Enabling Full-Stack DNN Accelerator Design and Evaluation on Synthesizable Hardware

Hasan Genc

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-161
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-161.html

August 8, 2024

Enabling Full-Stack DNN Accelerator Design and Evaluation on Synthesizable Hardware

By

Hasan Nazim Genc

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Krste Asanovic, Chair
Yakun Sophia Shao
Borivoje Nikolic
Vijay Janapa Reddi

Summer 2024

Enabling Full-Stack DNN Accelerator Design and Evaluation on Synthesizable Hardware

Abstract

Enabling Full-Stack DNN Accelerator Design and Evaluation on Synthesizable Hardware

by

Hasan Nazim Genc

Doctor of Philosophy in Computer Science

University of California, Berkeley

Krste Asanovic, Chair

The growing diversity of computationally demanding DNN workloads, together with the long-running decline of technology-scaling trends, has motivated the design of a great many diverse specialized hardware accelerators. While these accelerators provide significant improvements to performance and energy consumption — making the modern wave of AI innovation possible — they introduce significant challenges to computer architects, programmers, and hardware designers, due to the difficulty of (i) exploring the very broad design space they represent, (ii) translating such designs rapidly to high-quality RTL and software libraries, and (iii) evaluating such designs in realistic full-system contexts early on in the design process.

Prior work has attempted to address these difficulties by proposing new accelerator design frameworks which allow users to change only a few settings in a config file, or a few lines of a domain-specific language, from which they can rapidly generate new synthesizable hardware, or new high-level models that can guide architectural decisions. Many of these frameworks also attempt to make accelerator design more principled by *separating* the different concerns which go into accelerator design, so that each can be explored individually, and in relation to other design choices.

However, prior accelerator generators and design frameworks often lack the ability to provide users visibility into the impact that the *full* system and software stack have upon DNN accelerator performance, such as the potential for outer caches, virtual address translation mechanisms, or host CPUs to bottleneck performance if they have not been carefully tuned along with the accelerator's functional units or spatial arrays. Prior accelerator design frameworks which separate out different design concerns are also are not capable of generating high-quality RTL for *both* dense and sparse accelerator ASICs, which limits their ability to cover various modern workloads which sparsify DNN layers to improve performance or energy efficiency.

This thesis presents two projects, Gemmini and Stellar, which address these difficulties. Gemmini is a DNN accelerator evaluation framework which, while generating efficient spatial arrays and accelerators, is primarily intended to help users evaluate the impact of SoC components *outside* of the accelerator itself, such as external caches or virtual address translation mechanisms, upon overall DNN accelerator performance. Stellar is another framework which provides abstractions that help users to design and explore different components of both dense and sparse accelerators, while separating out the different concerns that go into accelerator design, such as an accelerator's functionality, its dataflow, the sparse/dense data formats it supports, its load-balancing strategies, and the private memory buffers it is equipped with. Gemmini-generated dense DNN accelerators achieve 87% the performance of prior state-of-the-art accelerators such as NVDLA on image classification networks such as ResNet50, and enable insights into how minor changes to system components such as TLBs can improve end-to-end DNN performance by up to 15%. Stellar-generated accelerators achieve up to 92% the performance of hand-written accelerators, with less than 15% area overhead and power overheads on various DNN layers as low as 7%.

To my son.

Contents

List of Figures

List of Tables

Acknowledgments

It may be cliche to say, but is undeniably true, that this PhD was only possible through the advice, input, and tireless help of a great many people, to whom I am deeply grateful.

First and foremost, I am grateful for the help and advice of Krste Asanović, my advisor, and Sophia Shao, who I have long regarded as a sort of unofficial advisor. Krste accepted me into his research group, and through all these years, has always been ready to provide me with invaluable feedback on everything from high-level research goals and directions to low-level hardware design problems. I would be well-content to understand anything by the end of my life as well as he understands computers. Sophia, on the other hand, joined our lab shortly after I did, as a new professor, and I cannot imagine how different this experience would have been without her there. She helped me find an angle by which to differentiate my first project, Gemmini, from the many DNN accelerators which came before and after it, allowing us to finally find a home for the work after earlier attempts had failed. Since then, she has never hesitated to help me identify research problems worth solving, ask the hard questions necessary to deliver a project to a satisfactory conclusion, or to dive deep into the cause of any hardware or software inefficiency.

I am also grateful to my other dissertation committee members, Borivoje Nikolic and Vijay Janapa Reddi. Both provided me with invaluable feedback from my quals all the way up to my final dissertation years later. Vijay, in fact, began advising me when I was only an undergrad sophomore, and I am thankful that he was willing to serve on my dissertation committee at the end when I at last graduated with my final degree.

I am also grateful to Ameer Haj-Ali, Alon Amid, Vighnesh Iyer, and Seah Kim, who were my earliest collaborators at Berkeley. One's earliest collaborators set the baseline for what one expects from future collaborators, and I was lucky that they set a high baseline for me indeed. Closer to the end of my PhD, Prashanth Ganesh and Hansung Kim also worked very closely with me, meeting the aforementioned standard easily. Without such friends and co-authors, none of my work would have gotten anywhere close to the finish line.

Furthermore, I worked with a great many other people during this process; I list some of them below, but by no means is this a complete list. John Wright, Daniel Grubb, Harrison Liew, and Colin Schmidt helped me complete my first tape-out; though the process was grueling, sitting around them (and Vighnesh) every day, all day, for several weeks on a large table in BWRC was the only thing that made it possible to squeak out a violation-free design at the end. Dima Nikiforov and Simon Guo helped me create tutorials for Gemmini; the slides they helped me make and present have since been downloaded hundreds (perhaps thousands) of times online. Jerry Zhao and Abraham Gonzalez helped me get unstuck whenever Chipyard, or hardware design in general, stumped me (I got stuck quite often). Qijing Jenny Huang and Grace Dinh knew much about topics of which I knew very little; their assistance saved me much time and effort. I am grateful also to Vikram Jain, Coleman Hooper, Josh Kang, Kris Dong, Gilbert Bernstein, Yuka Ikarashi, Charles Hong, Sehoon Kim, Amir Gholami, Ja Wattanawong, Albert Ou, Howard Mao, Kevin Anderson, Sahil Bhatia, Igor Kozachenko, Adam Izraelevitz, Brendan Sweeney, Richard Lin, Behzad Boroujerdian,

Matthew Halpern, Marcelino Almeida, Yazhou Zu, Srivatsan Krishnan, Ting-Wu Rudy Chin, David Biancolin, John Koenig, and Schuyler Eldridge. If there are others whom I have neglected to mention, the fault is with my memory and not with your contributions.

I was fortunate also to have a large number of undergraduate and Master's assistants over the past six years, including Richard Yan, Avinash Nandakumar, Leena Elzeiny, Divija Hasteer, Pranav Prakash, SooHyuk Cho, Sherry Fan, and Kareem Ahmad. They made excellent contributions to my projects, and often surprised me by surpassing the work that I would have expected from a PhD student.

The assistance of the staff at SLICE (formerly ADEPT [formerly ASPIRE]) was also indispensable and I am grateful therefore to Kostadin Ilov, Ria Briggs, and Tamille Chouteau. To Kosta, I apologize for the times I forced machines to reboot.

Finally, I am grateful to my family: to my wife, Selva, who followed me across states, made the long nights and debugging marathons much easier to bear, and who reminded me that a much bigger, more important world exists outside of the IDE and waveform viewer; to my parents, Emine and Ismail, who never tired of celebrating every small success and providing encouragement when facing every big difficulty; and to my sister, Fatma, who, unlike me, had the good sense to become a real doctor. Their constant caring love and support has been the pillar on which my whole life rests.

Chapter 1

Introduction

Modern computer architects find themselves squeezed from both ends: as DNN (deep neural network) workloads continue to place ever-higher compute and memory capacity demands on their underlying hardware, the transistor-scaling laws which traditionally made such hardware faster, smaller, and more efficient have gradually plateaued. The pressing need to accelerate such applications despite the (partial) demise of Moore's Law and other technology-scaling trends has motivated architects to search for optimizations further up the hardware-software-system stack: through microarchitectural innovations; through more careful scheduling or partitioning of DNN operations across multiple functional units, accelerators, or independent chips [43]; or through the *co-design* of separate hardware and software components so that neither imposes unnecessary inefficiencies on the other.

Such strategies have succeeded in accelerating DNN workloads, not only in academia, but also in commercial products [64]. However, these techniques typically require an increase in the complexity of both hardware and software designs. For example, simple systolic arrays may need to be made more reconfigurable [25]; balanced cache-hierarchies may need to become *im*balanced with each cache independently tuned for a different tensor [58]; sophisticated load-balancing techniques may necessitate expensive NoCs (networks-on-chip) to cope with unevenly-distributed workloads; or software schedules may become so complicated that new programming languages are needed for programmers to easily represent them across different accelerators [36].

The complexity and diversity these solutions impose spans across the hardware-software-system stack. The immense diversity of DNN models and software workloads is matched by an equal variety of co-designed specialized hardware accelerators. A single accelerator may need to run multiple workloads simultaneously, or a single DNN workload may need multiple accelerators for different operations that it executes — in either case, the exact partitioning of resources among competing hardware and software components, and the interactions across different layers of the stack can cause subtle and difficult-to-diagnose bottlenecks and unexpected inefficiencies.

To maintain high architect and engineer productivity in the face of this complexity, new tools are needed: to design and explore diverse accelerator solutions, and to evaluate their performance and efficiency once they've been integrated into a full-system, full-stack environment. This work describes two such tools, *Gemmini* [22] and *Stellar*, which enable developers to (i) rapidly explore accelerator designs for DNN (and other) workloads, (ii) automatically generate high-quality RTL for complete, programmable SoCs containing their accelerators, and (iii) evaluate how their accelerators run on real-world end-to-end workloads, when interacting with both external hardware on the SoC and with the higher-level software stack.

Gemmini is a DNN accelerator generator, which includes a flexible programming stack and various counters and profilers which are designed to evaluate the impact of SoC components *outside* of the accelerator itself, such as external caches or virtual address translation mechanisms, upon overall DNN accelerator performance. The accelerator generator, written in Chisel [3], primarily targets dense CNN and transformer inference workloads, although it also includes support for certain other applications such as DNN training. Gemmini covers a wide design-space of dense DNN accelerator designs, with different spatial arrays, integer or floating-point datatypes, dataflows, and memory-hierarchies, and generates RTL for them which achieves real-time performance on real-world workloads.

Stellar, on the other hand, is designed to cover a wider space of accelerator designs for both dense and sparse workloads. Stellar introduces abstractions that allow users to separate out the different concerns that go into accelerator design: the *functionality* of an accelerator, its *dataflow*, the dense or sparse *data formats* it supports, and its *load-balancing* strategies. Once users specify their accelerators using these abstractions, Stellar generates high-quality RTL which achieves comparable performance and area efficiency to handwritten Verilog designs from prior work.

Together, the two works together enable architects to more quickly design, generate, and evaluate novel accelerators in a full-system, full-stack context. Chapter 2 describes prior, related work and provides further background on accelerator design and accelerator generators. Chapter 3 provides an in-depth description of Gemmini, as well as case studies showing how it can be used to derive interesting insights into how "full-system components" affect DNN accelerator performance. Chapter 4 introduces Stellar, describes how its frontend enables architects to quickly design and explore accelerators, and how it then generates high-quality RTL based on users' frontend descriptions. Finally, Chapter 5 concludes the thesis and describes future research opportunities building on the work described here.

Chapter 2

Background: The Spatial Accelerator Design Space

The growing diversity of DNN workloads has spurred the development of an equally wide variety of specialized hardware accelerators, typically incorporating highly-parallel "spatial arrays" which are responsible for executing most arithmetic operations. To expedite the creation of so many accelerators, prior work has proposed a number of automated accelerator generators, which allow users to co-design or tune their hardware and software workloads at higher levels of abstractions and then generate high-fidelity models or high-quality RTL, as opposed to having users write new ad-hoc RTL by hand for every new accelerator design point. However, prior accelerator generators either fail to provide insight into the impact that all the different components of the hardware-software-system stack have upon accelerator performance, or they fail to generate actual synthesizable RTL for the full design space of sparse and dense DNN workloads — limitations which are addressed by Gemmini [22] and Stellar.

This chapter describes spatial accelerators from prior work (Section 2.1), as well as high-level generators and frameworks for both dense (Section 2.2) and sparse (Section 2.3) accelerator design. Section 2.4 describes how prior accelerator generators, for either sparse or dense workloads, provided only limited insight into how the *full* hardware/software stack determines overall accelerator performance and efficiency. Finally, Section 2.5 describes how our work addresses these limitations. Table 2.1 summarizes how our work compares to prior accelerator generators and design frameworks.

2.1 Spatial 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 [9, 18, 60, 86]. At the architecture level, different DNN accelerators exploit different reuse patterns to build specialized memory hierarchies [98] and interconnect networks [48] 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 [21] and NVDLA [80]. Many exploit either structured or unstructured sparsity in their workloads (typically imposed during or after training at a small cost to accuracy) in order to achieve higher performance or energy efficiency [14, 15, 31, 64, 71, 67].

The design space for dense accelerators is well understood as they typically differ in the dataflows they support (which may be fixed or runtime-configurable), the quantized or unquantized datatypes they operate on, the functional operations they can perform (e.g., ReLU, GeLU, or other activation functions for more recent DNNs), or in their resource constraints, e.g. for small accelerators targeting low-power edge devices or for high-performance accelerators located in the cloud.

Sparse accelerators differ in even more ways, due to the wide variety of sparsity distributions and the corresponding sparse data formats in sparse workloads. Some sparse accelerators are designed for extremely sparse workloads, where far fewer than 1% of elements are non-zeros [66, 82, 104, 102]. Conversely, accelerators optimized for sparse DNNs [68, 64, 72] target matrix densities ranging from 30% to 70%. The vast range of sparsity levels necessitates the adoption of distinct sparse data formats and hardware designs, exposing a substantial design space.

As a result, existing accelerator designs generally differ from each other in *multiple* ways. For example, recent sparse accelerator proposals commonly propose not only new hardware dataflows but also new sparse formats and load-balancing strategies [82]. This inherent diversity complicates the process of comparing different accelerators, making it challenging to discern *which* feature contributes to specific improvements or drawbacks. However, such nuanced comparisons are crucial for architects, providing insights into the key principles underlying each design and guiding the selection of optimal solutions for specific workloads.

Furthermore, although spatial arrays often attract a disproportionate amount of interest from hardware designers, they sit within a wider SoC and hardware/software stack, where components outside of the spatial arrays themselves may interact in unexpected ways to influence workload performance and efficiency. For example, cache and memory hierarchies, host CPUs, page-table walkers, and even operating system interrupt and prioritization schemes can impose delays which prevent the spatial accelerator from achieving high utilizations. Some prior work has attempted to address these sources of inefficiency from "external" components [35, 27, 57], but tools that provide users with visibility into how all these components interact holistically have traditionally not been available for DNN accelerator designers.

2.2 Dense Accelerator Generators

To help meet the ever-growing demand for custom dense spatial accelerator designs, prior efforts have developed hardware-generation frameworks that systematically enumerate the dense design space. These frameworks allow designs to be expressed along orthogonal design

		Dense Frameworks							***Sparse Frameworks***						
		PolySA/AutoSA	**Interstellar**	**NVDLA**	**VTA**	**MAGNet**	**Rubick**	**DNNWeaver**	**Sparseloop**	**TeAAL**	**SAM**	**DSAGen**	**Spatial**	**Gemmini**	**Stellar**
Design Parameters	*Functionality*	✓	✓	✗	✗	✗	✓	✗	✓	✓	✓	✓	✓	✗	✓
	Dataflow	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓	*	*	**	✓
	Sparse Data Structures	✗	✗	†	‡	✗	✗	✗	✓	✓	✓	✗	✗	✗	✓
	Load-Balancing	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗	✓
	Private Memory Buffers	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓
	Datatypes	✓	✗	✓	✗	✗	✗	✓	✗	✗	✗	✓	✓	✓	✓
Hardware Outputs	*Simulators*	✗	✗	✓	✓	✗	✗	✗	✓	✓	✓	✗	✗	✗	✗
	RTL	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓
Programming Support	*Application-Level*	✓	✓	✓	✓	✗	✓	✓	✗	✗	✗	✓	✓	✓	✓
	ISA-Level	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓
System Support	*Virtual Memory*	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓
	Full SoC	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓
	OS Support	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓

Table 2.1: A comparison of DNN accelerator generators and design frameworks.

∗ The dataflow specification is *implicit* to the functional description, and not separated.

∗∗ Various dataflow options are provided by the framework, but users cannot add their own custom dataflows.

† Supports sparse memory traffic, but not sparse execution on spatial arrays.

‡ Block sparsity.

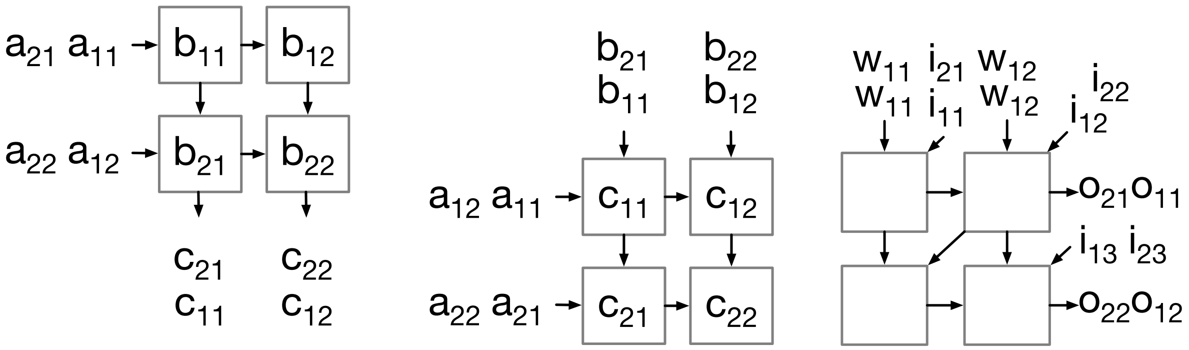
dimensions, such as *functionality* to describe an accelerator's expected outputs, *dataflow* to describe data reuse patterns, and *memory buffers* which describe memory hierarchies. Frameworks like PolySA [11], AutoSA [89], and Interstellar [97] allows users to succintly describe a broad set of of dense accelerators using these well-defined, orthogonal design dimensions. These frameworks can automatically synthesize RTL implementations for FPGAs or ASICs, together with application-level APIs as their programming interfaces, while offering a comprehensive separation of design concerns for systematic design space exploration. Other works, such as NVDLA [63], VTA [61], or Tabla [56] provide highly parameterized handwritten accelerators, or convenient pre-defined operators and templates which can be composed by users into more complex accelerators, but lack the ability to express certain design considerations, such as dataflows, independently of other design concerns.

The following subsections describe in further detail how these dense accelerator design frameworks specify and taxonomize different design considerations, such as dataflows and memory buffers, and their existing limitations.

2.2.1 Dataflow

Dense accelerator generators are often focused on expressing and generating different *dataflows* for spatial accelerators. A dataflow is a description of how inputs and outputs travel through a spatial array, and how they are re-used as they travel. For example, consider Figure 2.1, which illustrates spatial arrays for matrix multiplications holding either weights (Figure 2.1a) or outputs (Figure 2.1b) stationary in local registers, while other tensors travel between the processing elements (PEs), or Figure 2.1c, which shows a two-dimensional convolution where neither inputs, nor weights, nor outputs remain stationary.

Switching from one dataflow to another, as from Figure 2.1a to Figure 2.1b, does not necessarily change the number of functional units or the total maximum throughput of the



(a) Weight-stationary matmul.

(b) Output-stationary matmul.

(c) 2D convolution.

Figure 2.1: Spatial arrays with different dataflows for matrix multiplications and two-dimensional convolutions.

spatial array (although we will see later how dataflow inflexibility can limit performance for certain workloads). Dataflow exploration for dense accelerators is therefore often intended to minimize *energy consumption* by reducing the number of times which high-reuse elements must be read from or written into large SRAMs or register files, even if overall performance remains unchanged.

Which input or output elements will have the highest re-use depends on the exact workloads being run, and often varies significantly even within the same end-to-end workload. For example, convolutions in popular CNNs often have small filters which are reused hundreds or thousands of times across large input activation tensors, however, later layers in CNNs tend to have input activations with smaller planar dimensions which can significantly reduce the relative reuse of filter elements. The diversity of existing software workloads therefore requires diverse hardware dataflows to be explored by architects.

Beyond data-reuse opportunities, there are more subtle differences between different dataflows as well. For example, observe that the spatial array in Figure 2.1a accesses elements of a in a row-major order, but the spatial array in Figure 2.1b accesses them in a column-major order which will require either than data is laid-out differently in memory, or that hardware transposers are supplied to the accelerators to perform such transpositions on-the-fly.

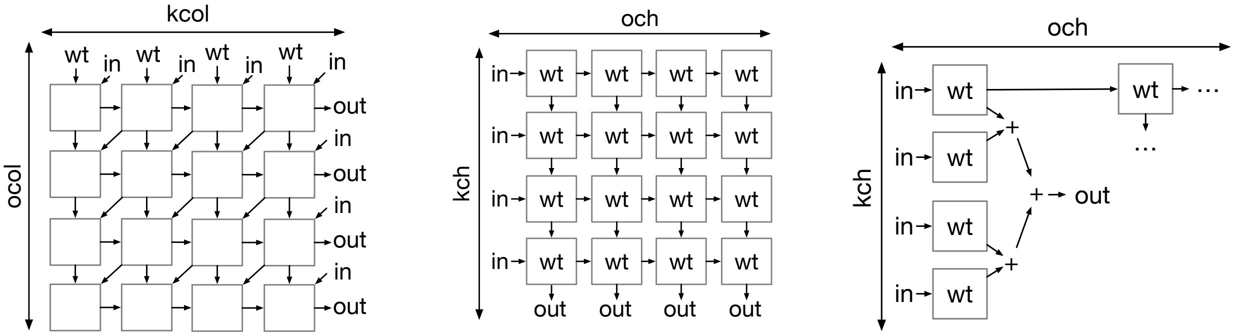
To make principled (and ideally automated) exploration of dataflows easier, numerous attempts have been made to construct dataflow taxonomies. For example, Eyeriss [7] introduces a dataflow classification scheme ultimately separating them into six classes: weight-stationary, three different types of output-stationary, dataflows with *no* stationary elements, and row-stationary (which was first introduced by the Eyeriss authors themselves).

Although this scheme is concise, easy-to-understand, and capable of explaining a great number of past accelerators, it is fundamentally limited: there are after all, far more than six possible spatial array dataflows. Any dataflow classification scheme which attempts to define dataflow solely as a set of pre-defined “enums” will be difficult to extend as new dataflows are discovered. (In fact, Eyeriss’s main innovation was a new dataflow which required the above classification scheme to be expanded so that it could cover it).

Interstellar [97] (described above in Table 2.1) describes dataflows in a more principled way: in terms of which loop axes are *spatially* or *temporally* unrolled across a physical spatial array. For example, Figures 2.2a to 2.2b illustrate different convolutional spatial arrays constructed by spatially unrolling different loops of a seven-nested convolutional for-loop (shown for convenience in Listing 2.1).

Listing 2.1: Three-dimensional convolution.

```
for (int b = 0; b < batches; b++)
  for (int orow = 0; orow < orows; orow++)
    for (int ocol = 0; ocol < ocols; ocol++)
      for (int och = 0; och < ochs; och++)
        for (int krow = 0; krow < krows; krow++)
          for (int kcol = 0; kcol < kcols; kcol++)
```



(a) Systolic dataflow.

(b) Systolic dataflow.

(c) Tree reduction.

Figure 2.2: Dataflows for convolutions specified by spatially unrolling the loop axes in Listing 2.1, as proposed by Interstellar [97].

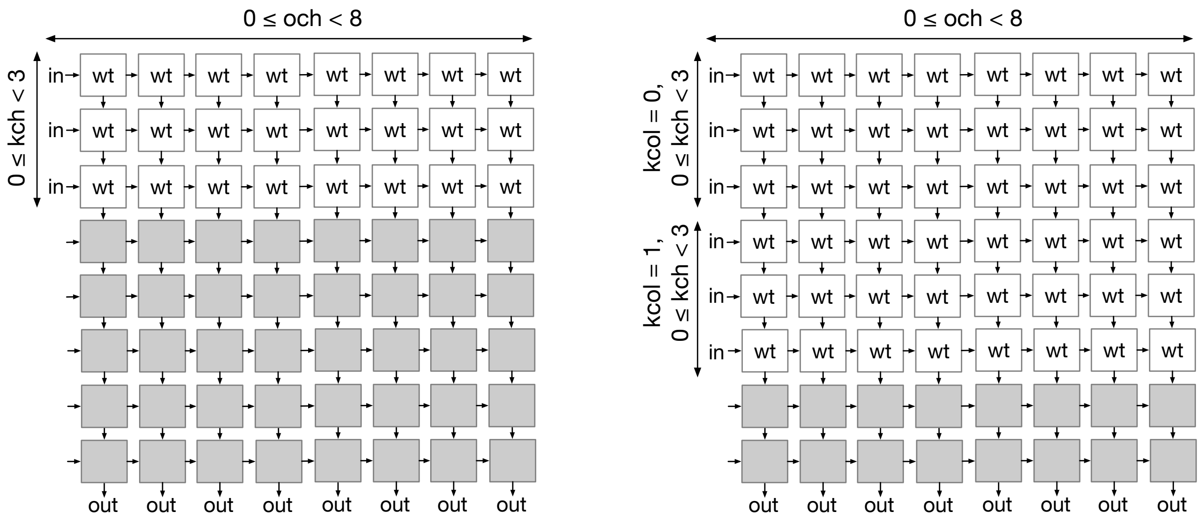
```
                for (int kch = 0; kch < kchs; kch++)
                  Out[b][orow][ocol][och] +=
                    In[b][orow+krow][ocol+kcol][kch] *
                    Wt[krow][kcol][kch][och]
```

The spatial arrays in Figures 2.2a to 2.2b are all rectangular and systolic [47], however, Interstellar also provides users with a boolean parameter that can be used to specify if a tree-based, recursive reduction should be used when unrolling loops instead, as in Figure 2.2c. A tree-based reduction can sometimes be used to construct shorter critical paths than a systolic one.

Unrolling loops spatially, however, can lead to PE underutilization and reduced performance when the loop dimension which is unrolled is smaller than the dimensions of the spatial array. For example, consider again the convolutional spatial array in Figure 2.2b, which spatially unrolls the input-channel and output-channel dimensions of convolution layers to construct a weight-stationary systolic array. Certain popular CNNs, however, have layers with very few input-channels, such as the depthwise convolutions in Mobilenet [33] which perform large numbers of convolutions with only a single input-channel each, or the first layer of a CNN like ResNet50 [30] which consumes an RGB image with only three input channels. Figure 2.3a illustrates how such a layer would be mapped onto a 16×16 spatial array with the weight-stationary dataflow from Figure 2.2b; note that the maximum possible utilization becomes 18.75%.

To support such types of workloads, Interstellar proposes a form of load-balancing which it calls "replication." As illustrated in Figure 2.3b, replication enables the dataflow to optionally unroll an *additional* dimension spatially whenever PEs are unutilized, at the cost of potentially greater hardware complexity.

Other works, such as PolySA [11], propose an even more flexible dataflow specification scheme [42, 50, 74, 75, 101], where rather than simply selecting several loop dimensions to

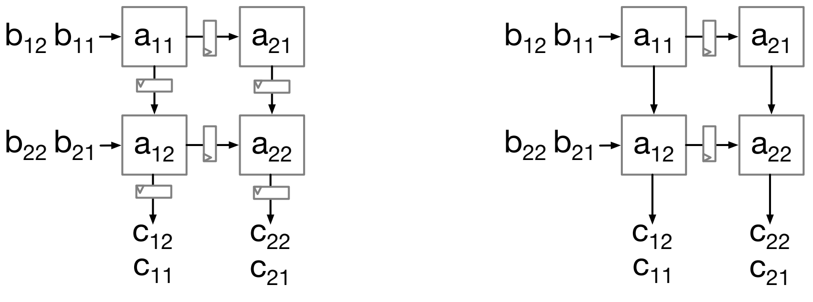


(a) Underutilized.

(b) Load-balanced.

Figure 2.3: PE underutilization caused by spatially unrolling the `kch` dimension from Listing 2.1, which is small for a subset of common DNN layers.

unroll spatially, a more general "space-time transform" matrix is constructed by the user. This space-time transform enables *multiple* loop axes to be unrolled spatially or temporally onto the same dimension of the physical spatial array, resulting in designs which may have different pipelining strategies based on how many dimensions were unrolled temporally, as in Figure 2.4, or in designs which are not even rectangular, as in Figure 2.5. Figure 2.5 unrolls *three* different axes of a matmul loop spatially onto a *two*-dimensional physical array. Build-



(a) Pipelined dot-product.

(b) Combinational dot-product.

Figure 2.4: Different pipelining strategies for spatial arrays whose dataflows keep the same elements stationary. The design in (a) pipelines the multiply-accumulates used to calculate a vector dot-product.

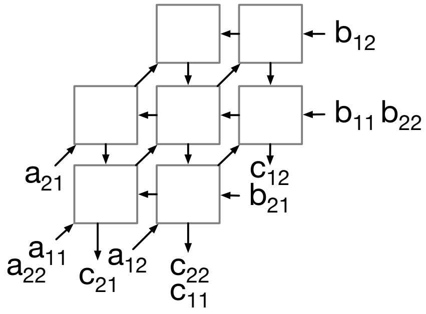


Figure 2.5: A spatial array with a hexagonal dataflow. This dataflow cannot be described simply by specifying a set of loop axes which will each be spatially unrolled on to a different physical spatial array dimension.

ing upon these space-time transform abstractions, more recent works, such as Rubick [53, 54], introduce frameworks which can decompose dataflows into separate specifications for the order in which tensor elements are accessed by a spatial array, and the layout with which these tensor elements are arranged in memory, enabling further automated optimizations. Later, Section 4.3.2 in Chapter 4 will describe in more detail how these space-time transform matrices are mapped to physical, synthesizable hardware.

2.2.2 Memory Buffers

Just as architects using dense accelerator design frameworks must choose dataflows which maximize data re-use in the spatial array, so must they construct memory hierarchies which exploit data re-use to perform the minimum number of energy-expensive SRAM reads and writes.

For example, Interstellar will construct separate memory buffers for each level of the loop being mapped to an unrolled spatial array[1], and automatically calculate the necessary capacities and bandwidths to keep the spatial array fully utilized when memory buffers are double-buffered. AutoSA, similarly, automatically constructs L1 buffers with optional double-buffering to feed spatial arrays with inputs at bandwidths which match their computational throughput.

MAGNet allows users to construct separate memory buffers for inputs, weights, and partial sum accumulations in a convolution, with small optional caches, constructed of latch arrays, providing an additional level of memory hierarchy. However, although separate memory buffers enable more aggressive hardware opimizations, note that memory buffers that are shared across different matrices, like a single large memory buffer for both weights and

[1] It is common for dense accelerator generators to construct one level of memory hierarchy for each level of a nested loop; note, however, that some prior work argues that "imbalanced" memory hierarchies which do not map one-to-one with loop nests can provide additional efficiency [58].

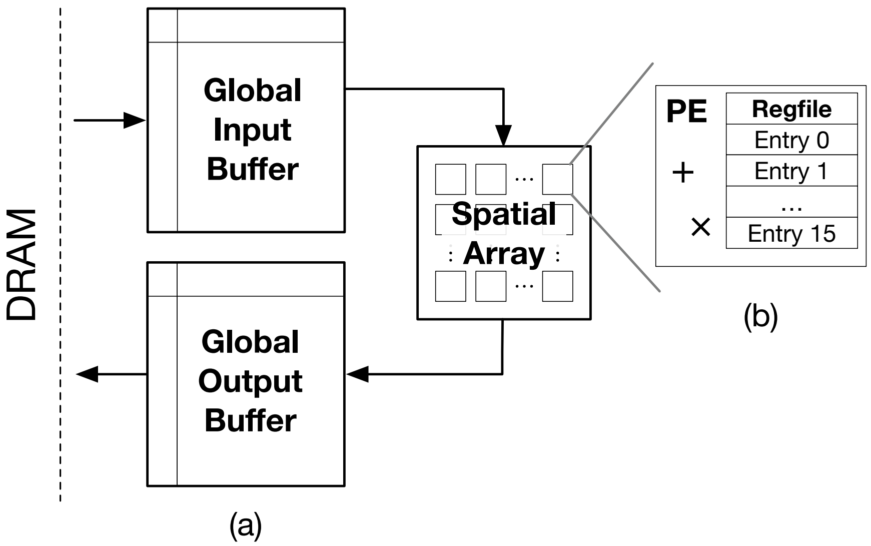


Figure 2.6: Memory buffers in a typical dense spatial accelerator. (a) shows centralized, global buffers which may be hundreds of kilobytes or megabytes large. (b) shows distributed, small register files local to each PE.

inputs, will enable an accelerator to sustain high performance across a wider set of workload dimensions or loop tile sizes. In addition to memory hierarchies external to the spatial array, as in Figure 2.6a, some accelerator generators [97] also enable users to explore different sizes for register files local to each PE, as in Figure 2.6b, to further reduce accesses to large SRAM buffers.

Finally, note that the memory buffers constructed by dense accelerator generators are oftentimes *explicitly-managed*, either by the programmer or by the accelerator generator framework, to maximize data reuse across the memory hierarchy without incurring the higher area or energy overhead of an implicitly-managed cache. For *dense* workloads specifically, it is possible for programmers or automated frameworks [69] to calculate the optimal tile sizes given a fixed memory hierarchy and DNN layer dimensions; explicit management is therefore the optimal choice.

2.2.3 Limitations

Although the hardware design frameworks described [11, 89, 56] above can effectively separate the concerns that go into dense accelerator design while generating high-quality RTL, they are challenging to extend to sparse accelerator design. Sparse accelerators introduce new key design considerations such as different sparse data structures and load-balancing techniques, which are often overlooked by frameworks designed exclusively for dense scenarios. Additionally, these frameworks typically expose only application-level programming interfaces which enable entire workloads to be offloaded to accelerators, but which lack the low-level ISA visibility which can help programmers who are targeting unusual or irregular workloads across a broad set of deployment scenarios.

2.3 Sparse Accelerator Generators

While existing design frameworks for dense accelerators provide end-to-end flows from abstract design specifications to RTL generation, frameworks dedicated to *sparse* accelerators have primarily focused on *modeling* and *simulating* sparse hardware designs and are not able to automate the RTL generation process. As such, they still leave significant manual work for hardware designers, and may sometimes fail to expose low-level performance bottlenecks not accounted for in higher-level simulators.

Specifically, recent works on modeling and simulating sparse hardware accelerators, such as TeAAL [62], the Sparse Abstract Machine (SAM) [34], and Sparseloop [95], introduce new abstractions for *sparse data formats* and *load balancing*, separately from other design concerns. These works define the functionality of sparse accelerators using convenient and intuitive syntax such as Einstein summations [19], the scheduling of operations on them using Halide-like loop transformations, and their sparse data formats using extensible abstractions such as fibertrees [84]. While TeAAL supports certain load-balancing schemes by allowing various tensor dimensions to be flattened for balanced distribution to spatial array PEs, it is difficult to describe more sophisticated load-balancing strategies [23], where individual PEs in a spatial array might have greater load-balancing capabilities than others.

The following subsections describe in further detail how these frameworks allow users to specify the sparsity formats, load-balancing schemes, and memory buffers needed.

2.3.1 Sparsity Formats

Sparseloop, TeAAL, and SAM all define the sparsity formats that their accelerators operate on using the fibertree notation [84]. As illustrated in Figure 2.7, a data format, such as

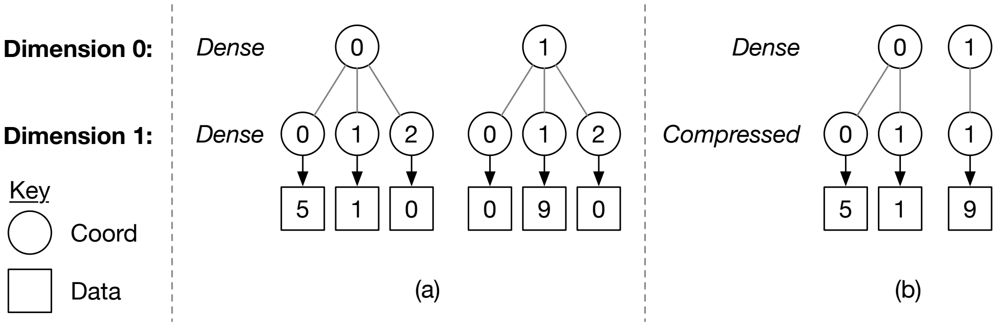


Figure 2.7: A 2×3 matrix represented using the fibertree notation in (a) as a dense matrix, and (b) in the CSR format.

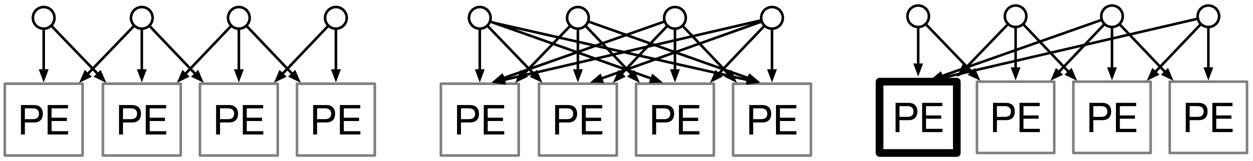
"Dense," "Compressed," or "Linked List," is attributed to every dimension of a tensor. A "Dense" dimension skips no elements at all, stores no associated metadata, and enables both in-order iteration and efficient random-access. A "Compressed" dimension, on the other hand, skips 0-elements entirely (whether these 0-elements are individual scalar values or empty rows/blocks), requires coordinate and pointer metadata to be stored along with data, and cannot efficiently perform random accesses. By composing these fibertree formats in different orders, many sparse data formats, such as CSR (*Dense-Compressed*) or block-CSR (*Dense-Compressed-Dense-Dense*) can be specified. A non-sparse two-dimensional matrix would simply be *Dense-Dense*.

Note that the fibertree notation does not necessarily indicate anything about the actual sparsity or distribution of nonzeros in a tensor. For example, a sparse tensor stored in the CSR format could have 0.1% density, 10% density, or (in theory if not in practice) 100% density. These nonzeros could also be distributed uniformly through the tensor, or following some Gaussian or power-law distribution (as with GCNs [23]). All of this, however, is independent of the actual data formats used to store the sparse tensors.

More recent work has proposed *structured* sparsity formats where the format requires specific nonzero distributions. For example, the NVIDIA A100's 2:4 sparsity format [64] requires two of every four adjacent elements to be zero, enabling far more efficient spatial arrays to be generated than unstructured formats such as CSR typically do. (This format has since been generalized to N:M sparsity formats [105]). The ELL [20] format is similar to the CSR format except that every row of the sparse matrix must have the same length, removing the need for lookups from a pointer array when iterating over the matrix. Finally, more recent sparsity formats proposed for large attention layers in transformers separate attention matrices into different regions, each of which may have diagonal, block-diagonal, or random sparsity distributions, and other regions which are completely dense [5, 10, 15, 73, 90, 100]. A single sparsity specification for the entire matrix is therefore not possible for such workloads.

2.3.2 Load-Balancing Schemes

Although load-imbalances can occur in dense workloads, as described above in Section 2.2.1, they are much more common in sparse workloads, where nonzero distributions can vary widely across rows (or other dimensions) of a tensor. Prior work proposes elaborate load-balancing schemes to maintain high PE utilization under such conditions; for example, AWB-GCN [23, 24] allows most PEs to take work only from overburdened *neighboring* PEs, but a small number of PEs are permitted to take work from *any* other PE, even if it is not a neighbor. The primary difficulty in designing load-balancing hardware is in avoiding large, unscalable interconnects when redistributing work from over-burdened PEs to underutilized PEs. For example, if every PE was permitted to share work with non-neighboring PEs, then the muxes and switches needed to redistribute work would grow vastly more expensive. Figure 2.8 illustrates how different levels of connectivity between PEs can lead to different costs and flexibilities for load-balancing.



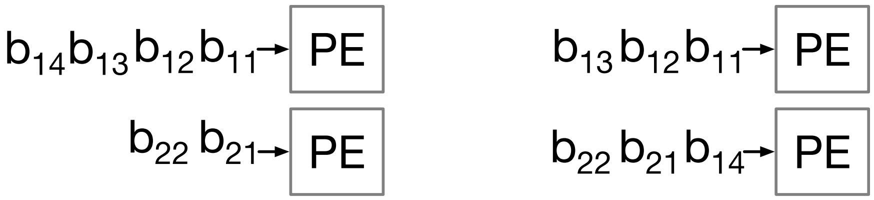
(a) Sharing work with only neighboring PEs.

(b) Sharing work with any PE, with expensive crossbars.

(c) Only one PE can take work from non-neighboring PEs.

Figure 2.8: NoCs required for different types of work-sharing and load-balancing between PEs in a spatial array whose workloads may be imbalanced.

Prior sparse accelerator design frameworks are not able to describe arbitrary load-balancing schemes separately from other design considerations; however, TeAAL is able to describe a subset of important load-balancing schemes by allowing tensor dimensions to be flattened and then re-tiled. For example, if a CSR matrix is unevenly distributed across different rows, TeAAL can flatten it into a single fiber which is then retiled so that equally-sized groups of nonzero elements (which may have originally been from different rows of uneven tensor) can be distributed to different PEs. Figure 2.9 illustrates how this flattening and retiling occurs. Some prior handwritten accelerators, such as Sextans [81] also balance sparse matmuls using this strategy.



(a) Imbalanced.

(b) Flattened and balanced.

Figure 2.9: Load-balancing a spatial array whose PEs consume an imbalanced tensor B by flattening and retiling the elements of B. In (a), each PE consumes a different row of B, but in (b), each PE can consume elements from any row of B.

2.3.3 Memory Buffers

Unlike memory buffers for dense accelerators, sparse memory buffers require additional storage for pointers, coordinates, or other metadata. The “bitmap” format, for example, stores small bitvectors adjacent to data buffers in order to determine exactly which coordinate a specific data value points to. The exact area or memory bandwidth overhead incurred by a sparsity format depends on the exact distribution of nonzeros within it’s tensors; the bitmap format would impose only a small extra storage requirement for a tensor with 40% density, but an unacceptably high overhead for a tensor with 0.01% density.

The fibertree notation described above can express a large number of sparsity formats, and sparse accelerator design frameworks can use such specifications to automatically calculate capacity or bandwidth requirements for sparse tensor memory buffers.

However, unlike dense accelerators, sparse accelerators often benefit from implicitly-managed caches, rather than lower-area explicitly-managed buffers, due to the difficulty of calculating the optimal tiling or access patterns for imbalanced, randomly distributed nonzeros before runtime. Handwritten sparse accelerators introduced in recent work [104, 102] propose implicitly-managed caches with novel prefetching or eviction strategies that are designed to maximize data re-use for specific sparse workloads. Some existing sparse accelerator design frameworks, such as TeAAL, can model caches with user-specified eviction policies to help cover such designs.

2.3.4 Limitations

Although the frameworks described above in this section enable rapid specification, evaluation, and simulation of sparse spatial accelerators, none of them *generate* actual RTL implementations. TeAAL and Sparseloop provide simulation and modeling capabilities and are primarily intended for early-stage exploration. SAM, while defining a set of hardware components for mapping dataflow, limits its evaluation to cycle-approximate simulations and CGRAs, rather than actual RTL implementations.

Prior work has proposed various hardware generation frameworks which *do* generate sparse accelerator RTL, but these lack the full separation of concerns necessary for effective design-space exploration. For example, DSAGEN [93] generates RTL for both dense and sparse workloads, however, it expects them to be defined as annotated C programs where sparse workloads are simply those with indirect memory accesses. These C descriptions are limited in their ability to separate out all the different concerns which go into sparse hardware design, so that each can be explored independently, as for the previously mentioned taxonomies. Other works, such as Spatial [45], introduce languages which describe custom hardware as sets of nested-loops, which generalize well to a variety of workloads, but which make it difficult to separate the functionality of an accelerator fully from its scheduling and dataflow, and which also lack higher-level constructions, such as data-format specifications, necessary to concisely express sparse workloads.

To enable faster hardware development, as well as to investigate various performance bottlenecks or area/power trade-offs which are only visible on actual RTL, hardware designers require tools which can generate synthesiable hardware designs from succint, expressive, separable abstractions. Although such frameworks do exist for dense accelerators, as described in Section 2.2, they are yet-to-be-developed for a broader set of dense and sparse hardware designs.

2.4 Full-Stack, Full-System Visibility

The frameworks described in Section 2.2 and Section 2.3 make it easier for users to design spatial accelerators across a broad design-space, but they do not provide users with visibility into the impact of the *full* hardware-software-system stack upon end-to-end performance and efficiency. Although a spatial accelerator is critical to the performance of a DNN workload, it is still only one component of a more complex SoC, where many different cores and accelerators may compete for access to shared resources, and where programming stacks and software runtimes must sometimes sacrifice full accelerator utilization for the sake of programmer convencience. Tools which allow architects and hardware designers to understand this full space are therefore necessary.

Prior accelerator generators typically do not enable users to explore SoC-level design parameters, such as host CPU configurations, virtual memory translation schemes, or arbitration over shared outer caches or system buses. However, industry evaluations have demonstrated that modern ML workloads could spend as much as 77% of their time running on SoC components outside of spatial arrays themselves, such as on CPUs which execute new operators that existing accelerators do not yet support, or when moving data between different CPUs and accelerators [32, 94, 29, 76]. Allowing architects to specify SoC-level design parameters, such the type of CPU to connect to an accelerator, enables the full hardware design to be tuned in tandem with the spatial accelerators themselves.

Finally, accelerator design frameworks need to provide easy-to-use programming interfaces 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 developers would prefer that the hardware programming environment be hidden by DNN development frameworks like PyTorch [70] or TVM [8] so that they don't need to worry about low-level development details, as in the case with VTA [60] and DNNWeaver [79]. 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 [87] and Maeri [48]. Accelerator generators should ideally provide "multi-level" programming interfaces which span the different levels of accelerator software development to satisfy users with different programming requirements. In addition to a lack of choice in the programming language/interface, prior accelerator generators and design frameworks typically neglect to add support for virtual memory, making it significantly more difficult for end-users to program their accelerators without special driver software.

2.5 Summary

As the demand for specialized hardware accelerators for DNN workloads grows, the need for faster, cheaper hardware design methodologies and frameworks becomes ever more acute.

Although prior work has introduced many excellent hardware design frameworks to meet this need, prior frameworks have several limitations: (i) they lack abstractions that allow independent exploration of different design parameters over the full space of dense and sparse accelerators, (ii) they could only generate actual synthesizable RTL for *dense* accelerators, and/or (iii) they do not provide architects with visibility into the impact of the *full* hardware-software-system stack upon overall performance.

Our work, Gemmini and Stellar, address these limitations. Gemmini is a full-stack, full-system DNN accelerator design framework that provides insight into how different SoC, programming, and system components affect end-to-end performance and efficiency. Stellar, on the other hand, is an accelerator design framework which separates out the different components of hardware design so they can be explored independently, and then generates synthesizable RTL for both dense and sparse accelerators which is comparable in performance and area overhead to handwritten RTL.

Chapter 3

Gemmini: Full-Stack Evaluation of DNN Accelerators

3.1 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 [60, 87, 11, 103, 96, 91, 99]. 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 [29, 94, 76]. 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

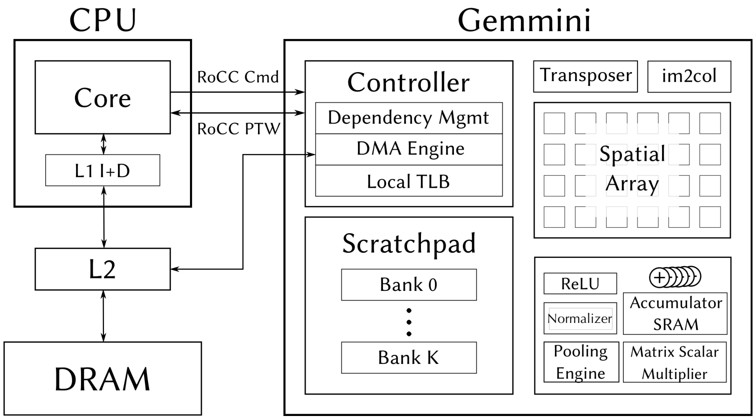


Figure 3.1: Gemmini hardware architectural template overview.

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 [80] 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.

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 3.2).

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 3.3).

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 3.4).

3.2 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. Table 3.1 summarizes the parameters that Gemmini users can set when exploring the design space for their accelerators.

3.2.1 Architectural Template

Figure 3.1 illustrates Gemmini's architectural template. The central unit in Gemmini's architectural template is an array with spatially distributed processing elements (PEs), each of which performs dot products and accumulations. The spatial array reads data from an 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.

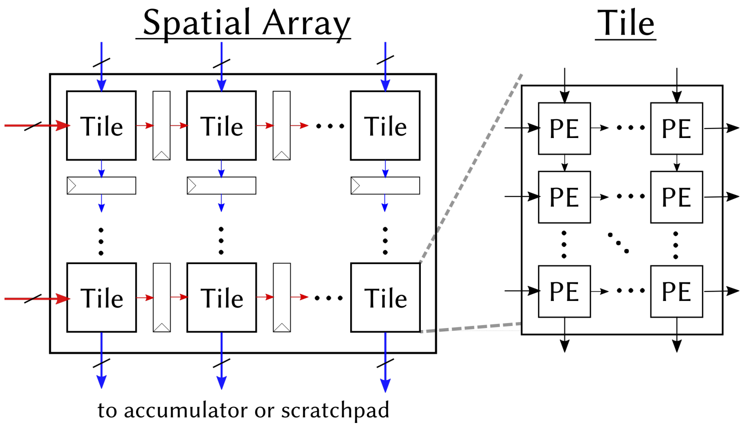
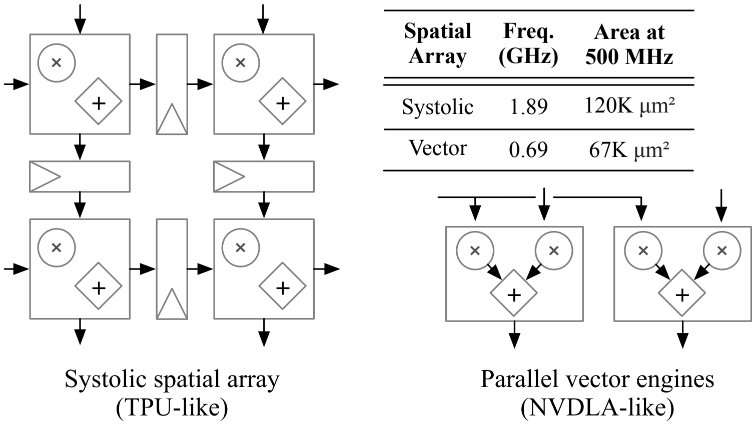


Figure 3.2: Microarchitecture of Gemmini's two-level spatial array.

Category	**Parameter**	**Recommended Range**
Spatial Array	Mesh Rows	1–256
	Mesh Columns	1–256
	Tile Rows	1–256
	Tile Columns	1–256
	Dataflow	Weight/output stationary, or both
Accelerator Memory	Scratchpad Capacity	256 bytes–16 MB
	Accumulator Capacity	256 bytes–8 MB
	Scratchpad Banks	1–4
	Accumulator Banks	1–4
	Scratchpad ports	1–2
	Accumulator ports	1–2
Execution Schedule	Reservation Station Entries	4–128
	Load Queue Entries	2–128
	Store Queue Entries	2–128
	Execute Queue Entries	4–128
Controller	PE Latency	0–4 cycles
	DMA Bus Width	64–256 bits
	DMA Block Size	32–64 bytes
	TLB Entries	2–64
Datatypes	Datatype	SInt/UInt/Float/User-defined
	Input Bitwidth	8–32 bits
	Output Bitwidth	8–32 bits
	Accumulator Bitwidth	16–64 bits
Operators	Multiply by Scalar	Present/Not
	Transposer	Present/Not
	Pooling	Present/Not
	Normalizers	Present/Not
	Backprop convs	Present/Not
	Im2col	Present/Not
System	Host Processor	Rocket, BOOM, etc.
	Number of Cores	1-64
	Number of Accelerators	0-Number of Cores
	Shared L2 Cache Size	256 KB – 16 MB
	Peripherals	UART, GPIO, SPI, JTAG, etc.
	IO Models	Network, DDR3, Block Device Latency-Bandwidth pipeline

Table 3.1: A summary of Gemmini hardware-configurable parameters. For the integer ranges, all power-of-2 values between the maximum and minimum are permitted. All parameters are independent of each other, and the size of the total search space is the cross-product of all possible parameter values.



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

Figure 3.3: 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.

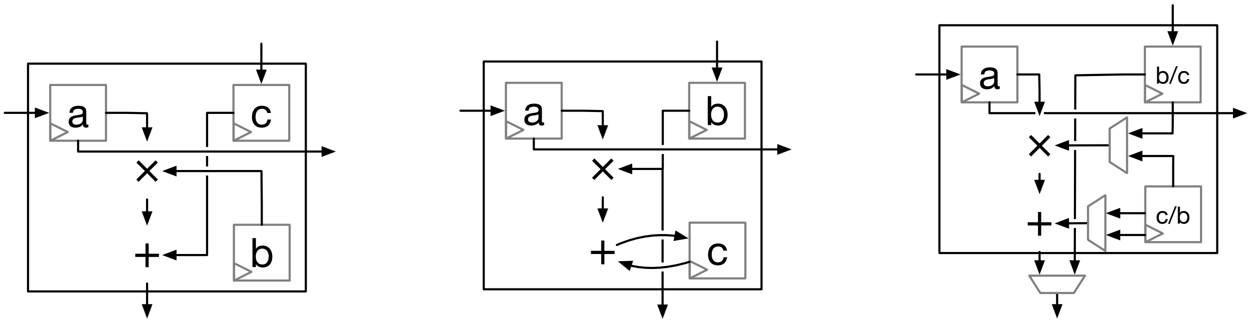
Spatial Array

We design Gemmini's spatial array with a two-level hierarchy to provide a flexible template for different microarchitectures, as demonstrated in Figure 3.2. 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 3.3 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 [1].

The dataflow can be either fixed at elaboration-time, or made configurable at runtime. Figure 3.4a illustrates a PE which supports only the weight-stationary dataflow, Figure 3.4b

[1] Note, however, that certain arithmetic dot-product optimizations, such as bulk normalization of floating-point values, have yet to be implemented in Gemmini.



(a) Weight-stationary (WS).

(b) Output-stationary (OS).

(c) Both OS and WS.

Figure 3.4: PEs that compute $A \times B = C$ with different dataflows.

shows a PE which supports only the output-stationary dataflow, and Figure 3.4c illustrates a PE which can switch between both at runtime; note that the runtime-configurability requires a small number of muxes to be added to the PEs. In the weight-stationary mode, PEs are preloaded with filter values before executing dot-products; in the output-stationary mode, they are preloaded with zeros or biases. All PEs are double-buffered so that they can be preloaded while dot-products are occuring; although this increases the area cost of the PEs, it helps ensure that matmuls and convolutions can continue at full throughput even when DNN matmul or convolutional layer dimensions do not enable much reuse of preloaded, stationary values.

Datatypes

Unlike some prior accelerator generators which provide support for only integer or only floating-point operations [11, 48, 79], Gemmini supports both. Users can generate spatial arrays with signed or unsigned integers, or with floating-point units with arbitrary exponent or mantissa widths, including IEEE standards such as double-precision, single-precision, and half-precision, or custom floating-point standards such as bfloat16 [38] or various proposed FP8 standards [59, 83]. The only limitations are that the total bitwidth of the datatype must be a power-of-2, and must be between 8 bits and 512 bits wide; these limitations only exist because they make the DMA design (described below) simpler.

Gemmini also provides an `Arihmetic` Scala typeclass which enables users to add their own custom datatypes by implementing a small number of operations such as additions, multiply-accumulates, and comparisons. Listing 3.1 shows an example of a custom complex datatype added to Gemmini.

Listing 3.1: An example of a custom complex datatype (with real and imaginary components) added to Gemmini.

```
class Complex(val w: Int) extends Bundle {
  val real = SInt(w.W)
  val imag = SInt(w.W)
}

// ...

implicit object ComplexArithmetic extends Arithmetic[Complex]{
override implicit def cast(self: Complex) =
new ArithmeticOps(self) {

  override def +(other: Complex): Complex = {
    val w = self.w max other.w

    Complex(w,
      self.real + other.real,
      self.imag + other.imag
    )
  }

  override def *(other: Complex): Complex = {
    val w = self.w max other.w

    Complex(w,
      self.real * other.real - self.imag * other.imag,
      self.real * other.imag + self.imag * other.real
    )
  }

  // ...
}
```

A Gemmini user's datatype specification can also interact with the dataflow selection to impact the spatial array design in more subtle ways. For example, 8-bit quantized DNNs typically accumulate partial sums into higher bitwidths, such as 32-bit integers, before they are scaled back down to 8-bit values which can be fed into the next DNN layer. With the output-stationary dataflow, 32-bit partial sums are accumulated in each PE; if the user wishes to pass these 32-bit results directly to the external accumulator where they can be scaled down, then 32, rather than 8, wires will need to be instantiated between each PE, increasing the spatial array's area overhead. If, instead, the user wishes to transmit only 8-bit values between PEs, then each PE must instead be instantiated with internal bitshifters or scaling units to reduce the 32-bit partial sums back down to 8-bit activations, also incurring an area overhead.

Scaling Units

As mentioned above, quantized DNNs typically require scaling units which cast higher-bitwidth partial sums down to lower-bitwidth activations for the following layer. In more area-conscious designs, these scaling units may be implemented as simple bishifters; when higher accuracy is required, they may be implemented as FP32 multipliers. To enable both area-conscious and high-accuracy use cases, Gemmini allows users to specify the scaling functions they require as arbitrary Chisel functions, although we also provide users with pre-written commonly-used scaling functions, such as integer bitshifts or floating-point multiplications with a variety of rounding modes.

These scaling units are optional and can be left out of accelerators which do not require them, such as DNNs which perform all operations with unscaled FP32 data. Note, however, that even floating-point non-DNN workloads, such as BLAS routines, often require such scaling units for both inputs and outputs; to support such use cases, we allow scaling units to be optionally instantiated at both the inputs to the scratchpad holding inputs and weights, and at the outputs of accumulators holding accumulated partial sums.

One existing limitation of Gemmini's scaling functions is that they are designed primarily to perform matrix-scalar multiplications. More recent DNN quantization schemes [16], including for commonly used large language model (LLMs) implementations such as `llama.cpp` [52], require separate scaling factors for different rows, or different blocks of adjacent values in input- or weight- matrices. Such scaling factors are not currently supported, though we expect that they will be added to Gemmini in future work.

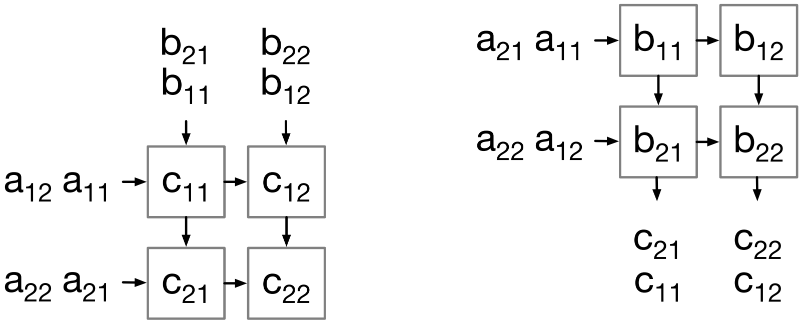
Activation Functions, Max-Poolers, and Normalizers

Gemmini-generated accelerators include optional functional units for commonly used activation functions such as ReLU or the I-BERT [44] variant of GeLU, as well as comparators for max-pooling. For weight-stationary matrix multiplications or convolutions, activation functions or max-pooling are applied as completed sums are being read out from the accumulator before being written out to DRAM or outer caches. In the output-stationary mode, activation functions can be optionally applied in the PEs before their internal scaling units reduce final partial sums down to low-bitwidth quantized activations for the next DNN layer.

To support more recent attention [85] models and LLMs, Gemmini also includes support for functional units which perform layernorm and softmax operations. To avoid the expensive lookup-tables needed to calculate exponents or other floating-point operations in parallel across large vectors or matrices, we instead implement polynomial approximations of these operations using compile-time integer constants [44]. Like weight-stationary activation functions or max-pools, normalizations are performed while accumulated partial sums are being read out from the accumulator memory so they can be written out to DRAM. However, normalizations such as layernorm require *multiple* read-outs of the same data from the accumulator; first to collect runtime statistics such as means or maximums of different vectors, and then again to use these collected statistics to scale and normalize the data.

Transposer

Figure 3.5a shows the order in which inputs and weights must enter an output-stationary spatial array, while Figure 3.5b shows the order in which they enter a weight-stationary array. Observe that for output-stationary matrix multiplications, elements of A must enter the array *transposed*, in column-major order.



(a) Output-stationary (OS).

(b) Weight-stationary (WS).

Figure 3.5: The order in which inputs stream into a matmul spatial array that computes $A \times B = C$ with different dataflows.

To perform such transpositions on-the-fly, Gemmini accelerators also include optional transposers, illustrated in Figure 3.6, which transposes data coming from the input/weights scratchpad into the matmul spatial array. The transposer is implemented as a small systolic array, but one which performs no computations and only reorders data. To transpose a DIM×DIM submatrix, the transposer will first take DIM cycles consuming rows of the submatrix from its left edge; it will then spend the next DIM cycles outputting the submatrix in column-major order along its upper edge. Note, however, that the transposer continues to consume a new untransposed matrix from its bottom edge while while outputting the transposed one from its upper edge; after DIM cycles, it will output another transposed matrix from its right edge. The transposer, therefore, operates at the same throughput as the matmul spatial array; there are no idle cycles or "bubbles" while it transposes a new matrix. Furthermore, the transposer's systolic design reduces its wiring congestion: every input port and output port can only connect to one of two PEs.

Table 3.2 shows which matrices Gemmini can transpose based on the dataflow the spatial array is configured to run. Because only one transposer is available in the accelerator, in the weight-stationary mode, only one matrix can be transposed at a time. However, in the output-stationary mode, both the inputs and weights can be transposed simultaneously: the weights by feeding them through the transposer, and the inputs by *not* feeding them into the transposer so that they enter the matmul spatial array in row-major order, rather than in column-major order as in Figure 3.5a.

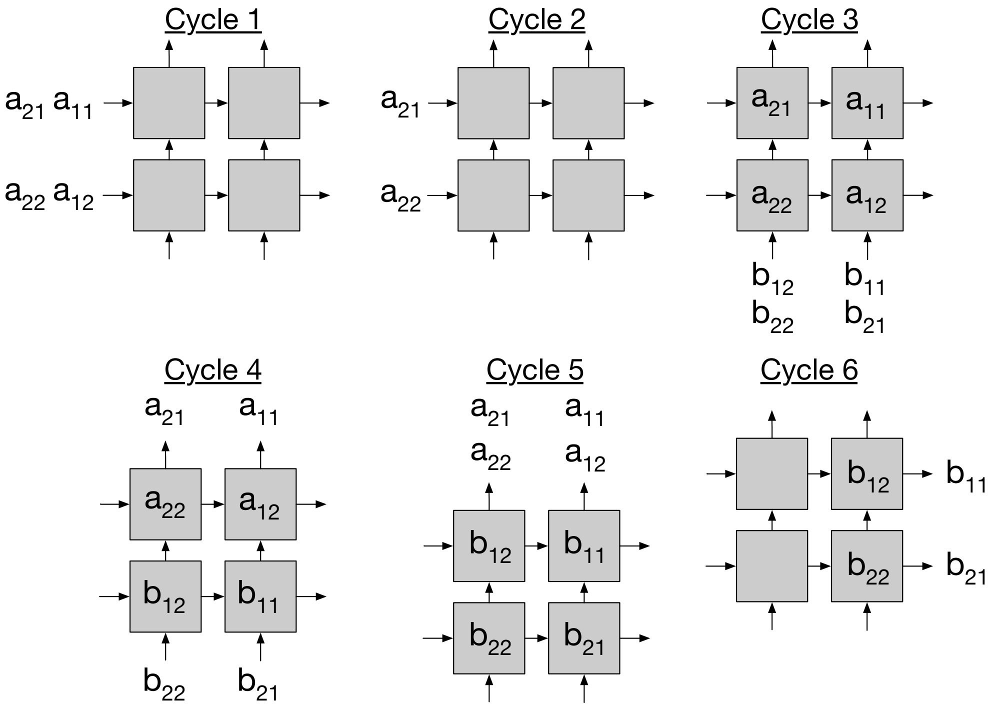


Figure 3.6: The systolic transposer included in Gemmini-generated accelerators while transposing two matrices, A and B, back-to-back.

Dataflow	Transpose A	Transpose B	Permitted?
OS	No	No	Yes
	No	Yes	No
	Yes	No	Yes
	Yes	Yes	Yes
WS	No	No	Yes
	No	Yes	Yes
	Yes	No	Yes
	Yes	Yes	No

Table 3.2: Legal and illegal transpositions on Gemmini.

Scratchpad and Accumulator Memories

Gemmini-generated accelerators include a scratchpad which stores inputs and weights for DNN layers, and an accumulator for partial sums. Inputs and weights share the same scratchpad, enabling a wide range of tiling sizes to be chosen by programmers during runtime, but partial sums can only be stored in the accumulator.

Table 3.1 describes the parameters that Gemmini users can set for the scratchpad and accumulator memories. Users can choose between different SRAM capacities, numbers of banks, and porting options. Bank conflicts in the SRAMs, or contentions of read/write ports in single-ported SRAMs will be handled transparently by Gemmini's hardware controllers and arbiters, although such contention may reduce performance. The partial sum accumulators must, by necessity, must be dual-ported, since they simultaneously perform reads and writes to add partial sums; however, if the accumulators are specified as being single-ported, then Gemmini will instantiate *two* single-ported accumulator SRAMs and interleave operations between them such that they appear to the rest of the accelerator as a dual-ported SRAM. The single-ported accumulators, however, do have certain limitations on striding patterns which the dual-ported SRAM accumulators do not, and cannot therefore support certain DNN training operations which perform convolutions and matmuls with more sophisticated striding patterns.

The scratchpad and accumulator memories in Gemmini are "row-addressed," where each row is DIM elements wide for a DIM×DIM matmul spatial array. The row-addressing scheme is illustrated below, in Figure 3.7, for an accelerator with a 2×2 spatial array, 8-bit integer quantized values in the inputs/weights scratchpad, and 32-bit partial sums in the accumulator.

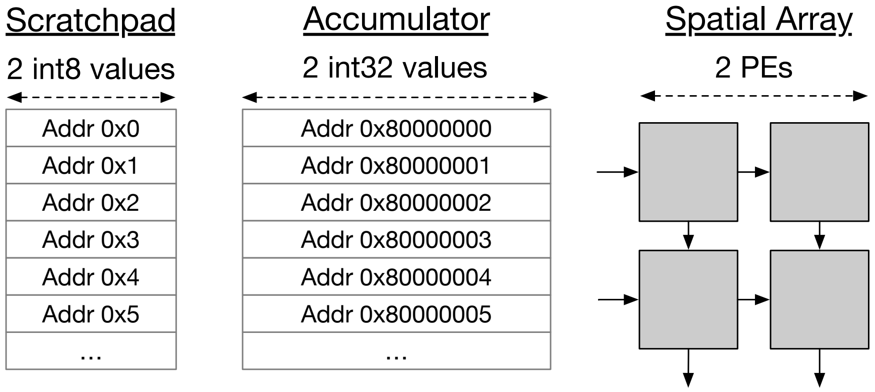


Figure 3.7: Scratchpad and accumulator addressing scheme for a 2×2 spatial array.

Because only rows of the scratchpad memories, rather than columns within them, can be addressed, the connections between SRAM ports and the spatial array PEs are quite efficient; each SRAM column is connected to only one or two PEs along the top, left, or lowermost edges of the spatial array, as illustrated in Figure 3.8. Enabling column-addressing would require more expensive crossbars to be instantiated between SRAM read/write ports and the spatial array PEs.

However, an important limitation of row-addressing is that it can lead to SRAM underutilization for awkwardly-shaped matrix multiplications or convolutions. For example, a one-column vector stored in the scratchpad SRAMs would only occupy one SRAM column;

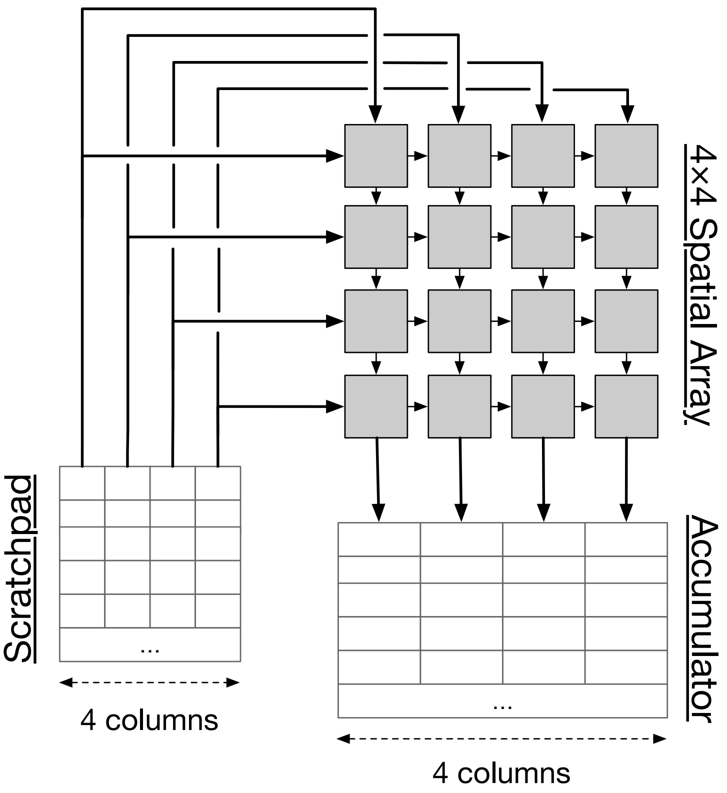


Figure 3.8: The scratchpad and accumulator columns each connect to only one or two PEs along the edges of the matmul array.

there is no way to pack multiple vectors across separate columns of the SRAMs for a series of independent matrix-vector operations because there is no way for the programmer to index or address individual columns of the scratchpad SRAMs so that they can separately be fed into the spatial array.

Every Gemmini scratchpad or accumulator address is 32 bits wide. The three most signficant bits are reserved, and have special meanings:

- Bit 31 (the MSB) is 0 if we are addressing the inputs/weights scratchpad, and 1 if we are addressing the accumulator.

- Bit 30 is ignored if we are addressing the scratchpad, or if we are reading from the accumulator. If, instead, we are writing to the accumulator, then bit 30 is 0 if we want to overwrite the data at that address, and 1 if we want to accumulate on top of the data already at that address.

- Bit 29 is ignored if we are addressing the scratchpad, or if we are writing to the accumulator. If, instead, we are reading from the accumulator, then bit 29 is 0 if we want to read scaled-down data from the accumulator, and 1 if we want to read the full, unscaled partial sums from the accumulator.

- If bit 29 is 1 for an accumulator read address, then we do not apply activation functions or scaling to the output of the accumulator.

This addressing scheme enables Gemmini to support residual additions or matrix-matrix additions very elegantly: programmers simply load data directly from DRAM into the accumulator by setting bit 31 (the MSB) to `1`, before loading another matrix on top of the same address by setting bits 31-30 to `11`. No specific matrix addition command is therefore required in the ISA; the memory addressing scheme instead encodes such information.

Currently, partial sums in Gemmini's accumulator cannot be transferred directly to the inputs/weights scratchpad unless they are first written out to DRAM by the DMA, and then read back in to the accelerator's memory. This limits Gemmini's ability to perform certain cross-layer optimizations when the accelerator is equipped with enough accumulator memory to store an entire layer's output tensors; however, we expect this limitation to also be addressed by future work.

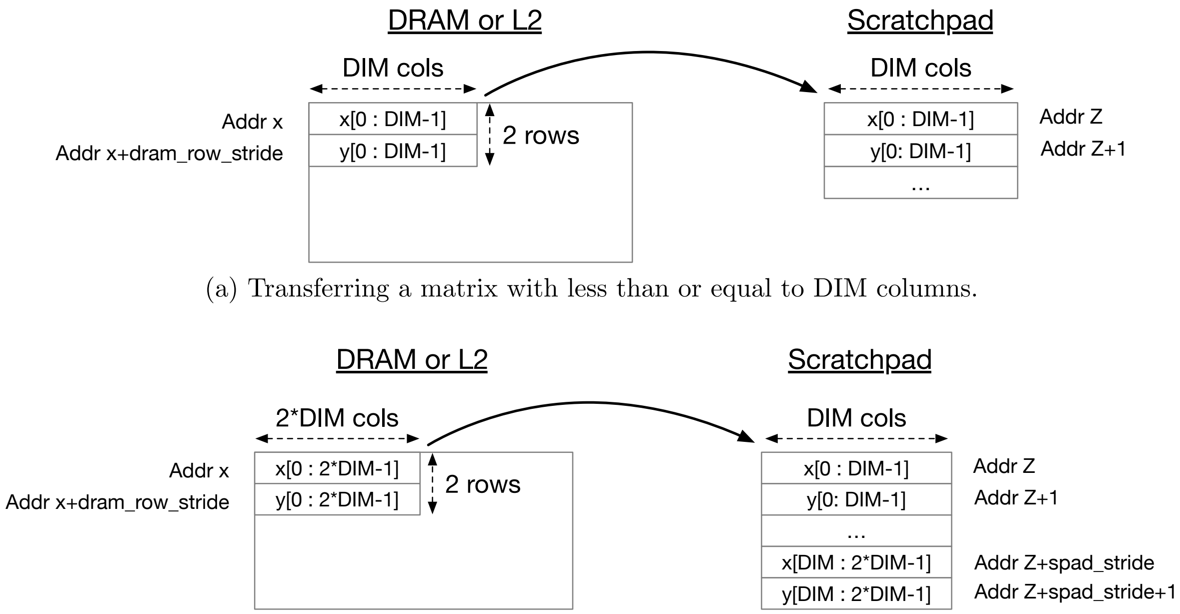
DMA

All data transfers between Gemmini's private scratchpad memories, and DRAM or outer caches, are handled by Gemmini's internal read DMA and write DMA. The read and write DMAs make memory requests to outer memory using the cache-coherent TileLink protocol [12]. The user can choose whether the two DMAs should share and arbitrate over the same TileLink ports to outer memory, or whether they should have separate ports, which can increase wiring congestion but enable reads and writes to happen simultaneously.

The read and write DMAs both move two-dimensional matrices with programmer-specified row-strides (the columns are assumed to be completely contiguous). Figure 3.9a shows how a load from outer memory to the scratchpad works. Figure 3.10b illustrates the special case where the number of columns moved-in to the scratchpad is greater than DIM for an accelerator with a DIM×DIM spatial array:

The read DMA also includes certain optimizations specifically useful for a DNN accelerator. For example, when the outer memory address specified for a move-in command is the null pointer, then the DMA will avoid making any TileLink requests and simply write zeros into the specified scratchpad addresses; this is helpful for padding in convolutional layers. The DMA also includes optimizations for loading biases into the accumulator. Biases in DNNs are typically one-dimensional vectors, but must be copied over many different rows of an accumulator. When the DRAM row-stride in Figure 3.9 is zero, the read DMA will make only a single TileLink request and copy the data returned from DRAM across multiple accumulator rows.

Finally, the read DMA can also assist with im2col for convolutions with very few input channels. By default Gemmini performs convolutions by unrolling the input-channel and output-channel dimensions spatially across the spatial array's rows and columns. However,



(a) Transferring a matrix with less than or equal to DIM columns.

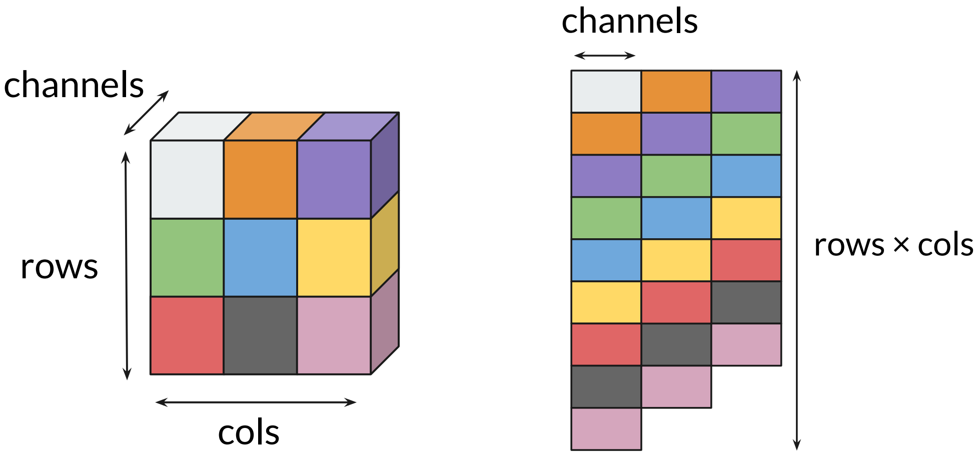
(b) Transferring a matrix with more than DIM columns.

Figure 3.9: How Gemmini's DMA moves matrices between DRAM or outer caches, and Gemmini's private scratchpad, based on programmer-defined strides.

when the number of input channels is very small, as in the first layer of a typical CNN, the resulting spatial array PE utilization and SRAM utilization can be very low. To improve utilization in such situations, the read DMA can automatically replicate input activation data across the scratchpad SRAM columns, as illustrated in Figure 3.10, similarly to how im2col replicates input activations in order to map convolutions to matrix multiplications [92].

3.2.2 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. 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



(a) Input activations.

(b) Layout in scrachpad.

Figure 3.10: Gemmini's DMA replicates input activation data in the scratchpad when the input channels are smaller than the number of scratchpad columns. When the number of input channels is larger, no such replication occurs.

dimensions of the spatial array, the dataflows supported, and the compute blocks that are included (such as pooling, im2col, or transposition blocks).

High-Level Programming Interface

Gemmini software libraries include a RISC-V fork of Microsoft's ONNX Runtime platform [65], which analyzes ONNX files and searches for operators, such as matrix multiplications or convolutions, which can be mapped directly to operators defined in Gemmini's handwritten mid-level prograamming interface (described below). Our fork adds support for integer quantization, as well as support for the NHWC format, which Gemmini uses for its input, weight, and output tensor layouts. We also add support for various operator fusions supported by Gemmini-generated accelerators, such as fusions of convolutions, activation functions, and max-pool operations.

When our high-level programming interface encounters an ONNX operator which it does not recognize, or which cannot be offloaded to Gemmini's spatial array or functional units, it falls back to running it on the CPU instead. The high-level programming interface is therefore able to maintain functional correctness even for novel, yet-to-be-invented neural network models, as long as a CPU fallback exists, although performance may be low for operations which can't run directly on Gemmini.

The high-level programming interface can also quantize and dequantize operations on-the-fly; for example, if a particular layer requires expensive floating-point operations which Gemmini does not support, our ONNX Runtime fork will automatically scale quantized integer layer outputs to dequantized floating-point values, and then scale them back to quantized tensors once they can be fed back into the Gemmini accelerator.

Mid-Level Programming Interface

Rather than using the high-level programming interface, which maps ONNX files directly to Gemmini, programmers can also run individual Gemmini-supported kernels in their own applications using our mid-level programming interface. Gemmini includes a C software library of handwritten, hand-tuned kernels for commonly used kernels such as convolutions, maxpools, residual additions, or layer normalizations. Table 3.3 summarizes the handwritten kernels which our mid-level programming interface currently supports.

Kernel	**Parameters**
`matmul`	Transpositions, input or output scaling factors, bias, activation functions, dataflow
`convolution`	Padding, input dilation, kernel dilation, fused max-pooling, scaling factors, activation functions, transpositions, depthwise
`resadd`	Input or output scaling factors, activation functions
`max_pooling`	Pooling dimensions and strides
`global_average_pooling`	Pooling dimensions and strides
`layernorm`	Fused activation function
`softmax`	Fused activation function

Table 3.3: DNN kernels available in Gemmini's mid-level programming interface.

At runtime, each kernel in the mid-level programming interface calculates tiling factors based on the dimensions of a layer's inputs and the hardware parameters of the accelerator. These tiling factors are calculated using heuristics which maximize the amount of data moved into the scratchpad per iteration. If the programmer wishes, the mid-level API also allows them to manually set tile-sizes for each kernel.

Low-Level Programming Interface

Finally, if the kernels we provide in the mid-level programming interface are not sufficiently optimized for a particular DNN, or if a DNN includes novel kernels which our mid-level API does not currently support, then programmers can write their own DNN kernels using Gemmini's low-level programming interface. At the low-level, programmers call Gemmini's ISA directly using low-level wrappers written in C, which are summarized in Table 3.4.

Instructions in Gemmini's low-level programming interface primarily operate on DIM×DIM matrices, where DIM is the dimension of the spatial array (although some instructions, like

	Instruction	Parameters
Loads	`config_mvin`	Row-strides and scaling factors
	`mvin`	Number of rows, number of columns, DRAM addresses, scratchpad addresses
Stores	`config_mvout`	Row-strides, scaling factors, activation function, max-pooling options
	`mvout`	Number of rows, number of columns, DRAM addresses, scratchpad addresses
Executes (i.e. matmuls, convs, dot-products,etc.)	`config_execute`	DRAM addresses, scratchpad addresses
	`preload`	Number of rows, number of columns, scratchpad addresses for preloaded values (weights or biases)
	`compute`	Number of rows, number of columns, scratchpad addresses for inputs/weights
Loop unrollers	`loop_matmul`	DRAM addresses of inputs, weights, biases, outputs, strides, transpositions, padding
	`loop_conv`	DRAM addresses of inputs, weights, biases, outputs, strides, transpositions, padding, dilation, fused max-pooling

Table 3.4: Gemmini's low-level ISA, summarized.

the `mvin` commands, can operate on larger matrices as described in Section 3.2.1). For example, `preload` loads a DIM×DIM matrix of weights or biases into the spatial array, `matmul` multiplies a DIM×DIM matrix with the preloaded values, and `mvout` moves the resulting DIM×DIM matmul result from the accumulator buffer into DRAM or outer caches.

RAW, WAR, or WAW dependencies between these instructions are tracked by a reservation station which attempts to overlap load, store, and matmul instructions for maximum performance, even permitting such instructions to execute out-of-order with respect to each other, while maintaining program order from the programmer's perspective. This reservation station can also be parameterized to have different capacities for load, store, or matmul instructions.

These DIM×DIM instructions can then be composed by the programmer into larger loops that perform matrix multiplications, convolutions, or other operations on arbitrarily-large tensors. However, composing instructions together into larger, tiled loops sometimes presents significant challenges to programmers, especially when DIM is small. When DIM is small, each low-level Gemmini instruction has a small granularity and may take only a few cycles to execute; the overhead of looping on the host CPU then can prevent Gemmini instructions from being issued fast enough to the accelerator to keep the spatial array or DMA fully utilized. Workarounds such as loop-unrolling, software pipelining, or double-buffering are difficult for programmers to do manually. To alleviate such difficulties and enable programmers to use the low-level programming interface more easily, Gemmini is also compatible with the Exo [36] programming language, which performs optimizations such as loop unrolling automatically.

Alternatively, programmers using the low-level programming interface can also use hardware loop unrollers that Gemmini-generated accelerators are all equipped with. These loop unrollers can be configured by programmers to automatically generate low-level ISA instructions and scratchpad addresses on-the-fly and feed them into Gemmini's reservation station, based on common looping patterns found in operations such as matrix multiplications and convolutions. Listing 3.2 shows an example of how the hardware loop unrollers can be invoked to replace tiled loops in software with code which performs equivalently, but doesn't require ardous manual loop-unrolling or software pipelining in order to maintain high instruction throughput from the host CPU to the accelerator's reservation station.

Listing 3.2: Invoking Gemmini's hardware loop unrollers to replace software loops composed of low-level ISA instructions.

```
for (int i0 = 0; i0 < N; i0 += TILE_I) {
  for (int j0 = 0; j0 < N; j0 += TILE_J) {
    for (int k0 = 0; k0 < N; k0 += TILE_K) {

      // Scratchpad tiles with software loops.
      for (int i1 = 0; i1 < N; i1 += DIM) {
        for (int j1 = 0; j1 < N; j1 += DIM) {
          for (int k1 = 0; k1 < N; k1 += DIM) {
            int i = i0 * tile_I + i1;
            int j = j0 * tile_J + j1;
            int k = k0 * tile_K + k1;

            A_addr = ...; B_addr = ...; C_addr = ...;

            if (j1 == 0)
              mvin(&A[i0*tile_I + i1][k*TILE_K + k1], A_addr);
            if (i1 == 0)
              mvin(&B[k0*tile_K + k1][j*TILE_J + j1], B_addr);

            preload(B_addr, C_addr);
            compute(A_addr);

      // Scratchpad tiles with hardware loop-unrollers.
      //   This one line replaces the many software
      //   looping lines written above.
      gemmini_loop_matmul(...); }}}}}}
```

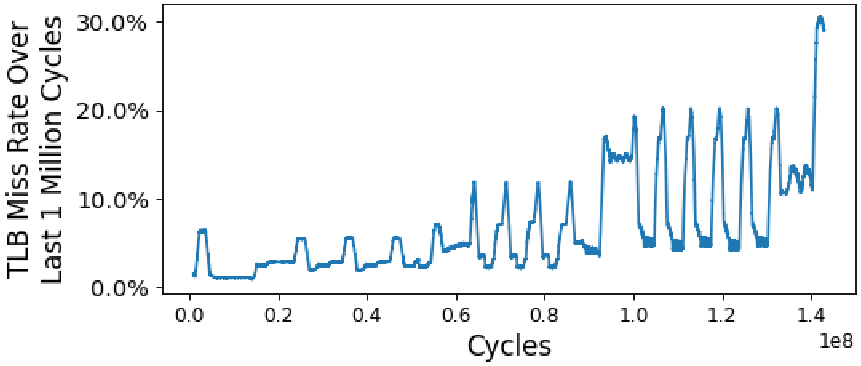


Figure 3.11: TLB miss rate over a full ResNet50 inference, profiled on a Gemmini-generated accelerator.

Finally, hardware loop unrollers also enable certain performance optimizations which software could not feasibly accomplish, even with perfect loop unrolling or software pipelining. For example, due to the unpredictability of outer cache accesses, it cannot be known until execution begins exactly how long each load or store instruction will take. A series of unexpectedly slow loads/stores can overwhelm the reservation station and leave no room for matmul instructions to be issued or executed, reducing overall performance as matmul instructions are no longer perfectly overlapped with memory transfer instructions. Gemmini's hardware loop unrollers, however, will monitor reservation station utilization during runtime and ensure that the number of load, store, and matmul instructions being executed simultaneously are balanced, even when loads or stores take unexpectedly long due to outer cache misses; such low-level, fine-grained optimizations are impossible for programmers to do solely with tiled software loops.

Gemmini's hardware loop unrollers cover a range of commonly used loops, summarized above in Table 3.4. In addition to commonly used DNN inference operations, they can also be used to accelerate certain backprop operations such as convolutions which insert 0-elements in between every row and column of an input-activation tensor, or convolutions which perform reductions over batch dimensions instead of input-channel dimensions so that the total gradient-step contribution of different images in a mini-batch can be accumulated together.

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 3.11 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 [55]. Later, in Section 3.4.1, 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.

Performance Profiling

To help programmers and architects identify the causes of performance bottlenecks, Gemmini-generated accelerators also included optional performance counters which track the number of cycles that the spatial array, DMA, or various functional units remain idle, and the reason for which they remain idle. For example, Gemmini includes counters that track how many cycles the DMA stalls while waiting for page-table walkers to translate virtual addresses, or how many cycles are spent stalling while waiting for long-latency loads to return from an outer cache miss.

3.2.3 System Support

Gemmini allows architects to integrate RISC-V CPUs with Gemmini-generated accelerators in the Chipyard [1] 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 3.12 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 the multicore, multicache memory system. Later, in Section 3.4.2, 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 mem-

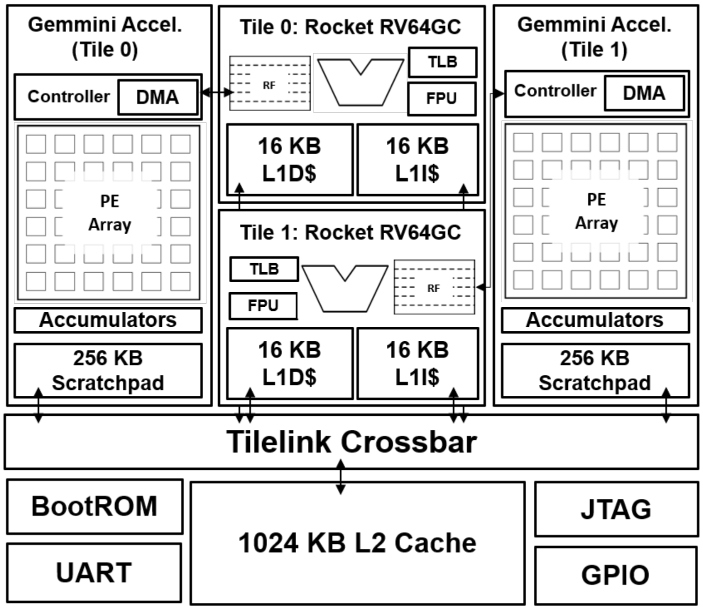


Figure 3.12: Example dual-core SoC with a Gemmini accelerator attached to each CPU, as well as a shared L2 cache and standard peripherals.

ory without the proper permissions. On a “baremetal” environment, these violations were silently ignored.

3.3 Gemmini Evaluation

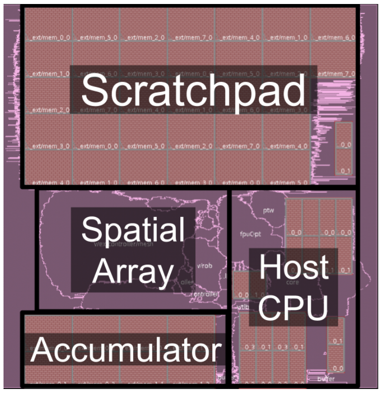
This section discusses our evaluation methodology and evaluation results of Gemmini-generated accelerators compared to both CPUs and state-of-the-art, commercial accelerators.

3.3.1 Evaluation Methodology

We evaluate the end-to-end performance of Gemmini-generated accelerators using the FireSim FPGA-accelerated simulation platform [40]. We evaluate five popular DNNs: ResNet50, AlexNet, SqueezeNet v1.1, MobileNetV2, and I-BERT. All DNNs are quantized to 8-bits, with 32-bit biases and accumulations, and run on a 16×16 spatial array. The CNNs use a 256 KB scratchpad and 64 KB partial sum accumulators, while I-BERT is evaluated with a 64 KB input/weights scratchpad and a 256 KB accumulator. All workloads are run 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 3.13, show that

Component size	**Area (μm^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.

Figure 3.13: Area breakdown and layout of accelerator with host CPU.

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.

3.3.2 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.

As illustrated in Figure 3.14, 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 40.3 frames per second (FPS) for ResNet50 inference when running at 1 GHz, which is a

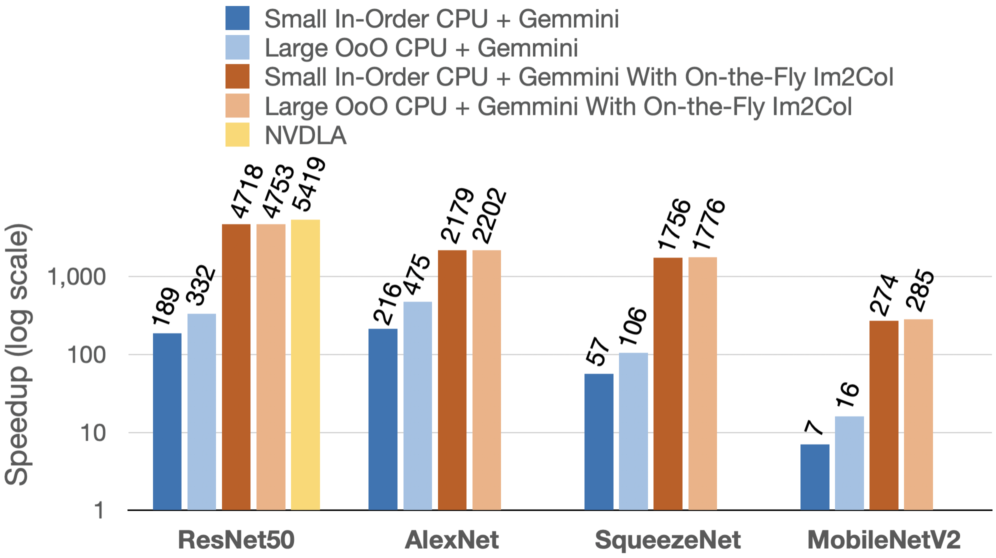


Figure 3.14: Speedup compared to an in-order CPU baseline. For CNNs, im2col was performed on either the CPU, or on the accelerator.

4,720x speedup over the in-order Rocket CPU and an 2,000x 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 255x speedup compared to the Rocket host CPU on MobileNetV2, reaching 37.5 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 3.13a.

The remaining performance gap with prior accelerators such as NVDLA is primarily caused by inefficiencies in Gemmini's DMA. For example, Gemmini's DMA breaks large matrix tiles that are being moved into or out of main memory into smaller DIM×DIM matrices, where DIM is the dimenion of the spatial array; tiling large memory transfers in this way can reduce spatial locality when iterting across the columns of a large matrix tile. More importantly, however, when replicating data into the scratchpad for convolutional layers with few input channels, as illustrated above in Figure 3.10, Gemmini's DMA will perform *multiple* writes to each SRAM row, which bottlenecks the bandwidth at which inputs can be moved from main memory into the private scratchpads. Because convolutional layers with few input channels can consume a significant portion of end-to-end runtime (over 15% of total runtime, in fact, on a ResNet50 inference on Gemmini), such inefficiencies can noticeably degrade end-to-end performance. We expect that by adding a set of simple shift registers to Gemmini's DMA, we will be able to avoid unnecessary writes to scratchpad

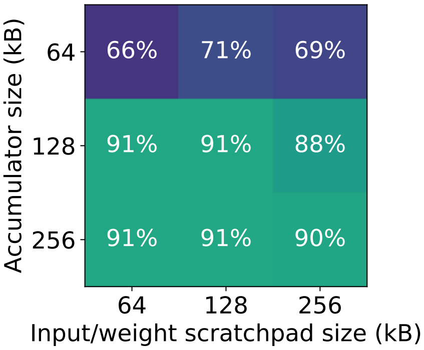


Figure 3.15: The matmul utilization while performing a BERT inference on Gemmini, with different scratchpad and accumulator sizes.

SRAM rows and close most of the remaining 13% performance gap with NVDLA.

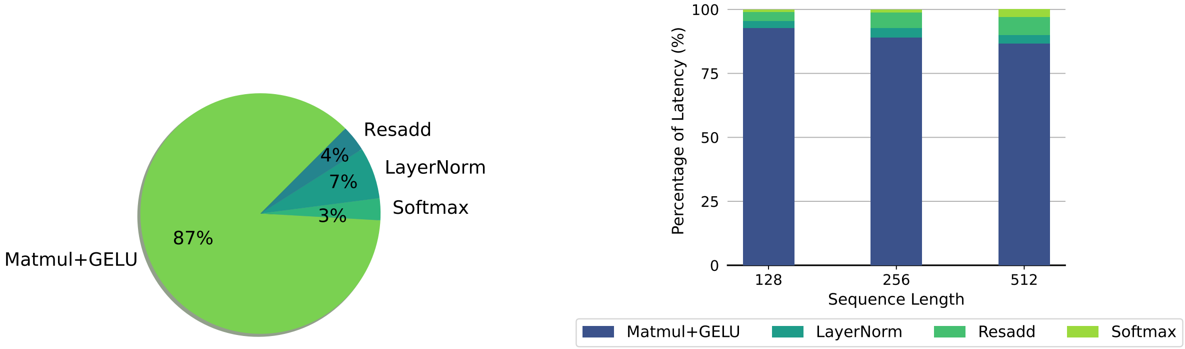
Language models such as BERT or its variants such as I-BERT have lower arithmetic intensity and different data re-use patterns than the CNN models evaluated above; they therefore have different optimal scratchpad and accumulator sizes. Figure 3.15 shows how the PE utilization of the spatial array varies with different scratchpad and accumulator sizes. Larger accumulators enable higher output-reuse which significantly improve the performance of transformer matmuls, which often have smaller, rectangular input matrices being multiplied to generate larger, more square output matrices.

Based on the results shown in Figure 3.15, we equip the Gemmini-generated accelerator with 256 KB of partial sum accumulator storage, and 64 KB of scratchpad space – the exact opposite of the scratchpad/accumulator provisioning ratio that we used for CNNs which had far higher weight or input reuse.

Figure 3.16 illustrates the time spent on different operations when running I-BERT on Gemmini. Matrix multiplications (fused with GeLU operations) consume the vast majority of runtime, with normalization operations like layernorm and softmax consuming the remaining 10%. As sequence lengths increase, the low-arithmetic-intensity, bandwidth-limited normalization operations consume a larger portion of total runtime.

Because layernorm and softmax operations are bandwidth-bound, it is natural to wonder why we did not fuse them with the more compute-bound matmul operations in Figure 3.16. In fact, not only would operator fusion in this case enable better overlapping of operations, but it would also allow Gemmini to avoid spilling 32-bit pre-normalization sums to DRAM, and enable only 8-bit normalized values to be written to DRAM instead, reducing bandwidth requirements for layernorm or softmax operations significantly.

Unfortunately, however, both layernorms and softmax require an *entire* matrix row to be resident in the accumulator for normalization statisics – such as means and variances – to be calculated. If layernorm and softmax operations are fused and fully overlapped with matrix multiplications in transformers, then the matmuls are forced to adopt much more



(a) Sequence-length of 512

(b) With different sequence lengths.

Figure 3.16: The time spent on different operations during a I-BERT inference. For all sequence lengths, the total execution time is dominated by matmuls.

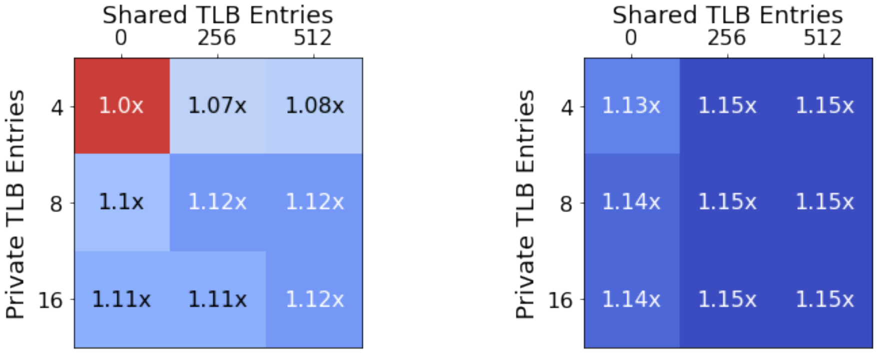
rectangular tile sizes, reducing the arithmetic intensity of each tile significantly and reducing overall inference performance. In fact, I-BERT matmul layers fused with layernorm have 73% lower arithmetic intensity with a sequence-length of 128 than when the layers are left unfused, given 256 KB of partial sum accumulator storage. Gemmini therefore leaves them unfused, sacrificing potential bandwidth savings in the normalization operations in order to maintain performance in the much higher-FLOP matmuls.

3.4 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.

3.4.1 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 [35], which calls for a highly parallel address-translation system with 128 page-table walkers (PTWs), to Cong et al. [27], 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,



(a) Without filter registers.

(b) With filter registers.

Figure 3.17: Normalized performance of ResNet50 inference on Gemmini-generated accelerator with different private and shared TLB sizes.

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 3.17a, 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 3.17a 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 3.17a 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 3.17b 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.

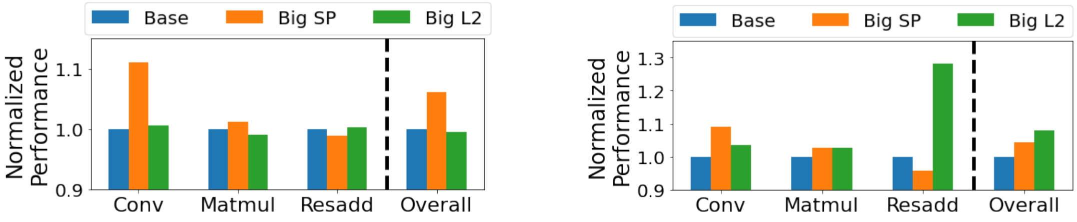
3.4.2 System-Level Resource Partition

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

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

Table 3.5: Gemmini SoC configurations for the system-level resource partitioning case study.



(a) Single-core SoC performance.

(b) Dual-core SoCs performance.

Figure 3.18: Performance of the various SoC configurations in the case study, normalized to the performance of the Base configuration in Table 3.5.

these constraints.

To demonstrate, we run ResNet50 inference on six different SoC configurations. These are the three different configurations described in Table 3.5, repeated for both single- and dual-core SoCs (as in Figure 3.12), 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 3.18a and 3.18b, 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 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 3.18b 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.

3.5 Summary

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, achieving 87% of the performance of prior handwritten accelerators such as NVDLA on workloads such as ResNet50, 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.

However, Gemmini does have certain limitations. For example, although it's parameters expose a broad design space, we do not yet include automated search algorithms to help users to search this space. (Section 5.1 in Chapter 5 discusses this challenge in more detail). Furthermore, Gemmini is limited to dense DNN workloads, such as CNNs like ResNet50 and MobileNet [78], or transformers such as I-BERT [44]; the following chapter describes how we create an accelerator design framework which helps cover the sparse accelerator space as well.

Chapter 4

Stellar: Designing and Synthesizing Accelerators

4.1 Introduction

In response to diminishing technology scaling trends and the growing computational demands of modern workloads, computer architects have increasingly turned to domain-specific accelerators and co-designed software optimizations for improved energy and area efficiency. However, the expanding landscape of diverse workloads has given rise to a large multitude of hardware designs and software co-optimization techniques. These diverse solutions range from fixed-function matrix-multiplication arrays for dense DNNs [22, 63, 18, 37, 61], to co-designed structured sparsity formats that remove low-priority weights or features from large DNN models [105, 64, 51, 88], to accelerators for extremely sparse, highly-imbalanced tensor operations [66, 104, 102, 82, 24, 13].

While this broad spectrum of hardware and software optimizations presents ample opportunities for accelerator and software co-design, it also greatly complicates the analysis, exploration, and design of specialized architectures. Prior work has attempted to address these challenges through methods ranging from ad-hoc hardware design to proposals for well-defined, expressive accelerator taxonomies. Ad-hoc design, a traditional and flexible approach, provides limited opportunities for automated and rapid design space exploration. Conversely, techniques like high-level synthesis enable swift hardware development via direct compilation from software to hardware, but fail to maintain a strong separation of concerns in the hardware design process, making it difficult to explore independent design choices without modifying unrelated parts of an architect's specifications. Recent efforts have introduced expressive abstractions and taxonomies in an attempt to disentangle various components of accelerator design. However, these approaches often fail to generate synthesizable sparse hardware designs, focusing predominantly on higher-level modeling, or they struggle to describe low-level aspects of hardware design that are of interest to architects.

In response to the complexity posed by the diverse landscape of accelerator design, we

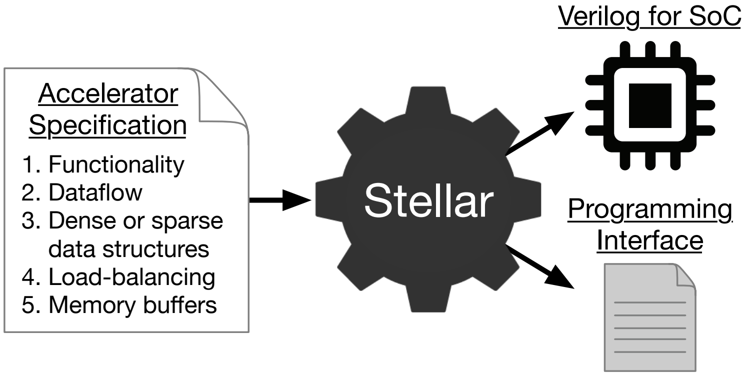


Figure 4.1: A simplified illustration of Stellar's accelerator specification and hardware generation process, from the user-specified inputs on the left to the Verilog and programming interface outputs on the right.

develop "Stellar", a new accelerator design framework for automated dense and sparse spatial accelerators. Stellar addresses these challenges by introducing three key features: (i) it provides expressive abstractions for the design of *both* dense and sparse accelerators, (ii) it maintains a strong separation of concerns between different design considerations, allowing independent specification and exploration, and (iii) it generates synthesizable Verilog implementations of user-specified hardware, together with RISC-V programming interfaces that can easily be incorporated into users' software applications. Stellar's design flow, summarized in Figure 4.1, enables the rapid development and generation of accelerators for both dense and sparse workloads.

Building upon prior taxonomies for dense and sparse accelerators [69, 49, 11, 95, 62], Stellar introduces new abstractions of interest to hardware developers, such as fine-grained load-balancing schemes and pipelining strategies. Users independently express a specialized hardware accelerator's (i) functional behavior, (ii) dataflow, (iii) supported sparsity patterns, (iv) load-balancing strategy, and (v) private memory buffers. They can modify these different design considerations in isolation and observe the subtle interactions between them to determine the best accelerator design choice.

Stellar maps these design specifications to hardware templates, and optimizes them to maximize data reuse and minimize area and wiring congestion. Stellar then outputs synthesizable Verilog implementations of a full SoC, including user-specified accelerators, optional host CPUs, and shared memory hierarchies. Our evaluation demonstrates that Stellar-generated hardware implementations perform competitively to hand-designed accelerators, and that they effectively expose various performance bottlenecks caused by either hardware or software design choices, which often cannot be exposed in high-level simulators or models. We provide an end-to-end, unified platform for both dense and sparse accelerator design, enabling systematic evaluation and comparison of diverse architectural design choices.

4.2 Specifying Accelerators in Stellar

Stellar is composed of a specification language that describes spatial accelerators, dense or sparse, and a compiler that translates these descriptions into Verilog. The specification language is designed to maximize an architect's separation of concerns when designing an accelerator; each of the five subsections below describes a separate design concern which users can specify and explore independently, even if the full impact of one design concern on the overall performance of the accelerator may depend partially upon another design concern.

4.2.1 Functionality

Stellar users specify the functionality of their accelerator with a Halide-like notation which has been used in prior work for dense accelerator design [75, 101]. The functional notation defines various tensor inputs and outputs, and how the outputs are calculated from the inputs.

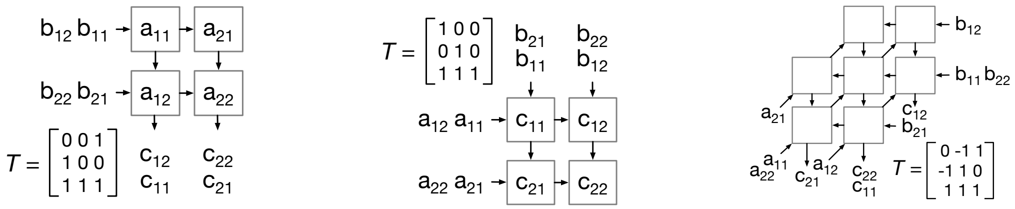
Consider, for example, Listing 4.1, which illustrates how a Stellar-user may define the functional behavior of a matrix-multiplication accelerator. We will refer back to this example repeatedly throughout this paper:

Listing 4.1: Functional behavior of a matmul accelerator with indices i, j, and k.

```
// Inputs
a(i, j.lowerBound, k) := A(i, k)
b(i.lowerBound, j, k) := B(k, j)
c(i, j, k.lowerBound) := 0
// Intermediate calculations
a(i, j, k) := a(i, j-1, k)
b(i, j, k) := b(i-1, j, k)
c(i, j, k) := c(i, j, k-1) +
      a(i, j-1, k) * b(i-1, j, k)
// Outputs
C(i, j) := c(i, j, k.upperBound)
```

Unlike an iterative for-loop, Stellar's Halide-like notation involves no state-mutations, and makes no assumptions about the order, time, or place of each multiply-accumulate operation within a hardware unit. The i, j, and k indices exist only in what we call the "tensor iteration space," and do not directly correspond to time or space coordinates on a physical hardware accelerator. Neither does the functional notation make assumptions about the sparsity distributions or sparse data formats of the input or output tensors. The following subsections describe how we specify such design considerations.

In addition to arithmetic operations, Stellar's functional notation also supports data-dependent accesses to the input and output tensors, which are sometimes useful for specifying merging and sorting algorithms for sparse workloads in particular. Output mergers



(a) Input-stationary dataflow (b) Output-stationary dataflow (c) Hexagonal dataflow

Figure 4.2: Examples of space-time-transforms (each named T) and the dense matmul dataflows that result from them.

and coordinate-matchers are necessary for many sparse accelerators, where they sometimes consume more area or power than the MAC arrays themselves [104]. For example, consider the Stellar code sample in Listing 4.2 below, which demonstrates how data-dependent Stellar code can be written for the shared-coordinate-matching unit of an inner-product sparse matmul accelerator:

Listing 4.2: Functional behavior of a matmul accelerator with indices i, j, and k.

```
matches(n) := A(ka(n)) == B(kb(n))
ka(n) := ka(n) + (A(ka(n)) < B(kb(n)))
kb(n) := kb(n) + (A(ka(n)) > B(kb(n)))
```

Stellar can therefore be used to construct a full pipeline for both sparse and dense accelerators, including functional units for arithmetic reductions and data-dependent pre- or post-processing. The functionality of these hardware units can also be specified independently of their dataflows, sparsity formats, load-balancing strategies, or private memory buffers.

Finally, although Stellar's functional provides a set of commonly used arithmetic operators, such as additions, multiplications, comparisons, bitshifts, or boolean operators, we also allow users to implement their own operators in raw Chisel, as shown in Listing 4.3. This extensibility helps support use cases where users wish to operate on custom datatypes such as bfloat16 [38], or perform complicated bit-level accesses which may be awkward to compose out of our pre-defined operators.

Listing 4.3: Custom floating-point MAC, in raw Chisel with the Hardfloat library [28], for Stellar's functional specification.

```
/* Specify the nputs and outputs for this custom
   operation, as in lines 8-9 of Listing 4.1 */
c(i,j,k) := Custom(ops=Seq(a(i,j-1,k), b(i-1,j,k), c(i,j,k-1)),
  function = { ops =>
    val Seq(a_bits, b_bits, c_bits) = ops

    /* A Chisel implementation of a multiply-addr, from
       the open-source HardFloat library */
    val muladder = Module(new MulAddRecFN(expWidth, sigWidth))

    muladder.io.op := 0.U // Multiply-accumulate

    /* HardFloat supports multiple rounding modes,
       but round-near-even is typically best for
      DNN workloads. */
    muladder.io.roundingMode := consts.round_near_even

    muladder.io.a := a_bits
    muladder.io.b := b_bits
    muladder.io.c := c_bits

    muladder.io.out
}, name="Mac")
```

4.2.2 Dataflow

Users define the dataflow of their accelerator by specifying a linear transformation (represented by an invertible matrix) from the tensor iteration space described above to physical space and time coordinates on a spatial array. Following the example of prior work [42, 50, 74, 75, 101], we call this linear transformation a “space-time transform”. For example, for the matmul example in Listing 4.1, the space-time transform would be T in the equation below:

$$T \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} x \\ y \\ t \end{bmatrix} \tag{4.1}$$

where T is a 3×3 invertible matrix, x and y are space coordinates, and t is a timestep. Every input, output, and intermediate MAC operation in the matmul in Listing 4.1 is mapped by T to a specific place and time on a two-dimensional physical spatial array with x rows and y columns. For example, if T is the identity matrix, then a MAC that takes place when $i = 1$, $j = 2$, and $k = 3$ would be mapped to the PE at position $(x = 1, y = 2)$ (i.e. row-1 and column-2 in the spatial array), and would occur when the time-step, t, equals 3.

Figure 4.2 illustrates various space-time transforms for matmuls and the spatial arrays that result from them. Note that by simply changing numerical values in the T matrix, users can create a wide variety of spatial arrays, including input-stationary, output-stationary, and hexagonal [4] designs.

Each of these space-time-transforms represents a separate *dataflow*, covering a *superset* of the dataflows proposed by some other dataflow-classification schemes which are instead

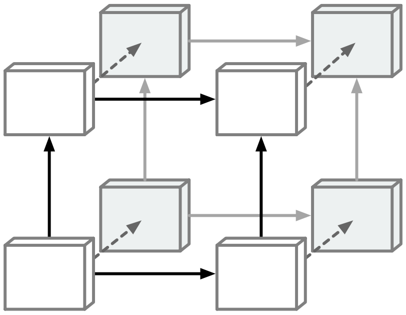


Figure 4.3: A three-dimensional spatial array generated by Stellar.

defined in terms of which tensor iterators are spatially or temporally unrolled [97], or by which inputs or outputs remain stationary during execution [7]. For example, a dataflow-classification scheme which only allows users to decide which iterators to spatially unroll could only produce 3D spatial arrays, as in Figure 4.3, if the user wanted to unroll all three indices (i, j, and k) in Listing 4.1. Stellar can express 3D arrays as well, but it can also express more niche spatial arrays that such dataflow-classification schemes cannot, such as the hexagonal array in Figure 4.2c which spatially unrolls all three indices onto a *2D plane*, yielding shorter wires which may be easier to route.

Stellar's dataflow specifications also give hardware designers more fine-grained control over lower-level hardware design decisions, such as the number of pipeline registers to place across different axes of the spatial array. Figure 4.4 illustrates how changing individual values in the lowest row (the *time* axis) of the dataflow-specification matrix T creates designs that are more or less aggressively pipelined.

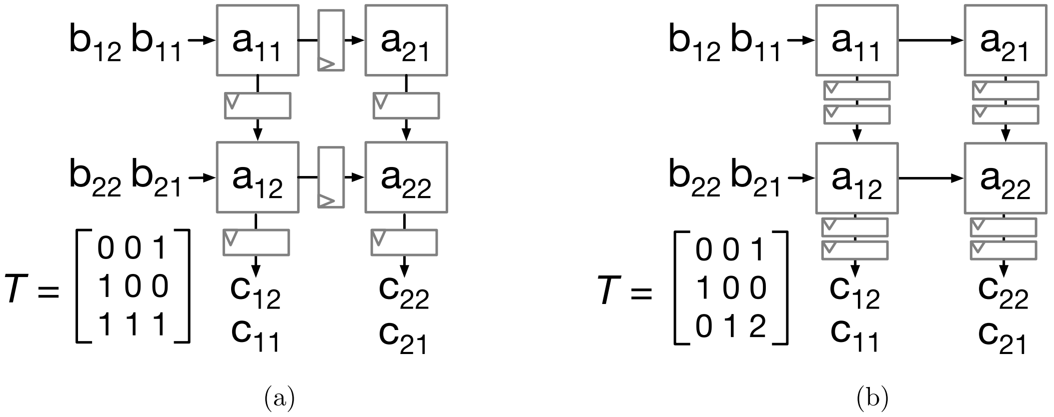
Stellar uses the dataflow specified by the user's space-time transform to construct "baseline" dense accelerators which maximize PE-to-PE data re-use, as in Figure 4.2. Later, Section 4.3.2 describes how these baseline spatial arrays are modified to skip zero-values in sparse workloads, based on the sparsity specifications given in the following subsection.

4.2.3 Sparse Data Structures

For sparse accelerators, the sparse data structures of the input and output tensors are expressed in Stellar in terms of which iterators in the tensor iteration space may be "skipped" and under which conditions they may be skipped. For example, consider the following sparsity structures we define for the matmul example introduced in Listing 4.1:

Listing 4.4: Specifying sparse data structures in Stellar.

```
// A*B=C where A and B are CSC/CSR
Skip i when A(i,k) == 0
Skip j when B(k,j) == 0
```



(a)

(b)

Figure 4.4: Different pipelining strategies for the input-stationary matmul accelerator in Figure 4.2a.

```
// A*B=C where A is diagonal
Skip i and k when i != k

// A*B=C where rows of A may be all 0
Skip k when A(i,->) == 0
```

Note that in Listing 4.4, we do not specify how exactly the tensors are stored in memory, or what metadata is associated with them; these details are irrelevant to the spatial array design. Listing 4.4 only specifies which tensor elements are skipped; e.g. whether we skip elements along rows, as in the CSR format, or along columns, as in the CSC format. By contrast, later subsections describe how users specify how tensors are actually stored and encoded in memory.

Once a sparsity structure is specified, Stellar determines which PE-to-PE connections in the baseline dense spatial array (as illustrated in Figure 4.2) are no longer *guaranteed* to transmit useful non-zero values in every single cycle. Under the assumption that these PE-to-PE connections are unlikely to carry useful data (as when the total non-zero density of a tensor is very low), Stellar removes these PE-to-PE connections and replaces them with IO connections that access the input- or output-tensors directly from outer register files.

For example, Figure 4.5 illustrates how the input-stationary matmul array in Figure 4.2a will look after the user specifies that the input-B matrix has the CSR format, causing Stellar to remove the vertical PE-to-PE connections which were previously being used to accumulate partial sums. Section 4.3.2 describes in further detail how Stellar calculates exactly which PE-to-PE connections to remove.

However, for some forms of structured sparsity, PE-to-PE connections are still valuable even if some of the data they carry will only be useful to a small number of PEs. For example, Figure 4.6 illustrates a Stellar-generated spatial array implementing NVIDIA's

A100 structured sparsity scheme [64] for matmuls, where two out of every four adjacent DNN weights are zeros. Some of the nonzeros will be used for useful computations in any particular cycle, while others will simply be forwarded through the array. To support such sparsity structures, Stellar provides the `OptimisticSkip` keyword, which, unlike the `Skip` keyword, does not remove PE-to-PE connections, but replaces them with wires that carry small bundles of potentially useful data, rather than scalar values.

4.2.4 Load-Balancing

Spatial array workloads are oftentimes extremely imbalanced, causing some PEs to idle while other PEs are performing useful arithmetic operations. Prior work proposes a wide variety of hardware techniques to redistribute work from over-burdened PEs to idle PEs at runtime [23, 24].

Stellar allows users to specify whether they want computations that would normally take place in certain regions of the tensor iteration space to be shifted towards other "target" iterations, but only if the target iterations would be idle otherwise. For example, consider the following load-balancing strategy for a sparse matmul described using Stellar's notation:

Listing 4.5: A simple load-balancing scheme in Stellar

```
Shift (/*i = */ N -> 2*N, j, k) to
      (/*i = */ 0 -> N, j, k+1)
```

where the tensor iterators, i, j, and k, are the same as those introduced for the matmul in Listing 4.1. For any iterator value $k = K$, if the target matmul iterations, where $0 \leq i < N$, are *all* idle due to a workload imbalance, then Stellar will shift future work that has not yet

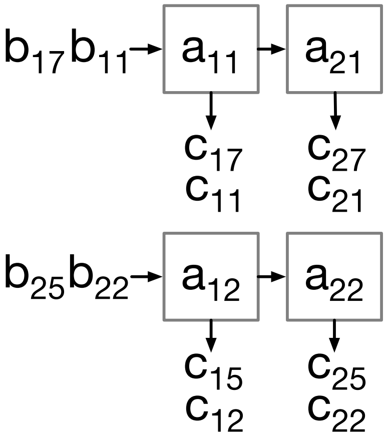


Figure 4.5: The input-stationary matmul array from Figure 4.2a after the B-matrix is specified as a sparse CSR matrix.

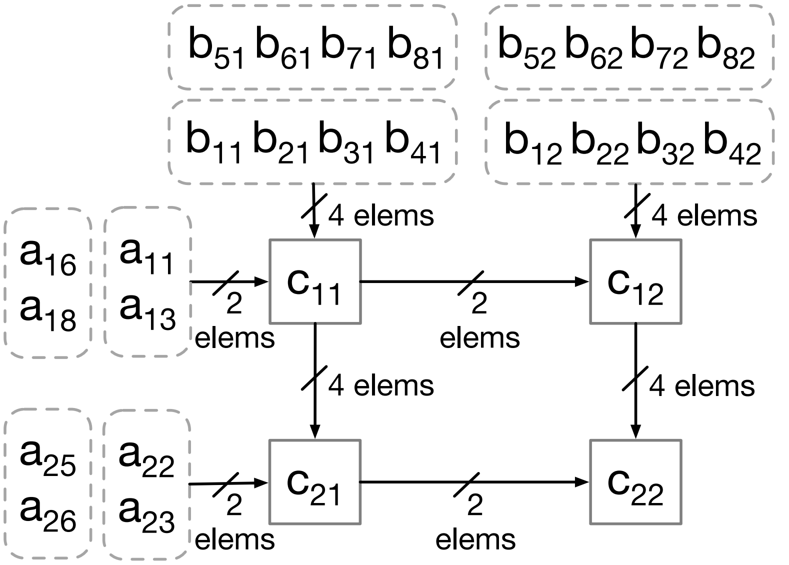
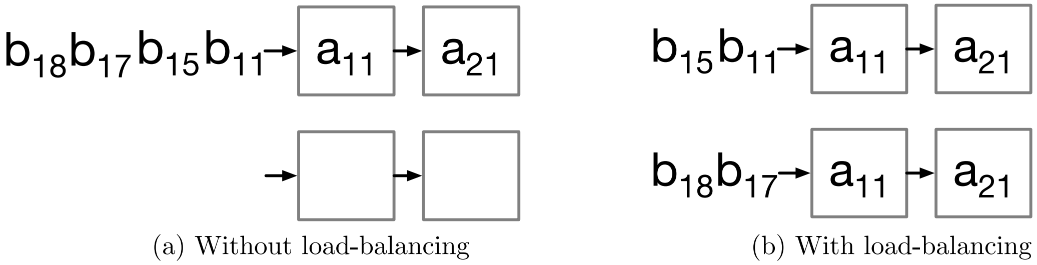


Figure 4.6: The output-stationary matrix from Figure 4.2b when the A-matrix conforms to the A100 2:4 sparsity format [64].

begun where $k = K + 1$ and $N \leq i < 2N$ onto the idle PEs, reducing the future workload of PEs which are currently busy.

Based on the exact dataflow specified, this might mean, for example, that adjacent rows of the spatial array can share work, but *only* if they are directly adjacent. Figure 4.7 illustrates a scenario where this happens. More flexible load-balancing schemes can share work across broader sets of PEs, but, as detailed later in Section 4.3.2, they may require Stellar to generate hardware with greater area and wiring congestion.

More fine-grained and sophisticated load-balancing schemes can also be specified. For example, some sparse accelerators, such as AWB-GCN [23], will give specific PEs the ability



(a) Without load-balancing

(b) With load-balancing

Figure 4.7: The sparse matmul array from Figure 4.5, executing an imbalanced B-matrix with and without load-balancing.

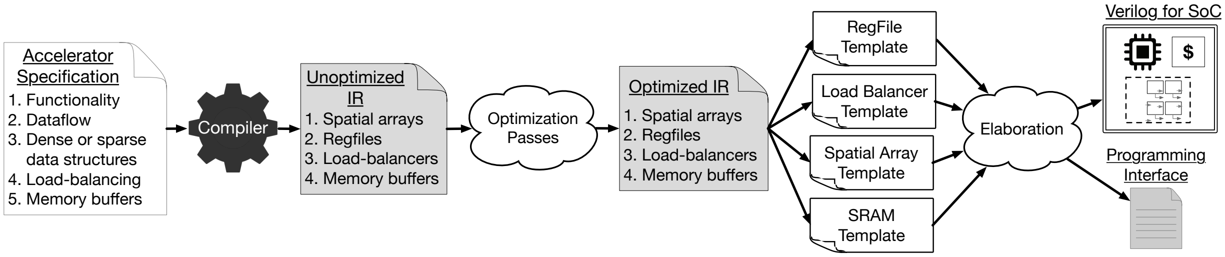


Figure 4.8: An overview of the hardware generation process for Stellar, from the initial architectural specification, to the unoptimized and optimized IRs, to the final Verilog and programming interface outputs.

to take work from any other overburdened PE, while other PEs can only share work with their close neighbors. To help support such strategies, Listing 4.6 shows an example load-balancing scheme where only iterations corresponding to a small subset of PEs will take work from other PEs:

Listing 4.6: Very flexible load-balancing for a limited set of PEs.

```
Shift (i,j,k) to (/*i=*/0, /*j=*/0->4, k)
```

Finally, note that Stellar's notation for specifying load-balancing schemes specifies which operations should be remapped only in the tensor iteration space (as described in Section 4.2.1). Therefore, users can express and explore different load-balancing strategies completely independently of the dataflow or sparsity structures themselves.

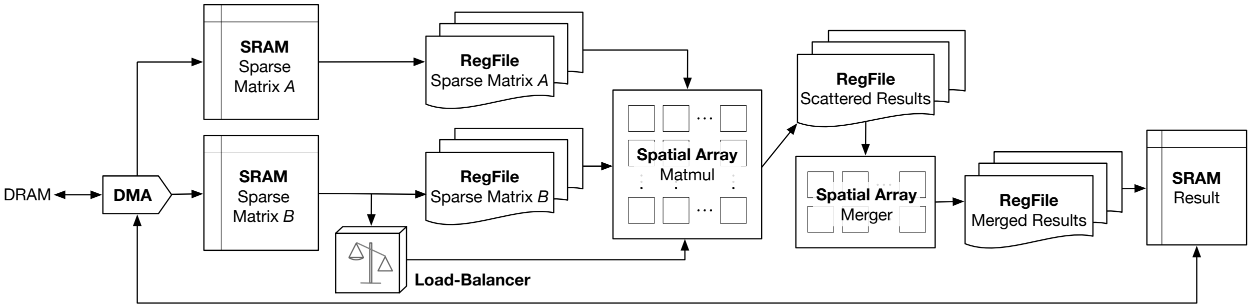


Figure 4.9: Hardware architecture overview for an example sparse matrix-multiplication accelerator.

4.2.5 Private Memory Buffers

To generate scratchpad memories and private memory buffers for their accelerators, Stellar users must specify the specific dense or sparse data formats they will support. To specify such data formats, we use the fibertree notation from prior work [84], where users specify a different dense/sparse format for every axis (i.e. dimension) of a tensor. For example, the CSR format would be specified by setting the outer-axis of a two-dimensional matrix to the *Dense* uncompressed format, while the innermost axis would be *Compressed*, composed of a list of coordinates and data. Stellar supports other formats as well, such as *Linked-Lists*; by composing these formats to different dimensions of a tensor, a wide variety of unique sparse tensor formats can be defined.

Note that the private memory buffer data formats do not necessarily have to conform exactly to a spatial array's expected sparse data structures (described above in Section 4.2.3). For example users can define spatial arrays that are completely dense connected to private memory buffers which store sparse data. The dense arrays will preserve short, low-overhead connections between PEs, unlike sparse spatial arrays which must often replace PE-to-PE connections with longer, more expensive, and more congested wiring to outer memory units. The sparse private memory buffers, meanwhile, can store sparse data in order to reduce DRAM bandwidth requirements. By allowing independent specification of both spatial array and memory buffer parameters, Stellar enables such designs to be explored and evaluated.

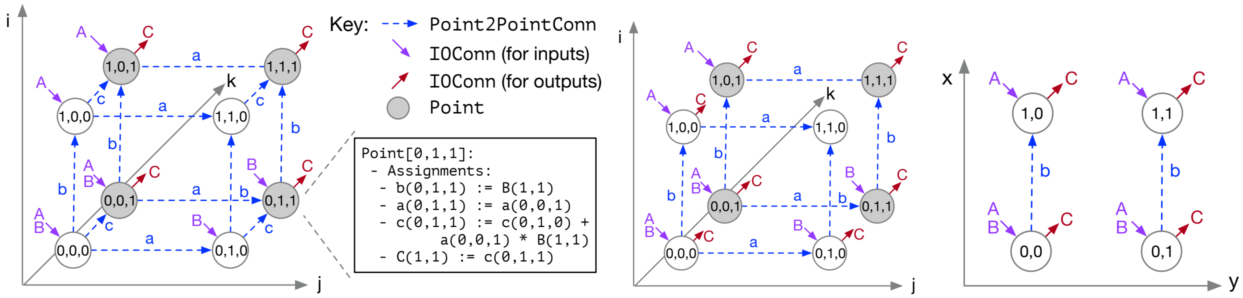
4.3 Hardware Generation in Stellar

After an accelerator's functionality, dataflow, sparse data structures, load-balancing schemes, and private memory parameters are specified in Stellar, our compiler elaborates these into an IR which represents a set of spatial arrays, register files, SRAMs, and load-balancers. These are optimized based on data-access patterns which can be determined at elaboration time, and the optimized IR is mapped to a set of Chisel [3] templates which are lowered into Verilog and RISC-V programming interfaces. Figure 4.8 illustrates this full hardware generation process, which is described in further detail in this section.

4.3.1 Architectural Overview

Stellar-generated accelerators are composed of spatial arrays, register files, private memory buffers, (optional) load-balancers, and a DMA. Figure 4.9 shows the overall hardware architecture of an example Stellar-generated accelerator that performs sparse matrix multiplications and merges the scattered partial sums into merged matrix results.

Spatial arrays perform compute operations such as matrix multiplications or the merging of scattered output results. These spatial arrays read and write their input and output tensors to register files, which may themselves be populated by or emptied into larger private memory buffers. Load-balancers monitor the regfile inputs and outputs to determine whether PEs will be idle or over-utilized. Finally, a DMA transfers tensors between off-chip DRAM or outer caches and the accelerator's private memory buffers. The following subsections describe how the aforementioned hardware components are generated and optimized by Stellar.



```
Point[0,1,1]:
 - Assignments:
  - b(0,1,1) := B(1,1)
  - a(0,1,1) := a(0,0,1)
  - c(0,1,1) := c(0,1,0) +
         a(0,0,1) * B(1,1)
  - C(1,1) := c(0,1,1)
```

(a) After functional description. (b) After sparsity pruning. (c) After dataflow.

Figure 4.10: The internal representation, called an `IterationSpace` for a spatial array performing a matmul as in Listing 4.1 as it is transformed from a purely functional description to a physically realizable two-dimensional spatial array.

4.3.2 Generating Spatial Arrays

Stellar initially constructs *dense* spatial arrays, based on the functional description of the accelerator and its dataflow. The PEs of these spatial arrays will request inputs or issue outputs when their physical coordinates and current time-step correspond to the indices these input/output operations are supposed to occur at. For example, the PE output on line 11 of Listing 4.1 occurs whenever the tensor iterator k is at its maximum value, `k.upperBound`. By multiplying a PE's space-time coordinates, (x, y, t) as in Equation 4.1, by the inverse of the space-time transform, T^{-1}, we can find the exact point in the tensor iterator space to which the current space-time coordinates correspond. If the value of k at that point is `k.upperBound`, then the PE output will be issued.

Figure 4.10a illustrates how an example spatial array is initially represented in the Stellar compiler's internal IR, based purely on the functional description in Listing 4.1, before the dataflow or sparsity specifications are applied to it. Stellar refers to this IR as an `IterationSpace`, where every `Point` within the `IterationSpace` corresponds to a different unique set of values for the tensor iterators (i, j, k). Furthermore, the `IterationSpace` includes a set of `Point2PointConns` (point-to-point connections) describing data dependencies between different points, and a set of `IOConns` (IO connections) representing input- or output-requests to external register files (described later in Section 4.3.4). Finally, every `Point` has a set of `Assignments` representing the different arithmetic operations, such as multiply-accumulates or variable initializations, that happen at a specific `Point`.

The baseline, dense spatial array that is initially constructed by Stellar is modified based on the user's sparsity structure specifications.. Dense spatial arrays typically achieve high data reuse by sharing data through PE-to-PE connections. However, sparse workloads often have much less data reuse, rendering many of these PE-to-PE connections obsolete. Stellar will remove the PE-to-PE connections which are no longer guaranteed to carry useful data between PEs, and replace them with direct connections to outer regfiles, as seen by the change between Figure 4.2a and Figure 4.5.

For an example of how Stellar determines which connections to remove, consider the accumulation of partial sums on lines 8-9 of Listing 4.1. We see that a PE at point (i, j, k) in the tensor iteration space computes the multiply-accumulate-sum, $c(i, j, k)$, based on $c(i, j, k-1)$, which means that the "difference vector" [101] for the variable c is $(\Delta i = 0, \Delta j = 0, \Delta k = 1)$. Multiplying the input-stationary space-time-transform T in Figure 4.2a by the difference vector yields the spacetime difference vector $(\Delta x = 1, \Delta y = 0, \Delta t = 1)$, which indicates that the partial sums travel vertically down the spatial array every time-step.

Now, suppose that B is in the CSR format, as in Listing 4.7:

Listing 4.7: Making the B-matrix CSR

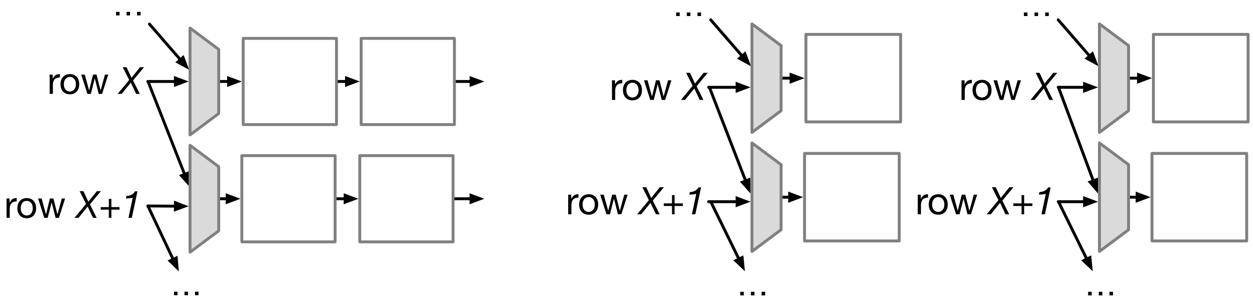
```
Skip j when B(k,j) == 0
```

In the CSR sparse format, finding the j-coordinate would require a series of indirect lookups and pointer arithmetic. Stellar abstracts these lookups away by expressing the "expanded" j-coordinate as some arbitrary function f whose inputs are k and the *compressed* j-coordinate: $j_{expanded} = f(k, j_{compresssed})$. Therefore, the difference vector for c now becomes $(\Delta i = 0, \Delta j_{expanded} = f(k, j_{compressed}) - f(k-1, j_{compressed}), \Delta k = 1)$. Because the j-component depends on indirect data lookups and can no longer be simplified into a scalar constant, Stellar can no longer assume that the partial sums will be unconditionally accumulated vertically across the spatial array. The corresponding vertical PE-to-PE connections are therefore removed, yielding the matmul array in Figure 4.5 with fewer PE-to-PE connections, and a larger number of output ports to outer register files. Figure 4.10b illustrates how the Stellar compiler's internal representation of a spatial array, an `IterationSpace`, appears after its `Point2PointConns` are pruned based on the equations described above.

Load-balancing schemes can also affect the design of spatial arrays. Figure 4.11b illustrates a load-balancing strategy where *any* PE within a row can operate on data that would otherwise have been sent to the upper row, if that PE would have otherwise been idle. Each PE can independently be redistributed work from the above row, and therefore it might no longer receive useful data along its horizontal PE-to-PE connections. Therefore, since horizontal PE-to-PE connections might no longer transmit the necessary inputs, Stellar must replace them with connections to outer regfiles. Contrast this to Figure 4.11a, where load balancing operates at the granularity of an entire row of PEs, preserving the horizontal connections.

Different sparsity and load-balancing schemes can therefore significantly impact the area and power overheads of the spatial arrays, by reducing the efficiency gains of cheap PE-to-PE communication that can be fully exploited when workloads are dense. Stellar's strong separation of concerns enables users to flexibly specify various sparse data structures and load-balancing schemes, and evaluate their impact on hardware.

After `Point2PointConns` have been fully pruned and replaced with `IOConns` based on the sparsity and load-balancing specifications, the space-time transform describing the dataflow (Section 4.2.2) is applied to generate a new `IterationSpace`, as illustrated in Figure 4.10c. Each `Point` in this new `IterationSpace` corresponds to a different PE in the final generated



(a) Balancing across entire rows

(b) Balancing across each PE

Figure 4.11: The effect of more or less flexible load-balancing strategies on PE-to-PE communication.

spatial array; multiple `Points` in Figure 4.10b may therefore map to the same `Point` in Figure 4.10c if they represent different operations which happen at different timesteps on the same PE. Therefore, this final transformation is not one-to-one, but many-to-one.

As described above, Stellar's `IterationSpace` supports affine linear transformations and the pruning of `Point2PointConns`; however, it does not yet support more complicated dataflow graph transformations such as transpositions, folding, or interleaving. We leave such transformations to future work.

Finally, every `Point` in Figure 4.10c is mapped to a highly generalizable Chisel template of a PE, shown in Figure 4.12. Every `Assignment` associated with the `Point` is translated to Chisel in the "User-Defined Logic" block. For a matrix-multiplication spatial array, this would consist primarily of multiply-accumulate operations. However, a single PE may perform different operations in different time-steps; for example, the variable c is initialized to 0 on line 4 of Listing 4.1, but is thereafter accumulated every subsequent cycle on line X.

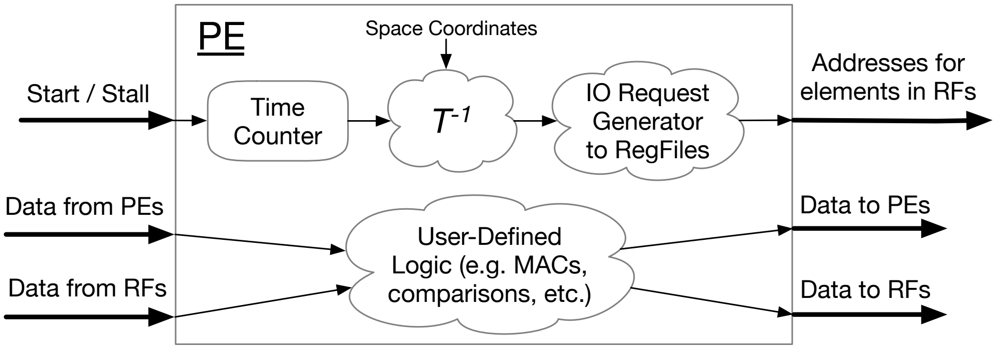

PE
Space Coordinates
Start / Stall
Time Counter
T^{-1}
IO Request Generator to RegFiles
Addresses for elements in RFs
Data from PEs
Data from RFs
User-Defined Logic (e.g. MACs, comparisons, etc.)
Data to PEs
Data to RFs

Figure 4.12: The architecture for a Stellar PE.

To calculate *which* operation to perform at a specific time-step, every PE includes a "Time Counter" register; when concatenated to the physical coordinates of the PE, a space-time vector, (x, y, t), can be generated and multiplied by the inverse of the space-time transform, T^{-1} to generate the original tensor iterators (i, j, k). Based on these tensor iterators, the exact operation to perform can be determined at runtime by the PE. Input- and output-requests to outer register files, each one corresponding to a different `IOConn`, are generated in the "IO Request Generator" when the tensor iterators match the values specified in the functional description in Listing 4.1. The PEs are then connected to each other if they have `Point2PointConns` in the spatial array's `IterationSpace`, as in Figure 4.10c.

4.3.3 Generating Private Memory Buffers

As described in Section 4.2.5, Stellar users specify the dense or sparse data formats, capacities, banks, and read/write bandwidths their private memory buffers will support. The data formats are defined using the fibertree notation, where different dense/sparse formats are defined for every axis (i.e. dimension) of a tensor.

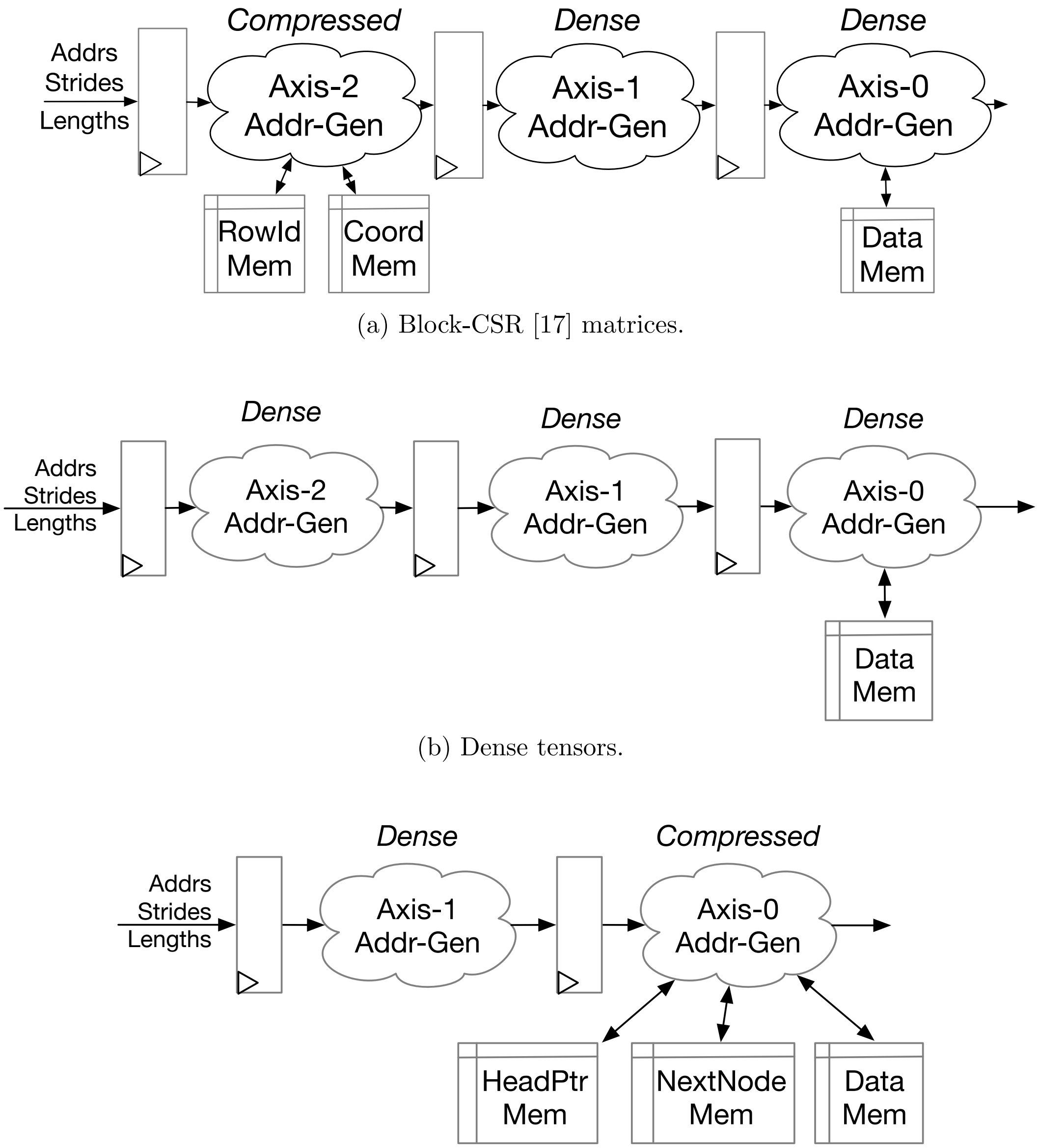
Stellar then generates multiple pipeline stages — one for each axis of the dense or sparse tensors that the buffer stores — which read/write requests made by programmers pass through. *Dense* axes generate simple address generators, while *Compressed* or *Linked-List* axes may require indirect lookups to SRAMs which store metadata to determine the final data addresses to read or write to. Figure 4.13a shows example pipeline stages generated for a private memory buffer holding tensors in the block-CSR [17] format.

Reads and writes each pass through separate pipeline stages, and Stellar-generated SRAMs are always dual-ported so that reads and writes can happen simultaneously. Only write pipelines will every write data to an SRAM, but certain metadata buffers, such as the `RowId` pointers in compressed axes, or the `NextNode` pointers in linked-list axes (illustrated in Figure 4.13c), may need to be read by both read and write pipelines. The SRAMs storing these metadata buffers therefore have round-robin arbiters that allow both read and write pipelines to share access to their read-ports.

Tiling

For every read or write request, the programmer must specify at runtime the address, length, and data and/or metadata strides for each axis. Because dense axes are implemented as basic address generators, Stellar users may wish to add extra dense axes to an SRAM to allow more complex striding or tiling patterns. For example, as in Figure 4.13b, a third outer dense axis might be added to a memory buffer storing a two-dimensional matrix, where the outermost dense axis is simply used to tile or block submatrices out of the larger 2D matrix.

However, simply adding an outer dense axis is sometimes not optimal when tiling imbalanced sparse matrices. Consider, for example, a workload where the user must tile small, two-dimensional submatrices out of a larger, sparse, CSR matrix with imbalanced rows, as



(a) Block-CSR [17] matrices.

(b) Dense tensors.

(c) Two-dimensional matrices where the rows are linked-lists.

Figure 4.13: The read/write pipeline stages for a private memory buffer storing tensors with different dense and sparse data formats.

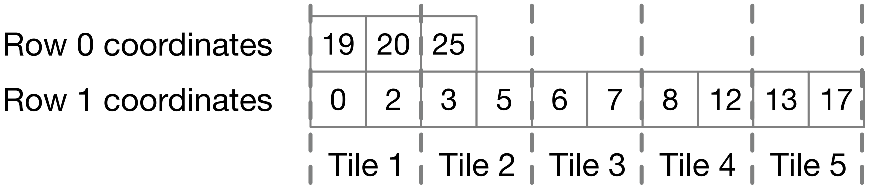


Figure 4.14: Tiling two-dimensional 2×2 matrices out of an imbalanced CSR matrix.

in Figure 4.14. We could tile it naively as in Listing 4.8 (which, in Stellar, would correspond to adding an outer dense axis):

Listing 4.8: Tiling a CSR matrix.

```
1 for (int tile_id = 0; tile_id < N_TILES; tile_id++) {
2   for (int row = 0; row < N_ROWS; row++) {
3     int start = RowId[row] + tile_id * TILE_SIZE;
4     int end = min(RowId[row+1], start + TILE_SIZE);
5     for (int ptr = start; ptr < end; ptr++) {
6       int data = CsrData[ptr];
7       int coord = CsrCoords[ptr];
8       // Perform computations on ‘data’ and ‘coord’...
9 }}}
```

However, this approach would result in many unnecessary reads of the `RowId` buffer in line 3 in order to re-determine the length of a row whose length had already been checked in the past. In fact, if the rows of the matrix being tiled are highly imbalanced, then many cycles may be spent checking the lengths of rows in line 3 which have already been entirely iterated over, leading to bubbles in the outputs from the memory buffer. To optimize for such tiling scenarios, Stellar users can parameterize their memory buffers to cache prior `RowId` reads or accesses to other pointers or metadata buffers, and only read data from rows of the matrix which have not yet finished being output.

Finally, naive tiling of sparse, imbalanced matrices sometimes fails to account for the fact that some regions of a tensor — for example, certain rows of a CSR matrix – may be consumed by a spatial array at a faster rate than others, at different points during runtime. For example, observe that the all the elements in row 0 in Figure 4.14 happen to have larger indices than all the elements in row 1. Suppose a Stellar user generates a spatial array which merges the different rows of a CSR matrix into a single sorted row. All the elements of row 1 will have to be read out before any elements of row 0 can be consumed; however, simple interleaving of rows in a round-robin fashion would cause elements of row 0 to be issued from the memory buffer even when they cannot be consumed by the spatial array. To optimize for such scenarios, Stellar's private memory buffers can also optionally monitor the number of elements that are currently resident in an output register file which have not yet been consumed by a spatial array, and the memory buffers will then interleave the rows (or other dimensions) that they output to prevent any row from occupying too much idle space in the register file.

Banking

The private memory buffers can be banked to support multiple simultaneous reads or writes from different addresses. As shown in Figure 4.15, banking causes the read and write pipelines to be duplicated; each read/write pipeline bank may then perform independent reads/writes to similarly banked SRAMs storing tensor data or sparse metadata.

The read/write pipeline banks are independent of the banking of the SRAMs; for example, users may create private memory buffers with four pipeline banks, but 16-banked SRAMs. Every pipeline bank has access to *all* SRAM banks (which leads to the creation of crossbars in the generated RTL). The more aggressively the SRAMs are banked, the less likely it will be that multiple pipeline banks attempt to access different addresses of the same SRAM bank simultaneously.

If two different pipeline banks, bank-X and bank-Y, attempt to read or write from the same SRAM bank, then pipeline bank-X will have priority if $X < Y$. However, as a minor optimization, if both pipeline banks are trying to read or write the *same* address in the same SRAM bank, then their read or write requests are consolidated so that they happen simultaneously.

Synchronization

The private memory buffers also support various synchronization and interleaving options. For example, reads and writes can be performed simultaneously on the same memory buffer, but with the writes "trailing" (i.e. following behind) the reads; this can be used, for example, to accumulate partial sums in a matmul. Section 4.5 describes Stellar's ISA which allows programmers to specify such synchronization options.

Hardcoding Memory Buffer Request Parameters

Finally, users can optionally hardcode certain read/write request parameters, such as striding patterns that will be used at runtime, before hardware generation begins in order to help Stellar make optimizations to both memory buffers and register files. For example, Listing 4.9 shows an example where a Stellar user hardcodes certain parameters to specify that the dense tensor memory buffer in Figure 4.13b will always produce 4×4 dense matrices. The internal adders and gates in the Dense Addr-Gen units in Figure 4.13b can be simplified based on these hardcoded parameters, for example by replacing runtime-configurable adders with simpler adders that have hardcoded operands.

Listing 4.9: Hardcoding memory buffer read parameters

```
def hardCoded(x: MemPipeline) = Map(
  x.read_req.spans(0) -> 4.U,
```

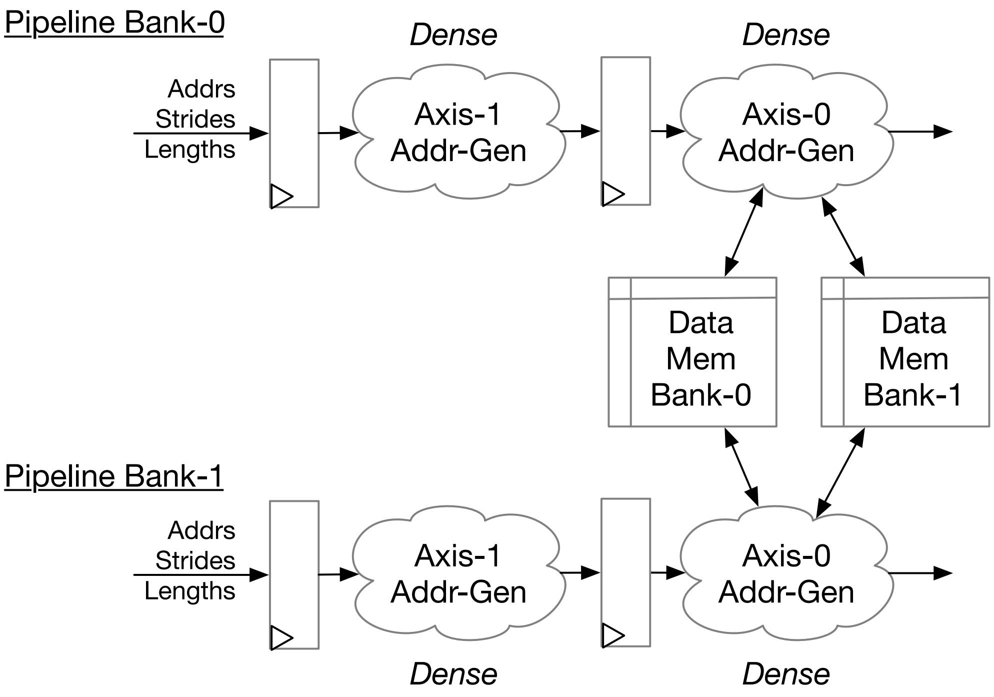


Figure 4.15: The read/write pipeline stages for a private memory buffer dense matrices, with two pipeline banks and two SRAM banks.

```
    x.read_req.spans(1) -> 4.U,
    x.read_req.data_strides(0) -> 1.U,
    x.read_req.data_strides(1) -> 4.U)
```

4.3.4 Generating Register Files

All spatial arrays in Stellar read from and write into *register files* (regfiles). For the matmul example in Listing 4.1, every input and output variable (A, B and C) must be stored in a separate register file before being accessed by the spatial array.

Note that these regfiles are also commonly included in handwritten spatial accelerators, even if the name they are given is different. For example, prior dense DNN accelerators, such as Gemmini [22], must delay certain inputs or outputs before they enter a spatial array because of their specific pipelining strategies; Figure 4.16 demonstrates how these delay registers may be arranged. In Stellar, this can be implemented as a low-area feedforward regfile with a triangular shape.

However, the default, baseline register file design in Stellar is more expensive, as illustrated in Figure 4.17a. Every input port and output port has access to *all* entries in the register file simultaneously, and outputs are performed by searching all entries to find one

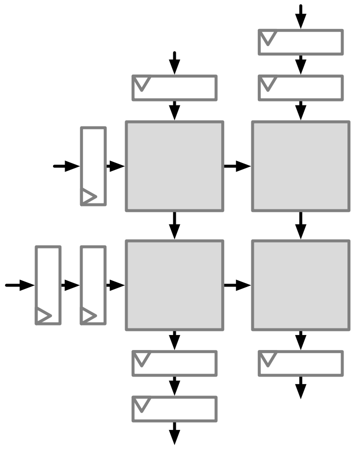


Figure 4.16: Delay registers surrounding a dense matmul array.

whose coordinates match the ones that a PE is requesting. The cost of *each* input and output port therefore scales by the total number of elements in the register file. The register file must also be large enough to store all elements that the spatial array will request during its execution, *no* element is popped off until the spatial array has completely finished its execution.

The baseline register file design is expensive because Stellar's functional specifications (described in Section 4.2.1) are highly flexible and support indirect accesses whose coordinates may not be known until runtime. It may also not be possible to know before runtime how many times exactly an element will be requested from a regfile. The baseline design therefore functions as a worst-case fallback for spatial arrays with complicated and unpredictable regfile access patterns, and despite its high area overhead and wiring congestion, may still be adequate for smaller accelerators. Fortunately, for the vast majority of dense and sparse accelerators, the register file parameters available in Stellar allow these overheads to be greatly reduced, and the scalability of regfiles to be greatly improved.

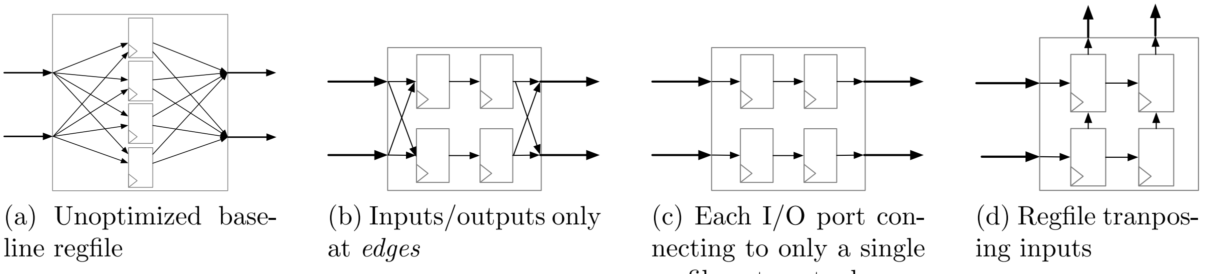
For example, to reduce the overhead of each individual input or output port, it might be sufficient for inputs and outputs to occur only at the *edges* of a regfile as in Figure 4.17b. With further optimizations, Stellar users can narrow the number of elements that need to be searched for a spatial array request even further, as in Figure 4.17c, where each output port is only responsible for observing a single element of the register file, to produce a simple feed-forward array of shift registers, also similar to the design previously shown in Figure 4.16. By selecting *which* edges to designate as entry and exit points for the regfile, Stellar regfiles can even perform various data layout transformations, such as transpositions, as illustrated by Figure 4.17d. The input and output port entry/exit points are also independent of each other; users can optimize only the exit points, for example, without optimizing the entry points if the regfile inputs need maximum flexibility. The entry/exit points can currently be `Anywhere`, `FIFO`, `Edge`, or `PerpendicularEdge`.

In addition to optimizing the number of elements that an input or output port needs to search, Stellar users can also benefit from other optimizations which do not change the functional correctness of a register file's RTL, but which can affect its area overhead. For example, if different regfile input or output ports can be determined to work "in lockstep", such as when two output ports in a regfile are known to always return two directly adjacent elements, then only one of the two ports will require circuitry to search for matching coordinates among the regfile elements, and the other can simply return the regfile element neighboring it. This is very common in dense spatial arrays, but also occasionally possible with sparse spatial arrays. Alternatively, if input ports write to hardware FIFOs in the register files, but the regfile's input ports have been specified to operate in lockstep with each other, then the regfile will not need independent, separate FIFO counters for each input-port; they will instead be able to share the same counters.

Other optimizations relate to the "popping" of elements from a register file. For example, if it is known that a spatial array will only access a regfile element once, then it can be "popped" (i.e. removed) immediately after being accessed, potentially enabling the register file to have fewer entries in total. If a spatial array, however, has complicated, data-dependent access patterns which may require a regfile entry to be accessed an indefinite number of times, then no regfile entry can be popped until the spatial array completely finishes execution, in which case the register file may need more elements in total so that all the inputs the spatial array could potentially need to access in a particular cycle are simultaneously available.

4.3.5 Generating Load Balancers

To support load-balancing, (described in Section 4.2.4), Stellar generates load-balancer modules which monitor regfile inputs and determine based on them whether the PEs that read from those regfiles have enough inputs available to do useful work, or whether they will be idle. The load-balancer modules are designed to be lightweight and to hold as little state as



(a) Unoptimized baseline regfile

(b) Inputs/outputs only at *edges*

(c) Each I/O port connecting to only a single regfile entry at edges

(d) Regfile tranposing inputs

Figure 4.17: Various register files generated by Stellar, with more or less aggressive optimizations. All regfiles in this figure have four entries, two input ports on the left, and two output ports on the right. Observe that when input/output ports can only connect to regfile *edges*, elements must travel through the regfile entries so they can reach the output ports.

possible: when feasible, bitvectors are used to record whether or not various spatial array inputs are available, with each bit corresponding to a different input known at hardware-generation time to be necessary for a PE to be utilized. When several layers of indirect lookups are needed to calculate the coordinates for a PE input, then extra state may need to be stored in the load-balancer module, because the exact coordinates corresponding to a required regfile input may be determined only at runtime.

Once the load-balancers determine that work should be redistributed between PEs, they calculate *space-time biases* to apply at runtime to the space-time transforms of the PEs to which work will be redistributed. A space-time bias is a vector addition, as seen in Equation 4.2, which modifies the unbalanced space-time transform in Equation 4.1 from Section 4.2.2:

$$T\left(\begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}\right) = \begin{bmatrix} x \\ y \\ t \end{bmatrix} \tag{4.2}$$

where b_1, b_2, and b_3 are scalar offsets which are calculated by Stellar based on the user's load-balancing specification. When the space-time bias gives the PEs new space-time coordinates at runtime, they *behave* as if they were other PEs in other parts of the spatial array, allowing them to take some of their workload.

4.3.6 DMA

Finally, Stellar generates a DMA module that handles data transfer between the private memory buffers buffers and main memory, using the cache-coherent TileLink protocol [12]. As with the private memory buffers, Stellar generates different address generation logic for each axis depending on the user's sparse format specification.

However, unlike the SRAM buffer design illustrated in Figure 4.13a, the DMA module generator does not instantiate separate pipelined stages, but instead a state machine that executes each address generation stage sequentially. We choose not to pipeline the DMA because we observe that memory transactions to outer memory are frequently bounded by DRAM bandwidth limitations, lowering opportunities to fully utilize hardware resources if using a more expensive pipelined design similar to the private memory buffers.

Regardless, the DMA does benefit from certain optimizations which are not relevant for the private memory buffers, such as prefetching or memory coalescing to minimize repeated consecutive accesses to the same outer L2 cache lines. The DMA is designed so that the naive, non-optimized implementation of any sparse or dense tensor transfer to/from DRAM, L2, or memory buffers should always be available, in order to guarantee functional correctness for any DMA accesses to any custom sparsity formats. However, for certain memory transfer patterns, such as for CSR matrix transfers, we manually added various extra optimizations, such as prefetching of row-ids (described in detail in the sub-sections below), to improve performance further.

Finally, for Stellar accelerators which have only dense spatial arrays and dense memory buffers, support for compressed or linked-list axes can be entirely left out of the DMA, simplifying it to a simple DRAM address generator and reducing its area or power overhead.

Prefetching Optimizations

As mentioned above, the effective bandwidth of DMA transfers is greatly improved by certain optimizations, such as prefetching of row-ids for CSR matrices from DRAM. For example, consider the DMA's actions when reading a two-dimensional CSR matrix from outer memory, such as L2 or DRAM, into a private memory buffer for a Stellar accelerator. A naive implementation of this CSR matrix transfer would have the DMA read each row-id pointer individually, in a separate TileLink read request, before reading the coordinates the row-id points to.

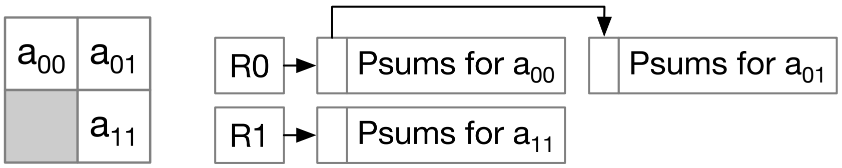
A more efficient strategy, however, is to read *multiple* row-ids simultaneously, in a single TileLink read request, which, in Chipyard, can return up to 64 bytes of data, equivalent to 16 4-byte row-ids. This strategy, where multiple row-ids are read and prefetched simultaneously, can greatly reduce the number of TileLink requests made to outer memory, reducing the number of DMA stalls while waiting for TileLink to return pointers that are necessary for the addresses of future memory requests to be calculated.

Pointer lookups can also be skipped entirely for sparse matrices in some cases, such as if the programmer wants to move an entire, untiled, contiguous CSR matrix from DRAM into SRAM, or vice versa. In such cases, all row-ids, coordinates, and data can be moved as dense blocks to or from DRAM, without any need to stall in the DMA while waiting for row-ids to be returned.

Making Multiple TileLink Requests In Every Cycle

For certain workloads, DMA performance can also be greatly improved if *multiple* non-contiguous TileLink requests can be performed every cycle. For example, OuterSPACE [66] is a handwritten sparse accelerator from prior work which performs matrix multiplications in two phases: one "matmul" phase which multiplies a CSC matrix, A, with a CSR matrix, B, to produced scattered, discontiguous partial sums that are written into DRAM, and a second "merge" phase which merges these partial sums into the final matmul results. OuterSPACE multiplies matrices to produce scattered partial sums, which are stored in a large array of linked-lists, as in Figure 4.18, where each node of each linked list is a small contiguous vector of partial sum values and coordinates.

Before any partial sum vector is accessed by the accelerator, the pointers to that vector must be read from DRAM. Despite comprising less than 10% of the total memory traffic in a typical OuterSPACE matmul, accesses to these pointers can pose a severe memory bottleneck for an OuterSPACE-like accelerator's performance due to control- and data-dependencies imposed by the pointer accesses. Any inefficiency in their reads or writes causes further latency-sensitive stalls in the accelerator's DMA.



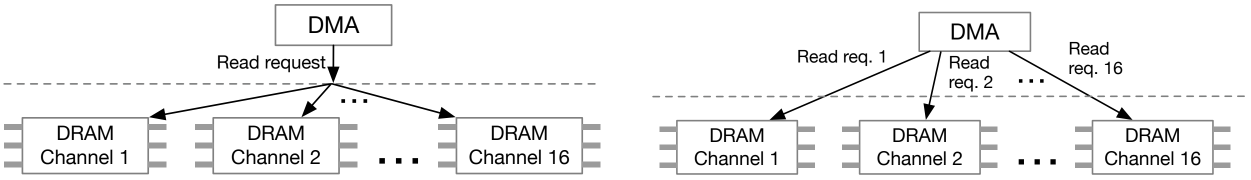
(a) *A (CSC)*

(b) Scattered partial sums *(array-of-linked-lists)*

Figure 4.18: Scattered partial sums generated by OuterSPACE.

This issue is compounded by simpler DMAs which can only make *one* new memory load-/store request per cycle, across the different DRAM channels, as illustrated in Figure 4.19a. For dense accelerators such as Gemmini [22], or many sparse accelerators such as SCNN [67], such DMAs are sufficient for high performance. Even when accessing the partial sum values and coordinates which make up most of OuterSPACE's total traffic, one request per cycle maintains high read bandwidth for most matmuls. However, when accessing the *pointers* to these vectors, one read request can only return a single scalar pointer, causing DRAM under-utilization and a series of costly stalls; these stalls then compound further because the DMA must wait for the pointer to return before it can make read requests for corresponding partial sum vectors.

To accommodate such pointer-chasing workloads, Stellar's DMA is can be parameterized by the architect to generate *multiple* independent DRAM read or write requests per cycle, as illustrated in Figure 4.19b, without necessarily increasing total DRAM bandwidth or the number of DRAM channels. When Stellar's DMA observes at runtime that the programmer is attempting to access multiple, discontiguous pointers, it will perform TileLink requests in parallel to reduce the chances of stalling while waiting for pointers to return.



(a) DMA that makes at most one new read-request every cycle.

(b) DMA that makes multiple new read-requests every cycle.

Figure 4.19: DMA designs that can be generated by Stellar. Both may access the same number of DRAM channels with the same maximum DRAM bandwidth, but (b) is better for pointer-chasing workloads.

4.4 Limitations

Although Stellar's front-end specification language and backend hardware generator support a wide variety of sparse and dense accelerator designs, the existing tool is limited in its ability to express or generate certain sophisticated cache hierarchies found in prior work [102, 104]. Stellar's private memory buffers are explicitly managed by the programmer; however some accelerators benefit most from hardware-managed caches with unusual eviction policies [104]. Fortunately, this limitation is mitigated to a degree by Stellar's integration with the Chipyard [1] framework, which can provision Stellar-generated SoCs with large L2 caches which can be shared by both CPUs and accelerators, although support for custom eviction or prefetching policies are left for future work.

For spatial arrays on the other hand, Stellar's dataflow descriptions can currently only express *affine* transformations, and cannot express recursive or hierarchical transformations such as tree-reductions. However, our functionality specification language is still general enough that such compute structures can be manually implemented by Stellar's users, though at the cost of blurring the separation of concerns between the functional behavior of an accelerator and the scheduling of operations spatially or temporally on it. For example, we were able with Stellar to express the complex hierarchical mergers described in SpArch [104]. After synthesis, we found that these mergers consumed 13× the area of simpler, non-hierachical mergers from OuterSPACE [66]. Therefore, even for recursive operations which don't map easily to Stellar's dataflow abstractions, our experience indicates that such designs can still be specified by the user and explored for area or performance tradeoffs.

Finally, the pre-defined operators in Stellar's functional specification (Section 4.2.1) are all assumed to take only a single cycle to execute, or to be fully-pipelined when extra pipeline registers are inserted between PEs. Certain operations, however, such as divisions or square-roots are typically implemented using *iterative* hardware which takes multiple cycles to return a result. Modern DNN workloads, such as large language models which perform layernorm or softmax operations, often include such iterative operations, which makes them difficult to map onto Stellar. Fortunately, however, Stellar's functional specification language is general enough that such operations, which take multiple cycles, can be implemented manually by the user using the pre-defined addition or bitshift operators.

4.5 Programming Interface

Stellar-generated accelerators are programmed using custom RISC-V instructions, summarized in Table 4.1. All instructions revolve around data transfers from one memory unit to another; for example, from DRAM to a private memory buffer, or from a private memory buffer to a register file. There are no commands to explicitly begin execution on a spatial array; instead, the spatial arrays immediately begin execution when the inputs they need appear in their register files.

For most Stellar instructions that move data from a source memory to a destination memory, users set certain values, such as addresses, strides, spans, and fibertree axis types [84], for the source and/or the destination. For example, to move a sparse CSR matrix from DRAM into a private memory buffer, programmers specify the addresses of the matrix's data and metadata arrays in DRAM, as well as the address within the accelerator's private memory that the data will be copied to.

To illustrate, Listing 4.10 shows two code snippets in C using Stellar's ISA. One snippet moves a dense matrix from DRAM into a private memory buffer called `SRAM_A`. The other moves a CSR matrix into `SRAM_B`. Programmers set strides to generate data- or metadata-addresses; for example, on lines 39-41, as we move through the outer dense axis, we increment the `ROW_ID` address of the innermost compressed axis.

Stellar's ISA also supports synchronization between data-transfer operations occurring simultaneously. For example, if we are accumulating a matmul result into a private memory buffer, we will need to simultaneously read and write partial sums in the memory buffer as they are gradually accumulated into their final results. If we set `should_trail_reads` in Table 4.1 to `true`, Stellar will prevent write-after-read hazards from mutating partial sums before they are read.

Stellar's codebase includes C/C++ libraries to enable easy integration of Stellar instructions into users' applications. Stellar-generated accelerators also come equipped with a variety of optional in-order [2] or out-of-order [6] RISC-V CPUs from Chipyard [1] which

Instruction	**Rs1[19:16]**	**Rs1[15:0]**	**Rs2**
`set_address`	For src, dst, or both	Axis	DRAM/SRAM address, or regfile
`set_span`	For src, dst, or both	Axis	Number of elements to move
`set_data_stride`	For src, dst, or both	Axis	Stride
`set_metadata_stride`	For src, dst, or both	Axis and metadata type (e.g. `ROW_ID` or `COORD`)	Stride
`set_axis_type`	For src, dst, or both	Axis	"Dense", "Compressed", "LinkedList", etc.
`set_constant`	N/A	ID of scalar or boolean constant to set: e.g. `should_trail_reads`, `should_interleave`, `interleave_axis`	True/false if boolean, scalar integer otherwise

Table 4.1: A representative subset of the commands in Stellar's RISC-V ISA. Each instruction has two 64-bit register arguments, *Rs1* and *Rs2*. Bits [63:20] in Rs1 are currently unused.

Listing 4.10: Moving matrices from DRAM into local memory.

```
// Moving in dense matrix
float matrix_A[DIM][DIM];

set_src_and_dst(DRAM, SRAM_A);

set_data_addr(FOR_SRC, matrix_A);

for (int axis = 0, axis < 2; axis++) {
  set_span(FOR_BOTH, axis, DIM);
  set_axis(FOR_BOTH, axis, DENSE);
}

set_stride(FOR_BOTH, /*addr-gen-axis=*/0, 1);
set_stride(FOR_BOTH, /*addr-gen-axis=*/1, DIM);

stellar_issue();

// Moving in CSR matrix
float matrix_B_data[DATA_SIZE];
int matrix_B_coords[DATA_SIZE];
int matrix_B_row_ids[N_ROWS];

set_src_and_dst(DRAM, SRAM_B);

set_data_addr(FOR_SRC, matrix_B);

set_metadata_addr(FOR_SRC, /*axis=*/0, ROW_ID,
  matrix_B_row_ids);
set_metadata_addr(FOR_SRC, /*axis=*/0, COORDS,
  matrix_B_coords);

set_span(FOR_BOTH, /*axis=*/0, ENTIRE_AXIS);
set_span(FOR_BOTH, /*axis=*/1, N_ROWS);

set_stride(FOR_BOTH, /*addr-gen-axis=*/0, 1);
set_metadata_stride(FOR_BOTH,
  /*addr-gen-axis=*/0, /*axis=*/0,
  COORDS, 1);
set_metadata_stride(FOR_BOTH,
  /*addr-gen-axis=*/1, /*axis=*/0,
  ROW_IDS, 1);

set_axis(FOR_BOTH, /*axis=*/0, COMPRESSED);
set_axis(FOR_BOTH, /*axis=*/1, DENSE);

stellar_issue();
```

can run arbitrary code while issuing instructions to Stellar-generated accelerators.

4.6 Evaluation

Stellar allows users to efficiently express state-of-the-art accelerator designs, and then automatically synthesizes RTL implementations which are comparable in performance and area

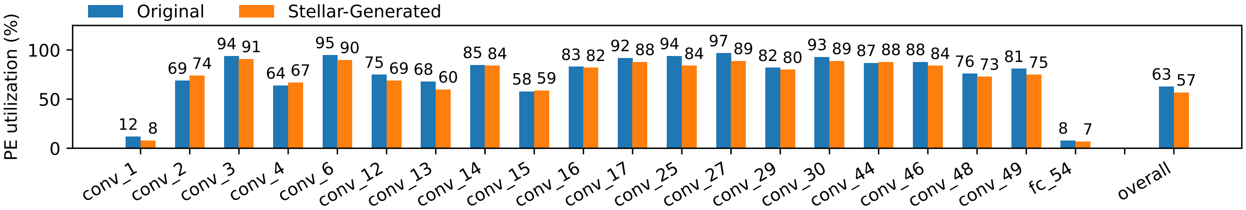


Figure 4.20: The PE utilization of both the handwritten and Stellar-generated Gemmini accelerators on ResNet50.

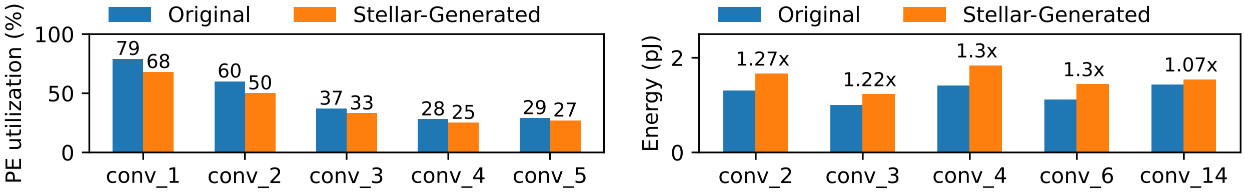
consumption to hand-designed accelerators when running real-world workloads. By generating actual hardware, Stellar also enables architects to make insights into area/power/performance trade-offs on real hardware which are which cannot be evaluated in higher-level simulators.

4.6.1 Methodology

To demonstrate that Stellar-generated designs are competitive with hand-written implementations, we generate two DNN accelerators from prior work: a dense DNN accelerator modeled after Gemmini [22], which performs convolutions and 8-bit quantized matrix multiplications with a 16×16 weight-stationary systolic array, and SCNN [68], which targets convolutional networks which have been pruned for unstructured weight and activation sparsity. Using cycle-accurate simulators [41], we compare the performance of both the hand-written and Stellar-generated implementations on the DNN workloads they were originally evaluated on in prior work: an end-to-end ResNet50 [30] inference for Gemmini, and AlexNet [46] for SCNN. For area and frequency comparisons, we synthesize designs using the ASAP7 PDK, and we evaluate energy consumption on Joules using the Intel 22nm process.

4.6.2 Performance and Area Overheads

The Stellar-generated Gemmini accelerator achieved 90% of the utilization of the handwritten Gemmini accelerator when both were synthesized to 500 MHz, as shown by Figure 4.20. However, the Stellar-generated accelerator was successfully synthesized at up to 1 GHz, while the handwritten Gemmini could only reach 700 MHz. The handwritten Gemmini includes complicated, centralized loop-unrollers, whose address generators failed to meet timing at higher frequencies; Stellar's more distributed memory-buffer address-generators were more scalable. Stellar's performance is also competitive with those of prior accelerator generators. For example, Interstellar [97] reports near 100% utilization for AlexNet-CONV3 when generating an accelerator using Gemmini's weight-stationary dataflow at 400 MHz; our Stellar-generated design achieves 92% utilization at 1 GHz.



(a) SCNN performance.

(b) Gemmini energy.

Figure 4.21: Performance and power consumption of Stellar-generated and handwritten dense and sparse accelerators.

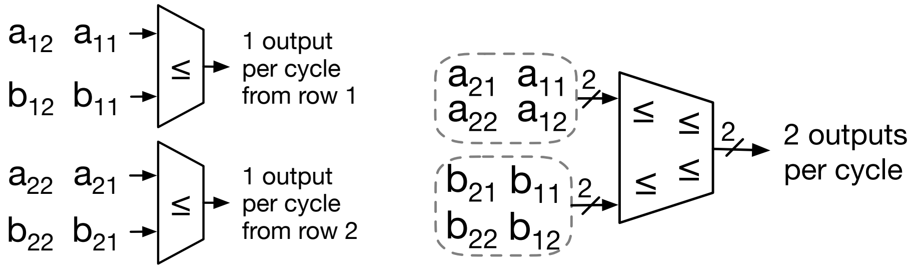
Furthermore, the Stellar-generated Gemmini accelerator only consumed 13% more area than the hand-designed accelerator when both were synthesized to 500 MHz, as shown in Table 4.2, demonstrating that Stellar's support for sparse accelerators does not compromise the competitiveness and efficiency of the dense accelerators that it generates. The area overhead for the matmul array comes partially from the larger amount of internal state in a Stellar-generated PE (such as the "time" register in Figure 4.12), compared to handwritten Gemmini PEs which have no internal counters. Furthermore, Stellar-generated spatial arrays include global signals that start and stall all PEs simultaneously. While this is useful for many workloads, it is not needed in Gemmini-like workloads where the memory buffers consuming partial sums from the matmul array will always be ready to consume spatial array outputs. These long global signals add further area overhead.

Stellar's power overhead ranges from 7% at best to 30% at worst compared to the handwritten Gemmini on various layers of ResNet50, as illustrated in Figure 4.21b when both were synthesized to 500 MHz with the Intel 22nm node.

Finally, as illustrated in Figure 4.21a, the Stellar-generated SCNN achieved 83%-94% of the hand-designed accelerator's reported performance when executing a sparse, pruned

	Original		**Stellar-Generated**	
	Area (μm^2)	Area (%)	Area (μm^2)	Area (%)
Matmul array	334K	10%	420K	11%
SRAMs	2,225K	68%	2,247K	61%
Regfiles	25K	1%	104K	3%
Loop unrollers	259K	8%	482K	13%
DMA	102K	3%	109K	3%
Host CPU	337K	10%	337K	9%
Total	3,282K	*100%*	3,699K	*100%*

Table 4.2: Area comparison between Gemmini accelerators.



(a) Row-partitioned mergers.

(b) Flattened merger.

Figure 4.22: Spatial arrays that merge scattered partial matrices. In (a), every PE merges a separate row of the partial matrices, and every PE only outputs a single element every cycle. In (b), the different rows of the partial matrices are flattened into a single fiber from which multiple elements are merged every cycle.

model of AlexNet. SCNN has a sophisticated design including a spatial array with a four-dimensional PE topology, but Stellar's abstractions covered this point in the design space while Stellar's hardware generation flow delivered synthesizable and programmable RTL.

4.6.3 Area and Performance Tradeoffs in Sparse Mergers

Stellar's generation of real RTL also enables architects to investigate performance, area, and hardware efficiency trade-offs which cannot be explored by more abstract, higher-level simulators. To illustrate, in this section, we show how Stellar can be used to significantly reduce the area of partial matrix mergers without compromising their performance on a variety of sparse matrix multiplications.

Prior work [102, 104, 66] on sparse tensor accelerators introduces various spatial arrays that *merge* the scattered partial matrices produced by sparse matrix multiplier arrays. Some works, such as GAMMA[102] and OuterSPACE [66], merge each row of a partial matrix on a different PE, each generating one element every cycle as shown in Figure 4.22a. The throughput of such mergers can be increased by scaling up the number of PEs to merge more rows in parallel. Other accelerators, such as SpArch [104], do not partition merging tasks in this way, but instead flatten the different rows in a partial matrix into a single contiguous fiber, and pop *multiple* elements from this fiber every cycle, as in Figure 4.22b.

The mergers which operate on different rows separately, such as with GAMMA, take up far less area than the ones which flatten partial matrices, such as the one used in SpArch. For example, SpArch's mergers consume over 60% of it's area, with 128 64-bit comparators used for a maximum throughput of 16 elements per cycle, while GAMMA-like mergers, when synthesized with Stellar, consume 13× less area. However, the cheaper, row-partitioned mergers are more sensitive to imbalances in the lengths of different rows of the partial matrices. SpArch's loop execution order generates many small partial matrices which can have highly imbalanced row-lengths, causing severe underutilization on GAMMA-like mergers.

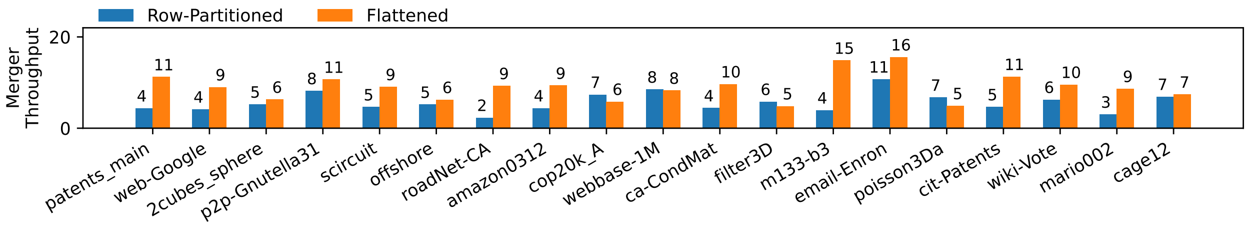


Figure 4.23: The number of merged elements generated every cycle by both row-partitioned and flattened mergers when merging partial matrices with SpArch's proposed execution order [104].

The original SpArch work [104] does not explore the potential performance and area tradeoffs of using cheaper, row-partitioned mergers at the cost of greater sensitivity to row-length imbalances. However, for accelerators with severe area or resource constraints, this trade-off may be worthwhile. With a framework such as Stellar, where diverse spatial array designs for components such as mergers can be easily generated alongside the necessary memory buffers, register-files, DMAs, and programming interfaces, such trade-offs can be easily explored.

To investigate, we generated row-partitioned, low-area mergers for SpArch with a maximum throughput of 32, and compared their performance to the more expensive flattened mergers from the handwritten accelerator, which have a smaller maximum throughput of 16 but more comparators in total in the mergers. As mentioned above, SpArch merges large numbers of small matrices; these matrices often have so few elements that most of their rows may be entirely empty of dense values. To compensate for this high level of "row-sparsity," we store the partial matrices in our private memory buffers in a custom sparsity format which skips empty rows entirely; Stellar's strong abstractions for private memory buffer design enable such optimizations.

As shown in Figure 4.23, the row-partitioned mergers achieve at least 80% of the flattened merger's performance on over a third of the SuiteSPARSE matrices that SpArch was tested on in its original publication (summarized in Table 4.3). In fact, on the matrices `poisson3Da`, `filter3D`, `cop20k_A`, and `webbase-1M`, the smaller, row-partitioned merger performed *better* than the larger, flattened merger from the original SpArch work, due to the row-partitioned merger's greater maximum theoretical throughput. On the remaining matrices, the row-partitioned mergers performed poorly due to the particular non-zero distributions across partial matrix rows. Architects who face area constraints and who expect that the matrices they merge will be similar to `poisson3Da` or `cop20k_A`, may prefer the row-partitioned mergers when building accelerators that merge matrices in the same order that SpArch does.

As noted previously in Section 4.4, SpArch's flattened mergers, illustrated in Figure 4.22b, are not the best fit for Stellar's dataflow specification language. However, despite this limitation, Stellar's functionality specification language was still generalizable enough to enable

Matrix	Nnz/row	Dimension	Matrix	Nnz/row	Dimension
patents_main	2.33	240,547	web-Google	5.57	916,428
p2p-Gnutella31	2.36	62,586	scircuit	5.61	170,998
roadNet-CA	2.81	1,971,281	amazon0312	7.99	400,727
webbase-1M	3.11	1,000,005	ca-CondMat	8.08	23,133
m133-b3	4.00	200,200	email-Enron	10.02	36,692
cit-Patents	4.38	3,774,768	wiki-Vote	12.50	8,297
mario002	5.38	389,874	cage12	15.61	130,228
2cubes_sphere	16.23	101,492	filter3D	25.43	106,437
offshore	16.33	259,789	poisson3Da	26.10	13,514
cop20k_A	21.65	121,192			

Table 4.3: The SparseSuite matrices we include in our evaluation.

SpArch's flattened mergers to be implemented so that they could be compared to the simpler, row-partitioned mergers more commonly used in accelerators such as GAMMA. Furthermore, Stellar's compiler was capable of generating the RTL for the memory buffers, regfiles, DMAs, and programming interfaces necessary to run these matrix merging and sorting workloads without writing custom Verilog for hardware components or testbenches.

4.7 Summary

Stellar enables the rapid design, exploration, and generation of both dense and sparse spatial accelerators, by allowing architects to cleanly separate the different concerns that go into designing an accelerator, and then generating synthesizable RTL implementations and software interfaces which are comparable to hand-written designs from prior work, based on architects' specifications. Stellar is also fully compatible with the Chipyard [1] chip design framework, enabling users to integrate their designs into complete, programmable SoCs.

Chapter 5

Conclusion

One of the greatest challenges for a hardware designer (and not just a hardware designer, but also a software programmer[1]) is to separate different design concerns: between an accelerator's functionality and its dataflow, between an accelerator's compute array and its cache hierarchy, or between any of the other design components which have such important cross-cutting impacts that they naturally tend to bleed into each other during ad-hoc implementation. This thesis describes two projects – Gemmini and Stellar – which demonstrate the advantages of maintaining a strong separation of concerns all throughout the design process, such as the ability to rapidly iterate on hardware designs, or the ability to accurately analyze the impact that often-overlooked design considerations such as TLB hierarchies have upon end-to-end performance. In particular, we make the following contributions:

- A detailed review of past work that examines (i) spatial arrays which accelerate DNNs (or other applications), (ii) past proposals for high-level abstractions for dense and sparse accelerator design, and (iii) the need for accelerator design/generation frameworks which allow architects to account for the impact of the *full* hardware-software-system stack upon DNN acceleration.

- Highly-parameterized hardware generators that produce high-quality, synthesizable RTL with only modest effort from architects. This RTL has been taped-out [26] (and continues to be taped-out in ongoing projects), and has been used for real-world workloads in domains such as computer vision, numerical analysis, and for large language models.

- Domain-specific languages that can describe a wide range of spatial arrays for both dense and sparse applications, while maximizing the separation of concerns between different design considerations such as dataflows and sparse/dense data structures.

[1] And not just a software programmer, but the creator of any complex system.

- Tools which leverage frameworks such as Chipyard [1] and FireSim [41] to enable architects to derive useful insights into the impact that system or software components external to the spatial array will have upon DNN acceleration.

5.1 Future Work

Although this projects described in this thesis help make DNN accelerator design easier and more efficient, they do have a number of limitations which provide opportunities for future work:

- Both Gemmini and Stellar expose broad design spaces for users, but neither work includes any method for *automated* exploration. It is our hope that the design space parameters and domain specific languages introduced are amenable to algorithmic exploration. (In fact, this was one of our aims; that is one reason why we preferred to define dataflows using simple matrices of which every element could, in theory, be swept by an automated search algorithm). However, designing such an algorithm still remains as future work.

- Stellar introduces expressive abstractions for both a front-end domain-specific language and an internal representation (the `IterationSpace` described in Chapter 4 Section 4.3.2) for spatial arrays. However, the abstractions for memory buffers, register files, DMA designs, and other memory storage and transfer components have more limited extensibility. We designed them as highly-parameterizable handwritten templates with a great number of elaboration-time parameters to choose from, but it remains possible that new accelerators can be proposed which require *new* parameters which would have to be added manually by the Stellar authors. Selecting the correct parameters for a given accelerator's memory buffers, register files, DMAs, etc. is also difficult. (Although Stellar includes baseline fallback designs for these components which should work in all scenarios, they are too unoptimized to be feasible for most large designs). Integrating high-level abstractions for memory buffer, cache, and DMA units into Stellar could make these units less like "templates with a great many parameters which must be carefully chosen," and more like the spatial array internal representations, which rely far less on handwritten optimizations to Chisel templates.

- Gemmini, by integrating into the Chipyard ecosystem, provides users with an SoC with a realistic programming stack (all the way up to the operating system running on the SoC), and with various performance counters that can identify performance bottlenecks caused by different system components. However, these performance counters were all identified and added manually by Gemmini's authors. It remains possible that future users will need to run workloads whose performance bottlenecks are caused by a system component that we did not forsee, and for which we added no performance counters ourselves. A more principled way to identify bottlenecks would make Gemmini more future-proof, and may provide opportunities for future work.

- Finally, the evaluations in this thesis were primarily DNN-focused. This is not unusual for research in this space, considering the economic importance of modern-day DNNs, but the full space of spatial accelerator workloads is clearly much broader. In particular, even AI applications are oftentimes bottlenecked by non-DNN kernels which require attention from architects – or even their own specialized hardware – to resolve. Prior work [77] has shown that in some end-to-end AI workloads for face recognition, over 40% of end-to-end cycles can be spent on non-DNN operations, rather than the matrix multiplications or convolutions which this thesis focuses on. For example, image processing operations such as resizing or cropping can consume over 15% of total runtime on some vision-based AI workloads [77]. Other prior work [39] shows that for heavily-optimized workloads such as ResNet or MobileNet, preprocessing operations, such as decoding image file formats, can be an order of magnitude more expensive than the actual DNN inference. Evaluating Gemmini and Stellar on such workloads (and extending them to better support such applications if need be) provides rich opportunities for future work.

5.2 Lessons Learned

Finally, this dissertation ends with a few pitfalls and challenges I encountered during this work, and the lessons I learned from them, in the hope of helping future students and researchers avoid them.

The projects in this dissertation attempted to provide principled abstractions and frameworks which made accelerator design, generation, and evaluation more efficient. However, a persistent challenge – for both Gemmini and Stellar – was that performance bottlenecks and energy overheads oftentimes came from those aspects of accelerator design which were initially overlooked by our high-level abstractions. Caches, transfers back-and-forth to main memory, and synchronization logic were frequent sources of such overheads, while, for both Gemmini and Stellar, our initial high-level abstractions focused instead on spatial array design, dataflow, and loop order configurations.

When we instead encountered bottlenecks from these other sources, we oftentimes simply hand-wrote optimizations for them, and added new scalar, enum, or boolean parameters to our templates to cover these design points. Over time, these boolean or scalar parameters gradually built up till they became difficult for users to fully understand or reason about. If, instead, we had taken the opportunity to rework our more principled, high-level abstractions when we encountered these unexpected bottlenecks, rather than simply appending an extra knob or switch on top of our existing framework, then the resulting design may have been simpler, more elegant, and potentially more future-proof as well. Of course, it is also possible that extending our high-level abstractions to cover all these extra design considerations in an elegant way might have been impossible, but it should have still been our initial inclination at all times.

Finally, this particular pitfall was primarily a consequence of the fact that we generated real RTL from all our frameworks (which, even now, is not all that common in this field). Generating higher-level models or simulators would have allowed us to ignore, for example, the overhead caused by certain DMA inefficiencies which only become apparent in real hardware, such as the examples given in Chapter 4, Section 4.3.6. Generating real hardware (and then taping it out) remains a worthy goal for computer architecture researchers; however, those trying to create principled, one-stop, end-to-end frameworks for hardware design and exploration would do well to remember the challenge of doing so when so many different (and sometimes, frankly, uninteresting) sources of inefficiency might exist in their SoC which are tempting to solve simply with ad-hoc manual RTL fixes.

Bibliography

[1] Alon Amid et al. "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs". In: *IEEE Micro* (2020).

[2] Krste Asanovic et al. *The Rocket Chip Generator.* Tech. rep. EECS Department, University of California, Berkeley, 2016.

[3] Jonathan Bachrach et al. "Chisel: Constructing Hardware in a Scala Embedded Language". In: *DAC.* 2012.

[4] MP Bekakos et al. "Hexagonal systolic arrays for matrix multiplication". In: *Highly parallel computaions: algorithms and applications.* 2001, pp. 175–209.

[5] Iz Beltagy et al. "Longformer: The long-document transformer". In: *arXiv preprint* (2020).

[6] Christopher Celio, David A Patterson, and Krste Asanovic. "The Berkeley Out-of-Order Machine (BOOM): An industry-competitive, synthesizable, parameterized RISC-V processor". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167* (2015).

[7] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks". In: *ACM SIGARCH computer architecture news* 44.3 (2016), pp. 367–379.

[8] Tianqi Chen et al. "TVM: An Automated End-to-end Optimizing Compiler for Deep Learning". In: *OSDI.* 2018.

[9] Y. Chen et al. "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices". In: *JETCAS* (2019).

[10] Rewon Child et al. "Generating long sequences with sparse transformers". In: *arXiv preprint* (2019).

[11] Jason Cong and Jie Wang. "PolySA: Polyhedral-based systolic array auto-compilation". In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* IEEE. 2018, pp. 1–8.

[12] Henry M Cook, Andrew S Waterman, and Yunsup Lee. *SiFive TileLink Specification.* Tech. rep. `https://www.sifive.com/documentation/tilelink/tilelink-spec/`. SiFive Inc., 2018.

[13] Vidushi Dadu et al. "Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 924–939. ISBN: 9781450369381. DOI: `10.1145/3352460.3358276`. URL: `https://doi.org/10.1145/3352460.3358276`.

[14] P. Dai et al. "SparseTrain: Exploiting Dataflow Sparsity for Efficient Convolutional Neural Networks Training". In: *DAC*. 2020. DOI: `10.1109/DAC18072.2020.9218710`.

[15] Steve Dai et al. "Efficient Transformer Inference with Statically Structured Sparse Attention". In: *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 2023, pp. 1–6. DOI: `10.1109/DAC56929.2023.10247993`.

[16] Steve Dai et al. "Vs-quant: Per-vector scaled quantization for accurate low-precision neural network inference". In: *Proceedings of Machine Learning and Systems* 3 (2021), pp. 873–884.

[17] Jack Dongarra. *Block Compressed Row Storage (BCRS)*. `https://netlib.org/linalg/html_templates/node93.html`. Accessed: 2023-11-21. 1995.

[18] Z. Du et al. "ShiDianNao: Shifting vision processing closer to the sensor". In: *ISCA*. 2015.

[19] Albert Einstein et al. "The foundation of the general theory of relativity". In: *Annalen Phys* 49.7 (1916), pp. 769–822.

[20] *ELL_Matrix Class*. `https://www.lanl.gov/Caesar/node223.html`. Accessed: 2024-06-04.

[21] Jeremy Fowers et al. "A Configurable Cloud-Scale DNN Processor for Real-Time AI". In: *ISCA*. 2018.

[22] Hasan Genc et al. "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration". In: *Proceedings of the 58th Annual Design Automation Conference (DAC)*. 2021.

[23] Tong Geng et al. *AWB-GCN: A Graph Convolutional Network Accelerator with Runtime Workload Rebalancing*. 2020. arXiv: `1908.10834 [cs.DC]`.

[24] Tong Geng et al. "I-GCN: A graph convolutional network accelerator with runtime locality enhancement through islandization". In: *MICRO-54: 54th annual IEEE/ACM international symposium on microarchitecture*. 2021, pp. 1051–1063.

[25] Soroush Ghodrati et al. "Planaria: Dynamic Architecture Fission for Spatial Multi-Tenant Acceleration of Deep Neural Networks". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2020, pp. 681–697. DOI: `10.1109/MICRO50266.2020.00062`.

[26] Abraham Gonzalez et al. "A 16mm2 106.1 GOPS/W Heterogeneous RISC-V Multi-Core Multi-Accelerator SoC in Low-Power 22nm FinFET". In: *ESSCIRC 2021 - IEEE 47th European Solid State Circuits Conference (ESSCIRC)*. 2021, pp. 259–262. DOI: `10.1109/ESSCIRC53450.2021.9567768`.

[27] Y. Hao et al. "Supporting Address Translation for Accelerator-Centric Architectures". In: *HPCA*. 2017.

[28] John Hauser. *Hardfloat*. `https://github.com/ucb-bar/berkeley-hardfloat`.

[29] Kim Hazelwood et al. "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective". In: *HPCA*. 2018.

[30] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: `1512.03385 [cs.CV]`.

[31] Xin He et al. "Sparse-TPU: Adapting Systolic Arrays for Sparse Matrices". In: *ICS*. 2020.

[32] G. Henry et al. "High-Performance Deep-Learning Coprocessor Integrated into x86 SoC with Server-Class CPUs Industrial Product". In: *ISCA*. 2020.

[33] Andrew G. Howard et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *CoRR* (2017).

[34] Olivia Hsu et al. "The Sparse Abstract Machine". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 2023, pp. 710–726.

[35] Bongjoon Hyun et al. "NeuMMU: Architectural Support for Efficient Address Translations in Neural Processing Units". In: *ASPLOS*. 2020.

[36] Yuka Ikarashi et al. "Exocompilation for productive programming of hardware accelerators". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2022, pp. 703–718.

[37] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *ISCA*. 2017.

[38] Dhiraj Kalamkar et al. *A Study of BFLOAT16 for Deep Learning Training*. 2019. arXiv: `1905.12322 [cs.LG]`. URL: `https://arxiv.org/abs/1905.12322`.

[39] Daniel Kang et al. "Jointly optimizing preprocessing and inference for DNN-based visual analytics". In: *arXiv preprint arXiv:2007.13005* (2020).

[40] S. Karandikar et al. "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud". In: *ISCA*. 2018.

[41] S. Karandikar et al. "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud". In: *ISCA*. 2018.

[42] Richard M Karp, Raymond E Miller, and Shmuel Winograd. "The organization of computations for uniform recurrence equations". In: *Journal of the ACM (JACM)* 14.3 (1967), pp. 563–590.

[43] Seah Kim et al. "MoCA: Memory-centric, adaptive execution for multi-tenant deep neural networks". In: *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2023, pp. 828–841.

[44] Sehoon Kim et al. "I-bert: Integer-only bert quantization". In: *International conference on machine learning*. PMLR. 2021, pp. 5506–5518.

[45] David Koeplinger et al. "Spatial: A language and compiler for application accelerators". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2018, pp. 296–311.

[46] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012).

[47] HT Kung and Charles E Leiserson. "Systolic arrays (for VLSI)". In: *Sparse Matrix Proceedings*. 1979.

[48] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. "MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Programmable Interconnects". In: *ASPLOS*. 2018.

[49] Yi-Hsiang Lai et al. "SuSy: A Programming Model for Productive Construction of High-Performance Systolic Arrays on FPGAs". In: *Proceedings of the 39th International Conference on Computer-Aided Design*. ICCAD '20. Virtual Event, USA: Association for Computing Machinery, 2020. ISBN: 9781450380263. DOI: `10.1145/3400302.3415644`. URL: `https://doi.org/10.1145/3400302.3415644`.

[50] Amy W Lim and Monica S Lam. "Maximizing parallelism and minimizing synchronization with affine partitions". In: *Parallel computing* 24.3-4 (1998), pp. 445–475.

[51] Liu Liu et al. *Transformer Acceleration with Dynamic Sparse Attention*. 2021. arXiv: `2110.11299 [cs.LG]`.

[52] *llama.cpp*. `https://github.com/ggerganov/llama.cpp`.

[53] Liqiang Lu et al. "Rubick: A Unified Infrastructure for Analyzing, Exploring, and Implementing Spatial Architectures via Dataflow Decomposition". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43.4 (2024), pp. 1177–1190. DOI: `10.1109/TCAD.2023.3337208`.

[54] Zizhang Luo et al. "Rubick: A synthesis framework for spatial architectures via dataflow decomposition". In: *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2023, pp. 1–6.

[55] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. "TLB improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level TLBs". In: *TACO* (2013).

[56] Divya Mahajan et al. "Tabla: A unified template-based framework for accelerating statistical machine learning". In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2016, pp. 14–26.

[57] Paolo Mantovani et al. "Agile SoC development with open ESP". In: *Proceedings of the 39th International Conference on Computer-Aided Design*. ICCAD '20. ACM, Nov. 2020. DOI: `10.1145/3400302.3415753`. URL: `http://dx.doi.org/10.1145/3400302.3415753`.

[58] Linyan Mei et al. "ZigZag: Enlarging Joint Architecture-Mapping Design Space Exploration for DNN Accelerators". In: *IEEE Transactions on Computers* 70.8 (2021), pp. 1160–1174. DOI: `10.1109/TC.2021.3059962`.

[59] Paulius Micikevicius et al. *FP8 Formats for Deep Learning*. 2022. arXiv: `2209.05433 [cs.LG]`. URL: `https://arxiv.org/abs/2209.05433`.

[60] Thierry Moreau et al. "VTA: An Open Hardware-Software Stack for Deep Learning". In: *CoRR* (2018).

[61] Thierry Moreau et al. "VTA: An Open Hardware-Software Stack for Deep Learning". In: *CoRR* (2018).

[62] Nandeeka Nayak et al. "TeAAL: A Declarative Framework for Modeling Sparse Tensor Accelerators". In: *arXiv preprint arXiv:2304.07931* (2023).

[63] NVIDIA. *NVIDIA Deep Learning Accelerator*. `http://nvdla.org/`. Accessed: 2019-08-1. 2019.

[64] *NVIDIA A100 Tensor Core GPU Architecture V1.0*. Tech. rep. NVIDIA.

[65] *ONNX Runtime: Optimize and Accelerate Machine Learning Inferencing and Training*. `https://microsoft.github.io/onnxruntime/`.

[66] Subhankar Pal et al. "Outerspace: An outer product based sparse matrix multiplication accelerator". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2018, pp. 724–736.

[67] Angshuman Parashar et al. "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks". In: *ISCA*. 2017.

[68] Angshuman Parashar et al. "SCNN: An accelerator for compressed-sparse convolutional neural networks". In: *ACM SIGARCH computer architecture news* 45.2 (2017), pp. 27–40.

[69] Angshuman Parashar et al. "Timeloop: A systematic approach to dnn accelerator evaluation". In: *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE. 2019, pp. 304–315.

[70] Adam Paszke et al. "Automatic differentiation in PyTorch". In: *NIPS-W*. 2017.

[71] E. Qin et al. "SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training". In: *HPCA*. 2020.

[72] Eric Qin et al. "SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training". In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2020, pp. 58–70. DOI: 10.1109/HPCA47549.2020.00015.

[73] Jiezhong Qiu et al. "Blockwise self-attention for long document understanding". In: *arXiv preprint* (2019).

[74] Patrice Quinton. "Automatic synthesis of systolic arrays from uniform recurrent equations". In: *SIGARCH Comput. Archit. News* 12.3 (Jan. 1984), pp. 208–214. ISSN: 0163-5964. DOI: 10.1145/773453.808184. URL: https://doi-org.libproxy.berkeley.edu/10.1145/773453.808184.

[75] S.K. Rao and T. Kailath. "Regular iterative algorithms and their implementation on processor arrays". In: *Proceedings of the IEEE* 76.3 (1988), pp. 259–269. DOI: 10.1109/5.4402.

[76] D. Richins et al. "Missing the Forest for the Trees: End-to-End AI Application Performance in Edge Data Centers". In: *HPCA*. 2020.

[77] Daniel Richins et al. "Ai tax: The hidden cost of ai data center applications". In: *ACM Transactions on Computer Systems (TOCS)* 37.1-4 (2021), pp. 1–32.

[78] Mark Sandler et al. "Mobilenetv2: Inverted residuals and linear bottlenecks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 4510–4520.

[79] Hardik Sharma et al. "From High-level Deep Neural Models to FPGAs". In: *MICRO*. 2016.

[80] Frans Sijstermans. "The NVIDIA Deep Learning Accelerator". In: *Hot Chips*. 2018.

[81] Linghao Song et al. "Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication". In: *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '22. Virtual Event, USA: Association for Computing Machinery, 2022, pp. 65–77. ISBN: 9781450391498. DOI: 10.1145/3490422.3502357. URL: https://doi.org/10.1145/3490422.3502357.

[82] Nitish Srivastava et al. "Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 766–780.

[83] Xiao Sun et al. "Hybrid 8-bit floating point (HFP8) training and inference for deep neural networks". In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[84] Vivienne Sze et al. *Efficient Processing of Deep Neural Networks: A Tutorial and Survey*. Springer, 2020.

[85] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[86] Swagath Venkataramani et al. "ScaleDeep: A Scalable Compute Architecture for Learning and Evaluating Deep Networks". In: *ISCA*. 2017.

[87] Rangharajan Venkatesan et al. "MAGNet: A Modular Accelerator Generator for Neural Networks". In: *ICCAD*. 2019.

[88] Hanrui Wang, Zhekai Zhang, and Song Han. "Spatten: Efficient sparse attention architecture with cascade token and head pruning". In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2021, pp. 97–110.

[89] Jie Wang, Licheng Guo, and Jason Cong. "AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA". In: *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2021, pp. 93–104.

[90] Sinong Wang et al. "Linformer: Self-attention with linear complexity". In: *arXiv preprint* (2020).

[91] Y. Wang et al. "DeepBurning: Automatic generation of FPGA-based learning accelerators for the Neural Network family". In: *DAC*. 2016. DOI: `10.1145/2897937.2898002`.

[92] Pete Warden. *Why GEMM is at the heart of deep learning*. `https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/`. Accessed: 2024-06-05.

[93] Jian Weng et al. "Dsagen: Synthesizing programmable spatial accelerators". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020, pp. 268–281.

[94] Carole-Jean Wu et al. "Machine Learning at Facebook: Understanding Inference at the Edge". In: *HPCA*. 2019.

[95] Yannan Nellie Wu et al. "Sparseloop: An analytical, energy-focused design space exploration methodology for sparse tensor accelerators". In: *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2021, pp. 232–234.

[96] Xuechao Wei et al. "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs". In: *DAC*. 2017. DOI: `10.1145/3061639.3062207`.

[97] Xuan Yang et al. "Interstellar: Using halide's scheduling language to analyze dnn accelerators". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 369–383.

[98] Xuan Yang et al. "Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators". In: *ASPLOS*. 2020.

[99] H. Ye et al. "HybridDNN: A Framework for High-Performance Hybrid DNN Accelerator Design and Implementation". In: *DAC*. 2020. DOI: `10.1109/DAC18072.2020.9218684`.

[100] Manzil Zaheer et al. "Big bird: Transformers for longer sequences". In: *NeurIPS* (2020).

[101] Eberhard Zehendner. "Systolic Systems". In: *Algorithms of Informatics*. Vol. 2. 2005. Chap. 12.

[102] Guowei Zhang et al. "Gamma: Leveraging Gustavson's algorithm to accelerate sparse matrix multiplication". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2021, pp. 687–701.

[103] Xiaofan Zhang et al. "DNNBuilder: An Automated Tool for Building High-performance DNN Hardware Accelerators for FPGAs". In: *ICCAD*. 2018.

[104] Zhekai Zhang et al. "Sparch: Efficient architecture for sparse matrix multiplication". In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2020, pp. 261–274.

[105] Aojun Zhou et al. *Learning N:M Fine-grained Structured Sparse Neural Networks From Scratch*. 2021. arXiv: `2102.04010 [cs.CV]`.