Implementation of a Computation Graph Framework

- a term project report for the course Multicore Processing Fundamentals (CSI-6505-01) -

Lukas Prediger

Yonsei University
`lukas.prediger@rwth-aachen.de`

1 Introduction - Project Goals and Outline

The term projects rules allowed for the choice of any topic that involves parallelization of computation either by leveraging multithreading on a CPU or the use of a GPGPUs massive parallelism capabilities and then comparing the parallelized version to a sequential baseline implementation. Instead of just re-implementing an existing sequential algorithm, I was interested in exploring how a general framework that allows for the parallel execution of arbitrary computations would have to be realized and how this generalization affects performance compared to an implementation specifically designed for a specific problem.

My project goal was thus the implementation of such a framework which allows to represent a mathematical computation on data as a hardware- and data-independent *computation graph.*[1]. To perform the computation represented by this graph, the framework compiles the graph into specific instructions to be executed on a hardware platform specified by the programmer. In compliance with the project rules, the two platforms implemented in the framework are a sequential CPU implementation and an implementation using OpenCL for execution on the GPGPU.

This report will briefly discuss the general architecture (Section 2) and present the compilation process of the graph (Section 3), as well as point out some features implemented specifically to increase performance (Section 4). It will then compare the performance of the CPU and OpenCL/GPU implementations on several examples as well as both framework implementation variants with a problem specific sequential CPU baseline implementation for one specific problem to examine the general abstraction overhead of the framework (Section 5). Finally the report will conclude with some remarks and options for future work (Section 6).

Please note that this report just aims to give an overview about the general workings of the framework and its performance. Thus, no code samples will be provided in the following. The frameworks source code including all examples mentioned in this report and instructions for compilation can be found in the

[1] TensorFlow (Abadi et al., 2016) was the obvious inspiration for this

github repository available at https://github.com/lumip/computegraphlib. All results reported in this document refer to commit 140b70e.

1.1 Computation Graphs

As mentioned in the paragraphs above, the framework will handle computation represented as *computation graph*s. These are directed graphs where each node represents a mathematical operation and the edges represent the flow of data: Incoming edges of a node are the inputs to that node from preceeding computations, outgoing edges accordingly mean that the results of the operation represented by the node will be used as input by the target node of the outgoing edge. Figure 1 shows a very simple graph representing the computation of $z = a \cdot (x + y)$.

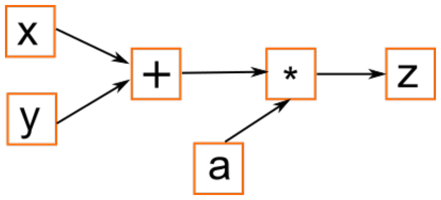


Fig. 1. A simple computation graph representing $z = a \cdot (x + y)$.

Note that the graph does explicitely not have to be circle-free and thus allows to feed back results to earlier stages of the computation. This does not affect a single evaluation of the graph, as each node is only evaluated once, but plays a role when the graph is evaluated multiple times. However, the framework currently requires every graph to have a single final node that is eventually reached from every other, i.e., if f is this final node, for every node v in the graph, there has to be a path from v to f.

Performing the computation represented by a computation graph will be referred to as *evaluating* or *executing* the graph in this report.

2 Framework Architecture

The major components of the framework are its collection of *Node* classes forming the API for creating hardware-agnostic computation graphs, the *GraphCompiler* and *CompiledGraph* classes which constitute the graph compilation API (along with some helper classes) and, finally, the implementations of the *GraphCompilationPlatform* interface along with the *Kernel* classes which offer platform-specific implementations to the *GraphCompiler* through a generalized interface.

2.1 *Node* Classes

As mentioned above, the *Node* classes form the frameworks API by which an application programmer can represent a computation graph. Each subclass of *Node* represents an operation. There are currently 16 such nodes implemented to represent operations such as element-wise addition or multiplication, matrix multiplication, slicing/stacking of data blocks and more. There are also special nodes to represent variables of and inputs to the graph.

Note that the framework assumes that data is passed along edges in matrix format, i.e., as a data block of size $m \times n$ for arbitrary natural number values of m and n. The data does not necessarily have to be a matrix in the mathematical sense but can also represent a batch/stack of vectors or even a single scalar value ($m = n = 1$). Some of the nodes allow for broadcasting operations, e.g., the *VectorAddNode*, which implements element-wise addition, allows inputs to be of size $m \times n$ and $1 \times n$ and will then treat the first input a batch of m vectors of size n and add the second input to each row of the first.

The subclasses of *Node* serve the sole purpose of representing the graph structure and are thus agnostic of the actual input data or even its dimension as well as the hardware platform the graph will be evaluated on. This allows application developers to implement graph structure in exactly the same way for different inputs and hardware platforms. It would also allow storing the graph structure into a file and porting it to a different system, although this is currently not implemented.

For an overview of the *Node* classes currently implemented, please consult the source code repository.

2.2 *GraphCompiler* and *CompiledGraph*

The *GraphCompiler* class implements the graph compilation process detailed in Section 3. It is written in a platform-agnostic way and relies on implementations of the *GraphCompilationPlatform* interface for platform specific operations. The two major platform-specific tasks are the allocation of memory for nodes and the compilation of *node kernels* which are the platform-specific implementation of the operation represented by a *Node* class in the graph.

An implementation of the *CompiledGraph* interface is the result of the compilation process and represents the platform-specific implementation of a graph. As such, it is tied to the platform it was compiled for and the concrete dimensions for the input variables, which have to be supplied to *GraphCompiler* at the start of the compilation process. *CompiledGraph* exposes methods to evaluate the graph for a certain input set as well as retrieve output data from the nodes of the graph.

2.3 *GraphCompilationPlatform* and *Kernel* classes

As mentioned above *GraphCompilationPlatform* is the interface abstracting a concrete platform implementation to the *GraphCompiler*. The framework currently implements *GraphCompilationCPUPlatform*, providing the sequential CPU platform, and *GraphCompilationGPUPlatform*, implementing the OpenCL/GPU platform.

The *Kernel* classes are the platform-specific implementations of node operations. For the most part there exists a *Kernel* class for each *Node* class per platform, except for cases where a node operation could be expressed using already existing *Kernel* implementations. *Kernel* classes receive references to the buffers that will hold the input and the output data at construction time and provide a method *Run()* which will perform the computation.

For the CPU platform, *Kernel* classes implement the computation operation in a straightforward way in their *Run()* method. *Kernel* classes of the GPU platform will compile OpenCL kernels at construction time and enqueue them to the GPU in their *Run()* method.

3 Graph Compilation Process

As briefly mentioned above, graph compilation is the process that translates the representation of the computation as a computational graph into actual kernel implementations for a specific hardware platform as well allocating required memory to store results for a given input size. The process is driven by the *GraphCompiler* class and has the following major steps:

1. Establish a topological order of the graph's nodes, starting from the leafs.
2. Determine the required size of the memory buffer each node needs to store its computation result in. (In the following, such a buffer will be referred to as a node's working memory.)
3. Determine whether subsequent nodes may share the same working memory.
4. Let the platform implementation allocate the required memory buffers.
5. Let the platform compile the kernels for all nodes.

3.1 Ordering the Nodes

The first step establishes a topological order of the graph's nodes which allows the following steps to iterate over a simple list rather than dealing with graph traversal. Having this list be in topological order is important because processing each node requires its input nodes to be already processed.

As mentioned previously, the framework allows graphs to contain circles, which prevents establishment of a true topological order. This is dealt with by only allowing circles that contain an instance of *VariableNode*. During the ordering, the input edge to this node is ignored and the corresponding node at the other end of this edge is treated as another "root" node with no outputs. Establishing a circle in a grahp that does not contain exactly one *VariableNode* instance will currently result in unspecified behavior of the compilation process.

3.2 Determining Buffer Size

Determination of the sizes of working memory buffers for each node proceeds linearly along the topologically sorted list of nodes. The process requires the application programmer to provide a mapping of graph inputs to input data dimensions and sets the working memory dimensions of *InputNode* instances accordingly. All other nodes will examine the working memory dimensions of the nodes that are their respective inputs and calculate their own working memory dimensions according to that.

3.3 Determining Buffer Reuse

After determining the required size of working memory for all nodes the process will try to minimize memory usage by reusing buffers among subsequent nodes. Let u be a node which is an input to node v, then v can reuse u's working memory iff

1. u's result is not used by any other node $w \neq v$
2. v is able to operate in-place
3. u's working memory buffer is equal or larger than v's in both dimensions.

As in the previous step, the process iterates over the topologically sorted list of nodes. For each node it encounters (called the current node), the algorithm iterates over its input nodes to see if it finds one that meets the above requirements. If so, it maps the current node to the same memory buffer. If not, it will map the current node to a new memory buffer.

The above outlines the general idea. The process is a bit more complicated in reality to deal with possible circles introduces by *VariableNode* instances: If the algorithm encounters an input node that does not yet have a buffer assigned to it but meets the requirements, it will allocate a buffer of sufficient size and then map both nodes, the current one and the input, to this buffer. Whenever it iterates to a new node in its main loop, it will first check whether this node already has a buffer associated with it. If that is the case but this buffer is not one of its inputs, it will proceed to look for an input buffer meeting the above requirements. If it finds one, it will merge its currently assigned buffer and the input buffer it found, i.e., assign all nodes assigned to one of the buffers to the other one and then delete that first buffer.

3.4 Allocating Buffers and Compiling Kernels

Note that no actual memory is allocated thus far to avoid constant allocation and deallocation of memory during buffer merges. It also allows the compilation process to perform this general computations without regard for the target platform. Thus, this in this next step, the *GraphCompiler* issues a call the the *GraphCompilationPlatform* to allocate memory and map the virtual buffers from the previous steps to actual memory handles.

In the next step, it will cause the *GraphCompilationPlatform* to instantiate *Kernel* objects for each node, which represent the concrete platform-specific implementation of the node operations. The kernels are stored in the same topological order as the nodes they correspond to. For the OpenCL/GPU platform, instantiation of a *Kernel* class implementation of a specific node will result in the compilation of OpenCL kernel source code. To avoid compiling the same kernel over and over again if the same node type is used several times in a graph, a cache is employed which maps OpenCL kernel source code to compiled OpenCL kernels, ensuring that each unique kernel source code is compiled only once.

Each *Kernel* instantiation encapsulates the platform-specific implementation of the operation represented by the corresponding node as well as the handles to the input and working buffers for that node. The ordered list of instantiated *Kernel* objects is returned encapsulated in a *CompiledGraph* object to the application code invoking the compilation process.

3.5 Graph Evaluation

Executing the graph is triggered by a call to the *Evaluate()* method of the *CompiledGraph* object. Application programmers have to make sure to first initialize potential *VariableNode* buffers by passing a mapping of variable names to variable initialization data to the *InitializeVariables()* method of the same object. The *Evaluate()* method expects a similar mapping for input names to input data. All data passed into the graph has to be a continuous memory buffer of the size specified by the input dimensions passed into the compilation process and will be treated as row-major linearization of the matrix layout represented in the graph.

The evaluation of the graph iterates over all *Kernel* objects and invokes the *Run()* method. As mentioned in Section 2.3, this simply executes the computation for the CPU platform and enqueues the kernels to the GPU for the OpenCL/GPU platform. Following the topological order of the graph nodes in this step is important on the CPU platform so that each nodes inputs are computed before its outputs. For the OpenCL/GPU platform, this is ensured using OpenCL events, such that the order in which nodes are enqueued is not important.

4 Framework Performance Features

Some features were implemented within the framework to reduce the performance and resource usage the impact of the abstraction overhead. As mentioned in the previous section, working memory reuse reduces the total amount of memory that is allocated by the framework and OpenCL kernel caching prevents multiple compilation of OpenCL programs. Two other features which will be discussed briefly in this section are mapped memory and parallel kernel execution capability for the OpenCL/GPU platform.

4.1 Mapped Memory

The execution of the graph requires the input data to be present in the working memory of the input nodes. This typcally involves costly copy operations from the memory locations holding the input data provided by the application programmer into the working memory during which the actual computation is idle. For the OpenCL/GPU platform this involves a memory transfer to device memory across the PCIe bus.

To reduce this copy overhead, the *CompiledGraph* class exposes the method *GetMappedMemory()* which takes an input or variable name as input and returns a handle to a region of memory that is mapped to the corresponding nodes working memory. The *CompiledGraph* class internally relies on the specific *GraphCompilationStrategy* implementation to provide this handle.

The CPU platform implementation will simply return the pointer to the working memory buffer of the node, allowing an application to read input data directly into it. This completely eliminates the need for a separate buffer on application side and thus the necessity of the memory transfer operation. At graph execution, the platform will check whether the buffer provided in the input data mapping is that node's working buffer and skips copying in this case. If the buffer in the mapping is different from the node's working memory, the copy operation is performed. Since the check is necessary in any case, even if only mapped memory is used, there is a small but, compared to copy time, negligible overhead still.

On the GPU platform, mapped memory is implemented by employing the OpenCL buffer allocation and mapping API to allocate (hopefully) pinned memory regions from which the copy to device memory can be performed by DMA, significantly improving throughput.

4.2 Parallel Kernel Execution

A computation graph of sufficient size typically has nodes that do not depend on each other (i.e., for two nodes u and v, u is not an input to v and v is not an input to u) and thus can, in principle, be executed in parallel. The framework supports this behavior for the OpenCL/GPU platform by allowing the OpenCL command queues to execute out of order and ensuring correct order of execution among dependent nodes by making use of OpenCL events and event waiting. If the OpenCL implementation and hardware supports parallel kernel execution, this allows kernels to be executed in parallel.

Note that the CPU platform as it is currently implemented is limited to executing kernels in sequence according to the topological order of nodes determined in the graph compilation process.

5 Performance Evaluation

To evaluate the performance of the framework, it has been subjected to several test runs which were intended to assess different aspects of the implementation.

As such, there were tests that compared only the platform-specicfic kernel implementations for a single node, a test concerning a larger graph with lots of independent nodes as well as a practical example which used the framework to implement and train a simple linear softmax classifier for the MNIST dataset.

All execution timings were taken using the PAPI library on a portable computer with an Intel(R) Core(TM) i5-3230M clocking at 2.6Ghz, 8GiB of main memory and an NVidia GTX 650M GPU with 2 compute units with 192 processing elements each and 2GiB of device memory.

All performance timings were obtained by executing the corresponding test executable 11 times, eliminating the result with the highest runtime and taking the mean of the remaining. The following results will only present charts to make the results easy to grasp. The Appendix A lists the results in numbers.

5.1 Kernel Implementations

The kernel implementation tests did not use the *GraphCompiler* class to maintain direct access to the underlying structures and thus be able to precisely measure only the runtime of that nodes kernel for the given platform. Almost no framework overhead is involved in the actual kernel execution such that a separate comparison to a pure CPU baseline implementation was not deemed to be necessary. Note that for the kernels to operate they require inputs to be present, such that *InputNode* objects (which perform no operation themselves) have been included in these tests as well as the actually tested node. The results presented in the following are for vector addition, matrix multiplication as well as a reduce mean operation.

Vector Addition (represented as *VectorAddNode* in the framework) is one of the examples that falls into the category of embarassingly parallel problems. As such, a major performance improvement can be observed for the parallelized OpenCL/GPU kernel. The left side of Figure 2 shows the time required for allocating memory and compiling the kernel (*setup time*), the time required for copying the input data into the working memory of the input nodes using mapped memory (*copy time*) as well as the time of the actual computation (*computation time*) for a vector batch of 50000×1000, amounting to 50 million elements. As was to expect, the OpenCL/GPU kernel outperforms the CPU kernel significantly with a speedup factor of about 12.5. It does, however, have a significantly higher setup time caused by the necessity for the complex OpenCL kernel compilation. It also has to spent more time for memory transfer, which is completely negligible for the CPU kernel. To further emphasize this point, the right side of Figure 2 shows only the setup and copy time with a logarithmic scale on the y-axis.

This overhead reduces the overall speedup of the OpenCL/GPU kernel compared to the CPU kernel to roughly 3. For larger amounts of data, it can be expected that the ratio of setup time to computation time will diminish further, meaning that the total speedup will increase further towards the 12.5 mark (but not reach it since the time to copy larger amounts of data will also increase for the OpenCL/GPU kernel).

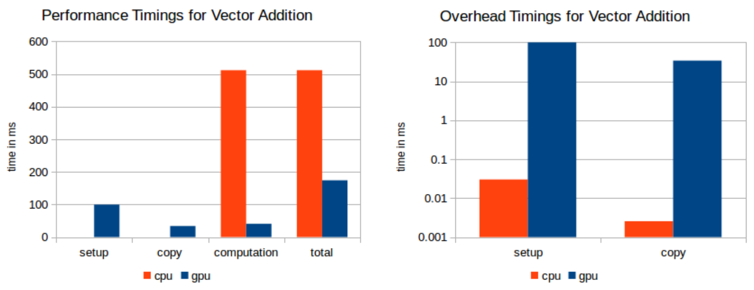


Fig. 2. Performance timings for *VectorAddNode*.

Matrix Multiplication (represented by the *MatrixMultNode* in the framework) presents a problem that allows for slightly different approaches to parallelization which do not perform equally well.[2] In the framework implementation, one OpenCL work item is created for each column of the result matrix which will calculate all row entries of that column (with the result that memory accesses of all parallelly executing work items can be coalesced). Figure 3 shows the results for a multiplication of matrices with dimensions 1000×10000 and 10000×1000 respectively. It is evident that, again, the parallelized OpenCL/GPU kernel per-

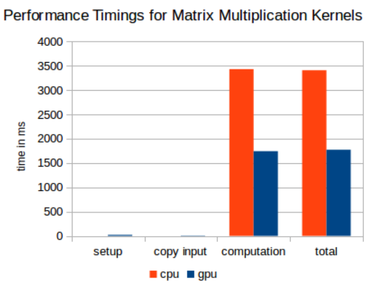


Fig. 3. Performance timings for *MatrixMultNode*.

forms faster than the sequential CPU kernel in terms of pure computation by

[2] I am referring to the lecture here.

a factor of about 1.97. Since the overall runtime for the computation is high, the relative significance of setup and copy cost, in which the CPU implementation trumps the OpenCL/GPU variant again, is very low: In total time the OpenCL/GPU kernel still achieves a speedup of 1.92. Note that parallelization benefit for OpenCL/GPU is more pronounced for larger matrices: For a multiplication of matrices with 4000×10000 and 10000×4000 elements, the speedup of total time is 6.32 (not shown in the figure; refer to Appendix A).

Figure 4 explores the effect of implementing mapped memory for both platforms in the matrix multiplication example. As can be seen clearly, the time spent in the copy routine reduces to almost none for the CPU platform. The usage of DMA transfer from a pinned memory buffer for the OpenCL/GPU platform reduces copying time by a factor of 2.14 in this example. However, since the data copying time is overall negligible in this example, the impact on overall runtime is low.

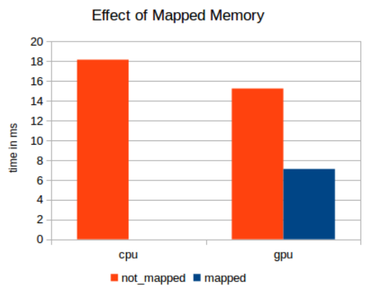


Fig. 4. Time for memory transfers (copy time) in the matrix multiplication example for mapped and not-mapped input data buffers.

Reduce Mean (represented by *ReduceMeanNode*) is the operation that takes the mean of all elements along one axis of the buffer (row or column) and places the results in a vector along the other axis. The process can be trivially parallelized with one worker per row/column. However, due to the row-major layout of data in the memory buffers, the choice of the reduction axis has an impact on the speedup that occurs using this parallelization scheme. Figure 5 illustrates this by showing evaluation times for both platform implemenatations along both axes for a buffer of size 15000×15000. Setup and copy time were omitted in the figure as they are not relevant to the point of interest.

As can be seen, the choice of reduction axis indeed plays a significant role. A reduction along columns (axis 0) is much faster than along rows on both platform and allows the OpenCL/GPU implementation to exploit its inherent parallelism and thus outperform the CPU implementation. For reduction along rows it seems to be unable to benefit from parallel exeuction and actually requires slightly more time for the computation. Thus, comparing column reducing to row reducing shows a drastic performance difference for the OpenCL/GPU platform implementation.

This effect is probably caused by the different memory access patterns depending on the axis. When reducing along columns, all work items for the OpenCL/GPU platform read adjacent memory locations in every iteration step, resulting in coalesced reads. For row reduction, the locations read by the work items are separated by a stride equal to the size of a row, making coalesced read impossible and thus decreasing memory throughput drastically to the point of effectively serializing the computation.

The CPU platform is implemented in a way that it will always iterate over the input in sequence (i.e. according to row-major layout) and add each element it encounters to an aggregation variable according to the reduction axis. When reducing columns, the cursor to the aggregation variables increases with each element and wraps around after reading a row of the input data. When reducing rows, the cursor increases which each row read. In theory, access to the input data as well as the aggregation variables should happen in a cache-friendly manner independent of the chosen axis. Interestingly, it is the variant where the input cursor changes less often and thus should put less strain on the cache that performs worse. This behavior puzzles me somewhat. My hypothesis is that in this case the single cache line storing the aggregation variable get swapped out of the L1 cache while iterating over the row because cache lines inside fall into the same cache line set.

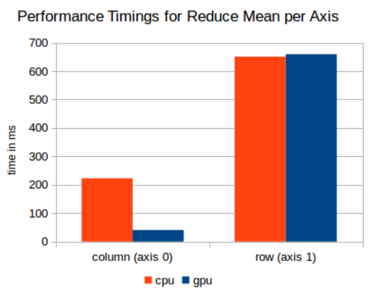


Fig. 5. Performance timings for *ReduceMeanNode* operating on both axes.

5.2 Full Graph Implementation: Massively Parallel Matrix Multiplications

To test the overall integrated working of compiling and executing a graph on the framework, a simple graph structure was used that also allows for parallel execution of kernels by having a large number of independent nodes. The graph constitutes of up to 8192 nodes that perform a matrix multiplication of the same two input matrices, which have a size of 200×100 and 100×200 respectively. Each such multiplication is thus quite small but the large number of them means that the overall graph execution time will be large if all operations are executed sequentially. The multiplication results are then summed up in a binary tree reduction scheme which can also be computed in parallel for each layer (/level of depth).[3] The structure of the graph is shown visually in Figure 6.

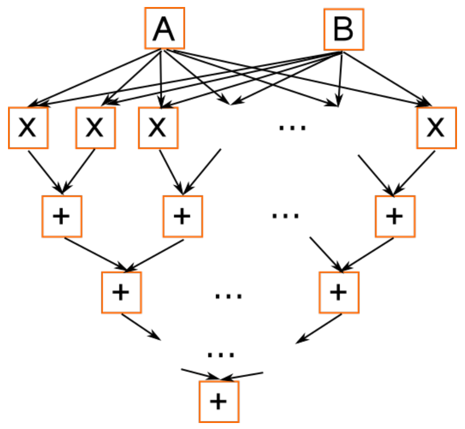


Fig. 6. Illustration of the matrix multiplication graph used in the performance evaluation.

Figure 7 shows graph evaluation times depending on the amount of nodes in the matrix multiplication layer on the left side. This includes the time to copy the input data using mapped memory. The number of nodes increases in steps of exponents of 2 from 512 to 8192. Both implementations show a linear increase, indicating that the OpenCL/GPU implementation fails to parallelize kernels. Further investigation suggests that this is an issue with either the OpenCL implementation or the hardware present in the testing system, which does not allow for parallel execution of OpenCL kernels.[4] However, since the execution time for each kernel is faster in the OpenCL/GPU implementation, the slope of the corresponding line in the graph is smaller and for larger amounts of nodes, a gap between evaluation durations becomes noticeable.

[3] Note that the buffer reuse described in Section 3 means that only memory for the intial layer of matrix multiplications needs to be allocated; the following summation layers can reuse the same buffers.

[4] For two subsequent calls to *clEnqueueNDRangeKernel()*, the second call will block if the kernel enqueued by the first call has not finished execution.

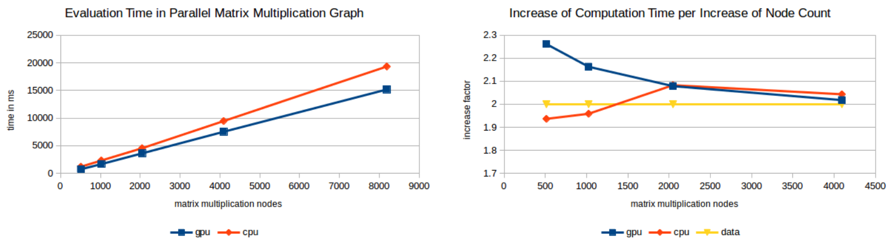


Fig. 7. Evaluation times (including copying) for the parallel graph example depending on the size of the multiplication layer (left) and relative increase in evaluation time to the previous layer size (right).

The right side of Figure 7 shows the increase factor in graph evaluation time for both executions depending on the increase in nodes between the steps (yellow line, 2). This further reflects the linear scaling in the number of nodes, as increase factors for both, CPU and OpenCL/GPU, implementations also are about 2. Interesting to note is that the increase factor for the OpenCL/GPU variant is slightly higher for small numbers of nodes but smoothly converges towards the value of 2. This is very likely due to the copy time overhead, which is significantly higher for this implementation than for the CPU variant but the impact of which diminishes as overall computation time increases. The CPU increase factor fluctuates very slightly around the baseline value of 2, which is probably just random noise.

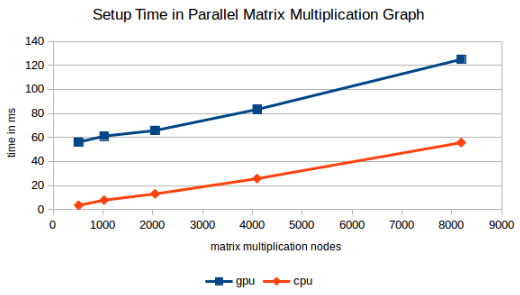


Fig. 8. Setup times for the parallel graph example depending on the size of the multiplication layer.

Finally, Figure 8 shows a similar linear dependency on the number of nodes for the graph setup/compilation time. In agreement with the results for single kernel performance measurements, the OpenCL/GPU implementation takes much longer here, beeing constantly at least 50 ms slower than the CPU one and also seems to have a slightly higher slope.

5.3 Practical Use Case: MNIST Softmax Classifier Training

As a final example the framework was tested with a small practical example in form of a graph representing a linear softmax classifier for the MNIST dataset and its training process. MNIST, introduced by LeCun, Bottou, Bengio, and Haffner, 1998, is a famous sample dataset for machine learning and consists of 60000 images of 28 × 28 pixels showing handwritten digits as well as a label providing the digit shown in the image. With this, a classifier is trained that predicts the digit corresponding to unseen samples out of a test set of 10000 more images also provided by the dataset.

To keep things simple I followed the corresponding tutorial for the TensorFlow framework (TensorFlow, 2017) to construct the graph representing the classifier. As the cited tutorial goes through the setup in detail, I won't replicate it here. For training the classifier, I added a feedback circle to the graph which updates the parameters of the classifier using gradient descent with a fixed learning rate. The following sections will first compare performance of both implementation variants on a single forward pass for different sizes of input batches. It will also include comparisons to two pure CPU benchmarks (one using AVX instructions to leverage instruction level data parallelism of the CPU) that were tailored towards the problem and do not make use of the framework to get an impression of the abstraction overhead. Afterwards, the performance of both platform implementations is compared for a full training process.

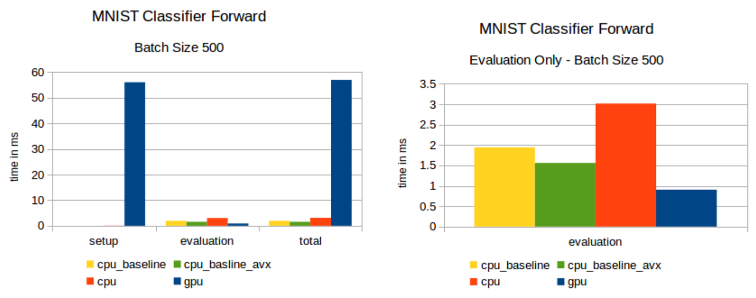
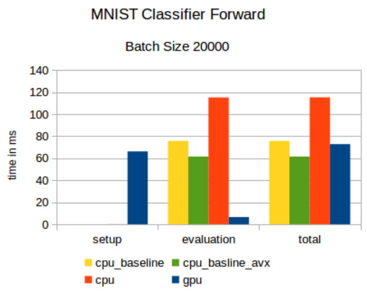


Fig. 9. Performance timings for a forward pass through the MNIST classifier with a batch size of 500 (left). The (right) chart is zooming in on evaluation time only.

Forward Pass Figure 9 shows the full performance timings for all four implementations using an input batch size of 500 (with 728 elements per input row) on the left. As in the previous section, the graph evaluation time includes time to copy the inputs (if copying occurs). It can be seen that the OpenCL/GPU framework implementation has a huge setup overhead time, which also dominates overall runtime. A single forward pass is fast for all implementations such that a large setup time has almost prohibitive impact.[5] Looking only at the true evaluation time (right side of the figure), the OpenCL/GPU framework implementation performs fastest, followed by the AVX baseline, plain baseline and then the CPU framework implementations in that order, as was to expect. This indicates that the kernel implementations of the OpenCL/GPU are performing very well and, if setup overhead can be accounted for, outperform problem-tailored sequential CPU implementations.

Figure 10 proves this point by showing the same timings for a batch size of 20000, where the problem size for a single pass throught the graph gets big enough for the setup time to start to pay off: The OpenCL/GPU implementation outperforms all other implementations except for the AVX baseline (although the plain baseline just barely). In terms of pure evaluation time, it clearly beats all other implementations with speedup factors > 9.



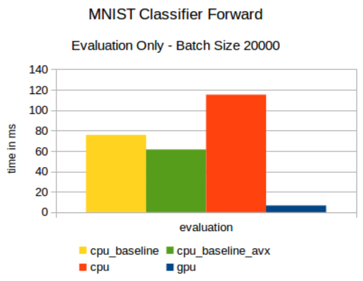


Fig. 10. Performance timings for a forward pass through the MNIST classifier with a batch size of 20000 (left). The (right) chart is zooming in on evaluation time only.

Finally, Figure 11 shows how the implementations scale in pure evaluation time with increasing batch sizes on a doubly-logarithmic scale (for easier readibility). As is to expect, all CPU implementations scale linearly due to their sequential nature. The OpenCL/GPU platform variant however, scales sublinearly and thus gains more advantage the bigger the input batches become, although this gap widens less towards greater batch sizes.

Training Figure 12 shows the performance timings for 10 epochs of training the MNIST classifier for batch sizes 500 (left) and 20000 (right). Note that an

[5] Of course, the baseline implementations have no setup (or copying) cost as they are not using the framework.

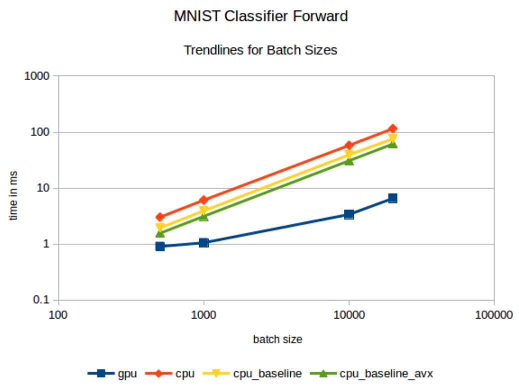


Fig. 11. Evaluation time over batch size for all implementations of the forward pass throught the MNIST classifier.

epoch always processes all 60000 samples in the training set, a different batch size just means that more batches are processed during one epoch (and thus the graph is executed more often). Since the pure computation time advantage of the OpenCL/GPU implementation is much more pronounced for larger batch sizes in a forward pass, it was expected that training is faster for this case and the difference between both platforms is larger for the larger batch size. Interestingly however, this is not the case. For the larger batch size, both platform variants got slightly faster, however not in any significant way (1 to 2 seconds). The difference between the platforms remained the same across different batch sizes. Note that setup time is negligible for both platform as the overall evaluation time is high.

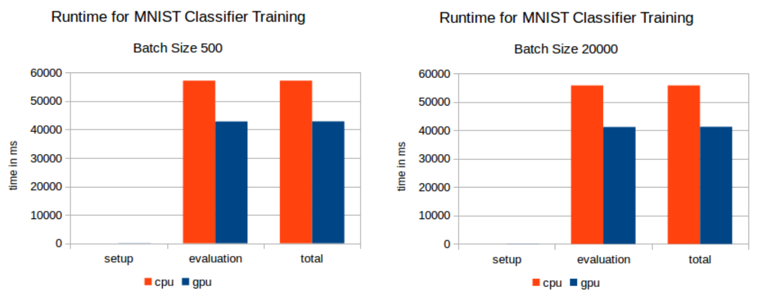


Fig. 12. Performance timings for 10 epochs of training the MNIST classifier for batch sizes of 500 (left) and 20000 (right).

Since the much faster evaluation of a single pass through the graph for larger batch sizes for the OpenCL/GPU implementation seems to not manifest any effect in the training here, I suspect that the whole process is bottelnecked by copying the batch input data into the working buffers, meaning that the actual implementation variant has only a small effect on overall runtime.

6 Conclusion

This report briefly summarized architecture and core features of the created framework and conducted an evaluation on several examples to highlight different aspects of framework performance. In general, the expectations towards the framework by the author were met by the results of the evaluation, demonstrating that the OpenCL/GPU implementation unsurprisingly outperforms the sequential CPU implementation in almost all cases, but is also capable of competing with problem-tailored sequential implementations if the problem size is large enough to amortize the graph compilation setup cost.

However, there is still a lot of work that could be done as some kernel implementations definetely need futher improvements. It would also be interesting to add a problem-tailored OpenCL baseline implementation to the MNIST forward pass evaluation to compare the framework overhead on that level as well. Another interesting aspect would be to find an OpenCL implementation that allows for parallel kernel execution for the parallel matrix multiplication graph as well as comparing the created framework to its source of inspiration, TensorFlow.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., … Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *OSDI* (Vol. 16, pp. 265–283).

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324.

TensorFlow, T. (2017). *MNIST for ML beginners.* (accessed December 19, 2017). Retrieved from https://www.tensorflow.org/get_started/mnist/beginners

A Performance Test Data

	setup	copy	computation	total
CPU	0.0300	0.0026	511.4991	511.5431
OpenCL/GPU	99.4795	33.8463	40.7044	174.0540

Table 1. Performance timings in ms for *VectorAddNode* for a vector with 50 million elements using mapped memory.

		setup	copy	computation	total
CPU (w/o mapped memory)	[1k rows]	0.0358	18.1613	3356.9046	3362.4792
OpenCL/GPU (w/o mapped memory)	[1k rows]	21.0623	15.2451	1742.0700	1778.2640
CPU	[1k rows]	0.0288	0.0027	3431.1673	3406.8097
OpenCL/GPU	[1k rows]	22.2867	7.1111	1742.5283	1771.9261
CPU	[4k rows]	0.0353	0.0025	49294.9499	49295.9897
OpenCL/GPU	[4k rows]	78.3387	27.0026	7690.6851	7796.1219

Table 2. Performance timings in ms for *MatrixMultNode* with and without mapped memory on a 1000×10000 (and transposed) matrix pair as well as on a 4000×10000 (and transposed) matrix pair with mapped memory.

	reduction axis	setup	copy	computation	total
CPU	row	0.0272	0.0022	650.5087	650.5417
OpenCL/GPU	row	193.2303	75.4013	659.3438	928.0013
CPU	column	0.0299	0.0023	222.7609	222.7971
OpenCL/GPU	column	196.3027	75.4104	40.7376	312.7600

Table 3. Performance timings in ms for *ReduceMeanNode* on a 15000×15000 elements large input matrix for both reductions directions using mapped memory.

	# nodes	512	1024	2048	4096	8192
CPU	setup	3.4295	7.7354	12.9619	25.6724	55.6570
OpenCL/GPU	setup	56.0852	61.0913	65.7772	83.3012	125.0189
CPU	evaluation	1197.4120	2318.8319	4541.9213	9459.3411	19327.2259
OpenCL/GPU	evaluation	739.5297	1672.0622	3615.8352	7515.5722	1516.4396

Table 4. Performance timings in ms for the parallel matrix muliplication graph example on 200×100 (and transposed) input matrices for different numbers of node counts in the multiplication layer using mapped memory (evaluation time includes copying of data).

	batch size	500	1000	10000	20000
CPU	setup	0.0663	0.0809	0.0837	0.0735
OpenCL/GPU	setup	55.9960	55.5887	61.8241	66.1387
CPU baseline	evaluation	1.9408	3.9098	39.3514	75.7029
CPU AVX baseline	evaluation	1.5600	3.1315	30.8252	61.4217
CPU	evaluation	3.0150	6.0797	57.9689	115.1675
OpenCL/GPU	evaluation	0.9042	1.0529	3.3746	6.5463

Table 5. Graph evaluation time in ms for the forward pass through the MNIST classifier example for different batch sizes using mapped memory (evaluation time includes copying of data). Also includes baseline implementation timings. The baselines do not have setup times.

	batch size	setup	evaluation	accuracy
CPU	500	0.1184	57120.4066	0.9115
OpenCL/GPU	500	53.5477	42780.0779	0.9115
CPU	20000	0.1471	55765.7366	0.8338
OpenCL/GPU	20000	64.3618	41156.1538	0.8338

Table 6. Evaluation time in ms for 10 epochs of training the MNIST classifier example for different batch sizes using mapped memory. The evaluation times include all copying of data (including assembling batches from the general input buffer into the mapped memory regions). The last column gives the accuracy of the trained graph on the test set (i.e. the ratio of correctly classified samples).