Accelerated Machine Learning Using TensorFlow and SYCL on OpenCL Devices

Mehdi Goli

Codeplay Software Ltd

June 22, 2017

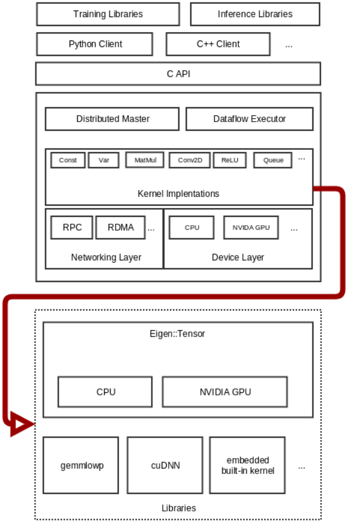
Motivation

Machine learning has been widely used in different areas

- image recognition, self-driving vehicles, etc.
- Existing frameworks
  - TensorFlow, Caffe, TinyDNN, Theano, etc.
- Challenges
  - Lack of OpenCL support
    - Do not support multiple architectures (exception Caffe)
    - Do not support performance portability
  - Embedded systems issues
    - Huge computational and communication demands
    - The stringent size, power and memory resource constraints
    - High efficiency and accuracy

TensorFlow

- Front-end: graph-based model
  - Tensor (input/output data)
  - Operations (unit of computation)
- Back-ends
  - Eigen (main): C++ template-based linear algebra library
    - Front-end: expression tree-based model
    - Backend: CUDA, CPU
  - CuDNN : NVIDIA neural network library
  - Embedded built-in operations



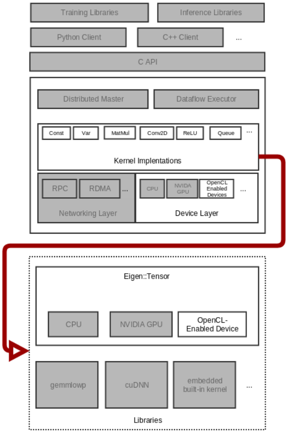
The Aim

Adding an OpenCL 1.2 backend to the existing TensorFlow framework.

- The added backend must be a non-intrusive approach
  - Should not change the front-end interface
  - Should be able to use the existing backend code as much as possible

Proposed Approach

- Adding SYCL backend for Eigen framework main backend of TensorFlow)
- Registering kernel implementation in TensorFlow for SYCL backend
- Registering OpenCL-enabled Devices as TensorFlow-supporting Devices



SYCL Programming Model

- A royalty-free, open standard from the Khronos Group
- ComputeCpp implementation used here
- Cross-platform performance portability
- Completely standard C++
- Single-source programming style

SYCL Simple Example

```
#include <array>
#include <CL/sycl.hpp>
using namespace cl::sycl;
template <typename T> class SimpleVadd;

template<typename T, unsigned long ORDER>
void simple_vadd(std::array<T, ORDER> &VA, std::array<T, ORDER> &VB,
                 std::array<T, ORDER> &VC) {
  // Queue creation
  queue q;
  // buffer creation
  buffer<T, 1> bA{VA.data(), range<1>{ORDER}};
  buffer<T, 1> bB{VB.data(), range<1>{ORDER}};
  buffer<T, 1> bC{VC.data(), range<1>{ORDER}};
  // queue submit scope
  q.submit([&](handler &cgh) {
    // convert host buffers to device accessors
    auto pA = bA.template get_access<access::mode::read>(cgh);
    auto pB = bB.template get_access<access::mode::read>(cgh);
    auto pC = bC.template get_access<access::mode::write>(cgh);
    // kernel scope
    cgh.parallel_for<class SimpleVadd<T> >(
        range<1>(ORDER), [=](id<1> it) {
          pC[it] = pA[it] + pB[it];
    });
  });
}
int main() {
  std::array<int,4> A = {1,2,3,4}, B = {1,2,3,4}, C;
  simple_vadd(A, B, C);
  return 0;
}
```

Why SYCL

Eigen's kernels follow a heavily C++-template-based expression tree model

- SYCL has the ability to dispatch device kernels from C++ applications, similar to CUDA, etc.
  - OpenCL 1.2 does not support C++
  - OpenCL 2.1 does support C++ templates inside the kernel
    - The kernel itself cannot be templated, therefore we still need different kernel registration per type
    - Expression tree-based kernel fusion is challenging without embedding a custom compiler

SYCL enables C++ code to run on OpenCL 1.2 which is widely supported on low-power platforms.

Why SYCL-(Continued)

Eigen uses the single-source programming model for both CUDA and CPU.

- SYCL supports single-source programming style
  - No need to implement separate kernel code for each operation
  - Use the same existing template code for both host and device
  - OpenCL needs to re-implement the backend and maintaining it would be hard

Challenges

- Eigen data storage
  - Standard pointer type for both CUDA and CPU
  - CUDA supports standard pointer
    - C-style pointer with no annotation will be created on the host side
    - The same pointer will be used on the device kernel
  - C-style allocation/deallocation of memory
  - OpenCL1.2 does not support standard pointer type to be created on the host as a device memory and to be used on the device kernel

Proposed Solution

- Introducing a *pointer mapper* structure to mimic standard pointer construction
- Using the template-based pointer class for leaf node in order to parametrize the expression type construction
  - Constructing the host expression using the pointer mapper structure
  - Preserving the Eigen expression interface
- Reconstructing the Eigen expression at compile time for the device kernel
  - Converting the pointer mapper structure of the leaf node to the actual device pointer at compile-time

Pointer Mapper

- Front-end: return a virtual pointer when memory allocation is called
  - Provide the same expression construction interface as CUDA and CPU
- Backend: a map structure
  - Create a one-to-one correspondence between the virtual pointer and the actual SYCL buffer on the device scope.

Pointer Mapper

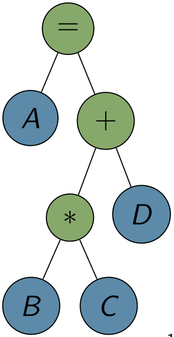
- On memory allocation:
  - $< KEY, VAL >$
  - KEY : virtual pointer
  - VAL : SYCL buffer
  - Return the virtual pointer
- On memory manipulation
  - Retrieve the buffer from the pointer
  - Apply the operation on the buffer
  - Arithmetic operations have been deduced from the virtual pointers and added to the buffers.
- On Memory deallocation:
  - Retrieve the buffer from the pointer
  - Delete the buffer
  - Remove the fragmentation in virtual pointer space

Expression Re-construction

Compile-time reconstruction of the actual expression from virtual pointer expression happens in 3 scopes.

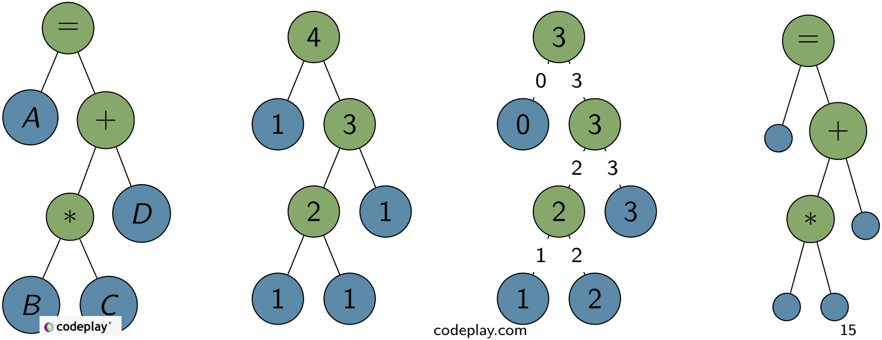
- Host Scope
- Queue Submit Scop
- Kernel Scope

- Example: A tree style representation of an expression. $A = B * C + D$



Expression Re-construction-Host Scope

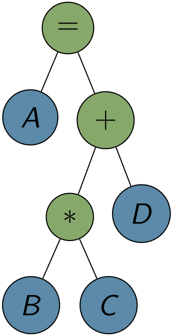
- Number the terminals of the expression, in depth-first order.
- Generate a placeholder expression type by replacing terminal types with a compile-time index type.
- Traverse the Expression tree in order to store all the stateful objects (e.g functors, dimensions)



Expression Re-construction-Queue Submit Scope

Compile-time reconstruction of the actual expression from virtual pointer expression happens in 3 scopes.

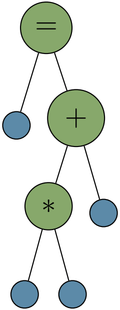
- Convert the leaf node buffers to accessors and store them on a tuple by traversing the Eigen expression.

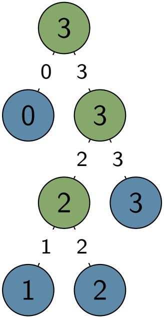


- $\ll Accessor(A - buffer),$
  $Accessor(B - buffer),$
  $Accessor(C - buffer),$
  $Accessor(D - buffer) \gg$

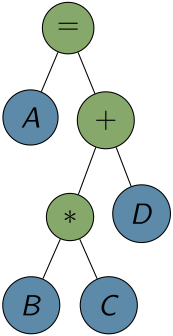
Expression Re-construction-Kernel Scope

- Convert the host pointers on the expression tree to the device pointers.
- Instantiate the device expression tree and run it on the device



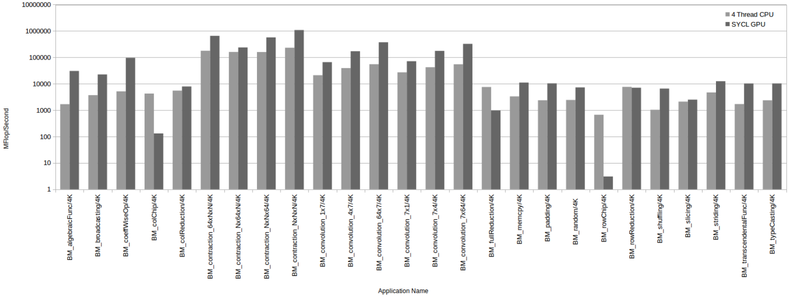


- $\ll Accessor(A-buffer), Accessor(B-buffer), Accessor(C-buffer), Accessor(D-buffer) \gg$



Performance Evaluation

Following is the execution of TensorFlow operators benchmarks using Eigen backend on Intel i7-6700K CPU backend @ 4.00GHz for CPU and AMD Radeon R9 FURY for SYCL backend. The result shows that for large scale tensor we will achieve up to one order of magnitudes speedup over 4 threads CPU when running on SYCL backend.



Conclusion & Future work

- Enabling OpenCL backend for TensorFlow
  https://github.com/lukeiwanski/tensorflow
- Enabling Eigen Tensor backend
  https://bitbucket.org/mehdi_goli/opencl/
- Achieving up to 5 times speedup over multi-threaded CPU code.
- Future work
  - Vectorising kernel Operations
    - Enabling Eigen-level vectorisation
  - Improving reduction operation
  - Completing the registration of all TensorFlow operations
  - SYCLBLAS
    https://github.com/codeplaysoftware/sycl-blas

Acknowledgement

This work has been supported by Low-Power Parallel Computing on GPUs 2 (LPGPU2) and the Knowledge Transfer Partnerships programme (KTP). LPGPU2 is a EU-funded research project under the category of Horizon 2020 projects (project number: 688759) for low powered graphics devices. Further information can be found at `http://lpgpu.org/wp/`