Towards Mapping Lift to Deep Neural Network Accelerators

Naums Mogers
University of Edinburgh
United Kingdom
naums.mogers@ed.ac.uk

Michel Steuwer
University of Glasgow
United Kingdom
michel.steuwer@glasgow.ac.uk

Aaron Smith
University of Edinburgh
Microsoft Research
United Kingdom
aaron.smith@microsoft.com

Christophe Dubach
University of Edinburgh
United Kingdom
christophe.dubach@ed.ac.uk

Dimitrios Vytiniotis*
DeepMind
United Kingdom
dvytin@google.com

Ryota Tomioka
Microsoft Research
United Kingdom
ryoto@microsoft.com

*Work done when author was employed by Microsoft Research

Abstract

Deep Neural Network (DNN) accelerators enjoy a rise in popularity due to the ubiquity of DNN applications. Devices to accelerate DNNs – CPUs, GPUs, ASICs, FPGAs – vary significantly and pose an increasingly difficult challenge to extract performance from them. Approaches proposed to address this problem lack in either portability or extensibility.

Lift is a novel approach that produces performance-portable GPU and CPU code for linear algebra, sparse matrix and stencil computations. Lift uses rewrite rules to detect and transform patterns for parallelism, memory configuration and instruction set of the target hardware. This paper presents preliminary work in applying Lift to the generation of optimised code for DNN accelerators by mapping expressions to coarse-grained ISA primitives; discussion of the additions to the IR, type system, code generation and rewrite rules makes a case for extensibility of Lift.

Author Keywords

Compilation, Deep learning, Performance Portability

ACM Classification Keywords

[General and reference]: Performance; [Software and its engineering]: Source code generation; [Hardware]: Emerging languages and compilers; [Computer systems organization]: Neural networks

1 INTRODUCTION

Deep Neural Networks (DNNs) are compute-intensive and benefit from specialised accelerators. Approaches to hardware acceleration vary from repurposing Graphics Processing Units (GPUs) to designing new Application-Specific Integrated Circuits (ASICs) and synthesising new architectures on Field-Programmable Gate Array (FPGA) devices. GPUs are a good fit for DNN computations due to the abundance of floating-point units and large memory, that are required for training and inference in DNNs; they are economically affordable for research, small-scale production and offline mobile computations of DNNs.

ASICs can provide the best performance for DNNs since they can be tuned for optimal fit in silicon. However, design and production costs coupled with ever-changing computational demands of rapidly evolving DNNs make ASICs prerogative of big industry players. Examples of DNN ASICs include the Tensor Processing Units (TPUs) [6], Huawei Da Vinci architecture [9], the DianNao family [3] and Movidius Myriad [5].

FPGAs are a compromise between GPUs and ASICs in that they allow a great deal of customisability without the need to bear costs of redesigning silicon. Notable examples are Microsoft project BrainWave [2], work by Qiu et al. [10], Suda et al. [14] and Lu et al. [7].

The difficulty with all these platforms is in producing efficient code that makes optimal use of hardware resources. Detecting opportunities to use built-in optimised primitives is not trivial in itself; so is mapping computations onto hardware in a way that utilises available memory and computational units in an efficient way. Each platform requires an unique combination of optimisations and extensive domain knowledge which makes it costly to produce efficient implementations across multiple devices.

This paper discusses preliminary work towards extending performance portability to hardware accelerators for DNNs with Lift [12]. The functional data-parallel Lift language abstracts algorithms from implementation while capturing useful algorithmic information; this information is leveraged by the compiler for expression transformation using rewrite rules. The rules are used to explore parallel and memory mappings, as well as to detect opportunities to use platform-specific optimised primitives. Lift was shown to generate efficient OpenCL code for NVIDIA, AMD and ARM Mali GPUs [4], and Intel CPUs [12] for various applications. The work presented in this paper discusses how the Lift compiler is extended to detect patterns that are frequent in DNN implementations and exploit platform-specific built-in primitives to accelerate the computation.

This paper offers insights into how rewrite rules can be used to detect and rewrite combinations of generic primitives into coarse-grained operations supported by the accelerator. The functional programming paradigm makes it easy to reason about the algorithm both in terms of detecting useful patterns and exploring parallelisation mappings. Rewriting rules are reused across different platforms and are extendable to support new platforms and optimisations. This preliminary work shows Lift could be extended with new primitives, types, rewrite rules and code generation stages to support DNN accelerators.

2 BACKGROUND

DNN accelerators based on both ASICs and FPGAs – BrainWave, TPU, DianNao, Movidius – all rely on matrix multiplication units to accelerate DNNs since the most compute-intensive tasks – weighing input data in layers – can be implemented as homogeneous blocks of GEMV operations. Making use of these units requires carefully designing the code for maximum unit occupation; combined with other optimisations such as tiling and memory access coalescing, the problem of detecting GEMV patterns becomes non-trivial. Even more difficult is this task when the hardware details are subject to change as is the case with FPGA-based BrainWave devices, which permit tweaking synthesised architecture.

Lift produces efficient OpenCL code using tiling, loop transformations, data layout and location optimisation, memory access number reduction [13]. Building on that, we discuss an extension to the Lift Intermediate Representation (IR) to allow detection of course-grained operations such as GEMV.

3 LIFT IR FOR DNNS

3.1 Data types

Lift DNN IR operates on the following types: Int, Floats of various bit-width and arrays. **Int** is used for literals. **Float8**, **Float16** and **Float32** types express the varying precision levels employed by DNN accelerators to reduce the amount of memory when possible. **Arrays** are used for storing vectors and matrices. The decision of whether an array is a vector or a matrix is made using array dimensionality and its memory type, which is embedded in its address space and reflects separation of memory banks in hardware.

The type checker traverses the expression to infer and check the types, which includes validating the requirements on memory types and array dimensionality imposed by parameter types and primitives.

3.2 Address spaces

Lift represents the hardware memory types by associating data with address spaces. For OpenCL, Lift uses *GlobalMemory*, *LocalMemory* and *PrivateMemory*. For DNN accelerators, we generalised these to *DRAMMemory*, *ChipMemory*, *LiteralMemory*, *InputMemory* and *OutputMemory*.

LiteralMemory is associated with all data that is not materialised in memory such as literals and expressions.

ChipMemory and **DRAMMemory** address spaces require specifying memory type as `VectorMemType` or `MatrixMemType`.

3.3 Lift primitives for DNNs

Lift primitives are used to specify memory access patterns, arithmetic operations and address space transfers. The primitives used for expressing DNNs include:

- **Map**, **Slide**, **Reduce**, **Zip**, **Join** and **Split** – these generic patterns are discussed in detail in [12].
- **toChip** moves data from the input memory or DRAM into the on-chip memory; **toDRAM** moves data from the on-chip to DRAM memory; **toOutput** writes from the on-chip to output memory.
- Arithmetic operators that are used for neuron activation in DNNs: **Add**, **Sub**, **Mul**, **Div**, **Neg**, **Mod**, **Tanh**, **ReLU**, **Sigm** and **Max**. Depending on argument types, some of these primitives generate one of several platform-specific commands; for example, Mul generates either `IntMul`, `VVMul` or `MVMul`.

4 CODE GENERATION

The code generation starts with searching for patterns that can be expressed using platform-specific coarse-grained primitives. The transformed expression is type-checked again; then, the Lift compiler performs a series of passes on the expression to infer information on address spaces, loop counter ranges and memory allocation. The compiler also traverses the expression Abstract Syntax Tree (AST) to build *Views* – data structures representing the memory access patterns for reading the input data and writing the output data at each AST nodes. Finally, the gathered information is used to build a platform-specific AST. The code generation process is discussed in more detail below.

During **address space inference**, the compiler assigns address spaces to subexpressions based on explicit memory transfers (`toChip`, `toDRAM` or `toOutput`), parameter types and arithmetic operators. This work extends the address space system to generalise over platform-specific memory spaces and introduces the concept of memory types for platforms that separate vector and matrix memory banks imposing restrictions on valid operations.

The **memory allocation** pass traverses the expression and infers the subexpression result memory based on primitives and data types and address spaces of their arguments. The allocated memory is either materialised as a buffer or used in code generation to access generated subexpressions for nesting.

For example, in an expression such as `ReLU(Head(Transpose(Id(X))))`, we allocate memory for each primitive, but only materialise that of `Id()`, which is an identity function that forces materialisation of its arguments. The *view* of `Relu()` argument looks like `ViewHead(ViewTranspose(mem123))`, where `mem123` is the memory of `Id()`. The address space of `Relu()` itself is `LiteralMemory`, so instead of materialising the generated expression will be saved for later using its memory as reference.

During **View building**, each subexpression is associated with input and output views for reading and writing into memory. Input views depend on nested expressions: the view of `Map` is `ViewMap`, the view of `Zip` is `ViewZip`, etc. Output views depend on outer expressions: for example, the output view of an expression followed by Join is `ViewSplit`.

```
// mapFusion:
Map(f)(Map(g)) ↦ Map(f(g))

// mapFissionWithZipOutside:
Map(fun(x => .. (f(Get(x, i))))))(Zip(.., y, ..)) ↦
Map(fun(z => .. (Get(z, i)))(Zip(.., Map(f)(y), ..))

// vectoriseMapZip:
Map(fun(y => f_abstract(Get(y, 0), .., Get(y, n)))))(
  Zip(X0, .., Xn)) ↦ f_vectorised(X0, .., Xn)
```

Listing 1: Examples of generic rewrite rules: map fusion joins maps, map fission splits Maps while taking care of preceding Zips, vectorisation removes Maps from outside the primitives that accept argument types on both sides of the rewrite rule

The final major stage is generating platform-specific AST. During code generation, the compiler lowers the abstract primitives such as **Add** and **Mul** to their typed counterparts such as **IntAdd**, **VVAdd** and **MVMul** depending on argument types. This work extends the compiler to allow deep nesting of generated subtrees as opposed to using memory to store intermediate results and depending on the platform compiler such as OpenCL to do copy propagation.

5 EXPLORATION

Rewriting in Lift is used to generate a search space of expression transformations with varying performance. For DNN expressions, we reuse generic rules introduced for OpenCL such as map fusion and fission; shown in Listing 1.

```
Join(Map(row => h(Reduce(Add(), 0)(
  Mul(row, vector))))(matrix))
↦ h(Mul(matrix, vector))
```

Listing 2: GEMV rewrite rule

```
Reduce(f, init) ↦ f(init, Reduce(f, 0))
```

Listing 3: The rewrite rule for extracting the initialising expression from Reduce

To make use of matrix-vector multiplication units in DNN accelerators, the rewrite rule presented in Listing 2 replaces the GEMV pattern with **Mul** primitive that is later translated into a platform-specific command `MVMul`. For generalisability, this GEMV rule is strict in matching AST nodes, so the compiler uses other rules to lower expressions to the matchable form.

Using the rewrite rule shown in Listing 3, Lift hoists the accumulator value expression outside of **Reduce**, which can only be applied to commutative functions. In this rule, zero has to be the neutral element with respect to f.

An example of rewriting is shown in Listing 4 using an expression implementing a fully connected layer with a bias (**B**) and an activation function (**ReLU**). In six steps, a generic Lift expression (top) is transformed into a compilable expression (bottom) that benefits from built-in platform-specific primitives. First, the compiler rewrites the GEMV pattern in a form where the pattern can be detected by the corresponding rewrite rule. The first four rewrites extract the bias value from inside `Reduce`. Then, `Map` is replaced with scalar addition with the addition operator, which can generate vectorised addition.

```
toOutput(ReLU(Join(Map(λ(neuron => {
    Reduce(Add(), Get(neuron, 1))(
      Mul(Get(neuron, 0), toChip(X)))
  }))))(Zip(W, B))

          ↧      extractInitFromReduce      ↧

toOutput(ReLU(Join(Map(λ(neuron => {
    Add(Get(neuron, 1),
      Reduce(Add(), 0)(
        Mul(Get(neuron, 0), toChip(X))))
}))))(Zip(W, B))

          ↧    mapFissionWithZipOutside     ↧

toOutput(ReLU(Join(Map(λ(neuron => {
    Add(Get(neuron, 1),
        Reduce(Add(), 0)(Get(neuron, 0)))
}))))(Zip(Map(λ(row => Mul(row, toChip(X))))(W), B))

          ↧    mapFissionWithZipOutside     ↧

toOutput(ReLU(Map(λ(neuron => {
  Add(Get(neuron, 1), Get(neuron, 0))
}))))(Zip(Join(Map(Reduce(Add(), 0))(
            Map(λ(row => Mul(row, toChip(X))))(W))),
          B))

          ↧           mapFusion             ↧

toOutput(ReLU(Map(λ(neuron => {
  Add(Get(neuron, 1), Get(neuron, 0))
}))))(Zip(Join(Map(λ(row => Reduce(Add(), 0)(
            Mul(row, toChip(X))))))(W),
          B))

          ↧          mvMulBuiltIn           ↧

toOutput(ReLU(Map(λ(neuron => {
  Add(Get(neuron, 1), Get(neuron, 0))
}))))(Zip(Mul(W, toChip(X)), B))

          ↧         vectorizeMapZip         ↧

toOutput(ReLU(Add(Mul(W, toChip(X)), B)))
```

Listing 4: Fully connected layer rewriting steps

6 RELATED WORK

Delite [15] is a compiler framework that provides tools to define Domain-Specific Languages (DSL) that reuse parallel patterns, optimise and generate platform-specific high-performance code using a single backend. Lift benefits from better performance portability thanks to reusability of rewrite rules across hardware platforms, whereas Delite hard-codes device-specific optimisations in each backend.

Halide [11] is another approach to simplify high-performance code generation: by decoupling algorithm descriptions from optimisations (schedules), Halide takes the problem of code tuning out of the hands of the application developer. Similarly to Delite, it suffers from limited portability due to hard-coded optimisations for a restricted set of platforms.

Building upon Halide principles and IR, TVM [1] and NNVM [8] together provide a comprehensive framework for cross-platform optimisation of neural networks. NNVM converts workloads described in different encodings to a standardised computational graph, while TVM handles high-level operator fusion, data layout transformation and tensor optimisation. TVM lacks in extensibility since each optimisation is hard-coded as a separate module, whereas Lift can be extended with principally new optimisations by adding new rewrite rules.

7 CONCLUSION

This work-in-progress paper describes a case study of how a function IR, coupled with rewriting, can be used to map computations onto DNN accelerators such as Brainwave, TPU and DianNao. This proposed extension of Lift aims to provide a way to express programs in a high-level platform-independent language and automatically tune generated code to various DNN accelerators through optimisation space exploration. Compared to existing solutions, Lift is extensible both in terms of optimisation methods and target platforms through a system of fine-grained rewrite rules and modular code generators.

Our preliminary work has shown that Lift is able to detect the most ubiquitous pattern in DNNs: matrix-vector multiplication. We have shown how to optimise a generic Lift expression for fully connected layer by transforming it to use a built-in primitive for GEMV. Stacking the produced expression, we get a Multilayer Perceptron, a neural architecture that is responsible for most of the TPU workload in Google servers [6].

We intend to build up on this in future work by evaluating the approach on DNN architectures such as VGG, ResNet and GoogleNet on a range of DNN accelerators such as TPU, BrainWave, Huawei Da Vinci architecture and Movidius Myriad. We are confident that the technique presented in this paper is generic enough to work on other types of layers and different hardware platforms.

ACKNOWLEDGEMENTS

We would like to thank Federico Pizzuti for help with Lift extension; this work was supported by Microsoft Research and by the Engineering and Physical Sciences Research Council (grant EP/L01503X/1), EPSRC Centre for Doctoral Training in Pervasive Parallelism at the University of Edinburgh, School of Informatics.

REFERENCES

1. Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, and others. 2018. TVM: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 12th USENIX conference on Operating Systems Design and*

Implementation. USENIX Association, 579–594.

2. Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Mahdi Ghandi, Daniel Lo, Steve Reinhardt, Shlomi Alkalay, Hari Angepat, Derek Chiou, Alessandro Forin, Doug Burger, Lisa Woods, Gabriel Weisz, Michael Haselman, and Dan Zhang. 2018. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro* 38 (March 2018), 8–20.
3. Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 92–104.
4. Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High performance stencil code generation with Lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM, 100–112.
5. Mircea Horea Ionica and David Gregg. 2015. The Movidius Myriad Architecture's Potential for Scientific Computing. *IEEE Micro* 35, 1 (2015), 6–14.
6. Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, and others. 2017. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE, 1–12.
7. Liqiang Lu, Yun Liang, Qingcheng Xiao, and Shengen Yan. 2017. Evaluating fast algorithms for convolutional neural networks on fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*. IEEE, 101–108.
8. Li Mu. 2017. Introducing NNVM Compiler: A New Open End-to-End Compiler for AI Frameworks. `https://aws.amazon.com/blogs/machine-learning/introducing-nnvm-compiler-a-new-open-end-to-end-compiler-for-ai-frameworks/`. (2017). [Online; accessed 25-November-2018].
9. Tony Peng. 2018. Huawei Leaps into AI; Announces Powerful Chips and ML Framework. `https://medium.com/syncedreview/huawei-leaps-into-ai-announces-powerful-chips-and-ml-framework-f9aa6ec87bcb`. (2018). [Online; accessed 25-November-2018].
10. Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, and others. 2016. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 26–35.
11. Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices* 48, 6 (2013), 519–530.
12. Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. *ACM SIGPLAN Notices* 50, 9 (2015), 205–217.

13. Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2016. Matrix multiplication beyond auto-tuning: rewrite-based GPU code generation. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. ACM, 15.

14. Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. 2016. Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 16–25.

15. Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 4s (2014), 134.