Architectural Support for Convolutional Neural Networks on Modern CPUs

Animesh Jain[1], Michael A. Laurenzano[1], Gilles A. Pokam[2], Jason Mars[1] and Lingjia Tang[1]

University of Michigan, Ann Arbor[1], Intel Labs[2]
{anijain,mlaurenz,profmars,lingjia}@umich.edu,gilles.a.pokam@intel.com

ABSTRACT

A key focus of recent work in our community has been on devising increasingly sophisticated acceleration devices for deep neural network (DNN) computation, especially for networks driven by convolution layers. Yet, despite the promise of substantial improvements in performance and energy consumption offered by these approaches, general purpose computing is not going away because its traditional well-understood programming model and continued wide deployment. Therefore, the question arises as to what can be done, if anything, to evolve conventional CPUs to accommodate efficient deep neural network computation.

This work focuses on the challenging problem of identifying and alleviating the performance bottlenecks for convolution layer computation for conventional CPU platforms. We begin by performing a detailed study of a range of CNN-based applications on a modern CPU microarchitecture, finding that designing a physical register file (PRF) capable of feeding computational units is the primary barrier that prevents the addition of more compute units in the CPU, limiting the performance improvements that can be achieved by CPU on convolution layers. We present the design of a novel, minimally intrusive set of microarchitectural and ISA extensions that address this problem and describe the code generation support needed to take advantage our design. Through a detailed evaluation that covers 5 state-of-the-art neural network applications, we observe that applying these extensions allows packing more compute in the CPU while keeping PRF energy in check, achieving a 2× performance improvement and a 2.7× energy-delay product improvement against a popular Intel Haswell server processor baseline.

CCS CONCEPTS

• **Computer systems organization** → **Single instruction, multiple data**; **Neural networks**;

KEYWORDS

Deep Neural Networks, Vector Instructions

ACM Reference Format:
Animesh Jain[1], Michael A. Laurenzano[1], Gilles A. Pokam[2], Jason Mars[1] and Lingjia Tang[1]. 2018. Architectural Support for Convolutional Neural Networks on Modern CPUs. In *International conference on Parallel Architectures and Compilation Techniques (PACT '18), November 1–4, 2018, Limassol, Cyprus.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3243176.3243177

PACT '18, November 1–4, 2018, Limassol, Cyprus

ACM ISBN 978-1-4503-5986-3/18/11.
https://doi.org/10.1145/3243176.3243177

1 INTRODUCTION

Deep Neural Networks (DNNs) have recently emerged as a core computational component in user-facing applications that include analyzing text, decoding speech, recognizing images and searching the web, among others [23, 28, 34–37, 51, 55, 59, 61]. For DNNs based on convolutional layers, GPUs and more specialized accelerators have gained significant traction as the hardware platforms of choice for running convolution computation [16, 17, 19, 48]. This *dual-device acceleration model* that our community has focused on involves adding the GPU or specialized accelerator to a conventional CPU platform, typically over a loosely-coupled interconnect such as PCIe, QPI or NVLink [3, 7, 8].

Looking at modern CPU offerings, it is clear that they offer substantially fewer raw floating point operations per second (FLOPS) than their GPU counterparts. However, CPUs are an indispensable part of the design of any system, meaning they are a well understood part of conventional system design practices while offering the benefit of a seamless, familiar programming model and software stack. Facebook uses CPUs for performing DNN inference at scale [27]. Recent work shows that the programming models for accelerators are non-standard and unfamiliar to many programmers [49]; it is especially problematic for asynchronous accelerator programming models that are error prone and difficult to master [13]. Due to the long history of solving challenges that arise with CPUs across the computing stack, we have a well-oiled computing stack for CPUs for quite sometime, and thus, general purpose CPU computing will remain widely deployable. Therefore, alongside designing dual-device acceleration platforms, it remains an equally important question to understand what are the performance bottlenecks for CPU in achieving high performance for CNNs and how to design CPU hardware that can improve performance of CNNs with minimum hardware changes while also performing all the non-accelerable tasks for which CPUs are essential.

CPU designs have a long history of incorporating hardware and ISA support for specialized domain-specific operations, evidenced by the near-universal support for cryptography, virtualization, security and multimedia operations in modern CPU offerings [4–6, 9, 10]. Unfortunately, despite the large body of work in our community on accelerating DNNs [11, 16, 18, 24, 25, 38, 41, 46–48], there is little understanding in the literature of the interplay among the factors involved in improving CPU performance on convolution layers. Simply increasing raw FLOPS by continuing down the path of scaling vector widths, such as in the progression from SSE

to AVX to AVX2 among x86 platforms [22, 43], is unlikely to continue for two reasons. First, the AVX2 vector width of 512 bits spans a full cache line, and thus longer vectors would necessarily touch multiple cache lines per vector register load, introducing significant performance penalties or substantial microarchitectural workarounds. Second, leveraging larger vector widths puts the onus on programmers and compilers to find additional sources of SIMD parallelism, an extremely difficult task even for current vector widths that remains an active, open area of research in the compiler community [12, 30, 45]. Thus, it is clear that improving the computational capability of CPUs for convolution layers requires an alternative approach, yet it remains unclear what that approach is.

This paper is the first to undertake a detailed characterization of the issues involved in improving CPU performance for convolution layers. We find first that scaling the read bandwidth of the physical register file (PRF) is one of the key constraints needed to deliver additional data to increasingly capable compute units. Second, we find that harnessing increasingly capable compute units requires crafting a solution that spans both hardware and software to take full advantage of the data reuse present in the core of the CNN computation.

Building on this insight, we design Locality Extensions for Deep Learning (LEDL). LEDL is a technique that spans both hardware and software, consisting of a novel set of microarchitectural and ISA extensions to increase the computational capabilities of modern CPUs for CNNs. We present the design in detail, which in hardware includes a handful of architecturally visible remote registers that reside within the VFMA units in the CPU and a set of inter-VFMA links that allow data to be passed between units directly. In software, LEDL's automatic code generator, ACG, is carefully designed to generate code that is robust to different microarchitectural implementations while taking full advantage of the reuse opportunities exhibited by convolution layer computation and aggressive prefetching mechanisms within CPUs.

The contributions of this paper are as follows:

- **Convolution layer/CPU Bottlenecks –** we present the first thorough characterization study of convolution layer performance on CPUs, identifying the hardware factors that constrain improving its performance, finding that scaling the PRF read bandwidth is the key constraint that needs to be solved to improve convolution performance on CPUs.
- **Extensible Hardware for Convolution Layers –** we describe the design and implementation of LEDL, a set of microarchitecture and ISA extensions that allow modern CPUs to seamlessly improve the performance of convolution layer computation, while keeping the energy-hungry physical register file in check. We also show that these extensions can accelerate certain varieties of fully-connected and long short-term memory (LSTM) networks. The extensions are evaluated across a large space of design points that offer differing levels of computational capability.
- **Robust Code Generation –** we describe the design and implementation of Automatic Code Generator (ACG), a code generator that takes advantage of the heavy data reuse exhibited within convolution layer computation and leverages aggressive register tiling and hardware prefetching mechanisms to produce high performance code across a large space of microarchitectural design points.

Together, the hardware and software components of LEDL produce a platform design capable of providing substantial performance and energy improvements to convolution layer computation on CPUs. When extending an Intel Haswell server processor design with LEDL, we observe that across 5 state-of-the-art neural networks we achieve performance improvements that average 2× and energy-delay product improvements that average 2.7×.

2 BACKGROUND AND MOTIVATION

2.1 CNN Computation

Machine learning research has been increasingly focused in recent years on convolution neural networks (CNNs), as CNNs have been shown to outperform the alternatives across a number of different machine learning tasks [20, 54]. It is also evident that convolution layers are becoming more prominent as time goes on, specifically for tasks like object recognition, video analysis, drug discovery and natural language processing [23, 28, 34–37, 51, 55, 59, 61]. These CNN-driven networks are becoming increasingly larger and deeper. For example, the Alexnet image recognition network had only 5 convolution layers [37], while the recently released ResNet can have hundreds of convolution layers [28].

CNN Characterization. Beyond making up a large number of layers in modern CNNs, convolution layers consume a large fraction of the computational cycles in the total execution time, an observation that is in line with similar observations made by prior work [16, 18, 46].

Convolution layer computation has a number of implementations that have been explored in the literature and adopted in software packages [1, 19, 21, 38, 56]. We observe that there are two main classes of implementations that appear in high performance implementations: IM2COL + matrix multiplication (IM2COLMM) and winograd transform [38]. While each of these different implementations differs in how broadly applicable they are and in their performance characteristics, the computational kernel underlying all of them is the SGEMM calculation. We observe that SGEMM computations on average contributes to 78% of total execution time for our application suite for IM2COLMM implementation. We observe similar trend for Winograd algorithm as well. In addition, this SGEMM computation has also been used as the underlying implementation of other widely used DNN layers like fully connected and long short term memory layers [33] (we briefly discuss these layers in Section 5.6). Thus, the key to increasing the performance of CNN computation on CPUs is to achieve higher performance on the SGEMM calculation.

2.2 CPU Bottleneck Identification

The current trend of increasing raw computational capability of the CPUs is to simply scale the vector width of the SIMD units. For example, the Intel x86 SIMD vector width extensions have increased from 128-bits in SSE to 256-bits in AVX2 to 512-bits in AVX-512. However, the vector width scaling trend is unlikely to continue for two reasons. First, scaling vector width beyond cache line width (512 bits) requires touching multiple cache lines per vector register load, possibly introducing complex microarchitectural workarounds to handle multiple variable-latency memory requests. Second, larger vector widths makes it

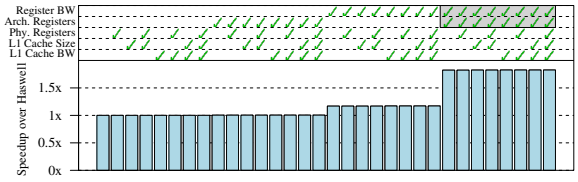


Figure 1: Performance impact of doubling five memory-related microarchitectural parameters, when VFMA units are increased to 4; Arch registers and Reg BW are the key factors

increasingly difficult for the application developers or compilers to find SIMD parallelism amenable to such large vector widths, which is a difficult problem to solve even for current SIMD widths [12, 30, 45]. Due to these issues with vector width scaling, the only other obvious solution to increase raw CPU FLOPS is to add additional vector math units.

However, adding more vector math units is not enough to achieve higher CPU FLOPS. It is equally necessary to supply data to these vector math units every clock cycle to take advantage of this additional CPU compute. This leads to the question - *which memory-related microarchitectural parameters need to be adjusted to keep vector units busy?*

Bottleneck Analysis. To begin to answer this question, we study five such microarchitectural parameters - L1 cache bandwidth, L1 cache size, number of architectural registers and number of physical registers and Register bandwidth to identify which memory structure(s) should be focused on. In this study, we increase the number of vector math units from 2 (Haswell processor baseline) to 4 and measure the impact of doubling the value of these five parameters in simulation, both in isolation and in conjunction with each other, on the performance of SGEMM kernel. We present our findings in Figure 1. There are 2 key observations. First, increasing cache bandwidth and/or size alone (first 16 bars) does not improve SGEMM performance. The reason is that SGEMM employs aggressive register tiling, reusing the data in registers multiple times before going back to caches. Current L1 cache size and bandwidth are sufficient for this usage. Similarly, current Intel machines have enough physical registers for this usage. Second, we observe that both the **number of architectural registers and register bandwidth** have to be increased simultaneously (the rightmost eight bars) to achieve substantial speedup. Register bandwidth is necessary to supply the data to the vector math units every cycle. And, increasing architectural registers is necessary to achieve higher tile size, reusing data multiple times before bringing more in from cache.

2.3 Challenges

Energy Consumption. Conventional out-of-order cores use Physical Register File (PRF) for register renaming which helps in extracting more instruction level parallelism. PRF size has been increasing with every new CPU offering, currently set at 168 physical floating-point registers in Haswell processors. This PRF size is large enough to support SGEMM kernel, given we have enough software-visible architectural registers. Therefore, the deciding parameter to keep vector math units busy is PRF bandwidth.

To understand the impact of PRF bandwidth, it is necessary to understand how SGEMM works. All SGEMM kernel operations can be realized using Fused Multiply Add (FMA) instructions. Fortunately, in recent years CPU vendors have introduced vector fused multiply add (VFMA) units in the processor that can be leveraged by SGEMM computation. Each VFMA operation requires 3 vector register reads. Therefore, adding a VFMA unit requires extra three read ports in the PRF, introducing several challenges.

Firstly, the energy per access increases rapidly as the number of PRF read ports increases. Thus, the inclusion of additional read ports to feed a larger number of vector compute units rapidly increases the energy per PRF read, which can quickly turn the PRF a major contributor to the energy consumption of the CPU. Secondly, additional read ports increase the access latency to the PRF, where even a modest number of read ports can begin to constrain clock rate. For instance, a PRF with 14 read ports at 22nm technology node can meet a 2.4GHz clock rate, while a PRF with 15 ports cannot.

Therefore, it is clear that PRF reads are expensive and have to be kept to a minimum to keep the energy-hungry PRF in check. We observe that SGEMM kernel has high amount of data reuse, which if exploited wisely can result in significant reduction in PRF reads. For example, considering multiplication of matrices A and B, first element of A is multiplied to every element in the first row of matrix B, providing opportunity to cut the PRF reads for first element of matrix A. And similarly, first element of matrix B is multiplied to every element in the first column of matrix A. These opportunities for reuse could, alongside register tiling, be leveraged to substantially reduce the number of reads to the PRF.

Code Generation. Another challenge is generating code that can efficiently take advantage of the additional compute in the CPUs. Libraries such as MKL are aggressively tuned to current CPU specifications, and thus these libraries cannot be readily ported to new hardware configurations having more VFMA units without significant additional manual labor. Increasing the number of VFMA units requires handling data movements between the memory, registers and the VFMA units in an effective manner to keep the VFMA units busy. In addition, this interplay changes with the number of architectural registers and VFMA units in the processor, requiring an automatic code generation technique that is robust to different microarchitectural implementations while taking full advantage of reuse opportunities exhibited by SGEMM calculation.

3 OVERVIEW

This work focuses on devising a set of solutions to the aforementioned physical register file (PRF) energy and performance limitations. This section presents a sketch of the solution components spanning both hardware and software that allow a general purpose CPU design to overcome those limitations.

Hardware. Our solution, Locality Extensions for Deep Learning (LEDL), takes advantage of the substantial data reuse opportunities inherent in the SGEMM calculations to efficiently utilize the scarce PRF bandwidth available on the CPUs. LEDL centers around two key modifications in the CPU microarchitecture to reduce the burden on PRF. First, we add an *architecturally visible register*, VFMA remote register, embedded in each VFMA unit. Second, we add *low-cost unidirectional inter-VFMA links* between the VFMA units, that a VFMA unit can use to pass on the data to the connected VFMA unit. These microarchitectural modifications enable the programmer to reuse

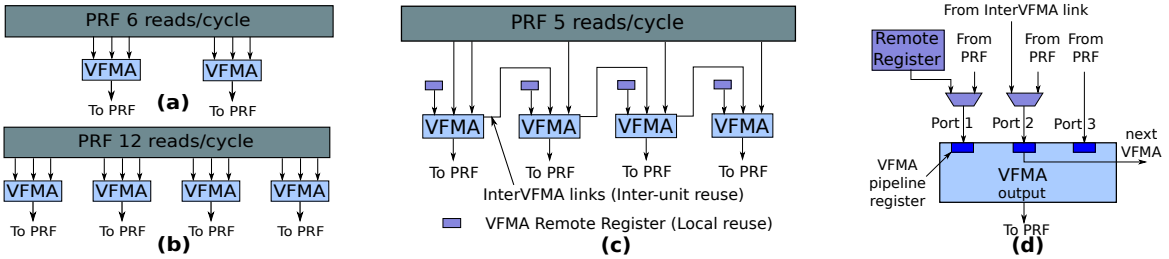


Figure 2: Architecture overview; (a) Haswell processor status with 2 VFMA units, (b) Straightforward extension to 4 VFMA units (c) LEDL introduces VFMA remote register and InterVFMA links, and (d) LEDL modifications to VFMA input ports

the data multiple times, both within and across the VFMA units, instead of reading from the PRF every time, effectively reducing the register reads per cycle while allowing to pack more VFMA units in the CPU. We contrast LEDL to related work in Section 6.

Software. We introduce Automatic Code Generator (ACG) that automatically generates code for SGEMM calculations suitable for a given number of architectural registers and VFMA units, while maximizing the data reuse. ACG leverages two optimization strategies - *Register tiling and Prefetcher-friendly layout transformation* - to keep compute units busy. These optimization parameters depend on the number of architectural registers and VFMA units. ACG analytically finds a suitable set of optimization parameters that structure the computation in a manner necessary to achieve high data reuse not only within the PRF, but also within and across the VFMA units as facilitated by LEDL microarchitectural additions.

4 DESIGN AND IMPLEMENTATION

CNN applications have high compute and energy requirements. Improving performance of CNN applications on CPUs requires adding more VFMA units, while keeping the energy-hungry PRF in check. In this section, we present LEDL hardware and software implementation details designed to improve CPU energy efficiently for CNNs.

4.1 Hardware Design

4.1.1 Energy-Efficient PRF Usage. LEDL's goal is to reduce the burden on PRF, while being able to pack more compute in the CPUs. It utilizes the data reuse inherent in SGEMM calculations to reduce PRF reads, effectively reducing the PRF bandwidth and energy requirements. We achieve this energy-efficient usage of PRF by making minor modifications in the VFMA units.

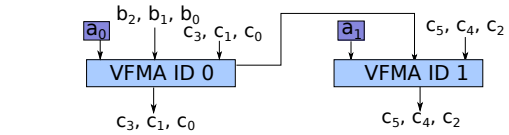
Figure 2 gives an overview of the current state of the PRF and FMA units and our microarchitectural extensions. Figure 2(a) shows the status of current Intel Haswell processor design having 2 VFMA units connected to the PRF. Each VFMA unit requires 3 register operands from the PRF and writes 1 register in the PRF, requiring a total of 6 PRF reads per cycle for Intel Haswell. Figure 2(b) shows a straightforward extension of Intel Haswell architecture, having 4 VFMA units. This configuration requires PRF bandwidth of 12 register reads per cycle, incurring significantly high energy cost. SGEMM calculations have high data reuse opportunity which can be exploited to reduce the number of PRF reads per cycle substantially. To utilize this data reuse, we extend each VFMA unit to achieve temporal reuse within and across the VFMA units, as shown in Figure 2(c). First, each VFMA unit has an architecturally visible register, referred to as VFMA remote register capable of reusing a vector register input locally (at the same unit) across multiple operations. Second, the VFMA units are connected with unidirectional links, referred to as InterVFMA links, adding opportunity of inter-unit reuse across VFMAs.

Local Reuse - VFMA Remote Register. To reuse a data value locally, each VFMA unit is augmented with an architecturally visible register. This register is different from other architectural registers in that it is coupled with a particular VFMA unit. It can be written from the caches or from the other registers like other architectural registers, but it cannot be written by the VFMA itself. It is used for storing an input value that can be reused multiple times, which would have otherwise come from PRF. Localizing the usage of the remote register to its VFMA unit, while also disallowing the VFMA to update it, results in little hardware overhead. VFMA remote register adds a capability of reducing the PRF bandwidth requirement by a maximum of one-third if the application data-reuse is efficiently utilized.

Inter-unit Reuse - InterVFMA links. Further, LEDL exposes inter-unit reuse capability in VFMA units by connecting them via a unidirectional link, as shown in Figure 2(c). The VFMA unit can obtain one of its operands from the InterVFMA link, instead of reading it from PRF. These links help in achieving inter-unit reuse, where an operand can be read just once from the PRF and then can be reused across VFMA units by using InterVFMA links. When coupled with VFMA remote registers, this further cuts down the PRF reads by around one-third by reusing the same value across different VFMA units. Similar to VFMA remote register, InterVFMA links transfer only the input data and do not support transfer of VFMA output to next VFMA input (discussed more in Section 6).

VFMA Input Ports. To take advantage of local and inter-unit reuse, we modify VFMA input port design so that it is flexible enough to take inputs from PRF, its Remote Register and InterVFMA link. Figure 2(d) shows the implementation details of VFMA ports. Typical VFMA unit has 3 input ports and 1 output port. In current Haswell architecture, each of these input ports is connected to the PRF. We modify input port 1 to take the input from either Remote register or PRF and input port 2 to obtain the input from either InterVFMA link or PRF. Input port 3 is kept unmodified, receiving the operand from the PRF. The VFMA output port is also kept unmodified, writing back the value in the PRF as usual.

4.1.2 Instruction Set Architecture. Here, we describe the ISA extensions required to utilize the microarchitectural data reuse capabilities exposed by LEDL. We use x86 operations to explain the workings of these ISA extensions, but the ideas can be applied to other ISAs as well.



Tags are shown by **[VFMA ID tag, Group ID tag]**

0 : Loads for values b_0, b_1 and b_2
1 : #1 - `vfma(a0, b0, c0)` **[0,0]**
2 : #2 - `vfma(a0, b1, c1)` **[0,1]** #3 - `vfma(a1, b0, c2)` **[1,1]**
3 : Cache miss for b_2 - Stalled
4 : Still waiting for b_2 - Stalled
5 : #4 - `vfma(a0, b2, c3)` **[0,2]** #5 - `vfma(a1, b1, c4)` **[1,2]**
6 : #6 - `vfma(a1, b2, c5)` **[1,3]**

Figure 3: Example of leveraging VFMA ID and Group tags for instruction scheduling

Remote Register Instructions. VFMA Remote Registers are architecturally visible registers that can be written by conventional move operations, moving the data from memory or other architectural registers to the remote registers. From a programmer's perspective, these are new registers that are dedicated to the VFMA units. An example of move operation from memory to a VFMA remote register(`%vfma0reg`) is:

```
vmov 0(%rcx), %vfma0reg
```

where `vmov` instruction transfers a vector word from the memory to the VFMA0 Remote Register.

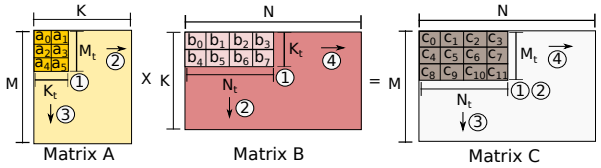
VFMA Instructions. Most of the our ISA extensions are restricted to VFMA instructions. These extensions provide the select signal for the multiplexers in the VFMA input ports shown in Figure 2(d), resulting in 4 categories of VFMA operations:

```
vfma <PRF>, <PRF>, <PRF>
vfma <RR>, <PRF>, <PRF>
vfma <PRF>, <LINK>, <PRF>
vfma <RR>, <LINK>, <PRF>
```

where `<RR>`, `<LINK>` and `<PRF>` denote that the operand is read from Remote Register, InterVFMA link and Physical Register File respectively. Note that the first category is the class of VFMA operations currently supported in a conventional microarchitecture, choosing all the operands from PRF.

VFMA and Group ID Tags. To facilitate precise instruction scheduling of VFMA instructions to take advantage of our design (discussed next in Section 4.1.3), we add two fields to the VFMA opcode specification. First, each VFMA unit is assigned a tag that can be specified in each instruction. The instruction scheduler extracts this tag from the VFMA instruction opcode and then issues the instruction to the specific VFMA unit as identified by the tag. Second, a Group ID tag provides another layer of precise scheduling capability by informing the instruction scheduler about the instructions that should be issued simultaneously. All the VFMA instructions that have the same Group ID tag must be scheduled simultaneously. This means that every instruction, in the group of VFMA instructions with same Group ID tag, must have its operands ready before the whole group can be issued. This can be seen as introducing a degree of in-orderness to the execution of these groups of instructions, however we show in Section 5.2 that this effect has minimal impact on the application performance.

4.1.3 Instruction Scheduling. Dynamic Instruction schedulers in CPUs have the responsibility of scheduling ready-to-issue instructions to the functional units as they become available. In current Haswell processors, whenever the dynamic



① computes partial sum of output tile C for input tile A and B
Repeating ①② computes one output tile to completion
Repeating ①②③ computes Nt columns of ouputs
Repeating ①②③④ finishes matrix multiplication

Figure 4: Register tiling steps performed by ACG

scheduler encounters a ready VFMA instruction, it schedules it on either of the VFMA units, whichever is available.

However, our extensions pose two challenges in the instruction scheduling - (a) The VFMA instructions need to be carefully scheduled to the relevant VFMA units. Since each Remote Register is local to its VFMA, and Inter FMA links are also unidirectional, the operations have to be orchestrated in a certain manner and cannot be scheduled randomly as done by the current dynamic scheduler. (b) In addition, some of the instructions in this pre-defined sequence might not be ready because one of their operands might be waiting for a cache miss to get resolved. To address these challenges, the instruction scheduler takes advantage of the two fields – VFMA ID tag and Group ID tag – included in the VFMA instruction specification.

Figure 3 shows the usage of these two tags, using an instruction sequence operating on 2 VFMA units connected via InterVFMA links. VFMA remote registers are already loaded with operands a_0 and a_1. The sequence of operands that go on the InterVFMA links is b_0, b_1 and b_2. The third operand, also the output register, comes from the register file and denoted by c_i. The figure shows the instruction sequence where each instruction has the associated tags in the square brackets [VFMA ID tag, Group ID tag]. In cycles 1 and 2, the VFMA ID tag directs the scheduling of instruction in the corresponding VFMA units. Also Cycle 1 and Cycle 2 have different Group IDs, forcing the instruction scheduler to follow the sequence. Cycle 3 shows an event where operand b_2 is unavailable due to a cache miss. Since instructions 4 and 5 share the same Group ID tag, even though instruction 5 is ready to be issued, instruction scheduler delays its issue until Cycle 5, when instruction 4 operand b_2 is also available. Group ID and VFMA ID tag, therefore, help in achieving precise instruction scheduling that is necessary to utilize the local and inter-unit reuse capabilities exposed by LEDL.

4.2 Code Generation

Increasing VFMA units in the CPU requires an automatic code generator that can generate the code as per the availability of hardware resources, while also maximizing the heavy data reuse exhibited in the SGEMM calculations. Our code generator, ACG, leverages two optimization strategies - Register Tiling and Prefetcher-friendly layout transformation - to maximize data reuse and keep VFMA units busy. Using these optimizations, ACG structures the computation in a manner, where data can be reused within and across the VFMA units. ACG, then, maps the computation to LEDL using the ISA extensions described in Section 4.1.2.

4.2.1 Register Tiling. SGEMM kernel calls have high data reuse, providing opportunities of achieving high compute

```
(a)
1  // The output tile is kept in registers - c0, c11
2  vmov b0, r0; vmov b1, r1; vmov b2, r2; vmov b3, r3 // Load tile B
3
4  vbroadcast a0, r4  // Read element from tile A
5  // Calculate partial sum for the first row of output tile
6  vfma r0, r4, c0; vfma r1, r4, c1; vfma r2, r4, c2; vfma r3, r4, c3
7
8  // Perform the same compuation for next output rows
9  vbroadcast a2, r4
10 vfma r0, r4, c4; vfma r1, r4, c5; vfma r2, r4, c6; vfma r3, r4, c7
11 vbroadcast a4, r4
12 vfma r0, r4, c8; vfma r1, r4, c9; vfma r2, r4, c10; vfma r3, r4, c11
13 // Repeat line 4-12 for next row of tile B and next colum of tile A (unroll Kt times)
```

```
(b)
1  // The output tile is kept in registers - c0, c11
2  vmov b0, rr0; vmov b1, rr1; vmov b2, rr2; vmov b3, rr3 // Load tile B
3
4  vbroadcast a0, r4; vbroadcast a2, r5; vbroadcast a4, r6
5
6  // Compute partial sums (Column is VFMA ID tag, row inst have same group ID tag)
7  vfma rr0, r4, c0;
8  vfma rr0, r5, c4; vfma rr1, Link, c1;
9  vfma rr0, r6, c8; vfma rr1, Link, c5; vfma rr2, Link, c2;
10                   vfma rr1, Link, c9; vfma rr2, Link, c6; vfma rr3, Link, c3;
11                                       vfma rr2, Link, c10;vfma rr3, Link, c7;
12                                                           vfma rr3, Link, c11;
13 // Repeat the same steps for next row of tile B and next colum of tile A
```

Figure 5: Code generation template for the partial sum output tile calculation for (a) non-LEDL and (b) LEDL hardware

to memory ratio. To take advantage of this reuse, it is necessary to perform aggressive vector register tiling in the CPUs. We show later in Section 5.8 that by utilizing the registers efficiently, we can achieve upto 5× performance improvements as compared to a software that underutilizes the registers.

The details of our register tiling approach are illustrated in Figure 4, showing the steps involved in applying register tiling when multiplying input matrices A and B to produce output matrix C. The tiling is performed for both input and output matrices. As shown in the figure, the input A tile size if $M_t \times K_t$, and the input B tile size is $K_t \times N_t$, resulting in an output tile size of $M_t \times N_t$. The output tile holds the partial sum for the multiplication of A and B input tiles. Structuring SGEMM calculations in this manner not only exposes data reuse in PRF, but also within and across VFMA units, where LEDL can be leveraged to achieve better energy characteristics.

We show the details of the partial output calculation for non-LEDL hardware in ① in Figure 4, while the corresponding code template is shown in Figure 5(a),where the tiling parameters – M_t, N_t, K_t – are set at (4,24,2). Firstly, first row of the input B tile (N_t elements) is read from the memory into the registers (line 2). These values are reused before moving on to the next row. Now, elements from the first column of the input tile A are read one-by-one and used to compute the partial sums for the first row of output (line 4 - line 12). Note that element A is a scalar, which has to be replicated by vector length (shown as `broadcast` instruction in line 4), as the same value is multiplied to each element in each vector of the current row of input tile B. Once all the elements in the column of A are used, we move to second column of tile A and second row of tile B. This essentially translates into unrolling the loop by K_t times.

Once this partial sum calculation finishes, there are several options to choose from. We observed that computing an output tile to completion results in the best performance, as it achieves maximum reuse possible for the output matrix. We achieve this by moving the tile horizontally in matrix A and vertically in matrix B, shown in the figure by ②, resulting in the completion of the output tile of elements $M_t \times N_t$. We then move the output tile vertically down shown by ③. Repeating ①, ② and ③ results in the completion of $M \times N_t$ output elements. Finally, we move to the next column, as shown by ④. Repeating ①, ②, ③ and ④ results in the completion of matrix multiplication.

While the underlying basics for performing register tiling using LEDL features remain same, the implementation details change slightly. The corresponding template is shown in Figure 5(b) which can be understood in conjunction with Figure 6 showing the values that are used within (local reuse) and the values that are used across the VFMA units (inter-unit reuse). The row elements of input tile B are brought into the VFMA remote registers (shown by `rr` in line 2), reusing these operands locally. All the column elements of input tile A are read into the registers before the actual computation starts (line 4). The values of these registers is now passed one by one to the first VFMA register which then transfers the value to the next units using InterVFMA links, enabling inter-unit reuse. While hoisting all the input tile reads to the start increases the register pressure, it results in better performance as it hides the memory latency to large extent.

Identifying Suitable Tiling Parameters. An objective of our code generation step is to find suitable tiling parameters that fit the hardware specifications, while also maximizing the data reuse opportunities. Analyzing the aforementioned template, we can easily find the relationship between the tiling parameters and the number of architectural registers. In addition, we can also calculate compute-to-memory-access ratio (CMAR) which captures data reuse at the PRF. ACG, using these relationships, generates a software variant by choosing an efficient set of tiling parameters that maximizes data reuse while fitting in available architectural register count.

As we can see from the template, for the software that does not use LEDL capabilities, ① requires 1 register for input tile A, N_t/VL registers for input tile B and $M_t * N_t/VL$ registers for input tile C, where VL refers to the Vector Length; the number of floating point elements that can fit into a vector. For compute-to-memory-access ration (CMAR), the template performs $M_t * N_t/VL$ VFMA operations for every 1 memory read from input tile A and N_t/VL memory reads from input tile B. Therefore, the resulting relationship between tiling parameters and register tiling and CMAR is

$$Arch\ Registers = 1 + N_t/VL + M_t * N_t/VL \quad (1)$$

$$CMAR = (M_t * N_t/VL)/(1 + N_t/VL) \quad (2)$$

Similarly, the relationships when we leverage LEDL capabilities are

$$Arch\ Registers = M_t + N_t/VL + M_t * N_t/VL \quad (3)$$

$$CMAR = (M_t * N_t/VL)/(M_t + N_t/VL) \quad (4)$$

Depending on whether the HW supports LEDL, ACG chooses the relevant equations and picks the tile parameters that has the highest compute-to-memory ratio, while also fitting inside the available architectural register file size.

4.2.2 Prefetcher-friendly Layout Transformation. We observe that for many convolution layers, even after applying aggressive register tiling, the generated code variants still have low VFMA utilization, sometimes as low as 50%. Upon further investigation, we find that CPU is heavily stalled on cache misses, even though the memory access pattern seems to be predictable for the cache prefetchers. The reason for this slowdown is that the prefetchers are not allowed to prefetch beyond page boundaries. In convolution layers, the matrices are typically large resulting in stride larger than a page boundary when the the data access pattern jumps to next row of the matrix.

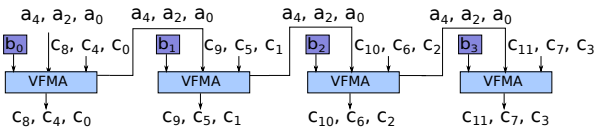


Figure 6: Computation and data movement for the LEDL code

To solve this issue, before starting any SGEMM computations, ACG performs a prefetcher-friendly layout transformation on the input matrices A and B, so that the memory access pattern becomes a continuous back-to-back sequence during the compute part. The overhead of performing this transformation (in the order of $O(M*K+K*N)$) typically gets amortized because of an order of magnitude higher number of FMA operations (in the order of $O(M*N*K)$), where the transformed data is reused multiple times. With this layout transformation, the prefetchers work very efficiently bringing most of the data in L1 caches before it is actually required, resulting in higher compute utilization. ACG uses the tiling parameters with vector memory operations (like Intel AVX *vmov* instruction) to generate the code for layout transformation. The transformation code is same irrespective of whether the code is utilizing LEDL reuse features. Figure 7 shows this transformation for input matrices A and B for the example discussed in Section 4.2.1. The transformation can be viewed as flattening the 2-dimension matrix into a 1-dimensional matrix, such that every next access is located contiguously in this flattened array.

Interleaving Transformation and Compute. To further reduce the cost of layout transformation, ACG interleaves some portion of compute with the layout transformation. Since layout transformation typically stalls on the memory, the interleaving utilizes the unused VFMA units to complete a small portion of SGEMM calculation in parallel. This technique is particularly useful for the cases where the amount of computation in the SGEMM computation is smaller, where the impact of hiding the overhead of the transformation becomes more visible.

5 EVALUATION

5.1 Methodology

Applications. We evaluate our hardware and software mechanisms on 5 state-of-the-art CNN applications – Alexnet, Overfeat, VGG_16, NiN and ResNet [28, 37, 42, 50, 52]. These are medium to large CNNs, presenting a large variation in convolution layer shapes and sizes. The number of convolution layers in the five CNN applications are 5, 5, 13, 12 and 50 respectively. Additionally, we evaluate our hardware on a variety of other widely used DNN layers like Fully Connected and Long short-term memory layers (a type of Recurrent layer). The configuration of these networks is detailed in Section 5.6.

Performance and Energy Measurement. We use Snipersim [15] to evaluate the performance impact of LEDL hardware and software mechanisms. We have augmented the Snipersim infrastructure to simulate the vector instruction extensions described in Section 4.1.2, along with VFMA Remote Register and InterVFMA link implementations. We took efforts to ensure that Snipersim achieved similar performance statistics in simulation to the characteristics observed on real Haswell processors for the Intel MKL and ACG generated software variants. Our experiments use McPAT infrastructure [40], extended to include the techniques described by Sam et al. [60], to model

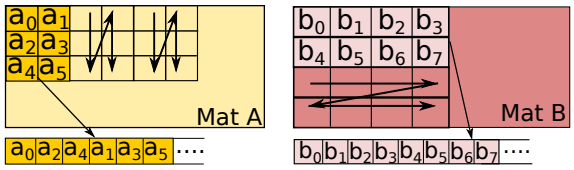


Figure 7: ACG's Prefetcher-friendly layout transformation

energy and area consumption. The energy and area measurements used throughout the evaluation include core and all three levels of caches.

Baseline and Hardware Configurations. Our baseline, where not stated otherwise, is derived from a currently available Intel Haswell server processor design whose configuration details are described in Table 1. Physical floating point registers in our designs are fixed to 168, similar to the Haswell baseline. We increase the number of architectural registers to 96, unless otherwise specified. We never observed structural hazards due to unavailability of physical registers in our experiments.

We study the impact of local and inter-unit reuse by evaluating across three supported modes of VFMAs:

- *NR* (No Reuse): The VFMA unit reads all 3 register operands from PRF, requiring 3 PRF reads per cycle.
- *LR* (Local Reuse): The VFMA reads one operand from its Remote Register and other two from the PRF, taking advantage of local reuse, requiring 2 PRF reads per cycle.
- *FR* (Full Reuse): The VFMA reads one operand from its Remote Register, one from its InterVFMA link and one from the PRF, utilizing both local and inter-unit reuse, requiring 1 PRF read per cycle.

Table 2 lists the hardware design points that we use for our evaluation. We observe that for 2, 3 and 4 VFMAs, the PRF can have enough read ports to support *NR* mode. However, 5 and 6 VFMAs require 15 and 18 PRF read ports, at which point PRF cannot meet the timing constraints. Using VFMAs in *LR* and *FR* modes does not require 18 read ports. Therefore, we use a hybrid design for 5 and 6 VFMAs, where the number of PRF read ports are kept to 12 (2 per VFMA). Unless otherwise specified, we use these hardware design points for evaluation.

Processor	8-wide OoO core, 2.4 GHz 192-entry ROB, 72-entry load queue
Private L1 cache	32 KB, 8-way, 2-cycle, 64 B block
Private L2 cache	256 KB, 8-way, 5-cycle, 64 B block
Shared LLC	8 MB, 16-way, 12-cycle, 64 B block
Main memory	1 GB, 65 ns latency
L1, L2 and LLC prefetcher	Line prefetcher

Table 1: Baseline hardware configuration, modeled after an Intel Haswell server configuration

Design point name	VFMAs	PRF read and write ports
2-VFMA (Baseline)	2	6 read and 2 write
3-VFMA	3	9 read and 3 write
4-VFMA	4	12 read and 4 write
5-VFMA-Hybrid	5	12 read and 5 write (*NR* not supported)
6-VFMA-Hybrid	6	12 read and 6 write (*NR* not supported)

Table 2: Hardware design points

5.2 Performance and Energy Improvements

In this section, we examine the characteristics of LEDL to understand the tradeoffs the different hardware design points offer in terms of performance and energy usage.

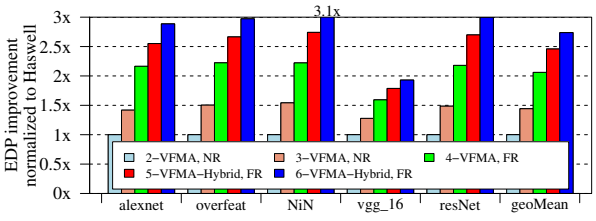


Figure 8: EDP improvement of increasing the VFMA units for end to end total convolution runtime.

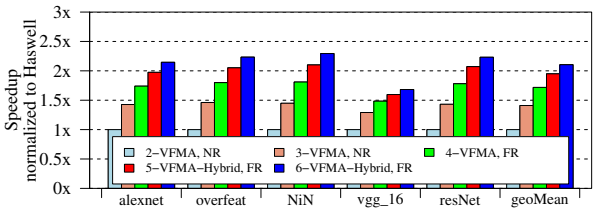


Figure 9: Performance improvements of increasing the VFMA units for end to end total convolution runtime

Energy Delay Product. In the first experiment, we use ACG to generate software for each convolution layer in the CNNs of our application suite. We then measure the energy consumption of each layer for our hardware designs points and each VFMA mode. We accumulate the energy for each convolution layer per DNN and measure the Energy delay product (EDP). The findings of this experiment are presented in Figure 8, showing the EDP improvement for the best FMA mode for each hardware design point, over the Intel Haswell baseline.

We observe that increasing the number of VFMA units results in significant EDP improvements over the Haswell baseline. LEDL extensions substantially reduce the number of PRF reads, resulting in average EDP improvements of 2.0×, 2.5× and 2.7× with *FR* mode on 4, 5 and 6 VFMA units. For lower number of VFMA units (2 and 3), *NR* mode achieves better EDP due to better tile characteristics.

Performance. Next, we perform the same experiment and measure the performance of each layer for our hardware designs points and each VFMA mode, giving us the total convolution runtime. The findings of this experiment are presented in Figure 9, showing the speedup of the best reuse mode for each hardware design point against the Haswell baseline.

We observe that adding VFMA units results in geometric mean speedup of 1.4×, 1.7× for 3 and 4 VFMA units for *NR* mode. Further, PRF cannot meet latency constraints for supporting *NR* mode when the number of VFMA units are increased to 5 and 6. Here, LEDL's reuse capabilities reduce the PRF bandwidth requirements, resulting in hybrid designs that improve compute capacity, achieving a performance speedup of 2.0× and 2.1× for *FR* mode on 5 and 6 VFMA units.

5.3 Impact of FMA modes

The LEDL-enabled FMA modes – *LR* and *FR* – reduce the number of PRF reads by taking advantage of local and inter-unit reuse, resulting in better energy consumption characteristics. In this section, we study the energy effect of these FMA modes by measuring the energy and execution time of each CNN layer in our application suite, giving us the total EDP of accumulated CNN layer execution. This experiment is performed for

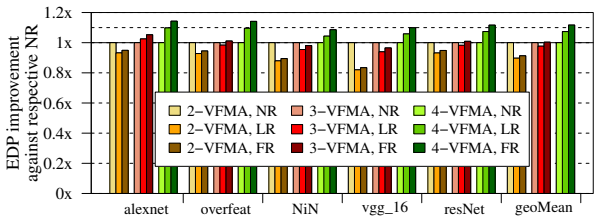


Figure 10: LEDL-enabled FMA modes comparison for 2, 3 and 4 VFMA units; LEDL-enabled modes achieve better EDP design point at 4 VFMA units

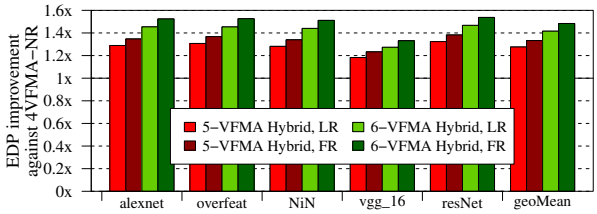


Figure 11: LEDL-enabled FMA modes comparison for 5 and 6 VFMA units. *NR* mode is not supported as PRF latency constraints cannot be met

all hardware designs on the 3 VFMA modes. We present the findings of this experiment in Figures 10 and 11.

First, we show the impact of FMA modes on 2, 3 and 4 VFMA units in Figure 10. The figure shows EDP improvement of LEDL-enabled *LR* and *FR* modes normalized to the currently-supported *NR* mode for 2, 3 and 4 VFMA units. We observe that for 2 and 3 VFMA units, the *LR* and *FR* reuse modes result in minimal improvement. This is because the tile characteristics of code variant for *LR* and *FR* modes have higher energy consumption compared to *NR* mode. In this experiment, we also observe that PRF power is 10% of the total power at 2 VFMA units, but increases to 18% for 4 VFMA units. Due to this high increase in PRF power, we observe that at 4 VFMA units, LEDL starts achieving better EDP characteristics than *NR* mode. On average, *LR* and *FR* achieve EDP improvements of 8% and 10% for 4 VFMA units, respectively, compared to *NR* mode.

However, beyond 4 VFMA units, *NR* mode is not viable because the PRF latency constraints could no longer be met. LEDL, on the other hand, relaxes PRF bandwidth requirements, packing more VFMA units while keeping PRF latency in check. We therefore compare the EDP characteristics of LEDL-enabled modes on 5 and 6 VFMA units to 4 VFMA units with *NR* mode, the best hardware design point currently supported by *NR* mode. This comparison in shown in Figure 11. We observe that LEDL-enabled modes result in significant EDP improvements, achieving an EDP improvement of 1.35× and 1.47× for 6 VFMAs with *FR* mode.

5.4 Impact of Microarchitectural Parameters

In this section, we study the impact of microarchitectural parameters on the energy characteristics of different FMA modes on our hardware design points. We perform the analysis on the conv2 layer of Alexnet (Alexnet's most time-consuming layer).

In this experiment, we measure EDP for Alexnet conv2 layer for different number of architectural registers and different hardware design points. The experiment is conducted for all three VFMA modes. We show the result of this experiment for *NR* mode, modeling the baseline Haswell processor configuration,

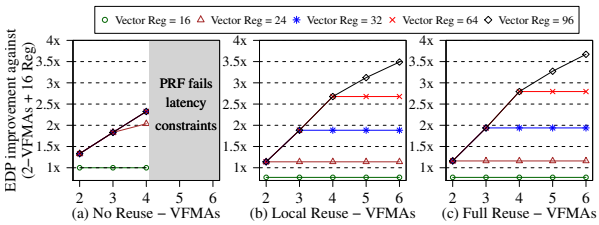


Figure 12: Impact of increasing architectural registers and VFMA units on Alexnet conv2 layer for different FMA modes

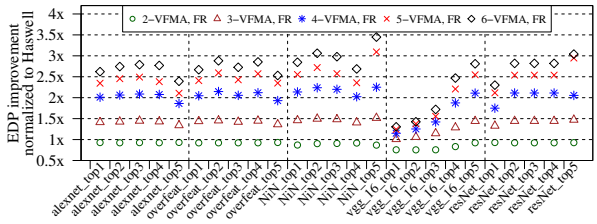


Figure 13: LEDL's EDP improvements on *FR* mode for the top 5 most time contributing layers of our application suite

and *LR* and *FR* modes, LEDL enabled modes that reduce the number of PRF reads, in Figure 12(a), (b) and (c) respectively. The figures show EDP improvements for different hardware design points against a hardware design point having 2 VFMA units and 16 architectural registers.

VFMA units and VFMA modes. First, we observe that increasing VFMA units result in significant EDP improvements for 96 architectural registers. But more importantly, we observe that this increase is limited to 2.4× for 4 VFMA units in *NR* mode. Adding any more VFMA units requires extra PRF bandwidth, reaching a point where PRF latency constraints could no longer be met for *NR* mode (shown by the grey box in (a)). *LR* and *FR* modes reduce the number of PRF reads, substantially reducing the PRF bandwidth requirements. This lets us pack more compute units, extending the number of VFMA units to 5 and 6 for *LR* and *FR* modes, increasing the EDP improvements to 3.5× and 3.7× respectively, as shown in (b) and (c).

Architectural Register Count. Next, we analyze the impact of architectural register count. There are three key observations. First, from (a), we observe that current Intel machines, which have 16 architectural registers, can improve their EDP by 35% just by increasing the architectural registers to 24. Second, number of architectural registers limit the EDP improvements when we increase the number of VFMA units for all VFMA modes. For example, in (a), 4 VFMA units achieve an EDP improvement of 2× at 24 registers, which can be increased to 2.4× at 32 registers. And last, we observe that *FR* and *LR* require larger number of architectural registers for same number of VFMA units as compared to *NR* (equations 1, 3). For example, for 3 VFMA units, *NR* requires 24 registers but *FR* requires 32.

5.5 Layer-by-Layer Analysis

Different convolution layers within the same network can have different performance and compute requirements. In this subsection, we present a layer-by-layer EDP analysis of our application suite. Due to space limitations we show only the top 5 layers in each network (ranked by their contribution to execution time).

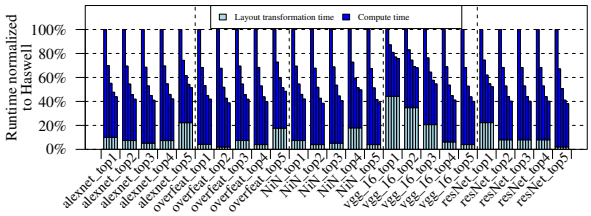


Figure 14: Breakdown of runtime in compute and layout transformation time, as the number of VFMA units are increased (2, 3, 4, 5 and 6 from left to right in each cluster)

EDP Analysis. In this experiment, we evaluate layer-by-layer EDP improvement when the number of VFMA units are increased. The number of architectural registers are kept fixed at 96. ACG takes the number of architectural registers and VFMA units as input and generates a code variant that uses *FR* mode for all the VFMAs. The findings of this experiment are presented in Figure 13, showing the EDP improvements over the Haswell baseline.

We observe that LEDL reuse features achieve significant EDP improvements when the number of VFMA units are increased. Alexnet, for example, achieve close to 2.5x EDP improvement for the top 4 most contributing layers. However we also observe, that some layers like top 3 layers of VGG do not achieve similar EDP improvements. This can be attributed to the high layout transformation time for these layers, which we evaluate next.

Layout Transformation Overhead. To understand the variation of EDP improvements across different convolution layers, we investigate the application execution time breakdown across compute and layout transformation steps. The findings of this experiment are presented in Figure 14, showing the breakdown of layer execution time across compute and layout transformation portion when the number of VFMA units are increased. The runtime is normalized to Haswell baseline.

There are two keys observations from this experiment. First, the compute time scales down gracefully as more VFMA units are added into the CPU. Second, the data transformation time remains constant and is not affected by the VFMA units. These two factors explain the EDP improvement variations observed earlier in Figure 13. VGG_16 layers have large data transformation cost, resulting in smaller EDP improvements. Such layers are characterized by small value of M (rows in matrix A) and large values of K and N (rows and columns, respectively, in matrix B). Therefore, the compute time, in the order of $O(M*N*K)$, in these layers is comparable to data layout transformation time, which is in the order of $O(M*K+N*K)$. Alexnet and ResNet, on the other hand, have low layout transformation overhead, leading to higher EDP improvements.

5.6 Applicability to Other DNNs

Fully Connected and Recurrent Neural Networks. We next measure LEDL impact on two other widely used DNN layers - Fully Connected (FC) and Long Short Term Memory (LSTM) layers (a type of recurrent layer) - that are also implemented atop SGEMM [33]. Recently, Google released an ASIC, having a fast matrix multiplication unit, to accelerate DNN inference – Tensor Processing Unit [33]. The research also showed that a subset of their FC and LSTM layers were compute bound on CPUs (refer to Figure 6 in [33]). Therefore, the extensions

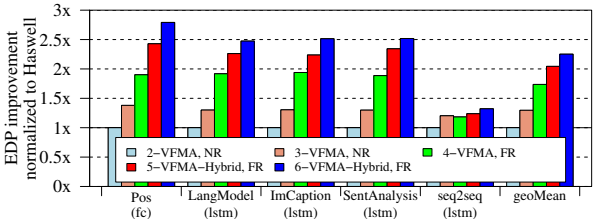


Figure 15: LEDL shows good EDP improvements for other widely used DNN layers – FC and LSTM

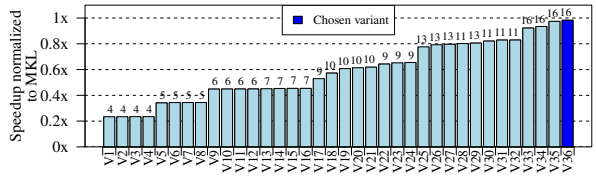


Figure 16: Performance of ACG variants against Intel MKL code. Variant register usage is shown at the top of each bar

offered by LEDL have the potential to improve energy and performance in these cases as well.

To study the applicability of LEDL on these layers, we evaluate a variety of FC and LSTM layers from five application domains – FC layer for Parts of Speech [26], LSTM layer of 200 cells for Language Modelling [62], LSTM layer of 128 cells for Image Captioning [39], LSTM layer of 500 cells for Sentiment Analysis [58] and LSTM layer of 1024 cells for Sequence to Sequence encoder [53]. In this experiment, we measure the EDP improvement for these layers for all hardware design points and VFMA modes. The findings of this experiment are presented in Figure 15, showing EDP improvement for the best VFMA mode for all hardware design points normalized to a conventional Intel Haswell baseline.

We observe that increasing VFMA units achieve significant EDP improvement for 4 out of 5 layers. As we increase the number of VFMAs to 4, *FR* mode starts showing better EDP characteristics, resulting in average speedup of 1.7×, 2.0× and 2.3× for 4, 5 and 6 VFMA units. The last application, seq2seq LSTM layer, shows low EDP improvement because this layer is memory bandwidth bound, resulting in diminishing improvements for additional VFMA units.

5.7 Area Overhead

Increasing raw computation capacity of a CPU requires adding more VFMA units as well as increasing the number of read ports in PRF. LEDL, in addition, introduces additional microarchitectural elements to reduce the PRF read bandwidth requirements. However, LEDL microarchitectural additions have minimal area overhead as VFMA remote register is local to its VFMA and InterVFMA links are also uni-directional with single link between two VFMAs. Therefore, the two major factors that govern the area overhead are VFMA units and PRF. We use McPAT to capture this area overhead for our hardware design points.

The area measurement is performed assuming a traditional CPU server, having 8 CPU cores, each having private L1 and L2 caches and sharing a LLC, whose parameters are listed in Table 1. We observe that the additional area for 3-VFMA, 4-VFMA, 5-VFMA and 6-VFMA-Hybrid design is 4%, 8%, 11% and 15% respectively. Most of this increase is because

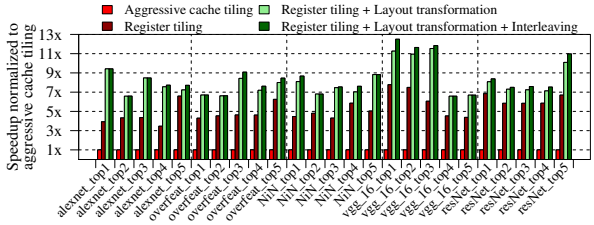


Figure 17: Speedup achieved by different ACG's optimizations

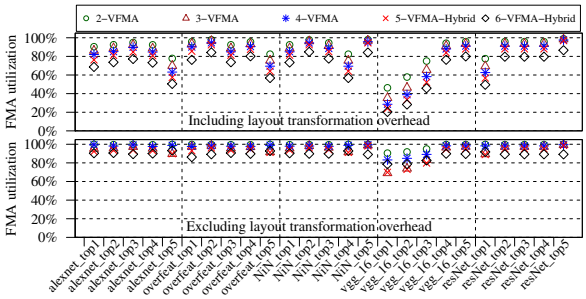


Figure 18: VFMA utilization achieved by ACG including and excluding the layout transformation overhead

of the additional VFMA units. For example, 14% area (compared to total of 15%) for 6 VFMA-hybrid design point is just because of additional VFMA units. Pollack's Rule states that performance increase due to microarchitectural advances are roughly proportional to the square root of increase in complexity, where complexity refers to the area [14]. We observe that LEDL leads to significant performance and energy improvements, that greatly outstrip the typical Pollack's Rule tradeoff.

5.8 Code Generator Efficacy

ACG is designed to generate codes that can take advantage of additional VFMA units, while also maximizing the local and inter-unit reuse. In this section, we evaluate the efficacy of the ACG, both on real hardware and simulation.

ACG Software Variants. In this experiment, we show the inner workings of ACG for Alexnet Conv2 layer on real Intel Haswell machines. Instead of choosing a particular set of tile parameters, we use ACG to sweep the tiling parameters over a small range to generate many software variants and measure their performance on real hardware. The results of this experiment are presented in Figure 16. The figure shows variants' performance against Intel MKL, an aggressively tuned code for Intel Haswell machines. The register usage of each software variant is presented at the top of its bar.

There are two key observations from the figure. First, to achieve the high performance, the SW variant has to efficiently utilize the register storage. Current Intel Haswell processor has 16 architectural registers. The figure shows that the highest performing variant utilizes all of these registers. Second, ACG achieves close to Intel MKL performance, which is an aggressively hand-tuned library.

ACG Optimization Breakdown. ACG uses Register tiling, Prefetcher-friendly layout transformation and Interleaving to achieve high performance on CPUs. In this experiment, we analyze the importance of each of these optimizations on Intel Haswell processor across the top 5 most contributing layers

for each network in our application suite. The performance speedup of the optimizations is presented in Figure 17.

We start with an aggressively Cache-tiled code, that performs cache tiling across L1, L2 and L3 caches. We observe this code performs poorly, leading to heavy under utilization of CPU resources. We then apply Register tiling to our software, leading to huge performance improvement for several convolution layers. Next, we apply the prefetcher-friendly layout transformation. This optimization makes accesses prefetcher friendly, again leading to substantial performance improvements. Finally, we apply Interleaving between compute and transformation, reducing transformation cost by overlapping it with some compute portion, leading to small additional performance improvements.

FMA Utilization. Finally, we study ACG performance when the number of FMA units are increased. In this experiment, we analyze the VFMA utilization for the top 5 most contributing layers for each of our network. We increase the number of VFMA units and instruct ACG to generate software using the *FR* mode of the VFMAs. We present VFMA utilizations of this experiment with and without the transformation overhead in Figure 18 (a) and (b) respectively. We observe that ACG efficiently utilizes the compute for majority of Alexnet, Overfeat and ResNet CNN layers. However, top three layers of VGG_16 have low utilization. To investigate this low VFMA utilization, we exclude the transformation overhead and measure the VFMA utilizations. The findings, presented in part (b), show that ACG achieves high VFMA utilization in SGEMM compute portion.

6 RELATED WORK

Systolic Arrays. A significant amount of accelerator research has been done on DNNs in past few years [16, 18, 24, 29, 33, 41, 46]. Spatial architectures, having distributed compute and memory, have been gaining attention as deep learning accelerators. The Catapult CNN accelerator for FPGAs [46], TPU [33] and Eyeriss [18] are examples of spatial architectures that use or can be configured as systolic arrays to transfer partial sums between the distributed compute elements. DianNao and DaDianNao research present DNN accelerators, focusing on minimizing off-chip as well on-chip data accesses [16, 17].

LEDL's InterVFMA links, enabling inter-unit reuse of data, have some similarities with the systolic dataflow model presented in the spatial architecture DNN research. However, there are substantial differences between the amount of compute and memory in CPUs as compared to spatial architectures. Distributed compute and memory helps spatial architectures divide up the work in a coarse-grained manner where several PEs can compute partial sums for a small subset of inputs in parallel and then transfer these partial sums between the compute elements. This is not possible in CPUs, because there is a centralized PRF and the amount of compute is also limited, preventing coarse-grained division of work. As a result, FMA latency becomes a critical constraint while passing partial sums between the VFMA units on CPUs, resulting in low performance for dataflows employing partial sum transfers. Therefore, instead of passing partial sums, we transfer the input elements between the compute units while maximizing the partial sum usage at PRF.

Multiply-Accumulate units. Similarly, other accelerators use MAC units instead of FMA units to store the intermediate results. However, switching to MAC units in CPUs is not a suitable alternative to get performance improvement for SGEMM kernel. As compared to accelerators, CPUs have very limited number of compute units. The required software dataflow (tiling and resulting memory access pattern) when using the MAC units in CPUs has a very small tile for the output matrix as governed by the number of vector units/intermediate registers. This small partial output tile prevents us from reusing the data in the tiles of input matrices very efficiently, resulting in underutilized hardware. This is in contrast with the FMA units, that has small input matrix tiles and a large partial output tile (in the PRF), providing opportunity to efficiently reuse the data across small input tiles. However, using MAC units might be an efficient design point in accelerators because they can pack more compute units, allowing a large tile for the output matrix.

Weight Pruning and Precision Reduction. Convolution layers show high opportunity of pruning weights, substantially reducing the data footprint and the costs associated with the data movements. Research efforts have focused on either achieving this pruning or designing hardware solutions taking advantage of the pruned datasets [11, 18, 24, 25]. In addition, many DNN applications do not require 32 bits of precision, further reducing the weight storage requirements. DNNs retain their accuracy even after converting the data format to 8/16-bit fixed point format [31, 32, 41, 47]. Many insights from these efforts are orthogonal to LEDL, resulting in additional speedups when applied in conjunction with LEDL.

Software. On software-focused efforts, there have been an increasing number of efforts in writing aggressively hand-tuned codes for hardware, like Intel MKL and NNPACK for CPUs, Nvidia CuDNN and Nervana Neon for GPUs [1, 2, 19, 21], extracting every last ounce of compute packed on the machines. In addition, there have been efforts to reduce the arithmetic complexity of convolution algorithms [38, 56]. Our code generator stands in a similar category of software efforts with focus on automatic code generation for a given number of CPU VFMA units, instead of hand-tuning it for one hardware design point [44, 57].

7 CONCLUSION

In this work, we focus on identifying and alleviating the microarchitectural bottlenecks that prevent us from improving CPU performance on CNN computations. Our study shows that designing a PRF capable of feeding computational units is the primary barrier on achieving higher CPU FLOPS. We present Locality Extensions for Deep Learning (LEDL) , a novel, minimally intrusive set of microarchitectural and ISA extensions that address this problem, along with an automatic code generator needed to take advantage of our design. Our detailed evaluation shows that applying these extensions allows packing more compute in the CPUs, and can achieve a 2× performance improvement and a 2.7× energy-delay product improvement compared to Haswell processors.

REFERENCES

[1] Intel Math Kernel Library. In *http://software.intel.com/en-us/articles/intel-mkl/*.

[2] NervanaGPU library. In *https://github.com/NervanaSystems/nervanagpu*.

[3] An introduction to the intel quickpath interconnect. In *http://www.intel.com/content/www/us/en/io/quickpath-technology/quickpath-technology-general.html*, 2009.

[4] Virtualization is coming to a platform near you. In *https://www.arm.com/files/pdf/System-MMU-Whitepaper-v8.0.pdf*, 2011.

[5] Intel advanced encryption standard (aes) new instructions set. 2012.

[6] AMD64 architecture programmer's manual. 2013.

[7] Nvidia nvlink high-speed interconnect. In *http://www.nvidia.com/object/nvlink.html*, 2016.

[8] Nvidia nvlink high-speed interconnect. In *http://www.nvidia.com/object/nvlink.html*, 2016.

[9] ARM architecture reference manual. 2017.

[10] Intel 64 and ia-32 architectures software developer's manual. In *Volume 3*, 2017.

[11] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[12] S. S. Baghsorkhi, N. Vasudevan, and Y. Wu. Flexvec: Auto-vectorization for irregular loops. In *Programming Language Design and Implementation (PLDI)*, 2016.

[13] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

[14] S. Borkar and A. A. Chien. The future of microprocessors. In *Communications of the ACM*, 2011.

[15] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[16] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[17] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. Dadiannao: A machine-learning supercomputer. In *International Symposium on Microarchitecture (MICRO)*, 2014.

[18] Y.-H. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[19] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cuDNN: Efficient primitives for deep learning. In *arXiV:1410.0759*, 2014.

[20] L. Deng and D. Yu. Deep learning: Methods and applications. Technical report, 2014.

[21] M. Dukhan. NNPACK: Acceleration package for neural networks on multi-core cpus. In *https://github.com/Maratyszcza/NNPACK*, 2016.

[22] F. Franchetti, S. Kral, J. Lorenz, and C. W. Ueberhuber. Efficient utilization of simd extensions. In *Proceedings of the IEEE*, 2005.

[23] E. Grefenstette, P. Blunsom, N. de Freitas, and K. M. Hermann. A deep architecture for semantic parsing. In *arXiV:1404.7296*, 2014.

[24] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: Efficient inference engine on compressed deep neural network. In *International Symposium on Computer Architecture (ISCA)*, ISCA '16, 2016.

[25] S. Han, J. Pool, J. Tran, and W. J. Dally. Learning both weights and connections for efficient neural networks. In *Neural Information Processing Systems (NIPS)*, 2015.

[26] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. In *International Symposium on Computer Architecture (ISCA)*, 2015.

[27] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[28] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. 2015.

[29] P. Hill, A. Jain, M. Hill, B. Zamirai, C.-H. Hsu, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars. Deftnn: Addressing bottlenecks for dnn execution on gpus via synapse vector elimination and near-compute data fission. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.

[30] J. Holewinski, R. Ramamurthi, M. Ravishankar, N. Fauzia, L.-N. Pouchet, A. Rountev, and P. Sadayappan. Dynamic trace-based analysis of vectorization potential of applications. In *Programming Language Design and Implementation (PLDI)*, 2012.

[31] A. Jain, P. Hill, S. C. Lin, M. Khan, M. E. Haque, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars. Concise loads and stores: The case for an asymmetric compute-memory architecture for approximation. In *International Symposium on Microarchitecture (MICRO)*, 2016.

[32] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[33] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *International Symposium on Computer Architecture (ISCA)*, ISCA '17, 2017.

[34] N. Kalchbrenner, E. Grefenstette, and P. Blunsom. A convolutional neural network for modelling sentences. In *arXiV:1404.2188*, 2014.

[35] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. Large-scale video classification with convolutional neural networks. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[36] Y. Kim. Convolutional neural networks for sentence classification. In *arXiV:1408.5882*, 2014.

[37] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Neural Information Processing Systems (NIPS)*, 2012.

[38] A. Lavin. Fast algorithms for convolutional neural networks. 2015.

[39] J. Li, D. Jurafsky, and E. H. Hovy. When are tree structures necessary for deep learning of representations? In *arXiV:1503.00185*, 2015.

[40] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *International Symposium on Microarchitecture (MICRO)*, 2009.

[41] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, and L. Zhong. Redeye: Analog convnet image sensor architecture for continuous mobile vision. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[42] M. Lin, Q. Chen, and S. Yan. Network in network. In *arXiV:1312.4400*, 2013.

[43] C. Lomont. Introduction to intel advanced vector extensions. In *Intel White Paper*, 2011.

[44] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti. Analytical modeling is enough for high-performance blis. *ACM Trans. Math. Softw.*, 2016.

[45] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for simd. In *Programming Language Design and Implementation (PLDI)*, 2006.

[46] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. Chung. Accelerating deep convolutional neural networks using specialized hardware. In *HotChips*, 2015.

[47] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[48] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler. Virtualizing deep neural networks for memory-efficient neural network design. 2016.

[49] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications? In *ACM SIGARCH Computer Architecture News*, volume 40, pages 440–451. IEEE Computer Society, 2012.

[50] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. In *arXiv:1312.6229*, 2014.

[51] K. Simonyan and A. Zisserman. Two-stream convolutional networks for action recognition in videos. In *arXiV:1406.2199*, 2014.

[52] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *arXiV:1409.1556*, 2014.

[53] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *arXiV:1409.3215*, 2014.

[54] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer. Efficient processing of deep neural networks: A tutorial and survey. In *arXiV:1703.09039*, 2017.

[55] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. 2014.

[56] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun. Fast convolutional nets with fbfft: A GPU performance evaluation. In *arXiV:1412.7580*, 2014.

[57] R. M. Veras, T. M. Low, T. M. Smith, R. A. van de Geijn, and F. Franchetti. Automating the last-mile for high performance dense linear algebra. *CoRR*, abs/1611.08035, 2016.

[58] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. In *arXiV:1411.4555*, 2014.

[59] I. Wallach, M. Dzamba, and A. Heifets. Atomnet: A deep convolutional neural network for bioactivity prediction in structure-based drug discovery. 2015.

[60] S. L. Xi, H. Jacobson, P. Bose, G. Y. Wei, and D. Brooks. Quantifying sources of error in mcpat and potential impacts on architectural studies. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[61] J. Yosinski, J. Clune, A. M. Nguyen, T. J. Fuchs, and H. Lipson. Understanding neural networks through deep visualization. In *arXiV:1506.06579*, 2015.

[62] W. Zaremba, I. Sutskever, and O. Vinyals. Recurrent neural network regularization. In *arXiV:1409.2329*, 2014.