Data vs. Instructions: Runtime Code Generation for Convolutions

Malith Jayaweera
malithjayaweera.d@northeastern.edu
Northeastern University
Boston, Massachusetts, USA

Yanzhi Wang
yanz.wang@northeastern.edu
Northeastern University
Boston, Massachusetts, USA

David Kaeli
kaeli@ece.neu.edu
Northeastern University
Boston, Massachusetts, USA

Abstract

Just-in-time (JIT) code generation techniques are increasingly gaining traction due to their ability to significantly speedup computations. Small and medium matrix multiplication has wide uses in the domains of Machine Learning (ML) and scientific simulations in high performance computing. These computations can benefit greatly from JIT compilation. Prior runtime code generation approaches have primarily focused on loop unrolling and address compression techniques that focus on optimizing the instruction flow. In contrast, we consider a unique code generation approach where data values are dynamically embedded in instructions as immediate values, effectively transforming a memory load into an immediate load. By utilizing Intel's AVX-512 vector extensions, we show that our technique achieves geometric mean speedups of 1.12× and 1.05× over MKL and MKL JIT, the current state of the art JIT library, for 32 input channels. We have open sourced our JIT library and plan to apply our approach to produce high performance convolution libraries.

CCS Concepts: • **Software and its engineering** → **Just-in-time compilers**; • **Computer systems organization** → *Single instruction, multiple data.*

Keywords: just-in-time compilation, SIMD Operations, Convolution, Winograd

ACM Reference Format:
Malith Jayaweera, Yanzhi Wang, and David Kaeli. 2018. Data vs. Instructions: Runtime Code Generation for Convolutions. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/1122445.1122456

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

1 Problem & Motivation

Small and medium-sized matrix multiplications are the computational engines powering ML inference on edge devices. For this class of problems, Intel's Math Kernel Library (MKL) [9, 13], LIBXSMM [2], and MKL-DNN [11, 12] are state-of-the-art JIT libraries used to accelerate convolutions.

Noticeably, existing JIT GEMM code generation techniques are limited in their ability to produce the best possible instruction sequence by utilizing runtime problem dimensions [1, 2]. While current approaches offer significant performance improvement (as compared to conventional GEMM), they typically do not exploit data characteristics to further improve performance.

To address this problem, we propose a JIT library to convert data memory accesses into instruction accesses, embedding data values as immediate operands, and avoiding many memory accesses. Our approach is motivated by the fact that in many deep neural network applications, weight data in convolution layers remain constant during inference until they are updated. To the best of our knowledge, this is the first JIT code generation technique which transforms data accesses into instruction accesses using SIMD (Single Instruction Multiple Data) instructions.

Below, we summarize our contributions.

- We develop a runtime code generation library (MARLIN) that transforms data accesses into embedded immediate operands, targeting Winograd [6, 14] convolutions. Our method can be easily extended to algorithms that utilize convolution operations.
- We show that existing JIT libraries can benefit by considering alternative code generation approaches. Our open source JIT implementation will enable the research community to experiment with a new flavor of code optimization.

2 Background & Related Work

Next, we provide an overview of JIT code generation, and discuss why it has been gaining traction recently in accelerating computations.

2.1 JIT Code Generation

Interpreted languages use JIT compilation techniques to execute instructions on the fly using a virtual machine (e.g., a Java Virtual Machine (JVM)) or an interpreter (e.g., the

Python interpreter). JIT code generation techniques have also been used in the context of compiled languages to optimize performance, in contrast to compile-time optimizations, which tend to be less aggressive. Compiled source code often utilizes external library functions which are dynamically linked at run-time. Thus, the scope of compile-time optimizations is restricted to the source code. Kistler et al. [5] address this issue by demonstrating that dynamic code generation can exploit inter-procedural optimizations when using dynamically linked libraries. They introduce a dynamic profiler and optimization modules that can perform Field Directed Optimizations (FDOs), which are applied in phases. The FDOs operate on a common intermediate representation (IR), known as guarded static assignment (GSA).

Nuzman et al. [10] explore the viability of feedback-directed IR-based dynamic code generation in the domain of statically compiled, rather than interpreted, languages. The authors demonstrate that fat binaries, which combine the IR together with the statically compiled executable, can enable software vendors to ship their software with the ability to be dynamically optimized, no longer relying on purely binary optimizations.

Template-based compilation in C++ has enabled programmers to keep the core logic of an algorithm abstract, while specializing their code to different data types/structures provided through template parameters. However, statically compiling all possible scenarios may not be the best choice due to the increased binary size, and not every code specialization can be identified at compile time. Finkel et al. [1] solve this problem in "ClangJIT", a framework built on top of the LLVM infrastructure to provide run-time template specialization.

All of the above code generation techniques rely on the availability of an Intermediate Representation (IR) and/or a IR-based JIT runtime. In contrast, there has been a trend recently to accelerate computations through JIT machine code generation, eliminating the overhead encountered with IR-based techniques.

2.2 JIT Code Generation Accelerating GEMM Based Computations

Heinecke et al. [2] show that JIT machine code generation can be used to accelerate small matrix computations using "LIBXSMM". They perform a study on loop unrolling and Enhanced Vector Extension (EVEX) address compression, performed by JIT compilation to reduce the instruction footprint, as well as to improve performance. The goal of LIBXSMM is to improve the performance of small matrices using Intel AVX2 and AVX-512 extensions, relying purely on an instruction-based approach.

Due to the characteristics of Deep Neural Networks (DNNs), low precision inference has been attractive in order to improve performance, while maintaining the same level of model accuracy. FBGEMM is a low precision matrix multiplication library for server-side inference developed by Khudia et al. [4]. FBGEMM is designed to maximize the benefits of quantization through JIT code generation techniques. In addition, FBGEMM achieves the same DNN inference accuracy and reduces the storage space requirements for weights.

Dynamic code generation has motivated the development of additional GEMM libraries, including MKL-DNN (which was renamed as oneDNN [11, 12]) and Intel's Math Kernel Library [8, 13]. Both of these libraries follow a similar approach as used in LIBXSMM and rely on performance optimizations through JIT instruction generation.

In contrast to previous work, MARLIN uses its own JIT backend, converting the matrix memory loads into immediate operands in instructions. We present a comprehensive study of the costs involved in each of the libraries evaluated and provide a comprehensive comparison across different problem sizes. We also consider AVX-512 instruction-level optimizations that can be used to further improve instruction-level parallelism.

3 Uniqueness of the Approach

In this section, we will present our unique code generation optimization which transforms data accesses into immediate operands in instructions. We will contrast the differences of our methodology with existing pure instruction-based JIT approaches. We will first describe the architectural features we exploit with our design, and provide details of our JIT code generation technique. Finally, we will highlight the underlying differences that translate to performance gains over existing solutions.

3.1 SIMD-Based Convolutions on CPUs

SIMD instructions on CPUs (both CISC - Complex Instruction Set Computing and RISC-Reduced Instruction Set Computing architectures) have been gaining traction recently. AVX-512 are 512-bit extensions to the 256-bit Advanced Vector Extension SIMD (Single Instruction Multiple Data) instructions for the x86_64 instruction set architecture. Intel microprocessors of the Skylake family have two AVX-512 Fuse Multiply Add (FMA) units. An AVX-512 FMA unit has 32 registers (ZMM0-ZMM31) that are 512 bits wide. Since Single Precision (SP) floating point numbers are 32 bits, a single AVX-512 register can hold up to 16 SP values.

Given the limited number of SIMD registers, when executing memory-bound applications, tiling becomes crucial to improve computational throughput and to improve data reuse. As shown in Figure 1, a Winograd 3×3 convolution uses an input tile size of 4×4 (16 input values). The 16 input tile values can be loaded in two iterations, with 8 rows in each iteration to 8 AVX-512 registers. Each row of 16 floats is loaded into a separate AVX-512 register. 8 values of the weight matrix are loaded into separate registers and broadcast across each register in order to achieve a matrix outer product. Due to the use of SIMD instructions, in a single

cycle, it is possible to operate on 16 input elements. In cases where the tile size is smaller than the available SIMD resources, mask registers ($K0 - K7$) are used to control the output written back to memory.

Expressing a program in a compact manner reduces the binary size and is important for libraries which are integrated with high performance applications. SIMD instructions are an ideal candidate to this end, as SIMD instructions allow compact representations of otherwise loop-based code blocks. The availability of the instruction set motivated us to explore the generation of specialized instructions with immediate values, all embedded within the code to relieve the CPU of issuing data-load requests. We will refer to this class of code generation techniques as data-oriented code generation, in contrast to existing code generation approaches that rely on purely instruction-based approaches.

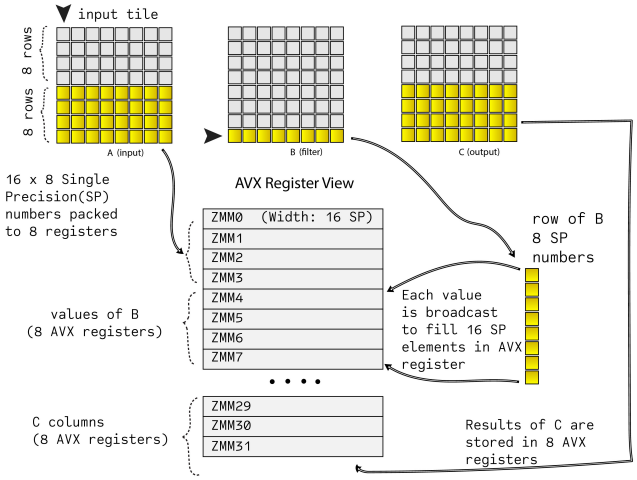

Figure 1. Tiling for a Winograd 3×3 element-wise product.

3.2 JIT Code Generation

Data-oriented code generation comes with challenges in terms of overall efficiency as well as the ability to amortize the cost of recompiling. Fortunately, current SIMD architectures, such as the Intel AVX-512, provide extensive support for performing vector operations on floats. When using single-precision floating point numbers, we show that a memory load value can be transformed into an immediate operation involving just two machine-level instructions, as shown in Figure 2a. Each immediate value is assigned to the *EAX* register, and then broadcast across the vector register using a **vpbroadcastd** [3] instruction. Essentially, this would increase the size of instructions in memory, as immediate values are embedded in instructions rather than fetched using additional load instructions. However, convolution weight matrices are often pruned to reduce storage cost, as well as improve performance [7, 15]. By exploiting this

```
0xb8, 0x00, 0x00, 0x00, 0x41,          // movd    eax, 0x41000000
0x62, 0xf2, 0x7d, 0x48, 0x7c, 0xc0     // vpbroadcastd zmm0, eax
```

(a)

```
0x62, 0xf1, 0x7c, 0x48, 0x57, 0xc0     // vxorps zmm0, zmm0, zmm0
```

(b)

Figure 2. (a) Machine instructions to generate immediate values. (b) Specialized machine instruction to generate a zero.

fact, whenever pruning results in the generation of zero values, immediate values can be expressed in a compact format, as shown in Figure 2b.

By generating instructions dynamically in a buffer, there needs to be a seamless integration with the convolution algorithm. Similar to existing solutions, we first allocate virtual pages via the operating system, and then initiate the code generation process to fill the code buffer. The pages are later marked as executable instead of readable / writable to allow dynamic execution. Based on the execution flow of the algorithm, by utilizing a metadata array, the required instructions are loaded and are dynamically executed.

3.3 Discussion on Performance Improvement

Existing code generation techniques, in the context of compiled GEMM library binaries, employ purely instruction-oriented methods. These libraries achieve performance improvements by applying dynamic loop-unrolling with the knowledge of problem dimensions (i.e., matrix dimensions) available at run-time; Another technique that can improve performance is to compute offsets and exploit variable-length instructions in the x86_64 ISA to compress load instructions.

In contrast, MARLIN leverages a data-oriented code generation technique, where data values are embedded within instructions as immediate values. This approach removes of overhead associated with the CPU issuing data loads, significantly reducing the number of load instructions executed. In addition, since weight data used in convolution operations can now be expressed as immediate operands, the values are effectively streamed through the L1-icache instead of the L1-dcache. Ultimately, when considering all levels of caching (i.e., L1, L2 and L3), this would result changes in cache behavior compared to existing code generation techniques. We hypothesize that performance benefits result from two sources: first, a reduction in cache misses will reduce overall memory latencies, and second, MARLIN takes pressure off the L2 cache bandwidth. Our experimental evaluation in section 4 provides insights as to the validity of our hypothesis.

3.4 Implementation

One of our key design considerations is the ease of integration with existing computations. We achieved this goal

by completely automating the code generation process, enabling the user to request the code generation when needed through an API call (1) (see Figure 3). Code generation is only performed once for the entire program execution, and a memory reference is used afterwards to trigger execution. The execution mechanism is similar to existing state of the art JIT libraries.

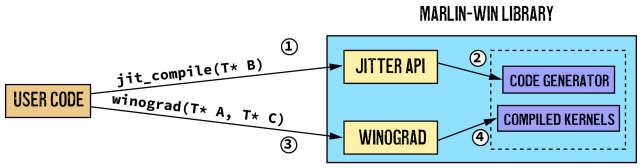

Figure 3. API calls, viewed from a user perspective.

To understand how the code generation and execution works under the hood, we present Figure 4, which shows the internal implementation of the *Code Generator* and *Compiled Kernels* (steps 2 and 4 in Figure 3). Our jitter leverages a set of code templates, mapping assembly instructions to machine code. For example, if we need to embed 7.0_{10} in an instruction, the last four bytes can be substituted. Following these step, we can embed any single precision floating point value as an immediate operand in an instruction.

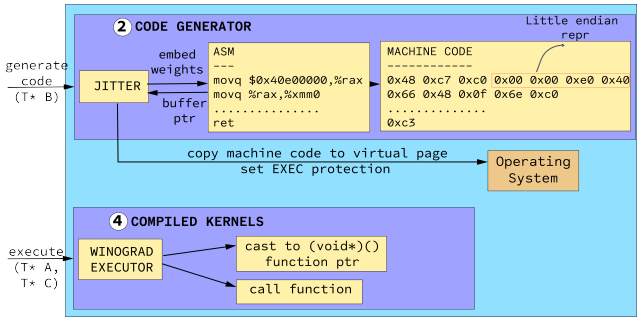

Figure 4. Internal software architecture of the JIT Code Generator (jitter) and executor.

Another key consideration is to exploit the available parallelism using SIMD vector extensions. Our library is currently supported on Intel processors that support the AVX-512F flag, but can be easily extended to other processors. For our experimental evaluation, we use a single-socket 12-core 3.50GHz Intel i9-9920X Skylake CPU, with hyper-threading enabled. Each core has access to two AVX-512 Fuse Multiply Add (FMA) units.

4 Results & Contribution

First, we evaluate improvements in execution speed as compared to existing state of the JIT GEMM and conventional GEMM libraries. Figure 5 shows the execution time of a 3×3 Winograd convolution using three library backends: Intel's MKL, MKL's JIT library, and our implementation (MARLIN).

We observe a performance gain of 1.12× over MKL and 1.05 × over MKL-JIT, for 32 input channels, and 1.11 × over MKL and 1.04×, when the number of input channels is 64. The batch size is set to 1, as the library is intended for real-time ML inference. We also observed that JIT libraries perform better than conventional BLAS libraries whenever:

$$\sqrt[3]{in\_channels \times out\_channels \times tile\_count} \leq 100 \qquad (1)$$

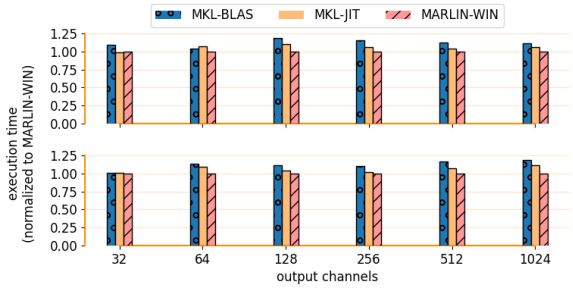

Figure 5. Execution time of Winograd convolutions, normalized to MARLIN. Top and bottom graphs correspond to 32 and 64 input channels, respectively, (lower is better).

Next, we analyze growth of the memory footprint produced by our JIT optimization to ensure that we are not exhausting memory resources. As shown in Figure 2, an instruction sequence consists of 11 bytes. Using this information, we can compute that the memory usage increases by approximately 2.75×. However, further optimizations allow the memory footprint to be reduced. For example, a zero can be expressed through a **vxorps** instruction, reducing the code bloat from 11 bytes to only 6 bytes which is a large improvement.

Finally, we explore the cache performance of Winograd convolution compared to existing GEMM libraries in Figure 6. The cache misses represent the number of times a memory block needed to be requested from main memory, measured through hardware counters on the CPU, since it was not present in the cache hierarchy. MARLIN produces an overall reduction in the number of cache misses compared to Intel's MKL and MKL's JIT library. As the number output channels increase, Intel MKL and MKL's JIT libraries are fine-tuned to better utilize the cache hierarchy.

The improvements in performance help motivate our efforts to exploit data-oriented code generation techniques and to expand available choices for JIT code generation in highly optimized applications.

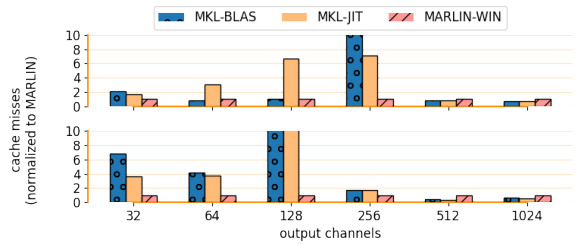

Figure 6. Cache misses recorded when performing Winograd convolutions, normalized to MARLIN. Top and bottom graphs correspond to 32 and 64 input channels, respectively, (lower is better).

5 Conclusion

Our findings and open-source JIT code generation library can help drive JIT based techniques in commercial and research machine learning applications. As an extension to our work, we intend to explore applications on mobile processors (ARM), where SIMD instructions are also available. For power-constrained devices, reducing the repeated data loads will translate into energy savings. We believe our work will motivate research in data-oriented code generation, in addition to conventional code generation techniques, enabling programmers to squeeze out every last drop of performance out of CPUs.

References

[1] H. Finkel, D. Poliakoff, J. Camier, and D. F. Richards. 2019. ClangJIT: Enhancing C++ with Just-in-Time Compilation. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, USA, 82–95. https://doi.org/10.1109/P3HPC49587.2019.00013

[2] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst. 2016. LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, USA, 981–991. https://doi.org/10.1109/SC.2016.83

[3] https://software.intel.com/. 2020. *Intel Intrinsics*. Intel. Retrieved 2020-07-20 from https://software.intel.com/sites/landingpage/IntrinsicsGuide/#text=broadcastd&expand=4953

[4] Daya Khudia, Protonu Basu, and Summer Deng. 2018. Open-sourcing FBGEMM for state-of-the-art server-side inference.

[5] Thomas Kistler and Michael Franz. 2003. Continuous program optimization: A case study. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 25, 4 (2003), 500–548.

[6] A. Lavin and S. Gray. 2016. Fast Algorithms for Convolutional Neural Networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 4013–4021. https://doi.org/10.1109/CVPR.2016.435

[7] Xiaolong Ma, Fu-Ming Guo, Wei Niu, Xue Lin, Jian Tang, Kaisheng Ma, Bin Ren, and Yanzhi Wang. 2020. PCONV: The Missing but Desirable Sparsity in DNN Weight Pruning for Real-Time Execution on Mobile Devices. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 04 (Apr. 2020), 5117–5124. https://doi.org/10.1609/aaai.v34i04.5954

[8] Intel MKL. 2020. *Intel Math Kernel Library. Developer Reference*. Intel Corporation, USA.

[9] Intel MKL. 2020. Intel® Math Kernel Library Improved Small Matrix Performance Using Just-in-Time (JIT) Code Generation for Matrix Multiplication (GEMM). Retrieved 2020-07-20 from https://software.intel.com/content/www/us/en/develop/articles/intel-math-kernel-library-improved-small-matrix-performance-using-just-in-time-jit-code.html

[10] Dorit Nuzman, Revital Eres, Sergei Dyshel, Marcel Zalmanovici, and Jose Castanos. 2013. JIT technology with C/C++ Feedback-directed dynamic recompilation for statically compiled languages. *ACM Transactions on Architecture and Code Optimization (TACO)* 10, 4 (2013), 1–25.

[11] oneDNN. 2020. oneAPI Deep Neural Network Library (oneDNN). Retrieved 2020-07-20 from https://oneapi-src.github.io/oneDNN/dev_guide_transition_to_v1.html

[12] oneDNN. 2020. oneAPI Deep Neural Network Library (oneDNN). Retrieved 2020-07-20 from https://github.com/oneapi-src/oneDNN

[13] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, USA, 167–188.

[14] Shmuel Winograd. 1980. *Arithmetic complexity of computations*. Vol. 33. Siam, USA. https://doi.org/10.1137/1.9781611970364.ch5 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611970364.ch5

[15] Shaokai Ye, Xiaoyu Feng, Tianyun Zhang, X. Ma, Sheng Lin, Z. Li, Kaidi Xu, Wujie Wen, S. Liu, J. Tang, M. Fardad, X. Lin, Yongpan Liu, and Yanzhi Wang. 2019. Progressive DNN Compression: A Key to Achieve Ultra-High Weight Pruning and Quantization Rates using ADMM. *ArXiv* abs/1903.09769 (2019).