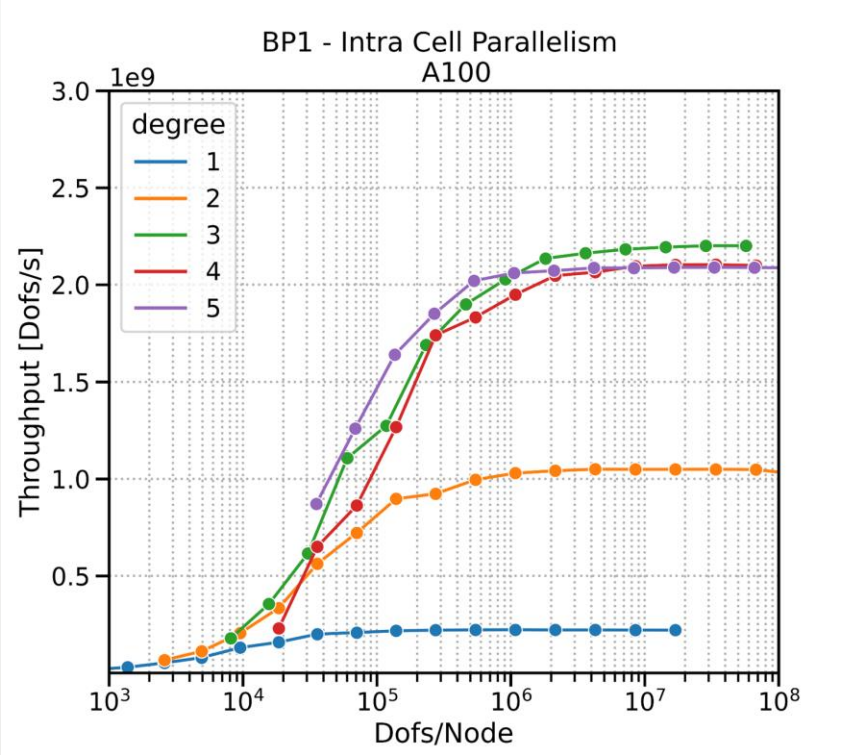




- Local Performance vs Local Problem Size
- Peak* Performance: 1.1 TDOF/s per node

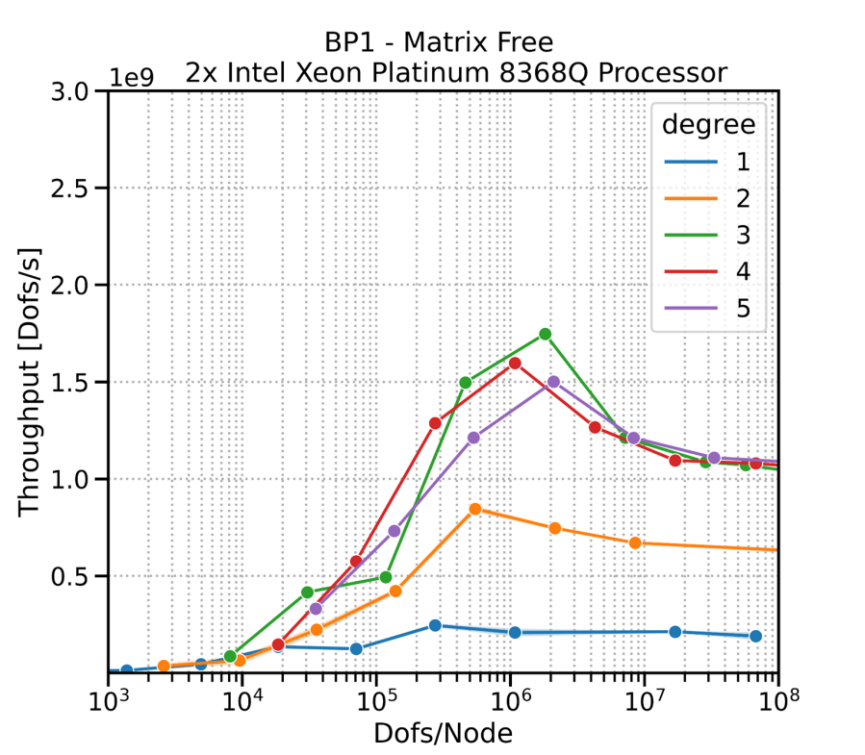
BP1 - Matrix-free - NVIDIA A100-80GB

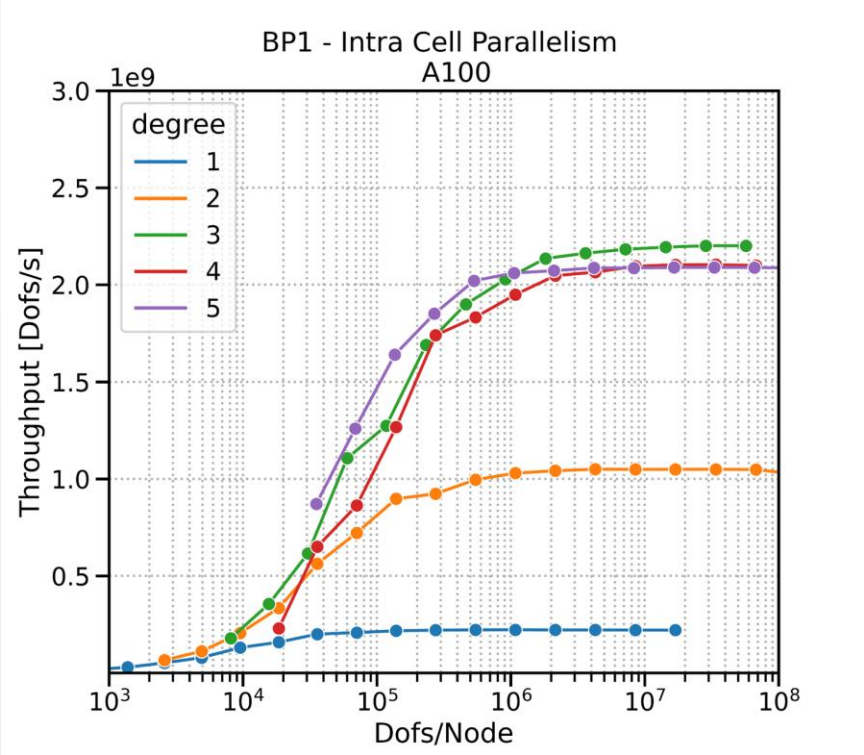




- One cell per thread vs one cell per block
- Peak Performance: 2.2 TDOF/s per node

CPU vs GPU implementations

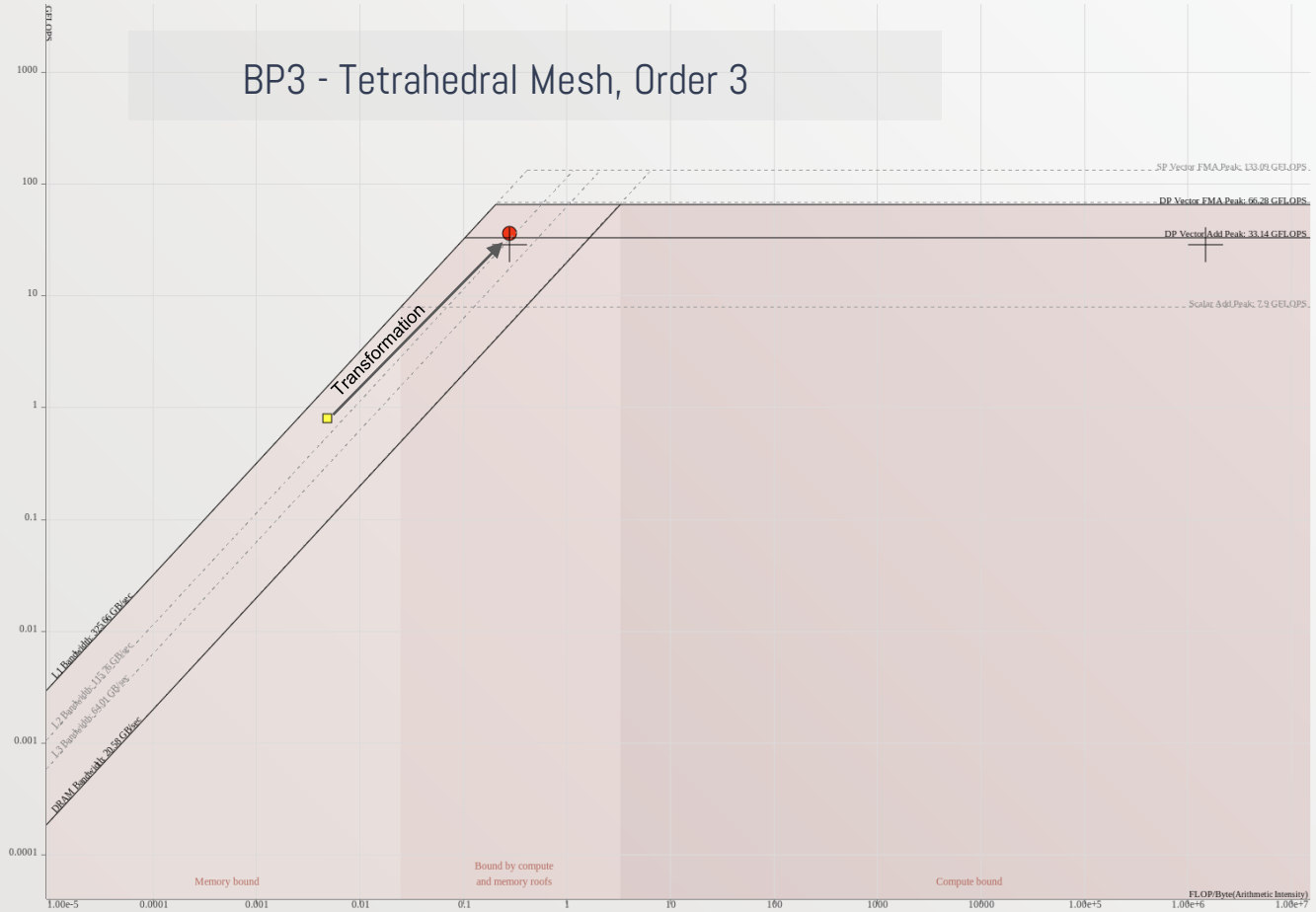




- DP Peak Performance: 4.6 TFLOPS/S
- Best CPU implementation: 1 cell per thread

- DP Peak Performance: 9.7 TFLOPS/S
- Best GPU implementation: 1 cell per thread block

GPU and CPU rooflines - matvec kernel



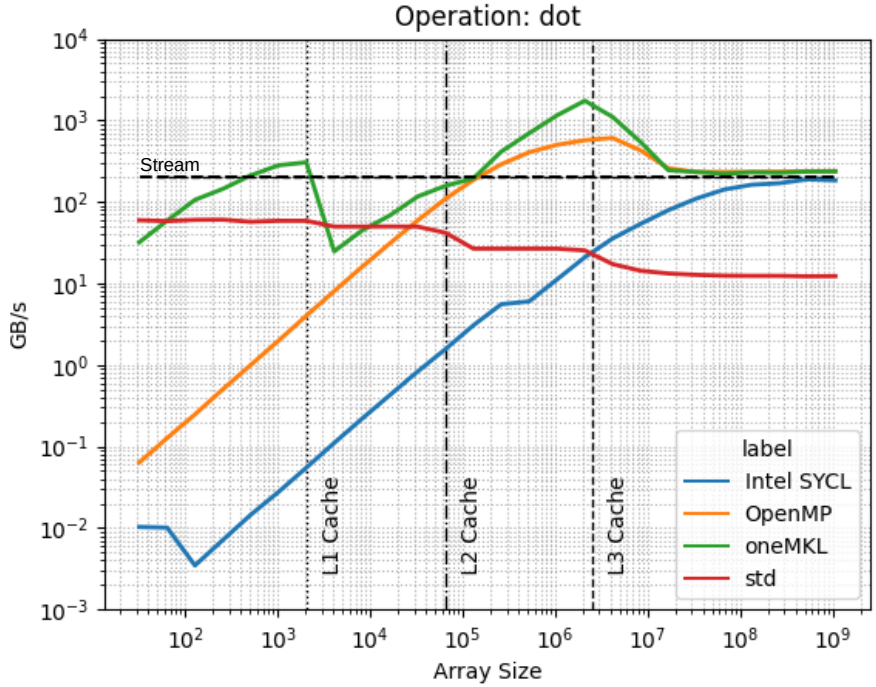
~50% of peak performance with AVX512

Strategies for reducing global memory traffic

Improved memory access pattern

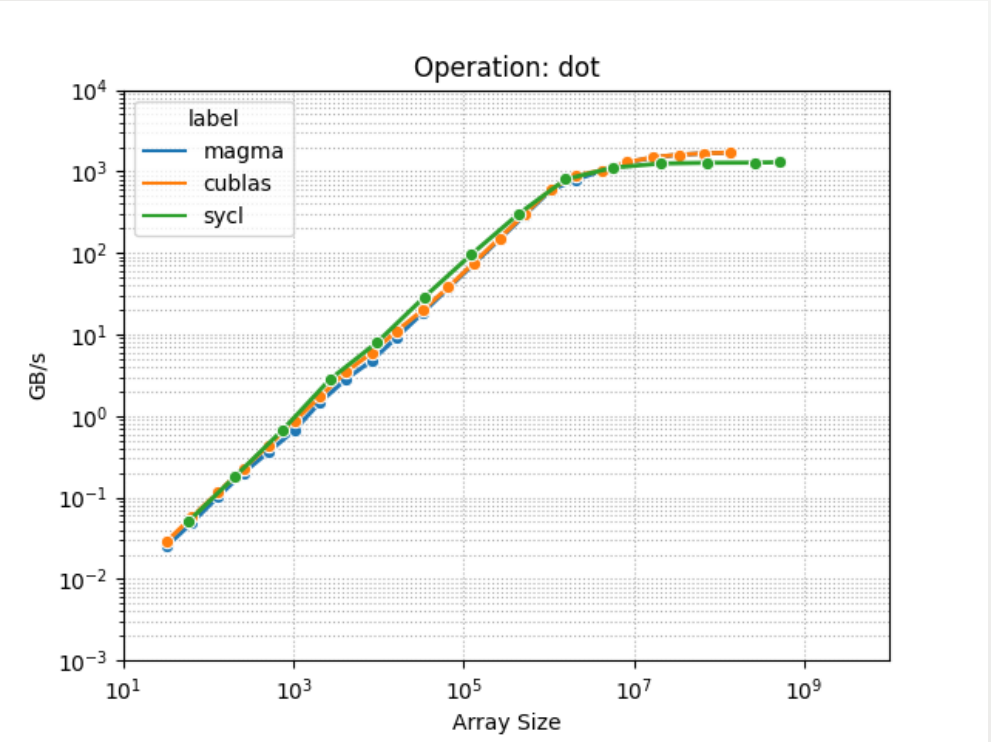
Improved generated code for auto-vectorization

dot: $\alpha = x \cdot y$ (Cascade Lake, 56 cores per node)




```
template <typename T, std::size_t BLOCK_SIZE=32>
T dot(sycl::queue q, std::size_t n, const T* x, T* y) {

 auto padded_length = BLOCK_SIZE * divceil(n, BLOCK_SIZE);
 sycl::range local{BLOCK_SIZE};
 sycl::range global{padded_length};

 auto sum = sycl::malloc_shared<T>(1, q);
 sum[0] = 0.0;

 auto event = q.parallel_for(sycl::nd_range<1>{global, local},
 sycl::reduction(sum,std::plus<T>()),
 [=](sycl::nd_item<1> it, auto& sum) {
   std::size_t i = it.get_global_id(0);
   if (i < n)
     sum += x[i] * y[i];
 });
 event.wait();

 T result = sum[0];

 return result;
}
```

dot: $\alpha = x \cdot y$ (A100 - 80GB)



SYCL code:

```
template <typename T, std::size_t BLOCK_SIZE=32>
T dot(sycl::queue q, std::size_t n, const T* x, T* y) {

 auto padded_length = BLOCK_SIZE * divceil(n, BLOCK_SIZE);
 sycl::range local{BLOCK_SIZE};
 sycl::range global{padded_length};

 auto sum = sycl::malloc_shared<T>(1, q);
 sum[0] = 0.0;

 auto event = q.parallel_for(sycl::nd_range<1>{global, local},
 sycl::reduction(sum,std::plus<T>()),
 [=](sycl::nd_item<1> it, auto& sum) {
   std::size_t i = it.get_global_id(0);
   if (i < n)
     sum += x[i] * y[i];
 });
 event.wait();

 T result = sum[0];

 return result;
}
```

Final observations

The range of problem sizes was chosen to span from the performance-saturated limit (>1M cells per core) to beyond the strong-scale limit (1 cell per core).

- SYCL implementation performed well in the performance-saturated limit (CPU and GPU).
- Small Problems: difficult to hide latency and thread overhead.

Where to go from here?

- In-depth profiling of the finite element GPU kernels.
- Testing on Intel GPUs underway (looks promising).
- Optimized MPI-X communication (hundreds of gpus).
- Code generation for GPUs (generalize to different differential operators).