Adding OpenCL to Eigen with SYCL

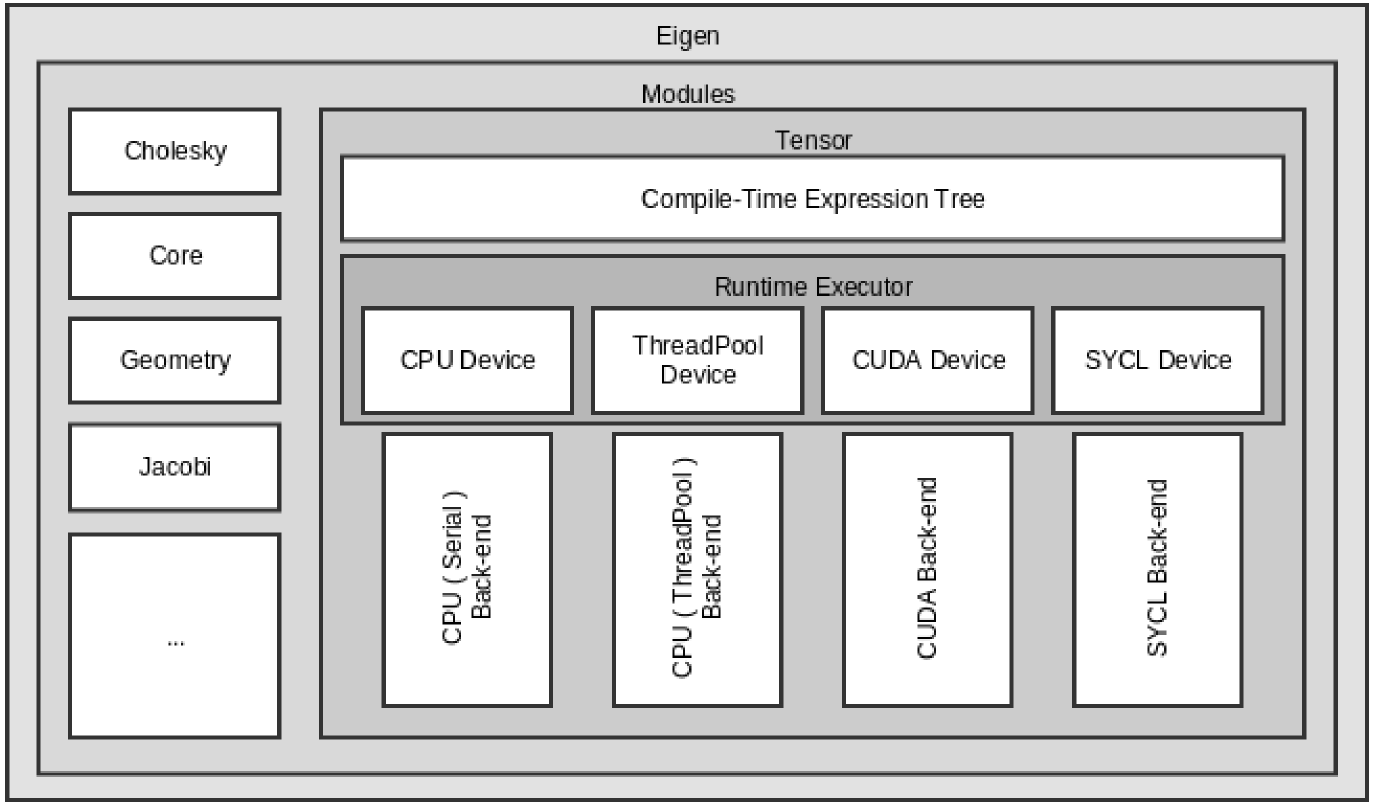
Mehdi Goli, Luke Iwanski, Andrew Richards

May 2017

Agenda

- Eigen
  - Expression Tree
  - Fusion
- Why SYCL?
- Requirements
- Challenges
  - Address Spaces
  - Explicit Data Movement
- Benchmarks
- What next?
- Questions?

Eigen


Eigen

Modules

Cholesky

Core

Geometry

Jacobi

...

Tensor

Compile-Time Expression Tree

Runtime Executor

CPU Device

ThreadPool Device

CUDA Device

SYCL Device

CPU (Serial) Back-end

CPU (ThreadPool) Back-end

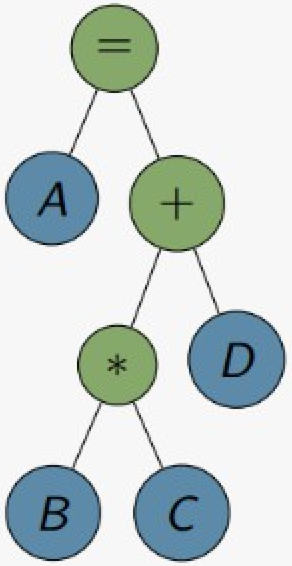
CUDA Back-end

SYCL Back-end

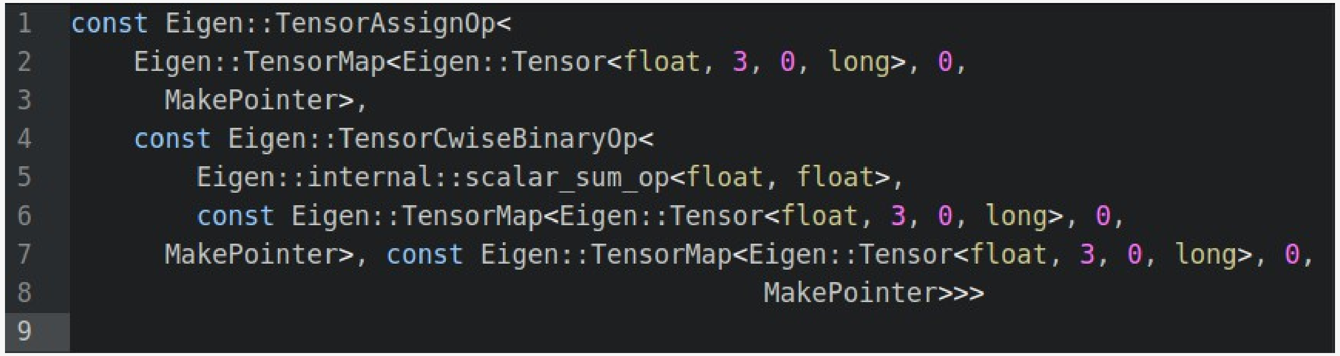
- C++ based high-performance dense linear algebra library.
- Modular
  - Linear algebra, matrix / vector operations, geometrical transformations, numerical solvers and related algorithms
  - Tensor (heavily used by TensorFlow)
- Headers only
- Expression templates meta-programming technique
- Generates compile-time DSL/EDSL based on the expression tree.
- Currently supports CPU and NVIDIA CUDA back-end and now SYCL

Expression Tree

A = B * C + D →

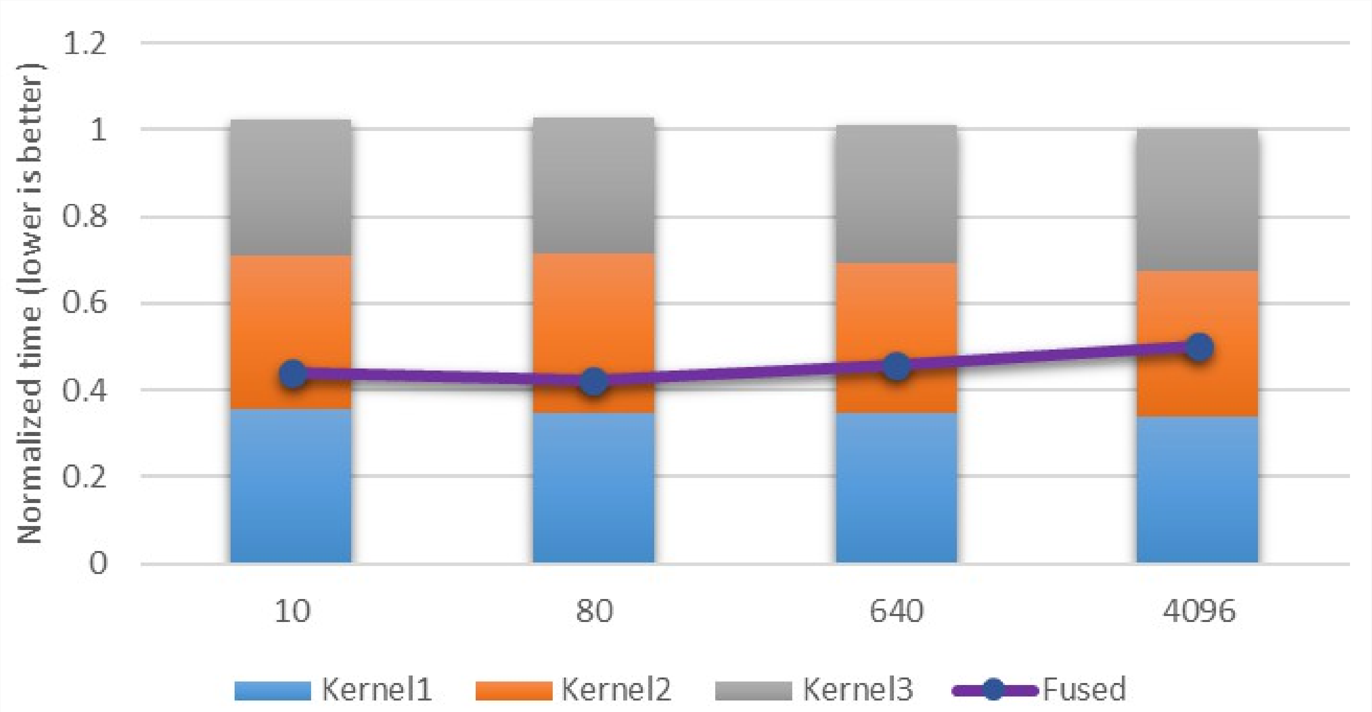


→



```
const Eigen::TensorAssignOp<
    Eigen::TensorMap<Eigen::Tensor<float, 3, 0, long>, 0,
      MakePointer>,
    const Eigen::TensorCwiseBinaryOp<
        Eigen::internal::scalar_sum_op<float, float>,
        const Eigen::TensorMap<Eigen::Tensor<float, 3, 0, long>, 0,
      MakePointer>, const Eigen::TensorMap<Eigen::Tensor<float, 3, 0, long>, 0,
                                            MakePointer>>>
```

Fusion



- Kernel1: C = A*A + B*B
- Kernel2: C1 = A1*A1 + B1*B1
- Kernel3: D = C + C1
- Fused: D = A*A + B*B + A1*A1 + B1*B1

Why SYCL?

- **SYCL is a standard** – not “yet another proprietary solution” bound to a specific device family
- SYCL can dispatch device kernels from C++ application, similar to CUDA
- OpenCL 1.2 does not support C++
- OpenCL 2.1 does support C++ templates inside the kernel
  - But, the kernel itself cannot be a template, therefore we still need different kernel registration per type
- Expression of the tree-based kernel fusion is challenging without embedding a custom compiler
- Single-source programming model
  - No need to implement separate kernel code for each operation
- Re-use of the existing template code for both host and device is possible
- OpenCL would need reimplementation of the back-end – maintenance overhead

Requirements

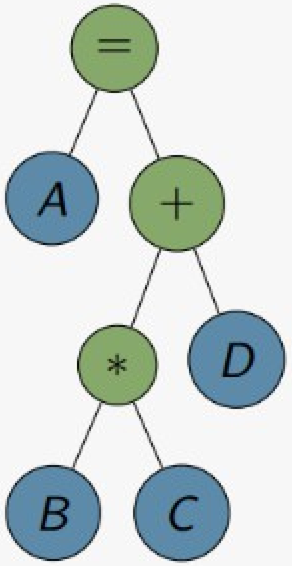
- The back-end must be **non-intrusive**
- Must re-use the existing code and modules in order to reduce maintenance effort
- Must exploit compile-time template meta-programming techniques in order to reduce the runtime overhead
- Must be consistent with the existing API design
- Open-Source projects do not like major changes in their existing code base

Challenge: Address Spaces

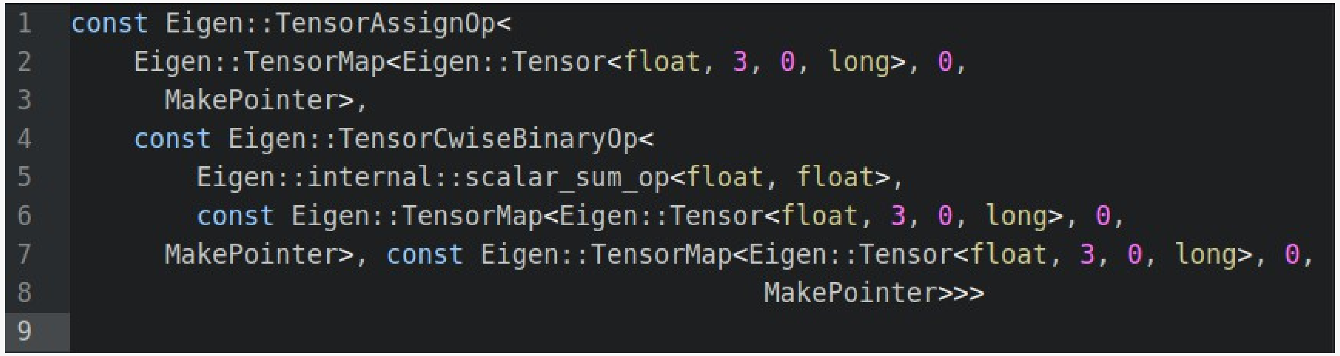
- Eigen expression specialisation uses Scalar pointer
- The difference in approaches: raw pointers (CPU/CUDA) VS. accessors and buffers (SYCL 1.2 /OpenCL 1.2)
- *cudaMalloc* returns “persistent pointer” that stays the same across kernels
- OpenCL 1.2 *cl_mem* object may be translated to non-persistent pointers – they might change across kernels
- OpenCL 2.x solves it via SVM
- Our target is 1.2 with wider range of targeted devices including mobile and embedded

Solution: Address Spaces

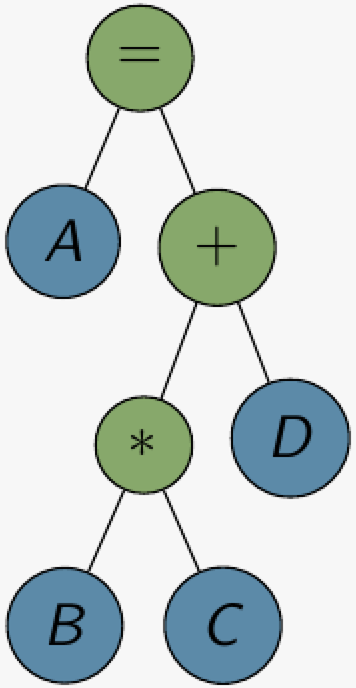
A = B * C + D →



→

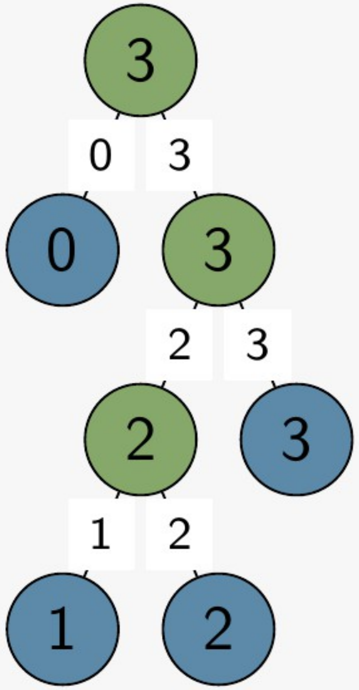

```
const Eigen::TensorAssignOp<
    Eigen::TensorMap<Eigen::Tensor<float, 3, 0, long>, 0,
      MakePointer>,
    const Eigen::TensorCwiseBinaryOp<
        Eigen::internal::scalar_sum_op<float, float>,
        const Eigen::TensorMap<Eigen::Tensor<float, 3, 0, long>, 0,
      MakePointer>, const Eigen::TensorMap<Eigen::Tensor<float, 3, 0, long>, 0,
                                              MakePointer>>>
```

Solution: Address Spaces

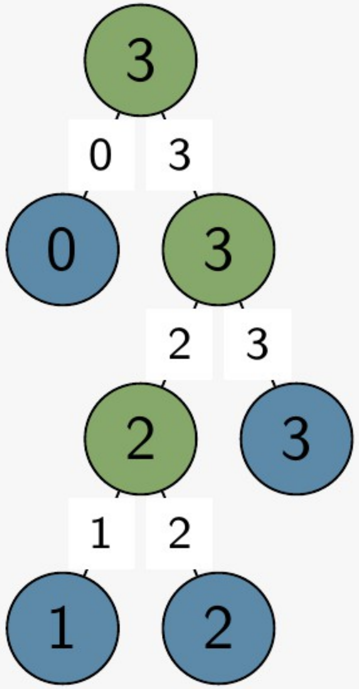


- The terminal nodes are counted recursively at compile time in order to replace each terminal node with a place-holder number
  - the place-holder number corresponds to the location of the relevant accessors in the accessors list

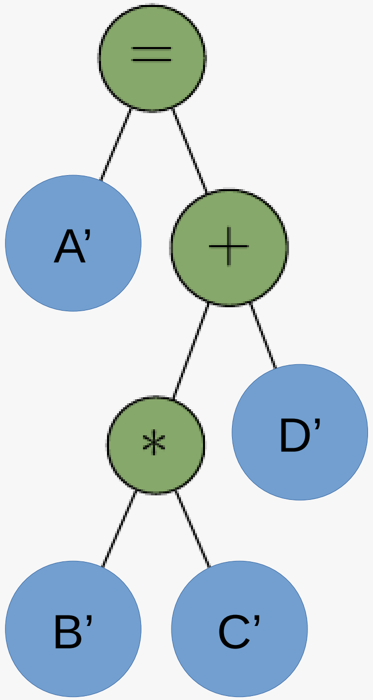
- Depth First Search algorithm is used both to:
  - label the leaf nodes (data nodes)
  - extract the accessors



Solution: Address Spaces



- The place-holder tree is recursively traversed in order to:
  - Re-instantiate the expression tree on the SYCL device
    - The host data pointer in the leaf node is replaced with the corresponding accessors from the accessors list



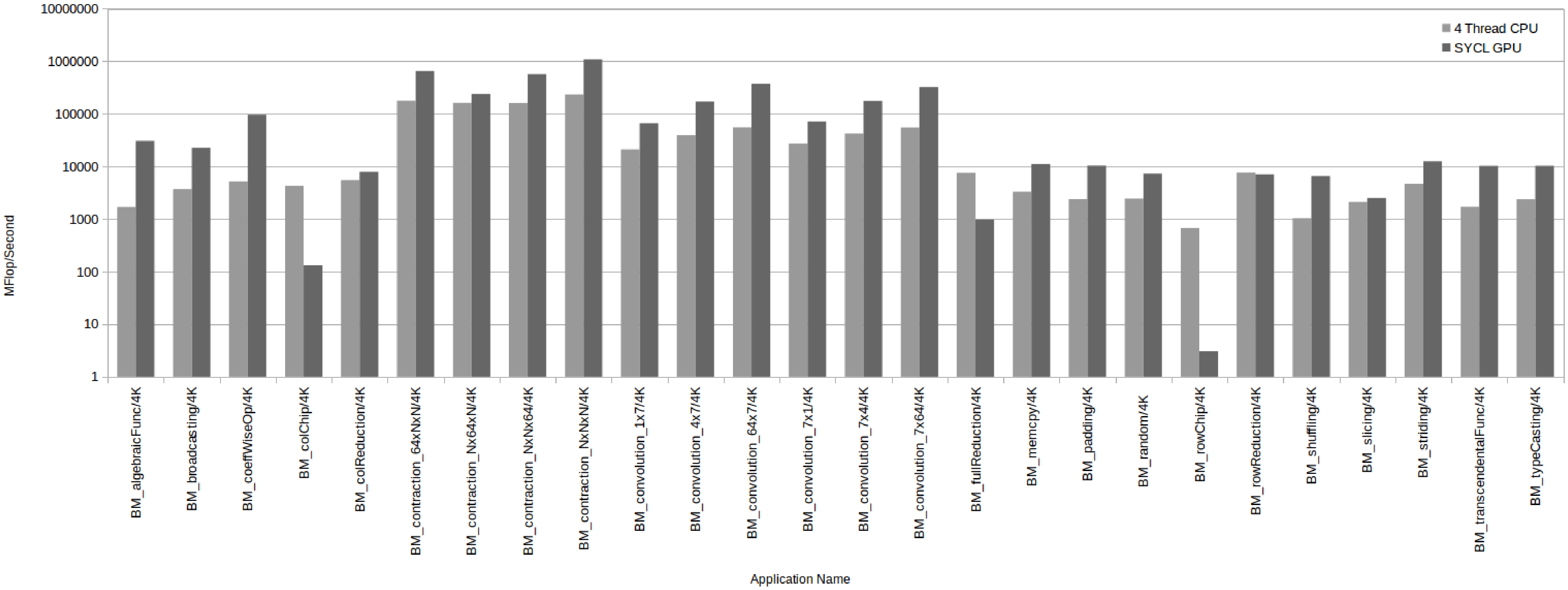
Challenge: Explicit Data Movement

- SYCL programming model is based on implicit data movement, but Eigen has its own data movement interface. These two approaches conflict.
- Eigen's device class provides its own pluggable scheduler for higher-level applications
- Each device can specify its interface - C-style design – methods:
- *allocateMemory, deallocateMemory, memcpy, memcpyHostToDevice, memcpyDeviceToHost, memset*
- Pointer is void and independent from the data type

Solution: Explicit Data Movement

- On the host side a buffer is created for each host pointer
- The buffer life time is coupled with that of the SYCL device instead of the expressions
- All the interface functions explicitly manipulate the corresponding SYCL buffer

Intel(R) Core(TM) i7-6700K CPU 4.00GHz VS AMD R9 Nano



What next?

- The current version of Eigen is the initial release of the SYCL back-end.
- Next steps are optimisation improvements and vectorisation
- We’ll keep you posted!

Thanks!
Questions?

luke@codeplay.com

https://sycl.tech
https://bitbucket.org/mehdi_goli/opencl