# Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration

Hasan Genc*, Seah Kim*, Alon Amid*, Ameer Haj-Ali*, Vighnesh Iyer*, Pranav Prakash*, Jerry Zhao*, Daniel Grubb*, Harrison Liew*, Howard Mao*, Albert Ou*, Colin Schmidt*, Samuel Steffl*, John Wright*, Ion Stoica*, Jonathan Ragan-Kelley†, Krste Asanovic*, Borivoje Nikolic*, Yakun Sophia Shao*
*UC Berkeley, †MIT
hngenc@berkeley.edu

2021 58th ACM/IEEE Design Automation Conference (DAC) | 978-1-6654-3274-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/DAC18074.2021.9586216

***Abstract*—DNN accelerators are often developed and evaluated in isolation without considering the cross-stack, system-level effects in real-world environments. This makes it difficult to appreciate the impact of System-on-Chip (SoC) resource contention, OS overheads, and programming-stack inefficiencies on overall performance/energy-efficiency. To address this challenge, we present Gemmini, an open-source[1], full-stack DNN accelerator generator. Gemmini generates a wide design-space of efficient ASIC accelerators from a flexible architectural template, together with flexible programming stacks and full SoCs with shared resources that capture system-level effects. Gemmini-generated accelerators have also been fabricated, delivering up to three orders-of-magnitude speedups over high-performance CPUs on various DNN benchmarks.**

## I. Introduction

Deep neural networks (DNNs) have gained major interest in recent years in application domains ranging from computer vision, to machine translation, to robotic manipulation. However, running modern, accurate DNNs with high performance and low energy consumption is often challenging without dedicated accelerators which are difficult and expensive to design. The demand for cheaper, high-productivity hardware design has motivated a number of research efforts to develop highly-parameterized and modular hardware generators for DNN accelerators and other hardware building blocks [1]–[7]. While the hardware generator efforts make it easier to instantiate a DNN accelerator, they primarily focus on the design of the accelerator component itself, rather than taking into consideration the system-level parameters that determine the overall SoC and the full software stack. Some industry perspectives have advocated for a more holistic exploration of DNN accelerator development and deployment [8]–[10]. However, existing DNN generators have little support for a full-stack programming interface which provides both high and low-level control of the accelerator, and little support for full SoC integration, making it challenging to evaluate system-level implications.

In this work, we present Gemmini, an open-source, full-stack DNN accelerator generator for DNN workloads, enabling end-to-end, full-stack implementation and evaluation of custom hardware accelerator systems for rapidly evolving DNN workloads. Gemmini's hardware template and parameterization allows users to tune the hardware design options across a broad spectrum spanning performance, efficiency, and extensibility. Unlike existing DNN accelerator generators that focus on standalone accelerators, Gemmini also provides a complete solution spanning both the hardware and software stack, and a complete SoC integration that is compatible with the RISC-V ecosystem. In addition, Gemmini implements a multi-level software stack with an easy-to-use programming interface to support different programming requirements, as well as tight integration with Linux-capable SoCs which enable the execution of any arbitrary software.

Gemmini-generated accelerators have been successfully fabricated in both TSMC $16nm$ FinFET and Intel $22nm$ FinFET Low Power (22FFL) process technologies, demonstrating that they can be physically realized. In addition, our evaluation shows that Gemmini-generated accelerators deliver comparable performance to a state-of-the-art, commercial DNN accelerator [11] with a similar set of hardware configurations and achieve up to 2,670x speedup with respect to a baseline CPU. Gemmini's fully-integrated, full-stack flow enables users to co-design the accelerator, application, and system all at once, opening up new research opportunities for future DL SoC integration. Specifically, in our Gemmini-enabled case studies, we demonstrate how designers can use Gemmini to optimize virtual address translation mechanisms for DNN accelerator workloads, and to partition memory resources in a way that balances the different compute requirements of different layer types within a DNN.

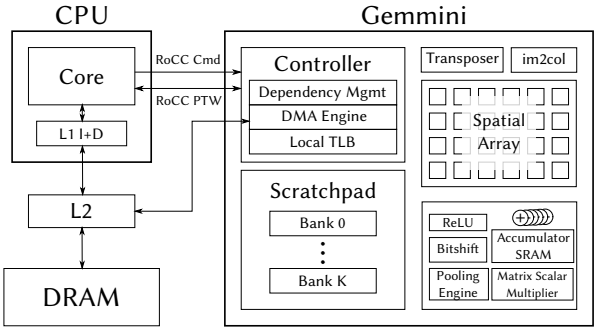



Fig. 1: Gemmini hardware architectural template overview.

In brief, this work makes the following contributions:

1) We build Gemmini, an open-source, full-stack DNN accelerator design infrastructure to enable systematic evaluation of deep-learning architectures. Specifically, Gemmini provides a flexible hardware template, a multi-layered software stack, and an integrated SoC environment (Section III).
2) We perform rigorous evaluation of Gemmini-generated accelerators using FPGA-based performance measurement and commercial ASIC synthesis flows for performance and efficiency analysis. Our evaluation demonstrates that Gemmini-generated accelerators deliver comparable performance compared to state-of-the-art, commercial DNN accelerators (Section IV).
3) We demonstrate that the Gemmini infrastructure enables system-accelerator co-design of SoCs running DNN workloads, including the design of efficient virtual-address translation schemes for DNN accelerators and the provisioning of memory resources in a shared cache hierarchy (Section V).

## II. Background and Motivation

The demand for fast and efficient DNN execution from edge to cloud has led to a significant effort in developing novel accelerator instances that are specialized for different DNN algorithms and/or different deployment scenarios. This section discusses recent advances

[1]https://github.com/ucb-bar/gemmini



Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on January 09,2026 at 02:34:19 UTC from IEEE Xplore. Restrictions apply.

# Gemmini：通过全栈集成实现系统化深度学习架构评估

哈桑·根奇*、西海·金*、阿隆·阿米德*、阿米尔·哈吉-阿里*、维格内什·伊耶尔*、普拉纳夫·普拉卡什*、杰里·赵*、丹尼尔·格鲁布*、哈里森·刘*、霍华德·毛*、阿尔伯特·欧*、科林·施密特*、塞缪尔·斯特夫尔*、约翰·赖特*、伊昂·斯托伊卡*、乔纳森·拉根-凯利†、克尔斯泰·阿萨诺维奇*、博里沃耶·尼科利奇*、雅昆·索菲亚·邵*

*加州大学伯克利分校，†麻省理工学院

hngenc@berkeley.edu

***摘要*——DNN 加速器的开发与评估往往孤立进行，未能充分考虑实际应用环境中的跨堆栈系统级效应。这种局限性使得人们难以全面评估片上系统（SoC）资源争用、操作系统开销及编程堆栈效率低下对整体性能与能效的影响。为解决这一难题，我们推出开源1的全栈 DNN 加速器生成器Gemmini。该工具基于灵活的架构模板，结合可配置的编程堆栈和具备共享资源的完整SoC架构，能够生成涵盖广泛设计空间的高效ASIC加速器。经实际验证，Gemmini生成的加速器在各类 DNN 基准测试中，相较于高性能CPU可实现高达三个数量级的加速效果。**

## I. 引言

近年来，深度神经网络（DNNs）在计算机视觉、机器翻译、机器人操控等应用领域引发广泛关注。然而，若没有专用加速器，要让现代高精度DNNs在高性能与低能耗之间取得平衡往往困难重重——毕竟这类加速器的设计既复杂又昂贵。为满足市场对低成本、高效率硬件设计的需求，学界已开展多项研究，致力于开发高度参数化、模块化的硬件生成器，用于 DNN 加速器及其他硬件组件[1]-[7]。尽管这些硬件生成器让 DNN 加速器的实例化变得容易，但它们主要聚焦于加速器组件本身的设计，而忽视了决定系统级架构（SoC）和完整软件堆栈的系统级参数。部分行业观点主张， DNN 加速器的研发与部署需要进行更全面的探索[8]-[10]。然而，现有 DNN 生成器对提供高/低层控制功能的全栈编程接口支持不足，且对全SoC集成的支持有限，这使得评估系统级影响颇具挑战性。

本研究推出Gemmini开源 DNN 加速器生成器，专为 DNN 工作负载设计。该工具支持端到端全栈实现与评估，可快速部署适配快速演进 DNN 需求的定制化硬件加速系统。Gemmini通过硬件模板与参数化功能，让用户能灵活调整硬件设计方案，实现性能、效率与扩展性的全方位优化。与传统聚焦单一加速器的 DNN 加速器生成器不同，Gemmini提供完整的软硬件解决方案，包含与RISC-V生态兼容的SoC集成。其多层级软件架构搭配直观编程界面，满足多样化开发需求，同时深度集成支持Linux的SoC，可执行任意软件。目前Gemmini生成的加速器已成功实现在 TSMC 16nm FinFET和英特尔22nm FinFET低功耗工艺的制造工艺中。

[1]https://github.com/ucb-bar/gemmini

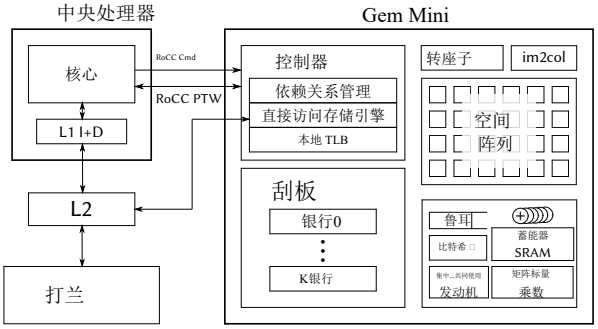



图1：Gemmini硬件架构模板概述。

(22FFL)工艺技术的验证表明其具备物理实现可行性。此外，我们的评估显示，Gemmini生成的加速器在硬件配置相近的情况下，性能可与最先进的商用 DNN 加速器[11]相媲美，相较于基准CPU最高可实现2670倍加速。Gemmini的全栈集成流程支持用户同步完成加速器、应用与系统的协同设计，为未来深度学习SoC集成开辟了新研究方向。具体而言，在Gemmini支持的案例研究中，我们展示了设计师如何利用Gemmini优化 DNN 加速器工作负载的虚拟地址转换机制，并通过内存资源分区平衡 DNN 中不同层级的计算需求。

简言之，本研究的贡献如下：

1) 我们开发了Gemmini，这是一个开源的全栈 DNN 加速器设计平台，用于系统评估深度学习架构。具体而言，Gemmini提供灵活的硬件模板、多层软件堆栈以及集成的SoC环境（第三部分）。

2) 我们采用基于 FPGA 的性能测量和商用ASIC综合流程，对Gemmini生成的加速器进行严格评估，以分析其性能与效率。评估结果表明，Gemmini生成的加速器性能与最先进的商用 DNN 加速器相当（第四节）。

3) 我们证明Gemmini基础设施能够实现运行 DNN 工作负载的SoC系统加速器协同设计，包括为 DNN 加速器设计高效的虚拟地址转换方案，以及在共享缓存层次结构中配置内存资源（第五节）。

## II. 背景与动机

从边缘到云端对快速高效 DNN 执行的需求，推动了针对不同 DNN 算法和/或不同部署场景的新型加速器实例开发的显著努力。本节将探讨最新进展。

.9586216



授权许可仅限以下机构使用： FUDAN 大学。下载于2026年1月9日02:34:19（UTC），来源：IEEE Xplore。使用受限制。

| | Property | NVDLA | VTA | PolySA | DNNBuilder | MAGNet | DNNWeaver | MAERI | Gemmini |
|---|---|---|---|---|---|---|---|---|---|
| Hardware Architecture Template | Datatypes | Int/Float | Int | Int | Int | Int | Int | Int | Int/Float |
| | Dataflows | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Spatial Array | vector | vector | systolic | systolic | vector | vector | vector | vector/systolic |
| | Direct Convolution | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Programming Support | Software Ecosystem | Compiler | TVM | SDAccel | Caffe | C | Caffe | Custom | ONNX/C |
| | Virtual Memory | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| System Support | Full SoC | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| | OS Support | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |

TABLE I: Comparison of DNN accelerator generators.

in DNN accelerators and DNN accelerator generators, motivating the need for a full-stack approach to evaluate deep learning architectures.

### A. DNN Accelerators

Researchers have proposed a large variety of novel DNN accelerators with different performance and energy efficiency targets for different applications across a diverse set of deployment scenarios [1], [12]–[14]. At the architecture level, different DNN accelerators exploit different reuse patterns to build specialized memory hierarchies [15] and interconnect networks [16] to improve performance and energy efficiency. Most existing hardware DNN architectures are largely spatial, where parallel execution units are laid out spatially either in a systolic fashion, as in the case of the TPU, or in parallel vector units like Brainwave [17] and NVDLA [11]. Based on these architectural templates, recent advances have also started exploring how to leverage applications' sparsity patterns [18]–[20] and/or emerging in-memory computing technology [21], [22].

### B. DNN Accelerator Generators

Recent research has designed hardware generators for DNN accelerators [1]–[3], [11], [16], [23]. Table I compares the different features supported by existing hardware generators compared to Gemmini. In contrast to building specific instances of hardware, generator-based approaches provide parameterizable architectural templates that can generate a wide variety of hardware and software instances, improving hardware design productivity. Here, we discuss the hardware, software, and system level requirements for DNN accelerator generators to enable full-stack, systematic DNN architecture evaluation.

DNN accelerator generators must provide flexible architectural templates to cover a wide variety of different DNN accelerator instances, each suited for a different execution environment and a different area/power/performance target. Most DNN accelerator generators today focus only on fixed-point representations and/or only support a single dataflow. In addition, today's generators only target a specific spatial array type, *i.e.*, systolic-based (as in the TPU) or vector-based (as in NVDLA), making it challenging to systematically compare against them. In contrast, Gemmini supports 1) both floating and fixed point data types to handle data representations in training and inference, 2) multiple dataflows that can be configured at design time and run time, 3) both vector and systolic spatial array architectures, enabling quantitative comparison of their efficiency and scalability differences, and 4) direct execution of different DNN operators.

Moreover, DNN accelerator generators also need to provide an easy-to-use programming interface so that end users can quickly program their applications for the generated accelerators. Different developers would prefer different software design environments based upon their targets or research interests. For example, DNN application practitioners would prefer that the hardware programming environment be hidden by DNN development frameworks like PyTorch or TVM so that they don't need to worry about low-level development details, as in the case of VTA [1] and DNNWeaver [23]. At the same time, framework developers and system programmers may want to interact with the hardware at a low level, in either C/C++ or assembly, to accurately control hardware states and squeeze every bit of efficiency out, as in the case of MAGNet [2] and Maeri [16]. Unlike other DNN generators that tend to focus on one of these requirements, Gemmini provides a multi-level programming interface to satisfy users with different requirements. In addition, Gemmini is the first infrastructure that provides hardware support for virtual memory without the need for any special driver software, making it significantly easier for end-users to program accelerators.

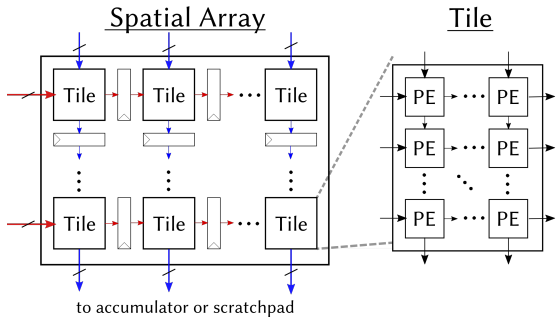



Fig. 2: Microarchitecture of Gemmini's two-level spatial array.

Third, system-level integration, including both the SoC and the system software, is also critical in DNN accelerator generators. Today's DNN accelerators are typically designed and evaluated in isolation. However, when they are eventually deployed, they need to be integrated as part of a larger system. In fact, recent industry evaluations have demonstrated that modern ML workloads could spend as much as 77% of their time running on CPUs, even in the presence of a hardware accelerator, to execute either new operators or to move data between the CPU and accelerators [8]–[10], [24]. However, unfortunately, none of the existing DNN accelerator generators support full SoC integration with host CPUs and shared resources like caches and system buses. Motivated by this observation, Gemmini has built-in system integration support where users can directly instantiate a complete SoC environment that can boot Linux, directly enabling architects to evaluate subtle trade-offs at the system level.

## III. Gemmini Generator

Gemmini is an open-source, full-stack generator of DNN accelerators, spanning across different hardware architectures, programming interfaces, and system integration options. With Gemmini, users can generate everything from low-power edge accelerators to high-performance cloud accelerators equipped with out-of-order CPUs. Users can then investigate how the hardware, SoC, OS, and software overhead interact to affect overall performance and efficiency.

### A. Architectural Template

Figure 1 illustrates Gemmini's architectural template. The central unit in Gemmini's architectural template is a spatial architecture



Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on January 09,2026 at 02:34:19 UTC from IEEE Xplore. Restrictions apply.

| | 财产 | NVDLA | VTA | 多聚唾液酸 | 深度神经网络构建器 | 磁力网 | 深神经网络编织机 | MAERI | Gem Mini |
|---|---|---|---|---|---|---|---|---|---|
| 五金器具<br>架构<br>模板 | 数据类型<br>数据流<br>空间阵列<br>直接卷积 | 整数/浮点数<br>✗<br>矢量<br>✓ | 肠<br>✗<br>矢量<br>✗ | 肠<br>✓<br>心脏收缩的<br>✗ | 肠<br>✓<br>心脏收缩的<br>✓ | 肠<br>✓<br>矢量<br>✓ | 肠<br>✓<br>矢量<br>✓ | 肠<br>✓<br>矢量<br>✓ | 整数/浮点数<br>✓<br>矢量/收缩的<br>✓ |
| 程序设计<br>支持 | 软件生态系统<br>虚拟存储器 | 汇编者<br>✗ | TVM<br>✗ | SD加速器<br>✗ | 咖啡<br>✗ | C<br>✗ | 咖啡<br>✗ | 自定义<br>✗ | ONNX/C<br>✓ |
| 体系<br>支持 | 全片上系统<br>操作系统支持 | ✗<br>✓ | ✗<br>✓ | ✗<br>✗ | ✗<br>✗ | ✗<br>✗ | ✗<br>✗ | ✗<br>✗ | ✓<br>✓ |

表I： DNN 加速器发生器对比

在 DNN 加速器和 DNN 加速器生成器中，这促使我们采用全栈方法来评估深度学习架构。

### *A. DNN 加速器*

研究人员针对不同应用场景提出了多种新型 DNN 加速器，这些设备在性能和能效方面各具特色，适用于多样化的部署场景[1]，[12]–[14]。从架构层面看，不同 DNN 加速器通过采用不同的复用模式来构建专用存储器层级结构[15]和互连网络[16]，从而提升性能和能效。现有硬件 DNN 架构大多采用空间化布局，其并行执行单元要么以 TPU 的系统级并行方式排列，要么采用 Brainwave[17]和 NVDLA [11]等并行向量单元。基于这些架构模板，近期研究还开始探索如何利用应用的稀疏性特征[18]–[20]和/或新兴的内存计算技术[21][22]。

### *B. DNN 加速器发生器*

近期研究已针对 DNN 加速器设计了硬件生成器[1]-[3]、[11]、[16]、[23]。表I对比了现有硬件生成器与Gemmini支持的不同功能。与构建特定硬件实例不同，基于生成器的方法提供了可参数化的架构模板，能够生成多种硬件和软件实例，从而提升硬件设计效率。本文将探讨 DNN 加速器生成器在硬件、软件及系统层面的要求，以实现全栈、系统化的 DNN 架构评估。

DNN 加速器生成器必须提供灵活的架构模板，以覆盖多种不同的 DNN 加速器实例，每个实例都适用于不同的执行环境和不同的面积/功耗/性能目标。目前大多数 DNN 加速器生成器仅专注于定点表示和/或仅支持单一数据流。此外，当前的生成器仅针对特定的空间阵列类型，*即*基于 TPU 的系统阵列或基于 NVDLA 的向量阵列，这使得系统性比较变得困难。相比之下，Gemmini支持：1）同时处理训练和推理中的浮点与定点数据类型；2）可在设计时和运行时配置的多数据流；3）支持向量和系统阵列两种空间阵列架构，从而能够定量比较其效率和可扩展性的差异；4）直接执行不同的 DNN 算子。

此外， DNN 加速器生成器还需提供易用的编程接口，以便终端用户能快速为生成的加速器编写应用程序。不同开发者会根据其研究目标或兴趣偏好不同的软件设计环境。例如， DNN 应用开发者更倾向于让硬件编程环境被PyTorch或 TVM 等 DNN 开发框架所隐藏，从而无需操心底层开发细节。

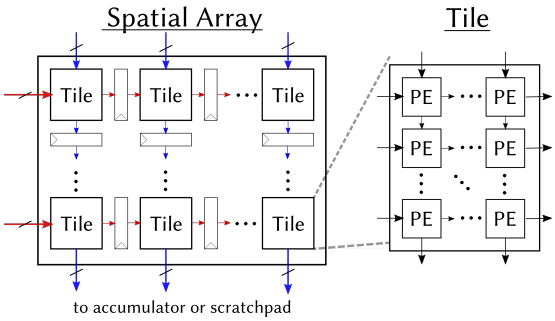



图2：Gemmini双层空间阵列的微架构。

具体实现细节可参考 VTA [1]和DNNWeaver[23]的案例。与此同时，框架开发者和系统程序员可能需要通过C/C++或汇编语言与硬件进行底层交互，以精准控制硬件状态并榨取每一分效率，例如MAGNet[2]和Maeri[16]的实现方式。与其他 DNN 生成器通常只关注其中某一方面不同，Gemmini提供了多层级编程接口，能够满足不同需求的用户。此外，Gemmini是首个无需特殊驱动软件即可支持虚拟内存硬件的基础设施，这使得终端用户编程加速器变得更为简便。

第三，系统级集成（包括SoC和系统软件）对 DNN 加速器生成器同样至关重要。当前 DNN 加速器通常采用独立设计和评估模式，但实际部署时必须作为整体系统的一部分进行整合。最新行业评估表明，即使配备硬件加速器，现代机器学习工作负载仍有高达77%的运行时间消耗在CPU上——无论是执行新算子还是在CPU与加速器间传输数据[8]-[10][24]。但遗憾的是，现有 DNN 加速器生成器均不支持与主机CPU及缓存、系统总线等共享资源的全SoC集成。基于这一发现，Gemmini内置了系统集成支持功能，用户可直接构建能启动Linux系统的完整SoC环境，使架构师能够直接在系统层面评估细微的性能权衡。

## III.伽米尼发生器

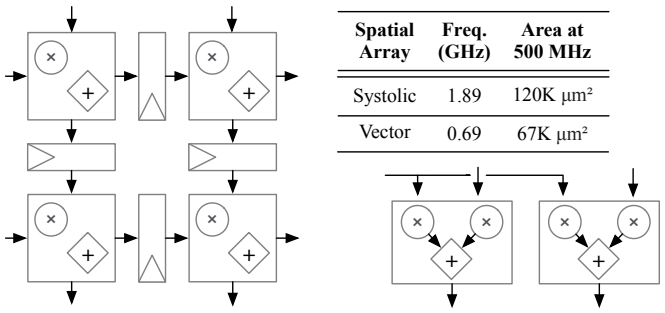
Gemmini是一款开源的全栈 DNN 加速器生成器，支持多种硬件架构、编程接口和系统集成方案。通过Gemmini，用户可生成从低功耗边缘加速器到配备乱序执行CPU的高性能云加速器等各类设备。用户还能深入分析硬件、SoC、操作系统和软件开销如何协同影响整体性能与效率。

### *A. 建筑模板*

图1展示了Gemmini的架构模板。Gemmini架构模板中的核心单元是一个空间架构



授权许可仅限以下机构使用： FUDAN 大学。下载于2026年1月9日02:34:19（UTC），来源：IEEE Xplore。使用受限制。



| Spatial Array | Freq. (GHz) | Area at 500 MHz |
|---|---|---|
| Systolic | 1.89 | 120K µm² |
| Vector | 0.69 | 67K µm² |

Systolic spatial array (TPU-like)

Parallel vector engines (NVDLA-like)

Fig. 3: Examples of two different spatial architectures generated by Gemmini. Both perform four multiply-accumulates per cycle though with different connectivities between multiply-and-accumulate units.

with spatially distributed processing elements (PEs), each of which performs dot products and accumulations. The spatial array reads data from a local, explicitly managed scratchpad of banked SRAMs, while it writes results to a local accumulator storage with a higher bitwidth than the inputs. Gemmini also supports other commonly-used DNN kernels, *e.g.*, pooling, non-linear activations (ReLU or ReLU6), and matrix-scalar multiplications, through a set of configurable, peripheral circuitry. Gemmini-generated accelerators can also be integrated with a RISC-V host CPU to program and configure accelerators.

We design Gemmini's spatial array with a two-level hierarchy to provide a flexible template for different microarchitecture structures, as demonstrated in Figure 2. The spatial array is first composed of *tiles*, where tiles are connected via explicit pipeline registers. Each of the individual tiles can be further broken down into an array of PEs, where PEs in the same tile are connected combinationally without pipeline registers. Each PE performs a single multiply-accumulate (MAC) operation every cycle, using either the weight- or the output-stationary dataflow. The tiles are composed of rectangular arrays of PEs, where PEs in the same tile are connected combinationally with no pipeline registers in between them. The spatial array, likewise, is composed of a rectangular array of tiles, but each tile *does* have pipeline registers between it and its neighbors. Every PE and every tile shares inputs and outputs only with its adjacent neighbors.

Figure 3 illustrates how Gemmini's two-level hierarchy provides the flexibility to support anything from fully-pipelined TPU-like architectures to NVDLA-like parallel vector engines where PEs are combinationally joined together to form MAC reduction trees, or any other design points in between these two extremes. We synthesized both designs with 256 PEs. We found that the TPU-like design achieves a 2.7x higher maximum frequency, due to its shorter MAC chains, but consumes 1.8x as much area as the NVDLA-like design, and 3.0x as much power, due to its pipeline registers. With Gemmini, designers can explore such footprint vs. scalability trade-offs across different accelerator designs.

### B. Programming Support

The Gemmini generator produces not just a hardware stack, but also a tuned software stack, boosting developers' productivity as they explore different hardware instantiations. Specifically, Gemmini provides a multi-level software flow to support different programming scenarios. At the *high level*, Gemmini contains a push-button software flow which reads DNN descriptions in the ONNX file format and generates software binaries that will run them, mapping as many kernels as possible onto the Gemmini-generated accelerator.

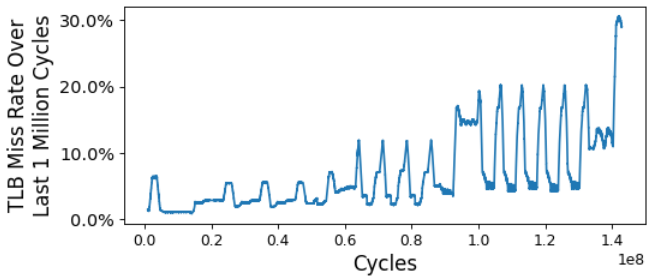



Fig. 4: TLB miss rate over a full ResNet50 inference, profiled on a Gemmini-generated accelerator.

Alternatively, at the *low level*, the generated accelerator can also be programmed through C/C++ APIs, with tuned functions for common DNN kernels. These functions must be tuned differently for different hardware instantiations in order to achieve high performance, based on scratchpad sizes and other parameters. Therefore, every time a new accelerator is produced, Gemmini also generates an accompanying header file containing various parameters, *e.g.* the dimensions of the spatial array, the dataflows supported, and the compute blocks that are included (such as pooling, im2col, or transposition blocks).

**Data Staging and Mapping:** At runtime, based on the dimensions of a layer's inputs, and the hardware parameters of the accelerator instantiation, Gemmini uses heuristics to maximize the amount of data moved into the scratchpad per iteration. Gemmini calculates loop tile sizes at runtime, and these tile sizes determine when and how much data is moved between DRAM, L2, and scratchpad during the execution of our tiled matrix multiplication, convolution, residual-addition, etc. kernels. If the programmer wishes, the low-level API also allows them to manually set tile-sizes for each kernel.

**Virtual Memory Support:** In addition to the programming interface, Gemmini also makes it easier to program accelerators by providing virtual memory support. This is useful for programmers who wish to avoid manual address translations as well as for researchers who wish to investigate virtual memory support in modern accelerators. Gemmini also enables users to co-design and profile their own virtual address translation system. For example, Figure 4 shows the miss rate of an example accelerator's local TLB profiled on Gemmini. As we can see, the miss rate occasionally climbs to 20-30% of recent requests, due to the tiled nature of DNN workloads, which is orders-of-magnitude greater than the TLB miss rates recorded in prior CPU non-DNN benchmarks [25]. Later, in Section V-A, we use Gemmini to co-design a virtual address translation system which achieves near-maximum end-to-end performance on accelerated DNN workloads, with only a few TLB entries in total.

### C. System Support

Gemmini allows architects to integrate RISC-V CPUs with Gemmini-generated accelerators in the Chipyard [26] framework. These can range from simple, in-order microcontrollers which are not expected to do much more than IO management, all the way up to out-of-order, high-performance, server-class CPUs that may be running multiple compute-intensive applications even as they are sending commands to the Gemmini-generated accelerator. SoCs can also be configured to host *multiple* host CPUs and Gemmini-generated accelerators, which can each operate on different tasks in parallel with each other. Figure 5 is one example of a dual-core system, where each CPU has its own Gemmini-generated accelerator. Additional SoC-level parameters include bus widths between accelerators and host CPUs, as well as the size, associativity and hierarchy of the caches in



Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on January 09,2026 at 02:34:19 UTC from IEEE Xplore. Restrictions apply.

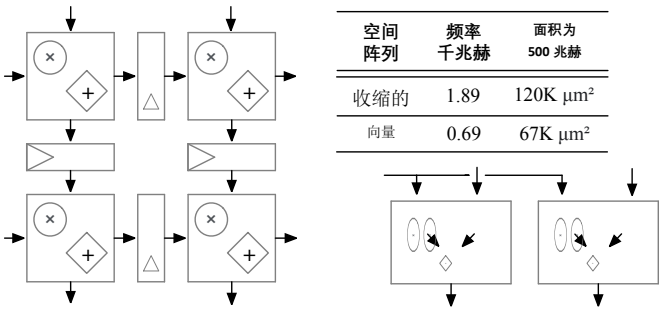



图3：Gemmini生成的两种不同空间架构示例。两者均实现每周期四次乘积累加运算，但乘积累加单元间的连接方式存在差异。

该系统采用空间分布式的处理单元（PEs），每个单元均能执行点积运算与累加运算。空间阵列从本地显式管理的SRAM缓存中读取数据，同时将运算结果写入位宽高于输入数据的本地累加器存储器。Gemmini还通过一组可配置的外围电路，支持其他常用DNN 内核，*例如*汇总运算、非线性激活函数（ReLU或ReLU6）以及矩阵-标量乘法。Gemmini生成的加速器还可与RISC-V主机CPU集成，实现加速器的编程与配置。

我们采用两级层次结构设计Gemmini的空间阵列，为不同微架构结构提供灵活模板，如图2所示。该空间阵列首先由*瓦片*构成，瓦片通过显式流水线寄存器连接。每个独立瓦片可进一步分解为处理器单元（PE）阵列，同一瓦片内的PE通过组合方式连接且无需流水线寄存器。每个PE在每个周期执行一次乘积累加（MAC）运算，可使用权重数据流或输出静态数据流。瓦片由矩形PE阵列构成，同一瓦片内的PE通过组合方式连接且中间无流水线寄存器。空间阵列同样由矩形瓦片阵列构成，但每个瓦片*确实*在其与相邻瓦片之间存在流水线寄存器。每个PE和每个瓦片仅与其相邻邻居共享输入和输出。

图3展示了Gemmini的两级架构如何灵活支持从完全流水线化的 TPU 架构到 NVDLA 类并行向量引擎（通过组合连接处理单元形成MAC缩减树）等各类设计，以及介于这两者之间的其他方案。我们使用256个处理单元对两种架构进行了仿真。研究发现， TPU 类架构由于MAC链路较短，最高频率提升了2.7倍，但其面积消耗是 NVDLA 类架构的1.8倍，功耗则因流水线寄存器的存在增加了3.0倍。通过Gemmini平台，设计者可以探索不同加速器架构在物理尺寸与扩展性之间的权衡关系。

### *B. 程序设计支持*

Gemmini生成器不仅能构建硬件堆栈，还能生成优化的软件堆栈，帮助开发者在探索不同硬件实例化时大幅提升生产力。具体而言，Gemmini提供多层级软件流程以支持多样化编程场景。在*高层*，Gemmini包含一键式软件流程，该流程可读取 ONNX 文件格式的 DNN 描述，并生成可运行的软件二进制文件，将尽可能多的内核映射到Gemmini生成的加速器上。

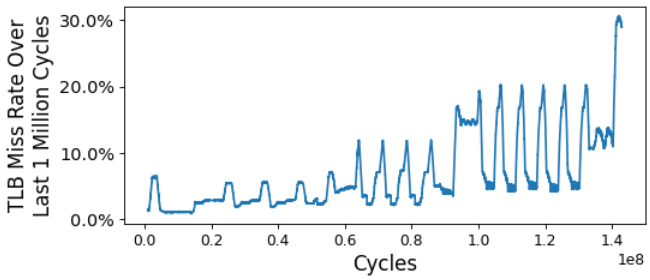



图4：在Gemmini生成的加速器上，完整ResNet50推理过程中的 TLB 漏检率分布。

在*低层级*，生成的加速器还可通过C/C++ API进行编程，针对常见 DNN 内核配置特定函数。为实现高性能，这些函数需根据不同硬件实例的存储器大小及其他参数进行差异化调优。因此，每当生成新加速器时，Gemmini都会自动生成包含各类参数的头文件，*例如*空间阵列尺寸、支持的数据流类型，以及所包含的计算模块（如池化、im2col或转置模块）。

**数据暂存与映射：**在运行时，Gemmini会根据图层输入维度和加速器实例的硬件参数，运用启发式算法来最大化每次迭代中移入暂存区的数据量。该系统会在运行时计算循环块大小，这些块大小将决定在执行矩阵乘法、卷积运算、残差加法等分块内核时，DRAM、L2缓存与暂存区之间数据的移动时机和传输量。程序员若需要，通过底层API还能手动为每个内核设置块大小。

**虚拟内存支持：**除了编程接口，Gemmini还通过提供虚拟内存支持简化了加速器编程。这对于希望避免手动地址转换的程序员以及希望研究现代加速器虚拟内存支持的研究人员都大有裨益。Gemmini还支持用户协同设计并分析自定义虚拟地址转换系统。例如，图4展示了在Gemmini上分析的某加速器本地 TLB 的未命中率。可以看出，由于 DNN 工作负载的分块特性，未命中率偶尔会攀升至近期请求的20%-30%，这一数值比先前CPU非 DNN 基准测试中记录的 TLB 未命中率高出数个数量级[25]。在后续的V-A章节中，我们将使用Gemmini协同设计一个虚拟地址转换系统，该系统在加速 DNN 工作负载上实现了接近最大端到端性能，且总 TLB 条目仅需少量。

### *C. 系统支持*

Gemmini架构支持架构师将RISC-V处理器与Gemmini自动生成的加速器集成到Chipyard[26]框架中。这些加速器的性能范围广泛，从仅需执行输入输出管理的简单按序执行微控制器，到能够同时运行多个计算密集型应用的高性能异步执行服务器级CPU——后者在向Gemmini加速器发送指令时仍可保持运算状态。系统级芯片（SoC）还可配置为同时承载*多个*主处理器和Gemmini加速器，各组件可并行处理不同任务。图5展示了一个双核系统实例，其中每个CPU都配备独立的Gemmini加速器。其他SoC级参数包括加速器与主处理器之间的总线宽度，以及缓存的容量、关联性与层级结构。



授权许可仅限以下机构使用： FUDAN 大学。下载于2026年1月9日02:34:19（UTC），来源：IEEE Xplore。使用受限制。

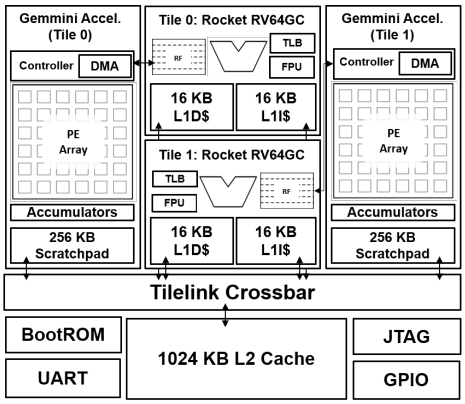



Fig. 5: Example dual-core SoC with a Gemmini accelerator attached to each CPU, as well as a shared L2 cache and standard peripherals.

the multicore, multicache memory system. Later, in Section V-B, we show how these parameters can be tuned, based on the computational characteristics of DNNs, to improve performance by over 8%.

RISC-V-based full SoC integration also enables deep software-stack support, such that Gemmini-generated accelerators can easily be evaluated running the full software stack up to and including the operating system itself. This enables early exploration of accelerated workloads in a realistic environment where context switches, page table evictions, and other unexpected events can happen at any time. These unexpected events can uncover bugs and inefficiencies that a "baremetal" environment would not bring to the surface. For example, our experience of running Linux while offloading DNN kernels to a Gemmini-generated accelerator uncovered a non-deterministic deadlock that would only occur if context switches happened at very particular, inopportune times. Running on a full software stack with an OS also uncovered certain bugs where Gemmini read from certain regions of physical memory without the proper permissions. On a "baremetal" environment, these violations were silently ignored.

## IV. Gemmini Evaluation

This section discusses our evaluation methodology and evaluation results of Gemmini-generated accelerators compared to both CPUs and state-of-the-art, commercial accelerators.

### A. Evaluation Methodology

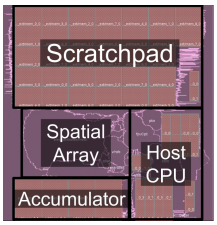
We evaluate the end-to-end performance of Gemmini-generated accelerators using the FireSim FPGA-accelerated simulation platform [27]. We evaluate five popular DNNs: ResNet50, AlexNet, SqueezeNet v1.1, MobileNetV2, and BERT. All DNNs are evaluated with a full Linux environment on a complete cycle-exact simulated SoC. We synthesize designs using Cadence Genus with the Intel 22nm FFL process technology and place-and-route them using Cadence Innovus. Our layout and area breakdown, described in Figure 6, show that the SRAMs alone consume 67.1% of the accelerator's total area. The spatial array itself only consumes 11.3%, while the host CPU consumed a higher 16.6% of area.

### B. Performance Results

We evaluated the performance of several Gemmini configurations, with different host CPUs and different "optional" compute blocks, to determine how the accelerator and host CPU configuration may interact to impact end-to-end performance. In particular, we evaluated two host CPUs: a low-power in-order Rocket core, and a high-performance out-of-order BOOM core. We used two different Gemmini configurations: one *without* an optional im2col block, and the other *with* an im2col block which allowed the accelerator to perform im2col on-the-fly, relieving the host CPU of that burden.

| Component size | Area ($\mu m^2$) | % of System Area |
|---|---|---|
| Spatial Array (16x16) | 116K | 11.3% |
| Scratchpad (256 KB) | 544K | 52.9% |
| Accumulator (64 KB) | 146K | 14.2% |
| CPU (Rocket, 1 core) | 171K | 16.6% |
| Total | 1,029K | 100.0% |

(a) Area breakdown.




(b) Layout.

Fig. 6: Area breakdown and layout of accelerator with host CPU.

As illustrated in Figure 7, when the accelerator is built without an on-the-fly im2col unit, its performance depends heavily on the host-CPU which becomes responsible for performing im2col during CNN inference. A larger out-of-order BOOM host CPU increases performance by 2.0x across all CNNs. The less complex the DNN accelerator is, the more the computational burden is shifted onto the CPU, giving the host CPU a larger impact on end-to-end performance.

However, when the accelerator is equipped with an on-the-fly im2col unit, the choice of host CPU is far less important, because the CPU's computational burden is shifted further onto the accelerator. Adding a small amount of complexity to the accelerator allows us to reduce the area and complexity of the host CPU to a simple in-order core while preserving performance. Gemmini enables hardware designers to easily make these performance-efficiency tradeoffs.

With the on-the-fly im2col unit and a simple in-order Rocket CPU, Gemmini achieves 22.8 frames per second (FPS) for ResNet50 inference when running at 1 GHz, which is a 2,670x speedup over the in-order Rocket CPU and an 1,130x speedup over the out-of-order BOOM CPU. The accelerator also achieves 79.3 FPS on AlexNet. Some DNN models such as MobileNet are not efficiently mapped to spatial accelerators due to the low data reuse within the depthwise convolution layers. Therefore, Gemmini demonstrates only a 127x speedup compared to the Rocket host CPU on MobileNetV2, reaching 18.7 FPS at 1GHz. On SqueezeNet, which was designed to be run efficiently on modern CPUs while conserving memory bandwidth, Gemmini still demonstrates a 1,760x speedup over the Rocket host CPU. Our results are comparable to other accelerators, such as NVDLA, when running with the same number of PEs as the configuration in Figure 6a. When running language models such as BERT, Gemmini achieves a 144x improvement over the Rocket CPU.

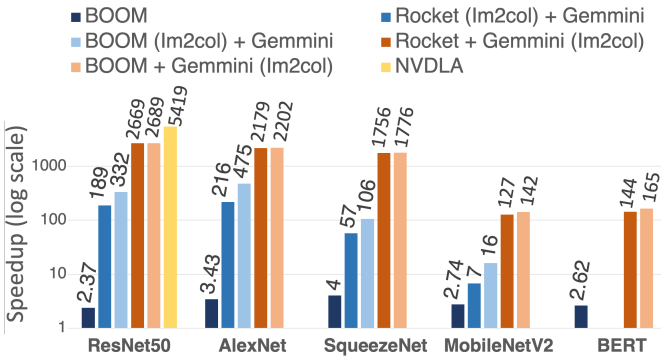



Fig. 7: Speedup compared to an in-order CPU baseline. For CNNs, im2col was performed on either the CPU, or on the accelerator.



Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on January 09,2026 at 02:34:19 UTC from IEEE Xplore. Restrictions apply.

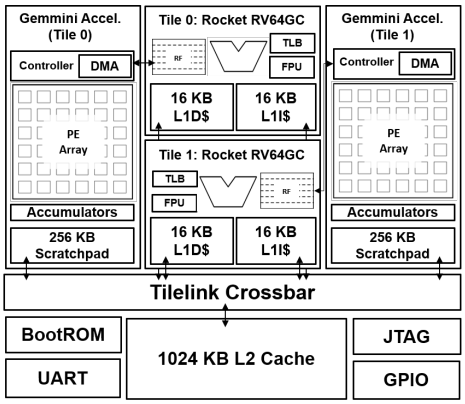



图5：示例双核SoC，每个CPU均配备Gemmini加速器，同时共享二级缓存（L2 cache）及标准外设。

多核、多缓存内存系统。随后，在第V-B节中，我们将基于深度神经网络（DNN）的计算特性，展示如何通过调整这些参数来提升性能，最高可提高8%以上。

基于RISC-V架构的全SoC集成还实现了深度软件栈支持，使得Gemmini生成的加速器能够轻松运行完整的软件栈（包括操作系统本身）进行评估。这使得在真实环境中（上下文切换、页表置换等意外事件可能随时发生）对加速工作负载进行早期探索成为可能。这些意外事件能揭示“裸机”环境无法暴露的漏洞和效率问题。例如，我们在将 DNN 内核卸载至Gemmini生成的加速器运行Linux时，发现了一个非确定性死锁问题——该问题仅在上下文切换发生在特定不恰当时刻时才会出现。在运行完整操作系统软件栈时，我们也发现了Gemmini在未获得适当权限的情况下读取物理内存特定区域的漏洞。而在“裸机”环境中，这些违规操作会被系统自动忽略。

## IV.gemmini评估

本节阐述了我们对Gemmini生成的加速器与CPU及当前最先进的商用加速器进行评估的方法学及结果。

### *A. 评价方法*

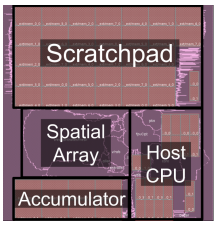
我们采用FireSim FPGA 加速仿真平台[27]对Gemmini生成的加速器进行端到端性能评估。测试了五种主流深度神经网络模型 ：ResNet50、AlexNet、SqueezeNet v1.1、MobileNetV2 和 BERT。所有模型均在完整周期精确模拟的SoC上，通过完整的Linux环境进行测试。我们使用Cadence Genus设计工具基于英特尔22纳米 FFL 工艺技术进行布局，并通过Cadence Innovus完成布线。如图6所示的布局与面积占比分析表明，SRAM模块单独就占用了加速器总面积的67.1%，空间阵列模块仅占11.3%，而主机CPU则占据了更高的16.6%面积。

### *B. 性能结果*

我们评估了多种Gemmini配置的性能表现，这些配置包含不同主机CPU和不同“可选”计算模块，旨在探究加速器与主机CPU配置如何相互作用影响端到端性能。具体而言，我们测试了两种主机CPU：一种是低功耗的顺序执行Rocket核心，另一种是高性能的乱序执行BOOM核心。同时采用两种Gemmini配置方案：一种是*不包含*可选im2col模块的配置，另一种是

| 组件尺寸 | 面积（$\mu m2$） | 百分比 体系区域 |
|---|---|---|
| 空间阵列(16x16) | 116K | 11.3% |
| Scratchpad (256 KB) | 544K | 52.9% |
| 蓄能器（64 KB） | 146K | 14.2% |
| CPU（Rocket，1核） | 171K | 16.6% |
| 共计 | 1,029K | 100.0% |

(a) 面积细目。




(b)布局。图6：加速器与主机CPU的区域划分及布局。

其他*采用*im2col模块的处理器，该模块使加速器能够实时执行im2col操作，从而减轻主机CPU的负担。

如图7所示，当加速器未配备即时im2col单元时，其性能高度依赖于主机CPU，后者需在CNN推理阶段执行im2col操作。采用更大规模的乱序BOOM主机CPU可使所有CNN的性能提升2.0倍。 DNN 加速器结构越简单，计算负担就越会转移到CPU上，从而对主机CPU的端到端性能产生更大影响。

然而，当加速器配备实时im2col单元时，主机CPU的选择就变得不那么重要了，因为CPU的计算负担会进一步转移到加速器上。通过在加速器中增加少量复杂性，我们可以在保持性能的同时，将主机CPU的面积和复杂度降低到一个简单的顺序核心。Gemmini使硬件设计师能够轻松实现这些性能与效率之间的权衡。

Gemmini搭载即时执行的im2col单元和简单的顺序式Rocket CPU，在1GHz频率下运行ResNet50推理时可达到22.8帧/秒（FPS），较顺序式Rocket CPU提速2670倍，较乱序式BOOM CPU提速1130倍。该加速器在AlexNet上同样实现79.3 FPS。由于深度卷积层内部数据复用率较低，MobileNet等部分 DNN 模型难以高效映射到空间加速器，因此Gemmini在MobileNetV2上仅较Rocket主机CPU提速127倍，1GHz频率下达到18.7 FPS。针对SqueezeNet（该模型专为现代CPU高效运行且节省内存带宽设计），Gemmini仍较Rocket主机CPU提速1760倍。当使用与图6a配置相同数量的处理单元（PE）时，我们的结果与其他加速器（如 NVDLA）相当。在运行BERT等语言模型时，Gemmini较Rocket CPU实现144倍性能提升。

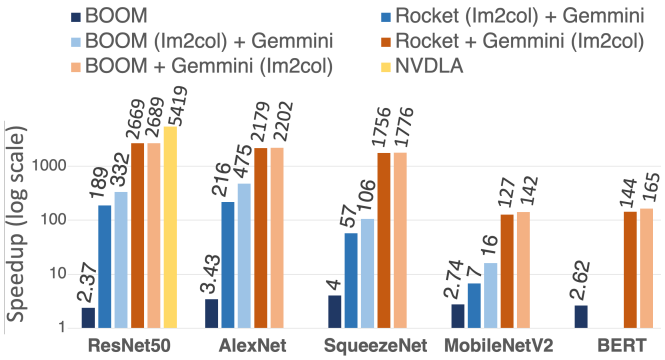
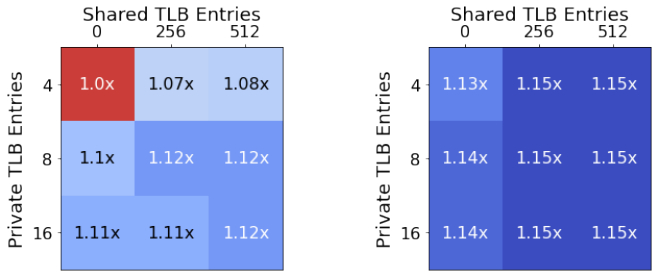



图7：与顺序CPU基准相比的加速比。对于CNN，im2col操作在CPU或加速器上执行。



授权许可仅限以下机构使用： FUDAN 大学。下载于2026年1月9日02:34:19（UTC），来源：IEEE Xplore。使用受限制。




(a) Without filter registers.

(b) With filter registers.

Fig. 8: Normalized performance of ResNet50 inference on Gemmini-generated accelerator with different private and shared TLB sizes.

## V. Gemmini Case Studies

This section demonstrates how Gemmini enables full system co-design with two case studies. We use Gemmini to design a novel virtual address translation scheme, and to find the optimal SoC-level resource partition scheme of a multi-core, multi-accelerator system.

### A. Virtual Address Translation

With an RTL-level implementation that supports virtual memory, users can co-design their own virtual address translation schemes based on their accelerator and SoC configuration. Prior works in virtual address translation for DNN accelerators have proposed very different translation schemes, from NeuMMU [28], which calls for a highly parallel address-translation system with 128 page-table walkers (PTWs), to Cong et al. [29], who recommend a more modest two-level TLB hierarchy, with the host CPU's default PTW co-opted to serve requests by the accelerator. This lack of convergence in the prior literature motivates a platform that allows co-design and design-space exploration of the accelerator SoC together with its virtual address translation system, for both hardware designers and researchers. Fortunately, with Gemmini, we can iterate over a variety of address translation schemes as we tune the accelerator and SoC.

To demonstrate, we configure Gemmini to produce a two-level TLB cache, with one private TLB for the accelerator, and one larger shared TLB at the L2 cache that the private TLB falls back on when it misses. Our design includes only one PTW, shared by both the CPU and the accelerator, which is suitable for low-power devices. We configure the accelerator for low-power edge devices, with a 16-by-16 systolic mesh and a 256 KB scratchpad. As shown in Figure 8a, we iterate over a variety of TLB sizes to find the design that best balances TLB overhead and overall performance, including over a design point where the shared L2 TLB has zero entries.

Figure 8a demonstrates that the private accelerator TLB has a far greater impact on end-to-end performance than the much larger shared L2 TLB. Increasing the private TLB size from just four to 16 improves performance by up to 11%. However, adding even 512 entries to the L2 TLB never improves performance by more than 8%. This is because our workloads exhibit high page locality; even with tiled workloads, our private TLB's hit rate remained above 84%, even with the smallest TLB sizes we evaluated. In fact, we found that 87% of consecutive *read* TLB requests, and 83% of consecutive *write* TLB requests, were made to the same page number, demonstrating high page locality. However, because reads and writes were overlapped, read and write operations could evict each other's recent TLB entries.

Although tuning TLB sizes improves hit rates, our private TLB hit latency in the tests shown in Figure 8a was still several cycles long. Fortunately, using the Gemmini platform, we were able to implement a simple optimization: a single register that caches the last TLB hit for read operations, and another register that caches TLB hits for write operations. These two registers allow the DMA to "skip" the TLB request if two consecutive requests are made to the same virtual page number, and help reduce the possibility of read-write contention over the TLB. These "filter registers" reduce the TLB hit latency to 0 cycles for consecutive accesses to the same page. As Figure 8b shows, this low-cost optimization significantly improves our end-to-end performance, especially for small private TLB sizes. Due to our high TLB hit rate and low TLB hit penalty, we found that a very small 4-entry private TLB equipped with filter registers, but without an expensive shared L2 TLB, achieved only 2% less than the maximum performance recorded. With such a configuration, the private TLB hit rate (including hits on the filter registers) reached 90% and further increases to either TLB's size improved performance by less than 2%, even if hundreds of new TLB entries were added.

Using Gemmini, we have demonstrated that a modest virtual address translation system, with very small private TLBs, a single page-table-walker, and two low-cost filter registers for the TLB, can achieve near maximum performance for low-power edge devices. Gemmini is designed to enable such co-design of the SoC and its various components, such as its virtual address translation system.

### B. System-Level Resource Partition

Gemmini also enables application-system co-design for real-world DNN workloads. To demonstrate, we present a case study describing a system-level design decision: memory partitioning based on application characteristics. We investigate memory partitioning strategies in both single-core and multi-core SoCs.

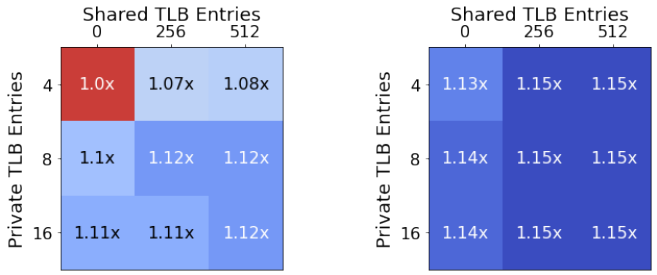
Real-world DNN applications, such as CNN inference, have diverse layer types which have different computational requirements and which contend for resources on an SoC in different ways. For example, ResNet50 includes convolutions, matrix multiplications, and residual additions, which all exhibit quite different computational patterns. Convolutions have high arithmetic intensity; matrix multiplications have less; and residual additions have almost no data re-use at all. Additionally, unlike the other two types of layers, residual additions benefit most if layer outputs can be stored inside the cache hierarchy for a long time, rather than being evicted by intermediate layers, before finally being consumed several layers later. These different layer characteristics suggest different ideal SoC configurations. To run with optimal performance over an entire DNN, a hardware designer must balance all these constraints.

To demonstrate, we run ResNet50 inference on six different SoC configurations. These are the three different configurations described in Figure 9a, repeated for both single- and dual-core SoCs (as in Figure 5), where each CPU core has its own Gemmini-generated accelerator. The dual-core SoCs run two ResNet50 workloads in parallel, while the single-core SoCs run just one. The base design point has a 256 KB scratchpad, and a 256 KB accumulator per core, as well as a 1 MB shared L2 cache. The scratchpad and accumulator memories are private to the accelerators, but the L2 cache is shared by all CPUs and accelerators on the SoC. We presume that we have 1 MB of extra SRAM that we can allocate to our memory system, but we need to decide whether to allocate these SRAMs to the accelerators' private memory, or to the L2 caches.

As shown in Figures 9b and 9c, convolutional layers benefit from a larger, explicitly managed scratchpad, due to their very high arithmetic intensity. Convolutional kernels exhibit a 10% speedup with one core, and an 8% speedup in the dual-core case, when the scratchpad and accumulator memory is doubled by the addition of our 1 MB worth of SRAMs. The matmul layers, on the other



Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on January 09,2026 at 02:34:19 UTC from IEEE Xplore. Restrictions apply.




(a) 无滤波器寄存器。 (b) 带滤波器寄存器

图8：ResNet50推理在Gemmini生成的加速器上不同私有和共享 TLB 规模下的标准化性能表现。

## V. GEMMINI 案例研究

本节通过两个案例研究，演示Gemmini如何实现完整的系统协同设计。我们利用Gemmini设计了一种新型虚拟地址转换方案，并为多核多加速器系统找到了最优的SoC级资源分配方案。

### *A. 虚拟地址转换*

通过支持虚拟内存的RTL级实现，用户可以根据加速器和SoC配置共同设计虚拟地址转换方案。此前针对 DNN 加速器的虚拟地址转换研究提出了多种不同方案：从需要128个页表游走器（PTWs）的高度并行地址转换系统NeuMMU[28]，到Cong等人[29]提出的较为保守的两级 TLB 层次结构——该方案将主机CPU的默认 PTW 资源用于加速器请求处理。由于现有文献缺乏统一标准，我们开发了Gemmini平台，使硬件设计师和研究人员能够协同设计加速器SoC及其虚拟地址转换系统，探索设计空间。幸运的是，通过Gemmini平台，我们可以在调整加速器和SoC配置时，对多种地址转换方案进行迭代优化。

为验证方案，我们配置Gemmini生成两级 TLB 缓存架构：加速器配备专用 TLB，L2缓存则设有容量更大的共享 TLB，当专用 TLB 未命中时可自动切换使用。该设计仅包含一个 PTW，由CPU和加速器共享，特别适合低功耗设备。针对低功耗边缘设备，我们配置了16×16的收缩网格架构和256KB临时存储器的加速器。如图8a所示，我们通过迭代不同 TLB 尺寸，最终确定了TLB 开销与整体性能的最佳平衡方案，其中包括共享L2 TLB 完全空置的设计点。

图8a显示，私有加速器 TLB 对端到端性能的影响远大于规模更大的共享L2 TLB 。将私有 TLB 规模从仅四个增加到16号方案可将性能提升高达11%。但即便在L2 TLB 中增加512个条目，性能提升也从未超过8%。这是因为我们的工作负载具有高度页面局部性——即使在采用分片技术的情况下，私有 TLB 的命中率仍保持在84%以上，即便在我们评估的最小 TLB 尺寸下也是如此。实际上，我们发现87%的连续*读取* TLB 请求和83%的连续*写入* TLB 请求都指向同一页号，这充分证明了页面局部性。但由于读写操作存在重叠，读写操作可能会互相驱逐对方的最新 TLB 条目。虽然调整 TLB 尺寸能提升命中率，但在图8a所示测试中，私有 TLB 的命中延迟仍长达数个周期。幸运的是，通过Gemmini平台，我们实现了简单的优化方案：采用单个寄存器缓存最近 TLB 命中数据。读取操作使用一个寄存器，写入操作则使用另一个缓存 TLB 命中结果的寄存器。这两个寄存器能让DMA在连续两次访问同一虚拟页号时“跳过” TLB 请求，从而降低 TLB 读写冲突的可能性。这些“过滤寄存器”将连续访问同一页面时的 TLB 命中延迟降至0周期。如图8b所示，这种低成本优化显著提升了端到端性能，尤其适用于小型私有 TLB 。得益于高 TLB 命中率和低 TLB 命中代价，我们发现配备过滤寄存器但未使用昂贵共享L2 TLB 的4条私有 TLB，其性能仅比最高记录值低2%。在此配置下，私有 TLB 命中率（包括过滤寄存器命中）达到90%，且 TLB 规模越大，性能提升幅度不超过2%——即便新增数百条 TLB 条目也是如此。

通过Gemmini，我们证明了一个适度的虚拟地址转换系统——配备极小的专用TLB、单页表跟踪器以及两个低成本 TLB 过滤器寄存器——能够为低功耗边缘设备实现接近最大性能。Gemmini旨在支持系统级芯片（SoC）与其各类组件（如虚拟地址转换系统）的协同设计。

### *B. 系统级资源划分*

Gemmini还支持针对真实 DNN 工作负载的应用系统协同设计。为验证该功能，我们通过一个案例研究展示系统级设计决策：基于应用特性的内存分区。我们研究了单核和多核SoC中的内存分区策略。

在实际 DNN 应用中，诸如卷积神经网络推理等场景会遇到多种不同类型的神经网络层，这些层不仅计算需求各异，还会以不同方式争夺片上系统的资源。以ResNet50为例，它包含卷积运算、矩阵乘法和残差加法这三类运算，它们的计算模式差异显著：卷积运算具有高强度的算术运算需求，矩阵乘法运算的算力需求较低，而残差加法则几乎不涉及数据复用。此外，与其他两类层不同，残差加法的性能优势主要体现在其输出数据能否长期保留在缓存层级中——而非被中间层逐层清除后才被后续层调用。这些差异化的层特性决定了理想的片上系统配置方案也各不相同。为了在整个 DNN 中实现最佳性能，硬件设计师必须平衡所有这些技术约束条件。

为验证性能差异，我们在六种不同SoC配置上运行ResNet50推理测试。这些配置包括图9a所示的三种不同架构，分别针对单核和双核SoC（如图5所示），每个CPU核心都配备Gemmini生成的专用加速器。双核SoC并行运行两个ResNet50工作负载，而单核SoC仅运行一个。基础设计配置包含每个核心256KB的暂存器和累加器，以及1MB共享的L2缓存。暂存器和累加器内存为加速器专属，而L2缓存则是SoC所有CPU和加速器共享。我们假设系统可额外分配1MB SRAM 用于内存系统，但需要决定将这些SRAM分配给加速器的专用内存，还是分配给L2缓存。

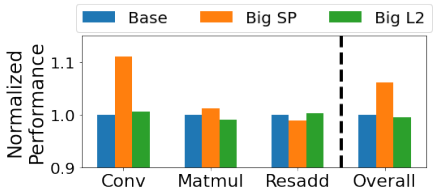
如图9b和9c所示，由于卷积层具有极高的运算强度，其性能可从更大且显式管理的临时存储器中获益。当通过增加1MB的SRAM使临时存储器和累加器内存翻倍时，单核情况下卷积核的加速比可达10%，双核情况下则为8%。而乘法层则



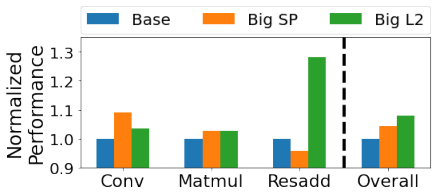
授权许可仅限以下机构使用： FUDAN 大学。下载于2026年1月9日02:34:19（UTC），来源：IEEE Xplore。使用受限制。

| Config Name | Scratchpad (per core) | Accumulator (per core) | L2 Cache |
|---|---|---|---|
| Base | 256 KB | 256 KB | 1 MB |
| BigSP | 512 KB | 512 KB | 1 MB |
| BigL2 | 256 KB | 256 KB | 2 MB |

(a) Resource contention SoC configurations




(b) Performance of single-core SoCs.




(c) Performance of dual-core SoCs.

Fig. 9: Performance of the various SoC configurations in the case study, normalized to the performance of the Base configuration.

hand, achieve only a 1% and 3% speedup when the scratchpad is enlarged in the single-core and dual-core cases respectively, due to their lower arithmetic intensity. Residual additions, which have virtually no data re-use and are memory-bound operations, exhibit no speedup when increasing the scratchpad memory size. Instead, they exhibit a minor 1%-4% slowdown, due to increased cache thrashing. In the single-core case, the increased convolutional and matrix multiplication performance is enough to make the design point with increased scratchpad memory, rather than increased L2 memory, the most performant design point.

However, Figure 9c shows that when we run dual-process applications that compete for the same shared L2 cache, allocating the extra 1 MB of memory to the shared L2 cache improves overall performance more than adding that memory to the accelerators' scratchpad and accumulator memories. Increasing the scratchpad size still improves convolutional performance more than increasing the L2 size, but this improvement in performance is more than negated by the 22% speedup of residual additions that the dual-core BigL2 design point enjoys. This is because each core's residual addition evicts the input layer that the other one is expecting from the shared L2 cache, increasing the latency of memory-bound residual addition layers. The dual-core BigL2 configuration, which increases the shared cache sizes, alleviates this contention, reducing the L2 miss rate by 7.1% over the full ResNet50 run, and increasing overall performance by 8.0%. The BigSP configuration, on the other hand, improves overall performance by only 4.2% in the dual-core case.

With Gemmini, we have demonstrated how the memory partitioning strategy, a key component of system-level design, can be decided based upon application characteristics, such as the composition of layer types and the number of simultaneous running processes.

## VI. Conclusion

We present Gemmini, a full-stack, open-source generator of DNN accelerators that enables systematic evaluations of DNN accelerator architectures. Gemmini leverages a flexible architectural template to capture different flavors of DNN accelerator architectures. In addition, Gemmini provides a push-button, high-level software flow to boost programmers' productivity. Finally, Gemmini generates a full SoC that runs real-world software stacks including operating systems, to enable system architects to evaluate system-level impacts. Our evaluation shows that Gemmini-generated accelerators demonstrate high performance efficiency, and our case studies show how accelerator designers and system architects can use Gemmini to co-design and evaluate system-level behavior in emerging applications.

## VII. Acknowledgements

This research was, in part, funded by the U.S. Government under the DARPA RTML program (contract FA8650-20-2-7006). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

## References

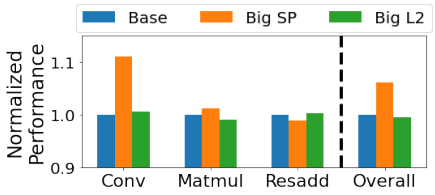
[1] T. Moreau *et al.*, "VTA: An Open Hardware-Software Stack for Deep Learning," *CoRR*, 2018.
[2] R. Venkatesan *et al.*, "MAGNet: A Modular Accelerator Generator for Neural Networks," in *ICCAD*, 2019.
[3] J. Cong *et al.*, "PolySA: polyhedral-based systolic array auto-compilation," in *ICCAD*, 2018.
[4] X. Zhang *et al.*, "DNNBuilder: An Automated Tool for Building High-performance DNN Hardware Accelerators for FPGAs," in *ICCAD*, 2018.
[5] Xuechao Wei *et al.*, "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," in *DAC*, 2017.
[6] Y. Wang *et al.*, "Deepburning: Automatic generation of fpga-based learning accelerators for the neural network family," in *DAC*, 2016.
[7] H. Ye *et al.*, "HybridDNN: A framework for high-performance hybrid dnn accelerator design and implementation," in *DAC*, 2020.
[8] K. Hazelwood *et al.*, "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective," in *HPCA*, 2018.
[9] C.-J. Wu *et al.*, "Machine Learning at Facebook: Understanding Inference at the Edge," in *HPCA*, 2019.
[10] D. Richins *et al.*, "Missing the Forest for the Trees: End-to-End AI Application Performance in Edge Data Centers," in *HPCA*, 2020.
[11] F. Sijstermans, "The NVIDIA Deep Learning Accelerator," in *Hot Chips*, 2018.
[12] Y. Chen *et al.*, "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices," *JETCAS*, 2019.
[13] Z. Du *et al.*, "ShiDianNao: Shifting vision processing closer to the sensor," in *ISCA*, 2015.
[14] S. Venkataramani *et al.*, "ScaleDeep: A Scalable Compute Architecture for Learning and Evaluating Deep Networks," in *ISCA*, 2017.
[15] X. Yang *et al.*, "Interstellar: Using Halide's scheduling language to analyze DNN accelerators," in *ASPLOS*, 2020.
[16] H. Kwon *et al.*, "MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Programmable Interconnects," in *ASPLOS*, 2018.
[17] J. Fowers *et al.*, "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *ISCA*, 2018.
[18] E. Qin *et al.*, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *HPCA*, 2020.
[19] X. He *et al.*, "Sparse-TPU: Adapting systolic arrays for sparse matrices," in *ICS*, 2020.
[20] P. Dai *et al.*, "SparseTrain: Exploiting dataflow sparsity for efficient convolutional neural networks training," in *DAC*, 2020.
[21] L. Song *et al.*, "PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning," in *HPCA*, 2017.
[22] H. Kim *et al.*, "Algorithm/hardware co-design for in-memory neural network computing with minimal peripheral circuit overhead," in *DAC*, 2020.
[23] H. Sharma *et al.*, "From High-level Deep Neural Models to FPGAs," in *MICRO*, 2016.
[24] G. Henry *et al.*, "High-Performance Deep-Learning Coprocessor Integrated into x86 SoC with Server-Class CPUs Industrial Product," in *ISCA*, 2020.
[25] D. Lustig *et al.*, "Tlb improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level tlbs," *TACO*, 2013.
[26] A. Amid *et al.*, "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs," *IEEE Micro*, 2020.
[27] S. Karandikar *et al.*, "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud," in *ISCA*, 2018.
[28] B. Hyun *et al.*, "NeuMMU: Architectural Support for Efficient Address Translations in Neural Processing Units," in *ASPLOS*, 2020.
[29] Y. Hao *et al.*, "Supporting Address Translation for Accelerator-Centric Architectures," in *HPCA*, 2017.



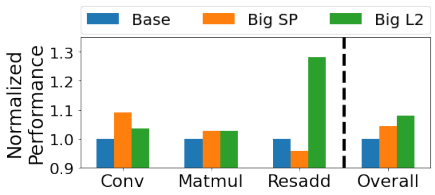
Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on January 09,2026 at 02:34:19 UTC from IEEE Xplore. Restrictions apply.

| 配置<br>名称 | 刮板<br>（每核心） | 蓄能器<br>（每核心） | L2<br>藏物处 |
|---|---|---|---|
| 基础 | 256 KB | 256 KB | 1 兆字节 |
| 大SP | 512 KB | 512 KB | 1 兆字节 |
| 大L2 | 256 KB | 256 KB | 2 兆字节 |

(a) 资源争用SoC配置




(b) 单核SoC性能




(c) 双核SoC性能

图9：案例研究中各类SoC配置的性能表现，以基础配置的性能为基准进行标准化。

在单核和双核场景中，当扩展临时存储区时，由于运算强度较低，仅能分别实现1%和3%的加速效果。而几乎不涉及数据复用且受内存限制的残差加法运算，其加速效果在扩大临时存储区时完全消失。相反，这类运算会因缓存抖动加剧而出现1%-4%的轻微减速。在单核场景中，卷积运算和矩阵乘法性能的提升足以使采用扩展临时存储区（而非增加二级缓存）的设计方案成为性能最优的方案。

然而，图9c显示，当我们运行需要争夺同一共享L2缓存的双进程应用时，将额外的1MB内存分配给共享L2缓存比将其分配给加速器的临时存储器和累加器内存更能提升整体性能。虽然增加临时存储器容量仍比扩大L2缓存容量更能提升卷积运算性能，但这种性能提升被双核BigL2架构在残差加法运算中获得的22%加速优势完全抵消了。这是因为每个核心的残差加法操作会挤占其他核心原本期望从共享L2缓存获取的输入层，导致内存受限的残差加法层延迟增加。通过增大共享缓存容量的双核BigL2配置有效缓解了这种竞争，使L2缓存未命中率较完整ResNet50运行降低了7.1%，整体性能提升了8.0%。相比之下，BigSP配置在双核场景下的整体性能提升仅为4.2%。

通过Gemmini系统，我们验证了内存分区策略——作为系统级设计的关键要素——如何根据应用特性（如层类型构成及并发进程数量）进行动态配置。

## 第六章.结论

我们推出Gemmini——一款全栈开源 DNN 加速器生成器，可系统评估各类 DNN 加速器架构。该工具采用灵活的架构模板，能捕捉不同类型的 DNN 加速器设计。此外，Gemmini提供一键式高级软件流程，显著提升开发效率。最终生成的完整SoC可运行包含操作系统在内的真实软件栈，助力系统架构师评估系统级影响。评估结果表明，Gemmini生成的加速器具有卓越的性能效率，案例研究则展示了加速器设计师与系统架构师如何利用Gemmini协同设计并评估新兴应用的系统级行为。

## 第七章.致谢

本研究部分由美国政府通过DARPA RTML 计划（合同号FA 8650-20-2-7006）资助。本文档所载观点与结论仅代表作者立场，不应被解读为美国政府（无论是明示或暗示）的官方政策。

## 参考文献

[1] T. Moreau等人，“VTA：一个用于深度学习的开放硬件-软件堆栈”，*CoRR*，2018年。
[2] R. Venkatesan等人，“MAGNet：一种用于神经网络的模块化加速器生成器”，收录于*ICCAD*，2019年。
[3] J. Cong 等人，“PolySA：基于多面体的收缩阵列自动编译”，载于*ICCAD*，2018年。
[4] X.张 等人，“DNNBuilder：一种用于构建高性能 DNN 硬件加速器的自动化工具，适用于FPGAs”，载于*ICCAD*，2018年。
[5] 薛超伟 等人，“基于FPGA的高吞吐量CNN推理自动化收缩阵列架构综合”，收录于*DAC*，2017年。
[6] Y. Wang等人，“Deepburning：基于FPGA的神经网络学习加速器的自动生成”，收录于*DAC*，2016年。
[7] H. Ye等人，“HybridDNN：高性能混合dnn加速器设计与实现框架”，收录于*DAC*，2020年。
[8] K. Hazelwood等人，“Facebook应用机器学习：数据中心基础设施视角”，见*HPCA*，2018年。
[9] C.-J.Wu等人，“Facebook机器学习:理解边缘推理”，*HPCA*，2019年。
[10] D.Richins等人，“只见树木不见森林:边缘数据中心的端到端AI应用性能”，见*HPCA*，2020年。
[11] F. 西斯特曼斯，《 NVIDIA 深度学习加速器》，载于*热芯片*，2018年。
[12] Y. Chen等人，“Eyeriss v2:一种用于移动设备上新兴深度神经网络的灵活加速器”，*JETCAS*，2019年。
[13] Z.杜等人，“视觉神经：将视觉处理更贴近传感器”，载于*ISCA*，2015年。
[14] S. Venkataramani等人，“ScaleDeep：一种用于学习和评估深度网络的可扩展计算架构”，见*ISCA*，2017年。
[15] X.杨 等人，“星际：使用Halide调度语言分析 DNN 加速器”，载于*ASPLOS*，2020年。
[16] H.Kwon等人，“MAERI：通过可编程互连实现 DNN 加速器上的灵活数据流映射”，收录于*ASPLOS*，2018年。
[17] J. Fowers 等人，“一种可配置的云规模 DNN 处理器用于实时人工智能”，载于*ISCA*，2018年。
[18] E.秦 等人，“Sigma：一种用于深度神经网络训练的稀疏不规则宝石加速器，具有灵活的互连结构”，载于*HPCA*，2020年。
[19] X. He等人，“稀疏 TPU：针对稀疏矩阵的收缩阵列自适应”，收录于*ICS*，2020年。
[20] P. Dai等人，“稀疏训练：利用数据流稀疏性实现高效卷积神经网络训练”，收录于*DAC* 2020年会议论文集。
[21] L. Song 等人，“PipeLayer：一种基于流水线ReRAM的深度学习加速器”，收录于*HPCA*，2017年。
[22] H. Kim等人，“内存神经网络计算的算法/硬件协同设计，以最小化外围电路开销”，收录于*DAC*，2020年。
[23] H. Sharma等人，“从高级深度神经模型到FPGA”，见*micro*，2016年。
[24] G. Henry等人，“高性能深度学习协处理器集成到x86 SoC与服务器级CPU工业产品中”，发表于*ISCA*，2020年。
[25] D. Lustig等人，“芯片多处理器的TLB改进：核心间协作预取器与共享最后一级TLB”，*TACO*，2013年。
[26] A. Amid等人，“Chipyard:集成设计、仿真和实现定制SoC的框架”，*IEEE Micro*，2020年。
[27] S.Karandikar等人，“FireSim:公共云中 FPGA 加速的精确扩展系统模拟”，载于*ISCA*，2018年。
[28] B. Hyun等人，“NeuMMU:神经处理单元中高效地址转换的架构支持”，见*ASPLOS*，2020年。
[29] Y. Hao等人，“支持加速器中心架构的地址转换”，收录于*HPCA*，2017年。



授权许可仅限以下机构使用： FUDAN 大学。下载于2026年1月9日02:34:19（UTC），来源：IEEE Xplore。使用受限制。