UNIVERSITY OF CAMBRIDGE

Finite elements on accelerators

An experience using **FEniCSx** and **SYCL**

Igor Baratta

Chris Richardson

Garth Wells

Table of Contents

What's Performance Portability?

And why do we care about it?

An application is performance portable if it:

- ✓ Achieves reasonable level of performance
- ✓ Requires minimal platform specific code

TOP500
The List.

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.

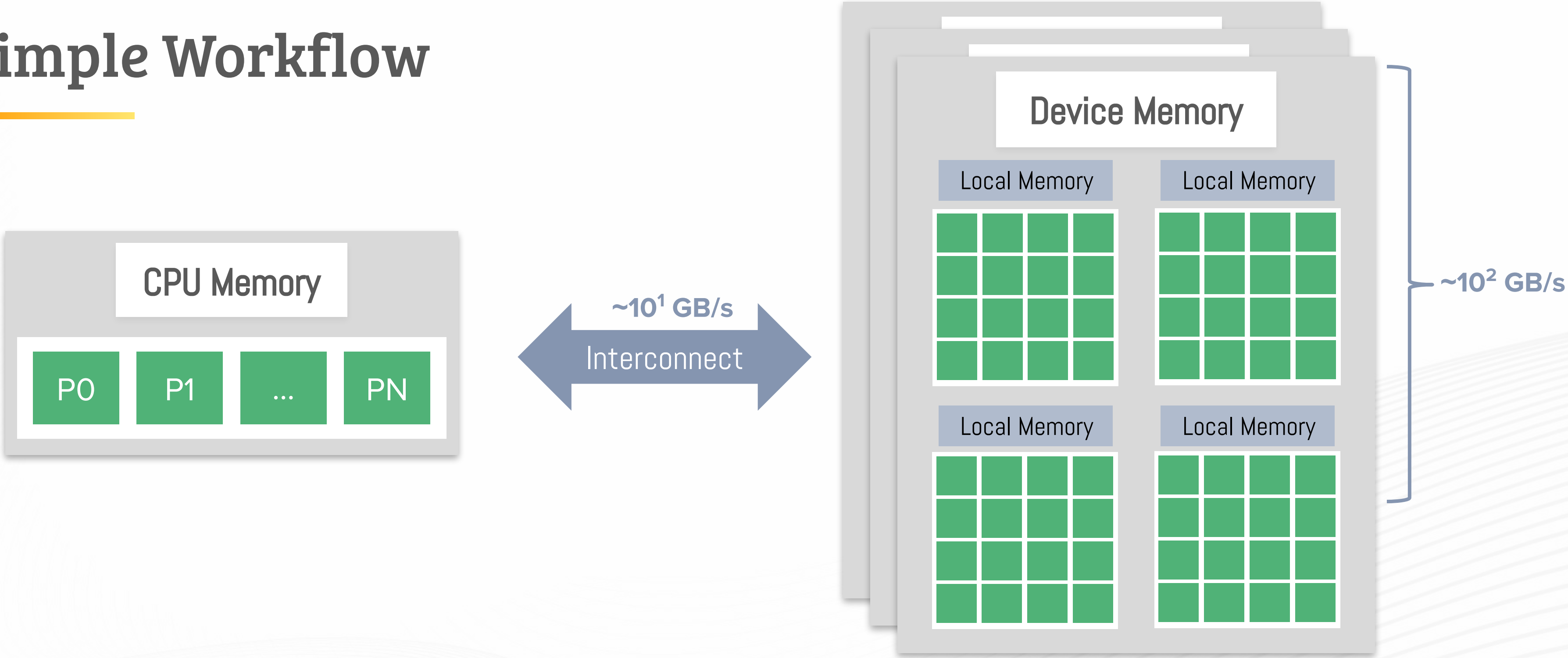
SYCL implementations:

- Intel SYCL*
- hipSYCL*
- Compute Cpp
- triSYCL*

**open source*

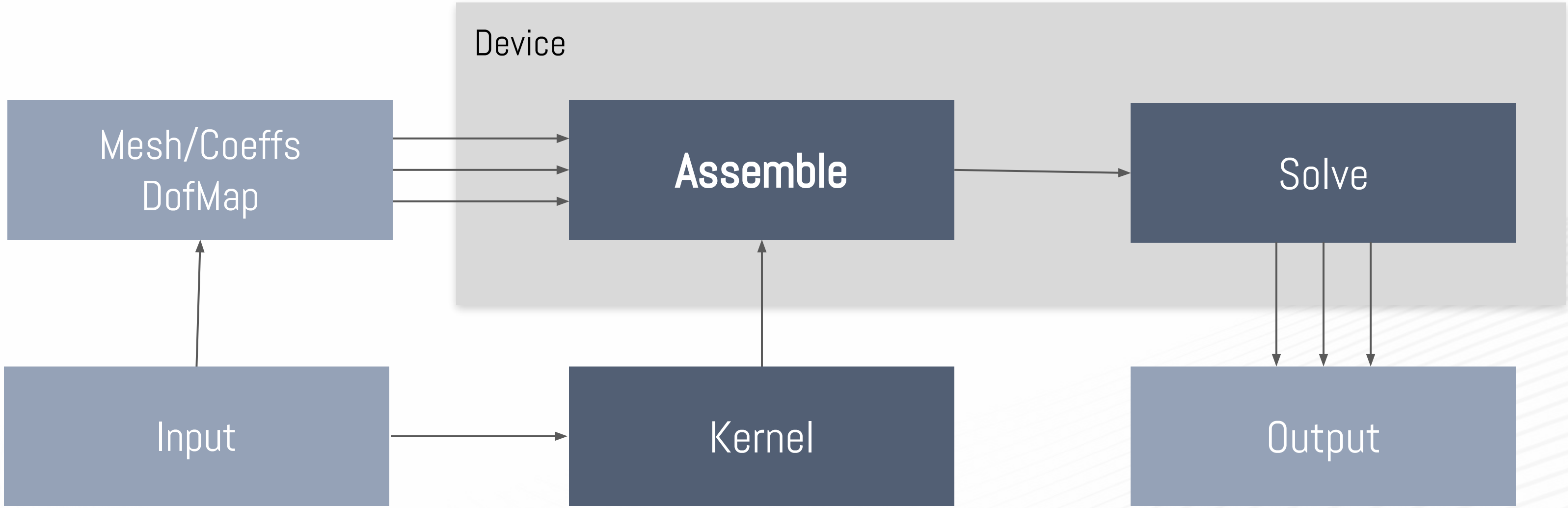
```cpp
cl::sycl::queue q{cl::sycl::gpu_selector()};

int N = 100;
auto a = cl::sycl::malloc_device<double>(N, q);
auto b = cl::sycl::malloc_shared<double>(N, q);

auto e = q.fill(a, 3.0, N);

q.parallel_for(cl::sycl::range<1>(N), e,
[=](cl::sycl::id<1> Id) {
  int i = Id.get(0);
  b[i] = 2 * a[i];
});

q.wait();

for (int i = 0; i < N; i++)
  assert(b[i] == 6.);
```

Simple Workflow



- 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**

An idealised modular Finite Element worflow

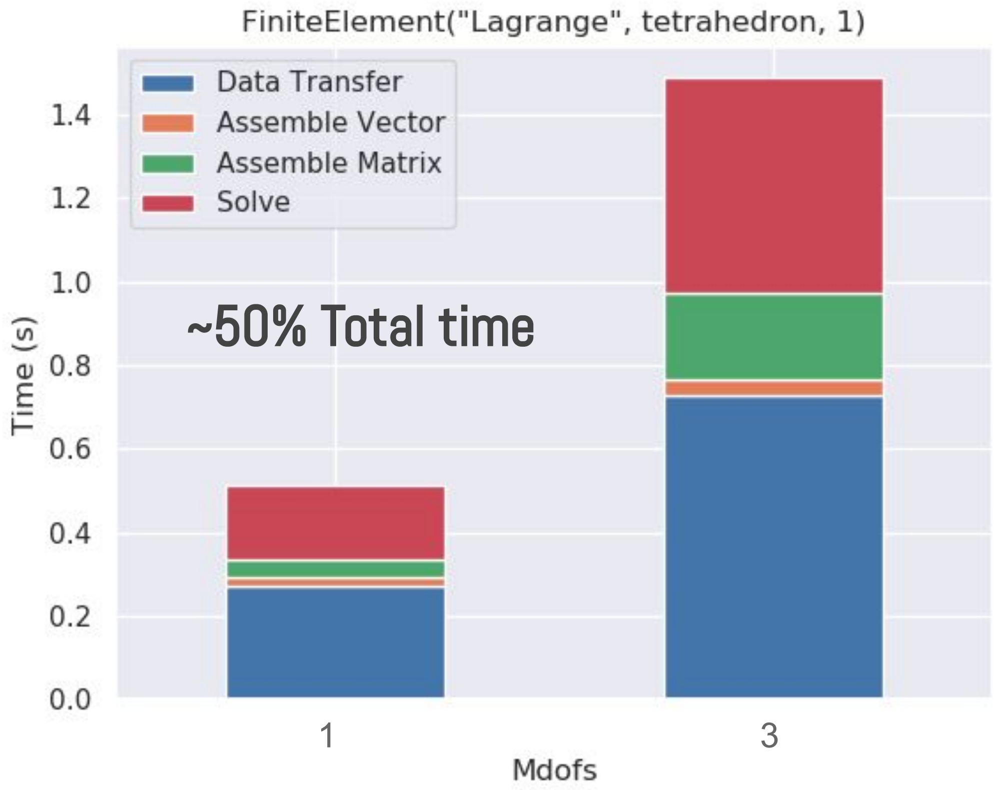


UFL File

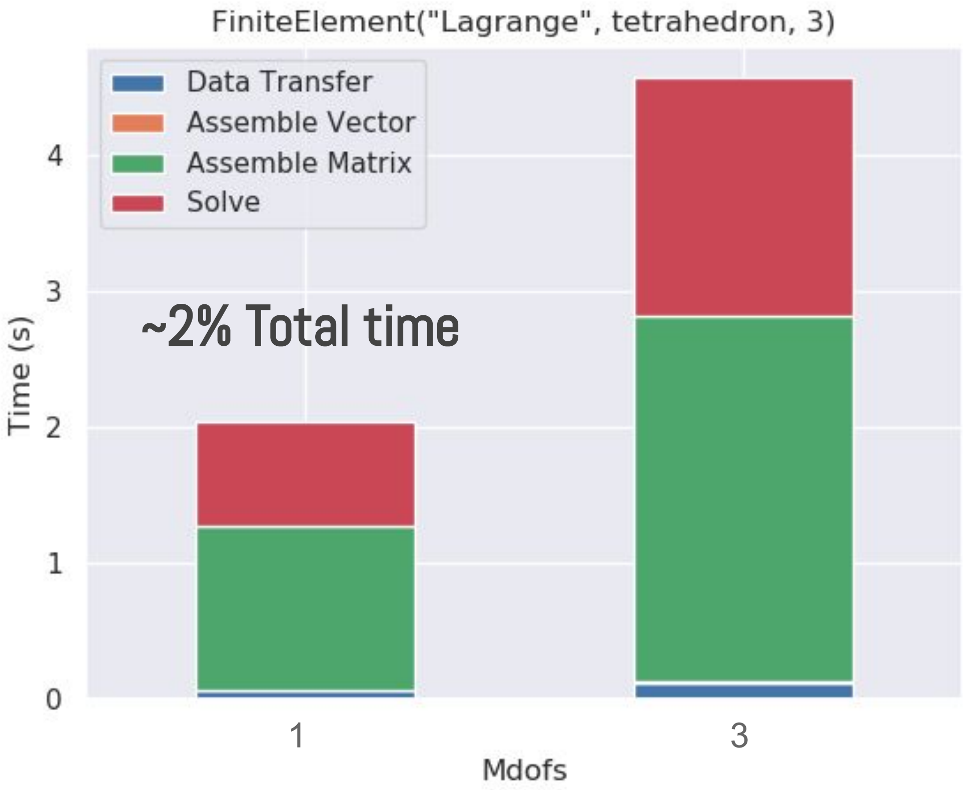
```
element = FiniteElement("Lagrange", tetrahedron, 3)
...
a = inner(grad(u), grad(v)) * dx + k*inner(u, v) * dx
L = inner(f, v) * dx
```

ffcx **--sycl_defines=True** problem.ufl

Data Transfer to Computation Ratio - P1



Data Transfer to Computation Ratio - P3



Matrix Assembly

For each cell:

01 Gather cell coordinates and coefficients

02 Compute element matrix

03 Update global CSR matrix

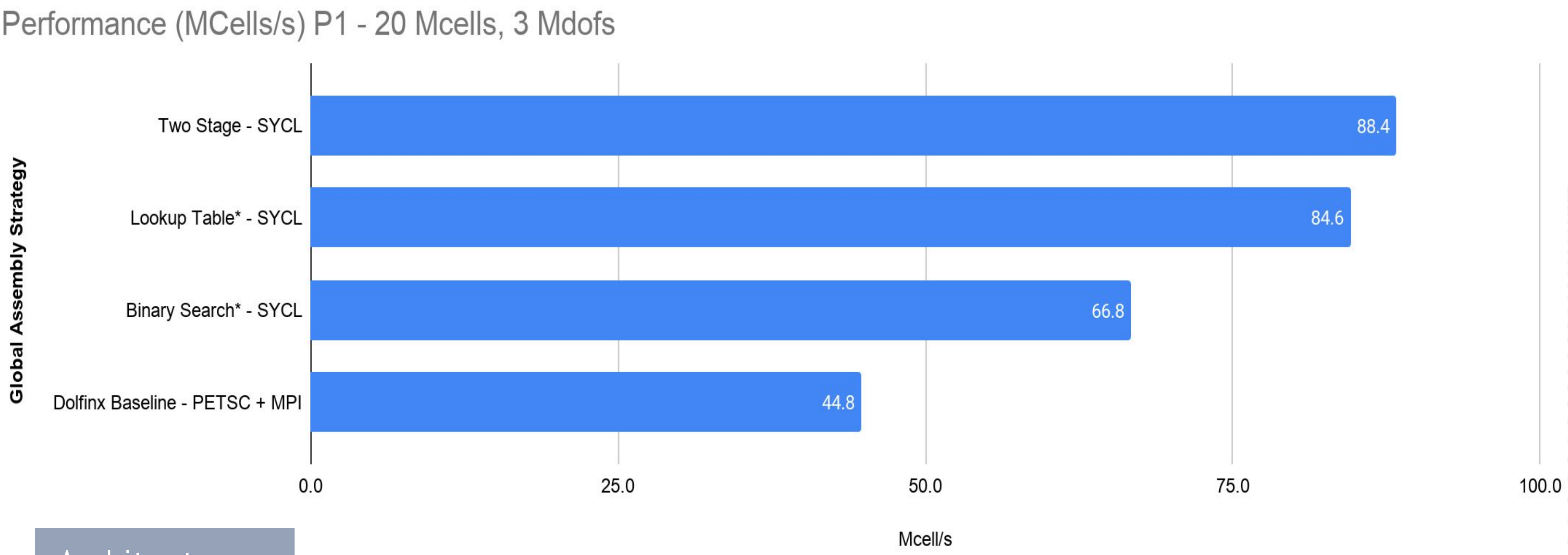
Global assembly strategies:

- Binary Search*
- Lookup Table*
- Two Stage

** atomic operations*

```
auto kernel = [=](cl::sycl::id<1> ID) {
   const int i = ID.get(0);
   ...
   double Ae [ndofs * nofs];

   // Gather cell coordinates and coefficients
   for (std::size_t j = 0; j < 4; ++j)
   {
     const std::size_t dmi = x_coor[i * 4 + j];
     for (int k = 0; k < gdim; ++k)
      cell_geom[j * gdim + k] = x[dmi * gdim + k];
   }
   ...
   // Compute element matrix
   tabulate_cell_a(Ae, coeffs, cell_geom);

   // Update global matrix - Binary Search
   for (int j = 0; j < ndofs; j++)
     for (int k = 0; k < ndofs; k++)
     {
       int ind = dofs[offset + k];
       int pos = find(indices, first, last, ind);
       atomic_ref atomic_A(data[pos]);
       atomic_A += Ae[j * ndofs + k];
     }
};
```

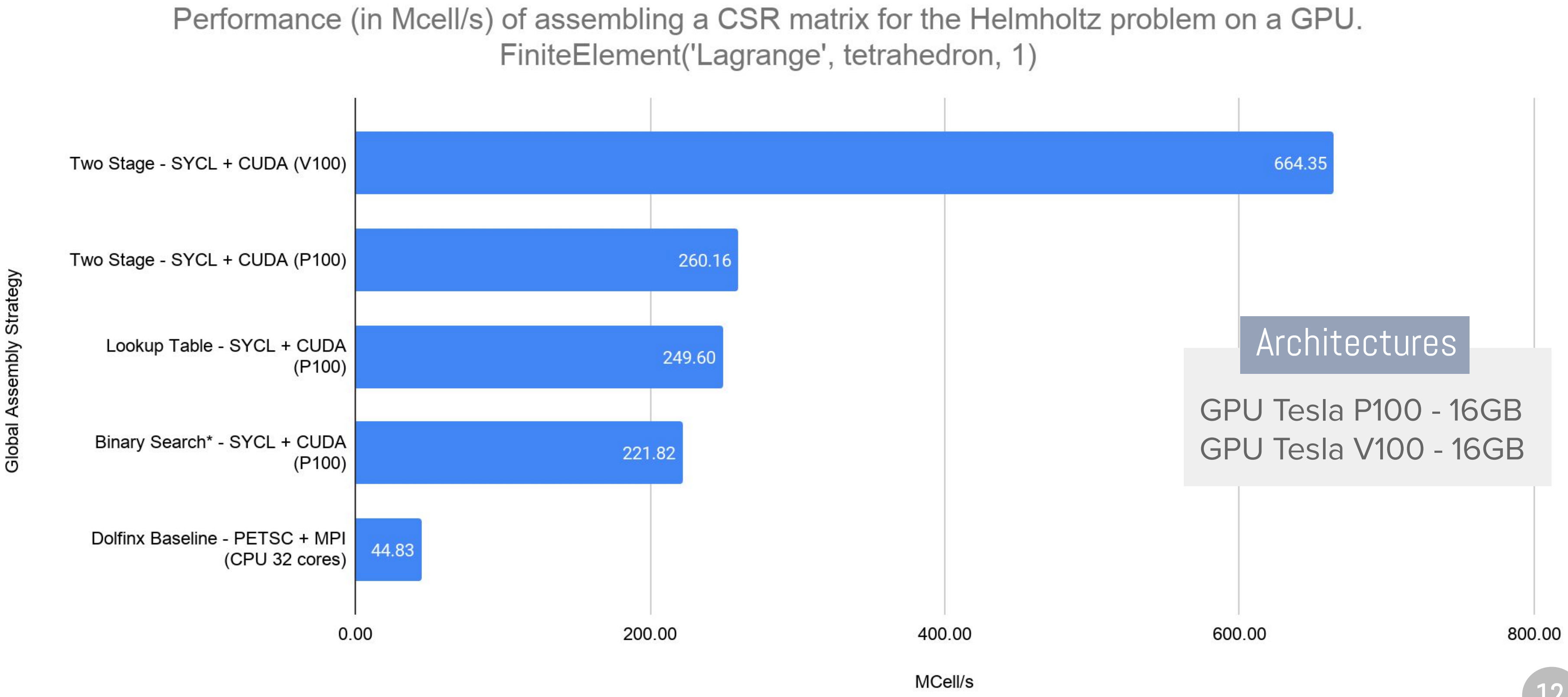
Matrix Assembly - CPU Performance



Architectures

2 x Intel Xeon Skylake 6142 processors, 2.6GHz 16-core
Theoretical peak performance: 2.7 TFlop/s.
192GB RAM

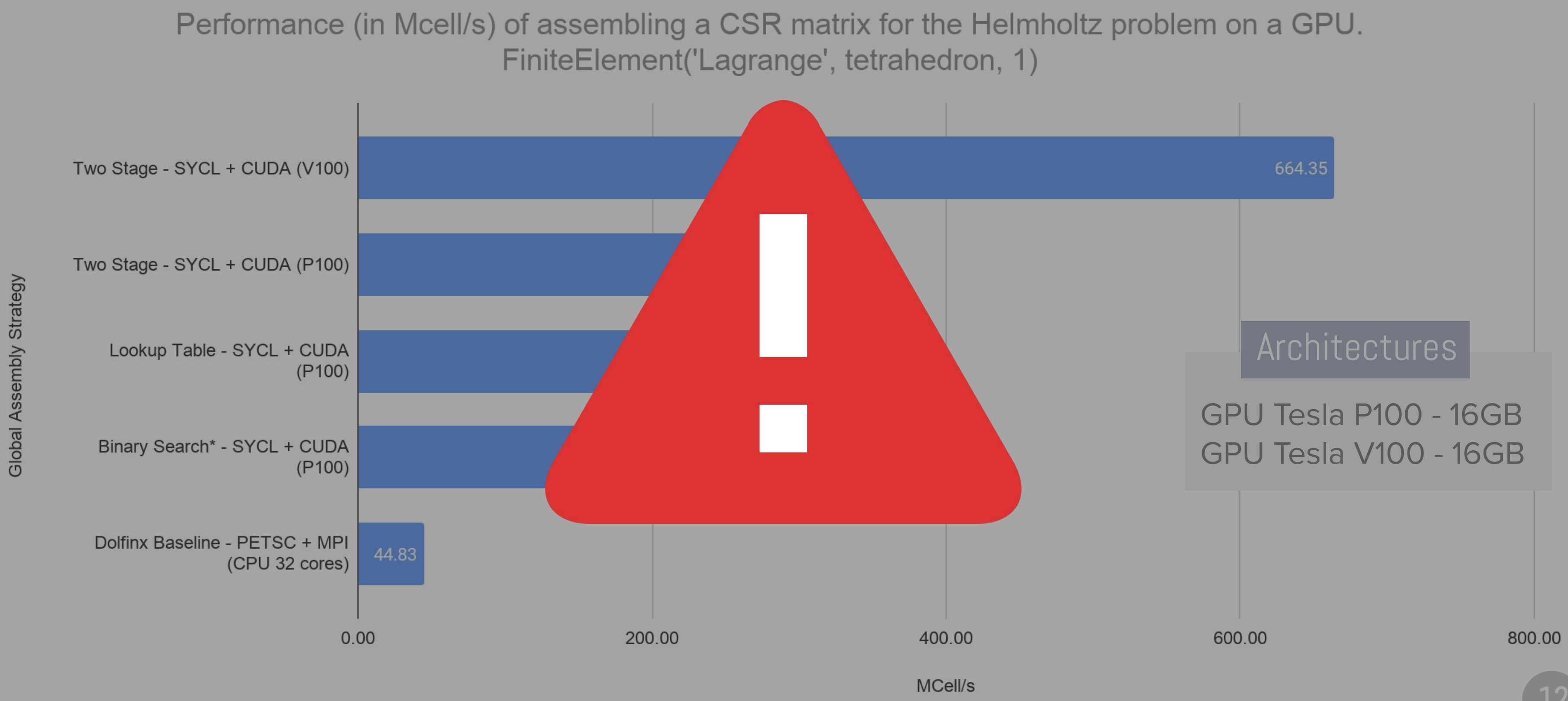
Matrix Assembly - GPU Performance



Architectures

GPU Tesla P100 - 16GB
GPU Tesla V100 - 16GB

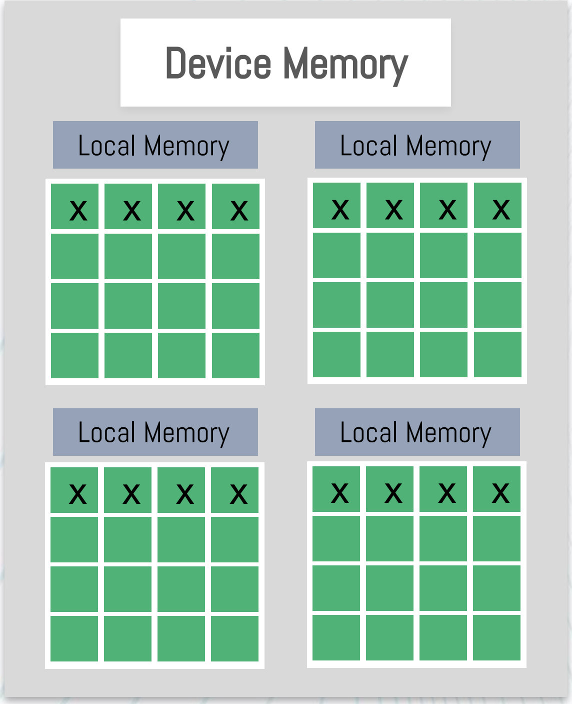
Matrix Assembly - GPU Performance



Architectures

GPU Tesla P100 - 16GB
GPU Tesla V100 - 16GB

Low Achieved Occupancy



Achieved Occupancy: ~25%
The occupancy limited by register usage.

Solution:
Use shared memory for precomputed tables.

Each thread block (work-group) has shared memory visible to all threads (work-item) of the block.

	Occupancy	MCell/s
1st Version	25%	664 MCell/s
Shared Memory	63%	1660 MCell/s
Reference CUDA[1]	*	1627 MCells/s

[1] James Trotter - High-performance finite element computations - Performance modelling, optimisation, GPU acceleration & automated code generation - Phd Thesis 2021.

Thank you!

The code and reproducibility instructions can be found at https://github.com/Excalibur-SLE/dolfinx.sycl

You can reach me via e-mail:
ia397@cam.ac.uk

Future/Ongoing Work

Different problems, and meshes

Linear Elasticity, Maxwell's equations

Profiling in a wider range of devices

AMD GPU, A64FX

Multi-GPU

MPI-based distributed memory computations

Code transformation

Improve generated code