



**FRIEDRICH-SCHILLER-  
UNIVERSITÄT  
JENA**

Faculty of Mathematics and Computer Science

**Fast Tensor Contractions  
on an  
XDNA Neural Processing Unit**

MASTER THESIS

for obtaining the academic degree

Master of Science (M.Sc.)

in the Computer Science degree program

Author: Tamino Steinert

Matriculation Number: 179670

First Reviewer: Prof. Dr. Alexander Breuer

Second Reviewer: Max Engel

Supervisor: Prof. Dr. Alexander Breuer

Jena, October 21, 2025

# Abstract

AMD's XDNA architecture is one of the leading-edge application-specific integrated circuit (ASIC) architectures available. It is shipped as a neural processing unit (NPU) on AMD's Ryzen AI processors. To utilise this NPU for fast tensor contractions, I have developed a framework that uses a tiled execution intermediate representation (TEIR) as interface. The TEIR can represent binary tensor contractions alongside a description of how to compute them. This enables the optimisation choices made during an optimisation phase to be encoded. To compile the TEIR into XDNA machine code, it is first lowered into MLIR-AIE code using AMD's *IRON* API. A general matrix-matrix multiplication (GEMM) kernel is also created by compiling either a hand-optimised assembly code or a vectorised *AIE API* code. The MLIR-AIE code using the GEMM kernel at its core is then compiled into XDNA machine code. The resulting NPU kernels achieve up to 78.5% hardware utilisation and 5,670 GFLOPS. The GEMM kernels compiled from assembly code or *AIE API* code achieve up to 92.8% and 90.5% hardware utilisation, respectively.

## Zusammenfassung

Die XDNA-Architektur von AMD ist eine der führenden ASIC-Architekturen (Application-Specific Integrated Circuit) auf dem Markt. Sie wird als neuronale Verarbeitungseinheit (NPU) auf den Ryzen-AI-Prozessoren von AMD ausgeliefert. Um diese NPU für schnelle Tensor-Kontraktionen zu nutzen, habe ich ein Framework entwickelt, das eine TEIR-Schnittstelle (Tiled Execution Intermediate Representation) verwendet. Die TEIR kann binäre Tensorkontraktionen zusammen mit einer Beschreibung ihrer Berechnung darstellen. Dadurch können die während einer Optimierungsphase getroffenen Optimierungsentscheidungen kodiert werden. Um die TEIR in XDNA-Maschinencode zu kompilieren, wird sie zunächst mit der IRON-API von AMD in MLIR-AIE-Code umgewandelt. Ein allgemeiner Matrix-Matrix-Multiplikations-Kernel (GEMM) wird ebenfalls erstellt, indem entweder ein handoptimierter Assemblercode oder ein vektorisierter AIE-API-Code kompiliert wird. Anschließend wird der MLIR-AIE-Code, der den GEMM-Kernel als Kern verwendet, in XDNA-Maschinencode kompiliert. Die resultierenden NPU-Kernel erreichen eine Hardwareauslastung von bis zu 78,5% und 5.670 GFLOPS. Die aus Assemblercode oder AIE-API-Code kompilierten GEMM-Kernel erreichen eine Hardwareauslastung von bis zu 92,8% bzw. 90,5%.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Background</b>	<b>6</b>
2.1	AMD XDNA NPUs . . . . .	6
2.1.1	XDNA Architecture . . . . .	6
2.1.2	Software Integration . . . . .	8
2.2	Einstein Summation Convention . . . . .	9
<b>3</b>	<b>Tensor Contractions on XDNA NPUs</b>	<b>12</b>
<b>4</b>	<b>GEMM Kernels on XDNA NPUs</b>	<b>13</b>
4.1	VLIW ISA . . . . .	13
4.2	GEMM Assembly Kernels . . . . .	15
4.2.1	Operation Packing . . . . .	15
4.2.2	4×4 Blocking of Output Matrix . . . . .	15
4.2.3	K-loop Unrolling . . . . .	17
4.2.4	M-N-loop Fusing with Branchless Pointer Updates . . . . .	17
4.2.5	Reducing Bank Conflicts . . . . .	19
4.3	GEMM <i>AIE API</i> Kernel . . . . .	19
<b>5</b>	<b>TEIR Lowering to MLIR-AIE</b>	<b>20</b>
5.1	Data Movement with MLIR-AIE . . . . .	20
5.2	TEIR Restrictions . . . . .	21
5.3	FIFO-queue-based Loops . . . . .	22
5.4	Uneven Distributed Dimensions . . . . .	24
<b>6</b>	<b>Evaluation</b>	<b>27</b>
6.1	Hardware and Software . . . . .	27
6.2	GEMM Assembly Kernels . . . . .	28
6.3	GEMM <i>AIE API</i> Kernels . . . . .	28
6.4	Binary Tensor Contractions . . . . .	30
<b>7</b>	<b>Discussion and Outlook</b>	<b>34</b>
<b>8</b>	<b>References</b>	<b>35</b>
<b>9</b>	<b>Appendix</b>	<b>38</b>

# 1 Introduction

As demand for AI applications continues to grow, the traditional processing units (PUs), such as CPUs and GPUs, struggle to keep up with the computational demands of these tasks. Consequently, chip companies are focusing on developing new customised hardware for these workloads [1–3]. This new type of PU, known as a neural processing unit (NPU), is optimised for the highly parallel and matrix-heavy computations that are fundamental to neural networks. NPUs offer significantly higher performance and energy efficiency compared to general-purpose processors, enabling faster training and inference of AI models. AMD’s XDNA NPUs are one of the leading-edge NPUs available, and are shipped on Ryzen AI system on a chips (SoCs) to enhance their performance on AI workloads. [4]

However, having a high-performing PU in the system does not guarantee that code is executed efficiently on the hardware. The code must be optimised for the hardware to achieve the full potential of it. AMD provides an MLIR-AIE repository [5], containing tools and examples to help developers write efficient code for their XDNA NPUs. This work is based on the matrix multiplication example in the MLIR-AIE repository. This example code can only handle matrix-matrix multiplication and imposes strict limitations on dimension sizes. In order to enable a wider range of machine learning workloads, I created a framework to handle binary tensor contractions. This framework uses the loop-over-general matrix-matrix multiplication (GEMM) approach to compile binary tensor contractions for a XDNA NPU. Furthermore, I adapted the GEMM kernels to impose fewer restrictions on GEMM sizes and enable all matrix layout configurations.

Specifically, I make the following contributions:

- I present a lowering algorithm for binary tensor contractions on the XDNA NPU. The interface is a tiled execution intermediate representation (TEIR) [6] adapted for the NPU, which is then lowered to MLIR-AIE code. This MLIR-AIE code is then compiled with a GEMM kernel into two ELF files containing the machine code for the NPU. The NPU kernel is then executed as specified in the TEIR.
- I provide two high-performance GEMM assembly kernels for XDNA NPUs, along with a detailed description of the applied optimisations.
- I improve GEMM *AIE API* kernel code to handle all combinations of transposed and non-transposed layouts and allow more GEMM sizes to be compiled.
- I evaluate the GEMM kernels and the NPU kernels created by the TEIR lowering algorithm on a *Phoenix* SoC. I provide an instruction-detailed breakdown of the performance of the assembly kernel. The two assembly kernels achieve 92.8% and 90.5% of the theoretical peak performance (TPP) in microbenchmarks. The NPU kernels achieve up to 78.5% hardware utilisation and 5,670 GFLOPS.

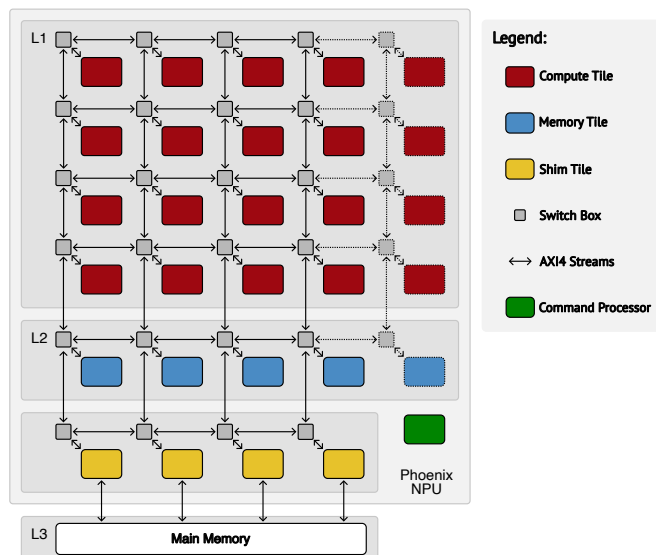
## 2 Background

### 2.1 AMD XDNA NPUs

Following its acquisition of Xilinx [7], AMD adapted and further optimised the Versal AI Engine (AIE) for machine learning (ML) workloads [8]. The AI Engine-Machine Learning (AIE-ML) is used in the AMD Ryzen 7040 series, which is the first x86 processor to feature an integrated NPU [4]. This work focuses on this first generation of NPUs, which was released on the SoC codenamed *Phoenix*.

#### 2.1.1 XDNA Architecture

The NPU of *Phoenix* uses the spatial dataflow architecture AMD XDNA. The XDNA architecture is described in the architecture manual[8] and technical paper "AMD XDNA NPU in Ryzen AI Processors"[4].



**Fig. 1: Overview of the XDNA Architecture on *Phoenix*, showing compute tiles (L1), memory tiles (L2), shim tiles and a separate command processor. The tiles are connected via AXI4 streams, which are configured by the switch boxes. Dotted components are not longer supported by the XDNA driver. (adapted from [9])**

**Layout.** The XDNA architecture has a 2D array layout consisting of AIE-ML compute tiles, AIE-ML memory tiles, and the AIE-ML interface tiles. A high-level overview is shown in Fig. 1. The AIE-ML interface tile, otherwise referred to as a shim tile, consists of a network on a chip (NoC) tile, a configuration tile, and a programmable logic (PL) tile. On the NPU of *Phoenix* SoC, the compute tiles are arranged in grid of five columns and four rows. Each column has a memory tile, and four columns have a shim tile.[9] The XDNA driver dropped support for the column without a shim tile with commit 2019485. As the driver no longer supports

using the column without a shim tile, only designs with the four columns will be considered, i.e all implementations use at most a  $4 \times 4$  grid of compute tiles.

**Compute Tiles.** Each AIE-ML compute tile contains a scalar unit, a vector unit, two load units, a store unit, and an instruction unit. Fig. 2 shows an overview of the units on one compute tile. The scalar unit is a simple 32-bit integer RISC processor. The vector unit supports multiple fixed- and floating-point precisions, and can issue one single instruction multiple data (SIMD) operation every cycle. Each of the 256-bit load and 256-bit store units has an address generator unit (AGU) which support on-the-fly pointer updates with a one-cycle latency. A Very Long Instruction Word (VLIW) Instruction Set Architecture (ISA) is used to manage the units. The VLIW ISA is described in detail in Section 4.1.

Each compute tile has 16 KB dedicated instruction memory (IMEM) and 64 KB data memory (DMEM). The DMEM is organized in four banks and acts as L1 memory. A compute tile also has full access to the DMEM of the neighboring compute tiles via the load and store units.

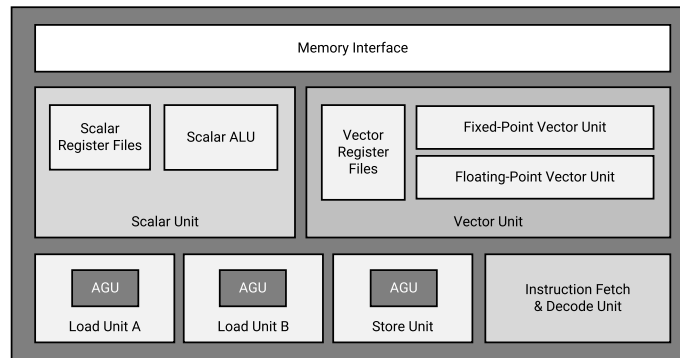


Fig. 2: Overview of functional units on a compute tile. [8]

**Register Files.** Each compute tile has several types of registers which fulfil different purposes. A complete list of the registers and their memory address layouts can be found in the register reference [10]. This section focuses on their role in ISA.

The vector unit has 12 512-bit vector registers prefixed with  $x$ , where each of these 512-bit registers can be accessed as a pair of two 256-bit registers prefixed with  $wl$  and  $wh$ . For instance, the 512-bit register  $x2$  can be accessed via the two 256-bit registers  $wl2$  and  $wh2$ , where  $wl2$  overlaps with the lower 256 bits and  $wh2$  with the higher 256 bits. In addition to the vector registers, the vector unit has accumulator registers, which can also be accessed at different bit widths. There are 8 1024-bit accumulator registers prefixed with  $cm$ . The lower and higher 512 bits of each of these registers can be accessed using the corresponding  $bm1$  and  $bm1$  registers. Each 512-bit register is also split into two 256-bit registers with the corresponding prefixes  $am1l$ ,  $am1h$ ,  $amh1$ , and  $amhh$ .

For scalar instructions there are 32 32-bit general-purpose registers ( $r0 - r31$ ), 8 20-bit modifier registers ( $m0 - m7$ ), 8 20-bit pointer registers ( $p0 - p7$ ) and other special registers.

The instruction unit has three registers for hardware loops. Two of them are 20-bit registers for the start address (**1s**) and end address (**1e**) of the loop, i.e. the addresses of the first and last instruction of the loop. The third one is a 32-bit register for the loop counter (**1c**).

**Memory Tiles.** Each memory tile has 512KB DMEM arranged into 16 banks and acts as L2 memory.

**Shim Tiles.** Shim tiles can move data between L3 (main memory) and L2 or L3 and L1. The *Phoenix* NPU has access to main memory through four LPDDR5x/DDR5 memory controllers with a 128-bit wide interface. This interface is shared with the CPU and GPU and provides a peak bandwidth of 120 GB/s. This interface allows zero-copy offloading from the CPU to the NPU.

**Data Transfer.** Each tile has a dedicated direct memory access (DMA) engine, which enables the data transfers between all tiles. It is primarily used to transfer data between L3 and L2, and between L2 and L1. The data transfers through a shim tile is handled by its DMA controller, which has access to 16 buffer descriptors (BDs).

Data is moved using AXI4-Streams, which are configured via AXI4-Stream Interconnects (also called switch boxes). A switch box on a shim tile and on a compute tile has two ports for reading from DMA streams and two ports for writing to DMA streams. On a memory tile, a switch box has six ports for each. Switch boxes have additional external ports that are used to connect different streams of neighboring tiles. They can also be configured to broadcast data from one input port to multiple output ports. Each AXI4-Stream can transfer 32 bits of data in one cycle, i.e. one stream has a bandwidth of 4 GB/s on 1 GHz clocked device.

### 2.1.2 Software Integration

AMD provides two software stacks [5, 11] for the XDNA NPUs. This work focuses on the workflow around the open-source MLIR-based toolchain [5].

**MLIR-AIE.** MLIR-AIE is a multi-level intermediate representation (MLIR) dialect developed by AMD. It is specifically designed for AMD’s NPUs. AMD created *IRON*, a Python-based API with a FIFO-queue-based interface for data transfer, to enable the creation of MLIR-AIE designs. [5] It is also possible to directly write scalar instructions for kernels in Python. The Python scalar instructions get transformed to LLVM code and are later compiled.

**AIE-ML Intrinsics.** To get high performing code, the vector units of the compute tiles have to be used. To enable this, AMD provides the *AIE API* [12], which comes as a header-only library and enables to use hardware intrinsics in C++.

**Compilers.** There are two compilers that can compile code for the compute tiles: One is AMD’s proprietary *xchesscc* compiler; the other is *peano*, an open-source LLVM-based compiler. Both compilers have support for with SIMD intrinsics and heuristic VLIWs packing. The lowering process from code in the

MLIR-AIE dialect to machine code is handled by *aiecc*, which is a compiler driver for MLIR tools. *aiecc* gets the MLIR code with any referenced object files and compiles them into one AIE program ELF file and one ELF file containing the kernels for the compute tiles.

**XDNA Driver and Xilinx Runtime (XRT).** AMD provides a Linux driver [13] and a runtime API for C, C++, and Python[14] to enable control over the NPU.

## 2.2 Einstein Summation Convention

Einstein summation convention, in short *einsum*, describes a multidimensional summation over indexed products.

***einsum* Notation.** Tab. 1 shows *einsum* expressions for some linear algebra examples. It operates on tensors. In *einsum* a tensor is a multidimensional array defined by a list dimensions. In the notation, each dimension is represented by a label, commonly a letters. That allows to represent a tensor by a list of labels. In an *einsum* expression, the input tensors, represented by their label lists, are comma-separated, followed by an arrow and the dimension labels of the output tensor. If multiple tensors share the same dimension, the same label is used. If a dimension does not appear in the output tensor, it is summed over. The rules of *einsum* are very simple, which makes it easy to use. Einstein used his original notation to reason about the dimensions of tensors, rather than to calculate them explicitly [15]. Today, there are many libraries that use this notation to simplify complex computations [16–20].

Operation	Math.	Summation	<i>einsum</i>
matrix-matrix product	$A \cdot B$	$c_{pr} = \sum_q a_{pq} b_{qr}$	pq,qr->pr; A,B
transp. mat.-mat. prod.	$A^T \cdot B$	$c_{pr} = \sum_q a_{qp} b_{qr}$	qp,qr->pr; A,B
Hadamard product	$A \odot B$	$c_{pq} = a_{pq} b_{pq}$	pq,pq->pq; A,B
sq'd Frobenius norm	$\ A\ _F^2$	$c = \sum_p \sum_q a_{pq} a_{pq}$	pq,pq->; A,A
Khatri-Rao product	$A \odot B$	$c_{pqr} = a_{pq} b_{qr}$	pq,qr->pqr; A,B
Kronecker product	$A \otimes B$	$c_{pqrs} = a_{pq} b_{rs}$	pq,rs->pqrs; A,B

Tab. 1: *einsum* Examples (adapted from [21])

**Computing *einsum* Expressions.** *einsum* expressions can have multiple input tensors and are usually broken down into binary tensor contractions [22]. Finding such a contraction sequence with minimum computational cost is an NP-hard problem [23]. However there are heuristics that find good contraction paths [24–26].

Once a contraction sequence has been selected, the next step is to decide the data layout of each intermediate tensor. Assuming all tensors are stored contiguously using general row-major storage, as most frameworks do, deciding the order of the dimensions of the tensor is the same as deciding the data layout. Dimensions can

also be split or fused, if applicable. This optimisation is important because the data layout significantly impacts the performance of the contraction. [21]

The final step is to create a kernel for each binary contraction that can efficiently compute it. A kernel for a binary contraction is usually created using the loops-over-GEMMs approach, where first a GEMM kernel is generated, and then the GEMM kernel is called within nested loops with updated pointers. The difficulties in this step are finding the optimal dimension sizes for the GEMM to generate a high-performing GEMM kernel, creating the GEMM kernel itself, and determining the order of the nested loops. The optimal order of the nested loops depends heavily on the compute throughput of the compute units and on the throughput and latencies of the memory and cache hardware. The best dimension sizes for the GEMM and the kernel generation depend heavily on the compute units, as most hardware has restrictions corresponding to dimension sizes, such as the vector lengths on vector units. [21]

The decisions made during optimisation can be represented using the TEIR. In the TEIR, a one binary contraction is represented by a tiled execution configuration (TEC). Fig. 3 shows the original TEIR domain and notation specification. The order of the dimensions in a TEC matches the loop order of the optimisation. The values in `dim_sizes`, `strides_in0`, `strides_in1`, and `strides_out` define the size and stride of each dimension of each tensor, respectively. The strides are set to 0 if a dimension does not exist in the tensor. The values of `dim_types` can be inferred from the tensor strides. The rule is shown in Tab. 2.

$$\begin{aligned}
 D &\in \mathbb{N}^+ \\
 \text{dim\_types} &= (t_0, \dots, t_{D-1}), t_i \in \{\mathbf{C}, \mathbf{M}, \mathbf{N}, \mathbf{K}\} \\
 \text{exec\_types} &= (e_0, \dots, e_{D-1}), t_i \in \{\mathbf{seq}, \mathbf{shared}, \mathbf{prim}\} \\
 \text{dim\_sizes} &\in (\mathbb{N}^+)^D \\
 \text{strides\_in0} &\in \mathbb{N}^D \\
 \text{strides\_in1} &\in \mathbb{N}^D \\
 \text{strides\_out} &\in \mathbb{N}^D \\
 \text{data\_type} &\in \{\mathbf{FP32}, \mathbf{FP64}\} \\
 \text{prim\_first} &\in \{\mathbf{None}, \mathbf{Zero}, \mathbf{ReLU}\} \\
 \text{prim\_main} &\in \{\mathbf{None}, \mathbf{Copy}, \mathbf{GEMM}, \mathbf{BRGEMM}\} \\
 \text{prim\_last} &\in \{\mathbf{None}, \mathbf{ReLU}\}
 \end{aligned}$$

**Fig. 3: Original TEIR domain and notation specification [6].**

There are further restrictions for a valid TEC. For example, in a TEC that loops over a GEMM, the value of `prim_main` must be set to GEMM. The last three dimensions must have `exec_types` set to `prim` and must have M, N, and K set as `dim_types`, respectively. These three dimension define the GEMM kernel, with their `dim_sizes` defining the kernel size.

I use a modified TEIR, which is adapted for the use on the NPU. Section

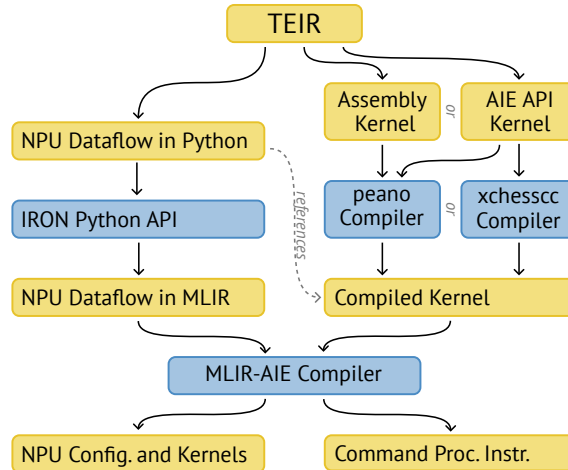
Type	Dim. exists in		
	in0	in1	out
C	✓	✓	✓
M	✓	×	✓
N	×	✓	✓
K	✓	✓	×

**Tab. 2:** The rules for inferring the dimension type from the strides of the tensors. A dimension exists in a tensor if the strides of the dimension in the tensor are non-zero.

5.2 states and explains the changes made to the original TEIR domain, and lists additional restriction on the TEIR.

### 3 Tensor Contractions on XDNA NPUs

In this section, the process of executing a tensor contraction described by a TEC is characterised. Section 4 describes the creation of the GEMM kernel, and Section 5 describes the generation of the MLIR-AIE code from a TEC. When compiling MLIR-AIE code with the GEMM kernel, a program ELF file and a kernel ELF file containing the kernels for the compute tiles are created. Fig. 4 shows an overview of the compilation process.



**Fig. 4: Overview of lowering and compilation process of a TEC. Files and intermediate representations are shown in yellow and used tools are shown in blue. (adapted from [9]).**

The XRT API is used to configure and control the NPU. First, a device handle for the NPU is acquired. Next, the kernel ELF file is loaded and registered to the device, and the kernel handle is acquired. Then, the program ELF file containing the shim tile data movement instructions is loaded into an XRT buffer object. The memory of an XRT buffer object must be 4096-byte aligned and cannot exceed 4 GiB in size.

The kernel is executed by calling the kernel handle with the program buffer object, its size and three buffer objects containing the tensors. All buffer objects must be synchronised before the first kernel call and after the last. The sequential loops in the TEC are realised with recursive function calls around the kernel call and sub-buffer objects of the tensors. A sub-buffer object allows offsets in a buffer object to be specified.

## 4 GEMM Kernels on XDNA NPUS

This section focuses on the creation of GEMM kernel for binary tensor contractions on the XDNA NPU. It starts with a description of how to program the hardware features present on a compute tile, explains how they can be used to create a highly optimised GEMM kernel in assembly language, and characterise how to write a general GEMM kernel with intrinsics through the *AIE API*.

When using either of the two compilers, it is only possible to specify hardware intrinsics. It is not possible to specify how the instructions should be packed into VLIWs. VLIWs are generated during the optimisation passes over which the programmer has no direct control. The only influence that programmers have over VLIW packing in the optimisation passes is to order the instructions in a way that helps the compiler find good VLIW packing. However, it is possible to create assembly files containing complete VLIWs, which gives programmers full control over VLIW packing. These assembly files can then be compiled using *p/aseano*.

### 4.1 VLIW ISA

The compute unit on the NPU uses a VLIW ISA with static in-order scheduling to exploit instruction-level parallelism (ILP). One VLIW can contain between 2 and 16 bytes and can contain up to six operations, one for each functional unit. If a VLIW does not specify an operation for a unit, a no operation (NOP) is implied. This decreases the program size.

Tab. 3 provides examples of operations for the compute tile. A VLIW can comprise of up to six operations, given as a semicolon-separated list of operations in the assembly code. In the assembly code, a suffix is used to indicate the corresponding unit in some operations: **a** and **b** for the load units; **s** for the store unit; **x** and **m** are for the scalar unit, with **x** controlling the arithmetic logic unit (ALU) and **m** used for register movement. The two operations for the scalar unit can be combined to allow for larger immediate values. Operations that interact with the vector unit have a **v** as a prefix.

The branching operations in this ISA have a latency of six cycles, meaning the next five instructions after the jump are always executed and they cannot include another jump operation. The five instructions in the delay slots can perform useful operations. The destination of a jump instruction must be 16-byte-aligned. The instruction unit features hardware loops designed to reduce the overhead associated with jumps in loop-like control flows. This hardware is programmed using the three loop registers. When the start and end address of the loop body are stored in the loop start and end register, and the loop count register is greater than one, the instruction unit jumps from the end address to the start address, decrementing the loop count register when the end instruction is reached. This occurs with no additional overhead. The first and last instruction of a loop must be 16-byte-aligned, and the last instruction must be 16 bytes in size. This is because the instruction unit fetches 16-byte-wide instruction values, and all jumps must be 16-byte-aligned.

The most important operation for a GEMM kernel is the multiply-accumulate (MAC) operation. A scalar MAC operation multiplies two registers, adds the result

Operation / Syntax	Notes	Latency
<b>nop</b> - no operation		
nop	do nothing	-
nop<a b s x m xm v>	do nothing in unit	-
<b>mov</b> - move		
mov <dest. r_>, <src. r_>	conflicts with nopm	1
mov <dest. r_>, #<immediate>	10b immediate	1
mov<a x xm> <dest. r_>, <src. r_>		1
mov<a x xm> <dest. r_>, #<immediate>	a,x: 8b imm.; xm: 32b imm.	1
vmov <dest. vec. reg.>, <src. vec. reg.>	registers have to be the same size	1
<b>ld</b> - load		
ld<a b> <r_>, [<p_>], <m_>	inc p_ by m_ after load	6
ld<a b> <r_>, [<p_>], #<immediate>	inc p_ by immediate after load	6
ld<a b> <r_>, [<p_>, <dj_>]	dj_ as offset	6
ld<a b> <r_>, [<p_>, #<immediate>]	immediate as offset	6
vld<a b> <w_>, [<p_>], <m_>		7
...		
vlda.conv.fp32.bfloat16 <bm_>, [<p_>], <m_>	converts bfloat16 to fp32	5
...		
<b>st</b> - store		
st <r_>, [<p_>], <m_>		1
...		
st.s16 <r_>, [<p_>], <m_>	stores lower 16-bit	1
...		
vst <w_>, [<p_>], <m_>		1
...		
vst.conv.bfloat16.fp32 <bm_>, [<p_>], <m_>	converts fp32 to bfloat16	1
<b>shuffle</b> - shuffle		
vshuffle <x_ bm_>, <x_>, <x_>, <r_>	r_ as config #28: <4x8>-<8x4>; #29: <8x4>-<4x8>	2
<b>add</b> - addition		
add <r_>, <r_>, <r_>	conflicts with nopx	1
...		
<b>padd</b> - pointer add		
padd<a b s> [p_], m_	value must be multiple of 4	1
padd<a b s> [p_], #<immediate>	a,s: 11b imm.; b: 10b imm.	1
<b>mul</b> - multiplication		
mul <r_>, <r_>, <r_>	conflicts with nopx	1
...		
<b>mac</b> - multiply accumulate		
vmac <cm_>, <cm_>, <x_>, <x_>, <r_>	r_ as config #776: int16<8x8>+=int8<4x8>*int8<8x8>	6
vmac.f <bm_>, <bm_>, <x_>, <x_>, <r_>	r_ as config #28: fp32<4x4>+=bf16<4x8>*bf16<8x4>	6
<b>comparisons</b>		
<gt lt ge le><u?> <r_>, <r_>, <r_>	> < >= <=; unsigned? conflicts with nopx	1
<b>j</b> - jump		
ret lr	return to address in link reg. conflicts with nopx	6
jnz <r_>, #<label>		6
...		

Tab. 3: Example operations of the AIE-ML ISA. The texts in angle brackets (<>) are placeholders for specific registers, unit suffixes or numbers. The bar (|) indicates an option.

to an accumulation register, and stores the final result in an accumulation register. Vectorised MAC operations on this hardware can perform a small matrix-matrix multiplication. For instance, the vector unit supports floating-point matrix-matrix multiplication, where the first vector register is treated as a  $4 \times 8$  `bfloat16` matrix, the second register is treated as an  $8 \times 4$  `bfloat16` matrix, and a  $4 \times 4$  `fp32` matrix is computed. This computation is performed when the scalar register in the `vmac.f` operation contains the value 28 and has the equal value of operations to 128 scalar MACs, achieving 256 floating-point operations (FLOPs) per cycle.

## 4.2 GEMM Assembly Kernels

The compute unit cannot stall an instruction, i.e. it issues an instruction every cycle, and each operation in an instruction has a fixed latency. This requires dependency conflicts to be resolved beforehand, with operations being scheduled accordingly. When there are no bank conflicts, the number of executed instructions is equal to the number of cycles needed to execute these instructions, and a number of cycles yields a fixed amount of time, since the NPU has a fixed clock frequency. Therefore, the program’s performance in terms of floating-point operations per second (FLOPS) can be estimated by counting the instructions executed. TPP is achieved by issuing the instruction with the most FLOPs every cycle. This is the `vmac.f` operation with the  $4 \times 8 \times 4$  layout. This results in 256 GFLOPS on a 1 GHz clock.

I created two highly-optimised example GEMM kernels with  $|M| \times |K| \times |N|$  set to  $64 \times 64 \times 64$  and  $64 \times 80 \times 64$ , which have a `vmac.f` operation in 96.3% and 95.3% of their executed instructions. They have a maximum of 80 bank conflicts, which should result in a performance above 93.4% and 92% of the TPP. The detailed calculations for the kernel size  $64 \times 80 \times 64$  are shown in Section 6.2. The optimisations applied to kernels to achieve this are described below.

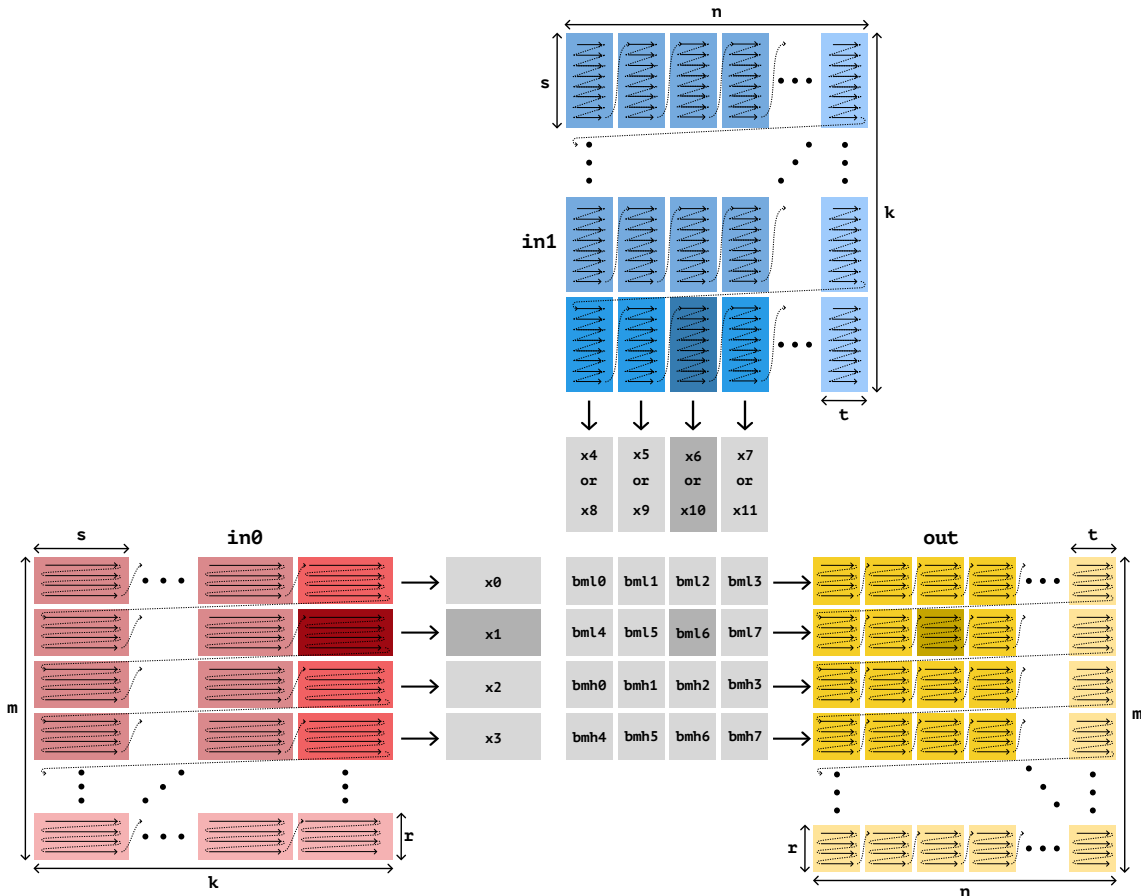
### 4.2.1 Operation Packing

As the compute tile uses VLIWs, it is possible to exert full control over the execution order and operations within an instructions. The vector unit is the unit with the greatest computational capability, and its most compute-intensive operation in terms of FLOPS is the `vmac.f` operation. Therefore, each instruction should contain a `vmac.f` operation, with other operations arranged around it. To achieve effective operation packing, the reuse rate of registers must be high, and operations must utilise the full potential of the units. For instance, instead of using a load operation (`vlda w0, [p0, #0]`) followed by a pointer update operation (`paddda [p0], m0`), a single load-with-pointer operation (`vlda w0, [p0], m0`) should be used.

### 4.2.2 $4 \times 4$ Blocking of Output Matrix

The input tiles for the GEMM arrive at the DMEM in a specific data layout. The data movement through the DMAs is described in Section 5.3. Fig. 5 shows the data layout of the tiles in the DMEM. The tiles themselves are self tiled, with the sub-tiles having their corresponding sub-matrix layout of the `vmac.f` operation. The data layout is equal to the *einsum* expression `mkrs, knst->mnrt` with sizes

$|M|/4, |N|/4, |K|/8, 4, 8, 4$  in alphabetical order, where  $r, s,$  and  $t$  correspond to their respective dimensions in the `vmac.f` operation and  $m, k,$  and  $n$  correspond to the other parts of the split dimensions  $M, K,$  and  $N$  of the GEMM tiles.



**Fig. 5:** Data layout and access pattern of tensors in DMEM. The first input, second input and output is colored red, blue and yellow, respectively. The register are colored gray. The sub-matrix sizes  $4 \times 8 \times 4$  for the `vmac.f` operation are annotated as  $r, s$  and  $t$ , respectively. The labels  $m, k,$  and  $n$  correspond to the other parts of the split dimensions  $M, K,$  and  $N$  of the GEMM tiles. The access in first  $M-N$  iteration and last  $K$  iteration is highlighted. Four sub-matrices of the first input are multiplied pairwise with four sub-matrices to compute  $4 \times 4$ -blocked sub-matrices of the output.

To achieve a high reuse rate on the vector and accumulation registers, a  $4 \times 4$  blocking of the output matrix is used in both GEMM kernels. Fig. 5 shows how the sub-matrices of the two input matrices and the output matrix are distributed across the registers. The first four sub-matrices of both input matrices are loaded into four vector registers each. The sub-matrices of the first input matrix align along the  $M$  dimension and the sub-matrices of the second input matrix align along the  $N$  dimension. The corresponding sub-matrices of the output matrix are loaded into the 16 accumulation registers and form a  $4 \times 4$  blocking along the  $M$  and  $N$  dimension. The input sub-matrices are then multiplied, where sub-matrices of the

first input are multiplied pairwise with the sub-matrices of the second input. Their results are added to the  $4 \times 4$ -blocked output sub-matrices. Then, the sub-matrices of both input matrices are strided for the next computations in the K-dimension loop. The process of loading the input sub-matrices, multiplying them, adding them to the output sub-matrices and shifting is repeated until the end of the K-dimension is reached. Afterwards, the 16 output sub-matrices are written back. The output sub-matrices are then strided, either for M or N dimension iteration, and the input sub-matrices are strided accordingly. Then, the next K-loop is computed. This whole process is repeated until the end of both the M and N dimension is reached.

In order to have a `vmac.f` in every cycle of a K-iteration, vector registers for the first instructions of a K-iteration must be loaded. This is achieved by loading the relevant vector registers in the previous iteration. Two sub-matrices of the first input matrix and all four sub-matrices of the second input matrix are loaded in the previous iteration. There are 12 512-bit vector registers available, allowing eight to be used for the second input. In each K-iteration, four of these eight vector registers are used to preload registers for next the K-iteration, while the other four are used to calculate the output sub-matrices. In the next K-iteration, the roles are switched. In order to preload the six vector registers for the first K-iteration, a warm-up phase is required.

### 4.2.3 K-loop Unrolling

To enable operator packing for the load and store instructions for the output sub-matrices, all K-blocks must be written down explicitly. As the load operation has a latency of five cycles, five of the output sub-matrices are loaded in the warm-up phase, with the remaining sub-matrices loaded inside the first K-iteration. As the `vmac.f` operation has a latency of six cycles, six store operations cannot be fused into the last K-iteration. The six input preload operations in the last K-iteration are adapted to load the corresponding input sub-matrices of the next iteration. There are five additional load operations in the last K-iteration, which preload the five output sub-matrices for the next M or N dimension iteration. This enables the subsequent preload phases to be skipped and the next iteration to start directly at the beginning of the first K-iteration.

### 4.2.4 M-N-loop Fusing with Branchless Pointer Updates

To avoid branching within the kernel that is not handled by hardware loops, the M-loop and N-loop are fused into one single loop, and the pointer updates are managed using branchless programming techniques. Listings 1, 2, and 3 demonstrate the two-step transformation from the C++ code with nested-loop code (Listing 1) into fused-loop C++ code with branchless pointer updates (Listing 3).

The C++ code after the first step of the transformation is shown in Listing 2. The rules for this transformation step are as follows: When two loops are fused, the iteration count of the fused loop is the product of the original loop counts. If the iteration variable `mn_i` is set to zero at the start and the iteration count of the inner loop is `N`, the operations inside the outer loop and before the inner loop must only be executed if `(mn_i % N == 0)` is true. Likewise, the operations inside the

```

1 for (int m_i=0; m_i<M; m_i++) {
2   for (int n_i=0; n_i<N; n_i++) {
3     // unrolled in kernel
4     for (int k_i=0; k_i<K; k_i++) {
5       ...
6     }
7     p_in0+=n_offset_in0;
8     p_in1+=n_offset_in1;
9     p_out+=n_offset_out;
10  }
11
12  p_in0+=m_offset_in0;
13  p_in1+=m_offset_in1;
14  p_out+=m_offset_out;
15
16 }

```

**List. 1: C++ Example of Nested M-N-Loops**

```

1 for (int mn_i=0; mn_i<M*N; mn_i++) {
2
3   // unrolled in kernel
4   for (int k_i=0; k_i<K; k_i++) {
5     ...
6   }
7   p_in0+=n_offset_in0;
8   p_in1+=n_offset_in1;
9   p_out+=n_offset_out;
10
11   if (mn_i % N == N - 1) {
12     p_in0+=m_offset_in0;
13     p_in1+=m_offset_in1;
14     p_out+=m_offset_out;
15   }
16 }

```

**List. 2: C++ Example of Fused M-N-Loop with Branching Pointer Updates**

```

1 for (int mn_i=0; mn_i<M*N; mn_i++) {
2
3   // unrolled in kernel
4   for (int k_i=0; k_i<K; k_i++) {
5     ...
6   }
7   p_in0+=n_offset_in0;
8   p_in1+=n_offset_in1;
9   p_out+=n_offset_out;
10
11   int apply_m_offset=(int) (mn_i % N == N - 1);
12   p_in0+=apply_m_offset*m_offset_in0;
13   p_in1+=apply_m_offset*m_offset_in1;
14   p_out+=apply_m_offset*m_offset_out;
15
16 }

```

**List. 3: C++ Example of Fused M-N-Loop with Branchless Pointer Updates**

outer loop and after the inner loop must only be executed if  $(mn\_i \% N == N - 1)$  is true. This can be achieved by having the operations inside the outer loop and outside the inner loop within two if-blocks with the corresponding condition.

The next step is to remove the branching of the if-blocks. If the if-blocks contain only basic arithmetic operations, these operations can be transformed so that they have an effect only when the if-statement evaluates to true, allowing them to be executed unconditionally. A simple addition increment can be transformed by first evaluating the if-statement and treating the result as an integer value. This integer value is then multiplied with the summand and the result is added to the variable. The resulting C++ code is shown in Listing 3.

This entire transformation is applied to the M-loop and N-loop, resulting in a single large fused loop with no branching. The hardware loop is used for this fused loop to have no branching overhead.

### 4.2.5 Reducing Bank Conflicts

As this hardware has four banks in the DMEM, and to minimise bank conflicts, only one load operation is performed per cycle in the inner K-blocks. However, multiple load operations cannot be avoided in the first and last K-iteration, due to the loading and storing of the output matrix. In the first K-iteration, only load operations from different pointers occur within the same cycle. When the data resides in different banks, there is no bank conflict. In the last K-iteration, the accumulation registers must be written and preloaded for the next iteration. Five cycles write to and load from the output matrix, which is likely to result in bank conflicts. In the current implementation, these bank conflicts can only be avoided by using two cycles for each of the five instructions, resulting in five extra cycles. This cancels out the benefit of reducing bank conflicts, given that bank conflicts result in one extra cycle per each conflicting operation.

## 4.3 GEMM AIE API Kernel

It is not feasible to write all possible GEMM kernels by hand in assembly language, given the sheer number of different sizes and configurations. Therefore, an example GEMM kernel was adapted [5]. Written with the *AIE API*, it uses a fixed  $4 \times 4$  blocking of sub-matrices for the output matrix and can handle a transposed second input and output matrix. This kernel was adapted to also handle a transposed first input matrix and all sizes for dimensions M and N that are divisible by 4, as this is the sub-matrix size in these dimensions.

At its core, the adapted kernel also uses a  $4 \times 4$  blocking for the output matrix, but also has an edge treatment for the non-fitting blocks. First, the edge treatment ensures that all remaining blocks can be computed efficiently with  $4 \times 4$  blocking. Where possible, it uses the `constexpr` keyword to inform the compiler that the evaluation of the if-statements does not change at runtime, e.g. whether an edge treatment is needed for pointer updates regarding transpose options. In the source code, the loads and corresponding pointer updates are positioned next to each other to facilitate operator packing into a single load operation with pointer update, rather than two separate operations, during compilation. There are also optimisation hints on the for-loops to help the compiler achieve higher performance.

## 5 TEIR Lowering to MLIR-AIE

This section describes how to lower a TEC to a high-performing MLIR-AIE code for the NPU. This is achieved using an optimised GEMM kernel from the previous section and the loops-over-GEMM approach.

### 5.1 Data Movement with MLIR-AIE

Data movement between NPU tiles is programmed using MLIR-AIE with the abstraction *ObjectFIFO*, which uses `object_fifo` instructions to describe data movement between tiles and `object_fifo_link` instructions for linking `object_fifos` together to create complex data flows. The `npu_dma_memcpy_nd` instructions define read and write data movement between the NPU and the main memory.

`object_fifo`. An `object_fifo` instruction is compiled into DMA configurations. `object_fifo` parameters include a name, the producer tile, a list of consumer tiles, a buffer depth, a multidimensional array description, and a data layout transformation. The multidimensional array description defines the size, data layout, and data type of the transferred memory. A data layout transformation can have up to four dimensions, which are specified as dimension sizes and strides. This describes the data layout of the input data, and the output data is transformed into a contiguous general row-major layout with matching sizes. The reading DMA performs this transformation. Switch boxes are used for broadcasting to multiple receiving tiles. The buffer depth parameter specifies the number of ring buffer elements for this FIFO queue.

`object_fifo_link`. Multiple `object_fifos` can be linked together with an `object_fifo_link` instruction to enable complex data flow patterns, including joining, distributing and simple forwarding. Depending on the pattern, the operation can receive offsets that describe the data flow between the input and output `object_fifos`.

`npu_dma_memcpy_nd`. An `npu_dma_memcpy_nd` instruction configures one of the 16 BDs of a shim tile, thereby determining how the shim tile's DMAs access the main memory. The parameters are the metadata of an `object_fifo`, a BD id, a main memory reference, a list of sizes, and a list of strides. Depending on the `object_fifo`, the DMA reads or writes data in the given data layout, which is specified by the lists of sizes and strides (they can have up to four elements). In order to reuse a BD, its data transfer must be finished. To wait for the end of a transfer, `dma_wait` can be called with the metadata of the corresponding `object_fifo`. All `npu_dma_memcpy_nd` operations after a `dma_wait` are only started if the `npu_dma_memcpy_nd` operation corresponding to the `dma_wait` is finished. If multiple BDs correspond to the same `object_fifo` and `dma_wait` is called on this `object_fifo`, the shim tile waits on the first unsynced BDs, i.e. `dma_wait` must be called multiple times to sync with all BDs.

## 5.2 TEIR Restrictions

The original TEIR must be adapted for the NPU. Fig. 6 shows the adapted version of the TEIR. The NPU’s parallelism differs from standard shared memory parallelism on a CPU. To reduce confusion, the `shared` execution type has been replaced with a new type, called `npu`. The NPU’s hardware restrictions limit the TECs that can be lowered. Currently, only GEMM kernels have been implemented. This restricts the valid options for `prim_main` to `GEMM` and for `prim_last` to `None`. The used GEMM kernels only support the `bfloat_16` data type. Therefore, `bfloat_16` is the only valid `data_type` option. There is also a restriction on the supported kernel sizes. The total memory of all three tiles must be less than 31.5 KB, as double buffering is employed and a compute tile has 64 KB of DMEM. By default, 1 KB of DMEM is reserved as stack. Additionally, their kernel sizes in M and N dimension must be divisible by four and the size of the K dimension must be divisible by eight, this is due to the data layout used in the `vmac.f` operation. The DMA can only access 4 bytes, which does not introduce a new restriction as the GEMM kernel indirectly requires even larger data access.

As a compute tile only has two DMA inputs, there is no efficient way to load initial values of the output tensor. Therefore, the output tensor can only be initialised with zeros, which reduces the valid options for `prim_first` to `Zero`. This also requires an output stationary contraction. Consequently, all dimensions of type K that are not of type `prim` must come directly before the `prim` dimensions and must have an execution type of `npu`. Currently, only one K dimension of type `npu` is supported.

$$\begin{aligned}
 D &\in \mathbb{N}^+ \\
 \text{dim\_types} &= (t_0, \dots, t_{D-1}), t_i \in \{\text{C, M, N, K}\} \\
 \text{exec\_types} &= (e_0, \dots, e_{D-1}), t_i \in \{\text{seq, npu, prim}\} \\
 \text{dim\_sizes} &\in (\mathbb{N}^+)^D \\
 \text{strides\_in0} &\in \mathbb{N}^D \\
 \text{strides\_in1} &\in \mathbb{N}^D \\
 \text{strides\_out} &\in \mathbb{N}^D \\
 \text{data\_type} &\in \{\text{bfloat\_16}\} \\
 \text{prim\_first} &\in \{\text{Zero}\} \\
 \text{prim\_main} &\in \{\text{GEMM}\} \\
 \text{prim\_last} &\in \{\text{None}\}
 \end{aligned}$$

**Fig. 6: Supported TEIR domain.**

The DMA of the shim tile imposes restrictions on the allowed strides. In each tensor, a `prim`-typed dimension must have a unit stride. All other strides must be even, because all data access must be 4-byte aligned. The strides in an `npu_dma_memcpy_nd` instruction cannot exceed  $2^{21}$ , as the DMA uses 20 bits to

represent 4-byte data access. The composition of the arguments given to each `npu_dma_memcpy_nd` instruction is described in the next section. This applies to the strides of the `prim`-typed dimensions, the `npu`-typed K dimension, and the last `npu`-typed dimension that is not of type K. If the second of these dimensions is of type M or N, its strides cannot exceed  $2^{19}$ , as it gets distributed on the NPU and the strides in the access are multiplied by 4. Furthermore, the stride of the last `npu`-typed M dimension of the output tensor cannot exceed  $2^{21}$ , as this dimension also appears in an `npu_dma_memcpy_nd` instruction.

### 5.3 FIFO-queue-based Loops

In order to perform a binary tensor contraction on the NPU using the loops-over-GEMM approach, the input tensors must be moved into the DMEM of the compute tiles, and the output tensor must be moved back into main memory. As discussed in the previous section, a compute tile has only the 64 KB DMEM, and in general, the three tensors do not fit into there. Therefore, the tensors are loaded and written in blocks, i.e. tiles in the TEC, with sizes matching the GEMM kernel sizes.

**Data Movement on the NPU.** Fig. 7 shows the movement of tensor tiles within the NPU. Each compute tile receives a different combination of input tensor tiles and computes the corresponding output tile. To bring the eight input tensor tiles into the NPU, each shim tile loads a tile from each input tensor. The tiles from the first input tensors align in the last M dimension with execution type NPU, and the tiles from the second input tensor align in the last N dimension with execution type NPU of the TEC. Each shim tile forwards its input tiles to its corresponding memory tile. Each memory tile then broadcasts the first input tile along a row and the second input tile along a column of compute tiles. The row or column number of the receiving compute tiles matches with the column number of the memory tile. In the same step, the DMAs of the memory tiles transform the tensor tiles according to the required sub-matrix layout for the `vmac.f` operation. The resulting data layout in the DMEM is shown in Fig. 5. Each compute tile then computes its output tile, forming a  $4 \times 4$  grid of output tiles. This process is described in detail in the next paragraph. The output tiles are joined along a column and written to the memory of the corresponding memory tile. They are then written through the corresponding shim tile to main memory. During the reading from the memory tile, the output tiles are transformed from the sub-matrix layout of the `vmac.f` operation to the layout specified in the TEC.

**Nested Loops on Compute Tiles.** The nested loops on the compute tiles are implemented using the structured control flow (SCF) MLIR dialect. Listing 4 illustrates the generation of MLIR code for a compute tile using the *IRON*-provided Python API. The GEMM kernel call is contained within nested `scf.for` operations. Before the `scf.for` operation that matches the NPU-typed K-dimension (i.e. is the innermost loop), a buffer of the output tensor `object_fifo` is acquired and set to zero. Inside the innermost loop, a tiled tensor is acquired from both input tensor `object_fifos`, the GEMM kernel is called and both tiled tensors are released. After

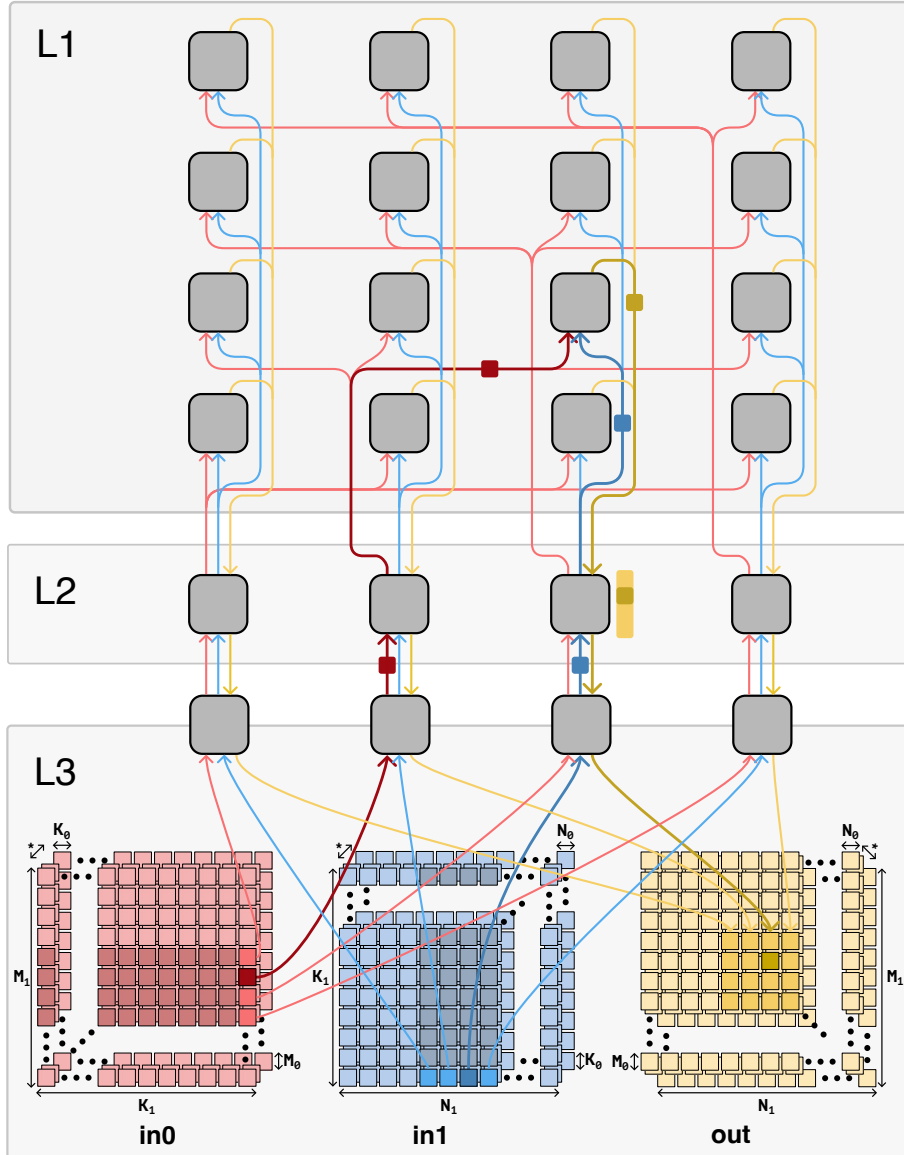


Fig. 7: Standard data movement of tiled input and output tensors. Dimension annotations refer to the dimension types, with the index indicating the position in the TEC starting from the end. This only applies to *prim* and *npu*-typed dimensions. The asterisk (\*) refers to all other dimensions.

the NPU-typed  $K$ -dimension loop, the buffer with the finished tiled output tensor is released, and tiled output tensor is passed along its `object_fifo`.

**Data Movement from Main Memory.** Reading and writing of the tiled tensors to and from main memory is programmed using `npu_dma_memcpy_nd` operations. The last two sizes and strides of each operation match the last two relevant sizes and strides of the corresponding tensor in the TEC, these are the three *prim*-type dimensions. In the reading operations, the second size and stride match the NPU-type  $K$ -dimension. The second size of the write operation is set to the number of

```

1 @core(comp_tiles[2][3], f"gemm_kernel.o")
2 def core_body():
3     for _ in range(...): # nested loops
4         ... # nested loops
5         elem_out = OUT_l112_fifos[2][3].acquire(Produce, 1)
6         zero(elem_out)
7         for _ in range(K_NPU_SIZE):
8             elem_in0 = IN0_l211_fifos[2].acquire(Consume, 1)
9             elem_in1 = IN1_l211_fifos[3].acquire(Consume, 1)
10
11             gemm_kernel(elem_in0, elem_in1, elem_out)
12
13             IN0_l211_fifos[2].release(Consume, 1)
14             IN1_l211_fifos[3].release(Consume, 1)
15             OUT_l112_fifos[2][3].release(Produce, 1)

```

**List. 4:** Python code with *IRON* API for nested loops on a the compute tile with index (2,3). The GEMM kernel call with the input tile handling is inside a the loop for the NPU-typed K-dimension. The data movement and initialisation of the output tile are directly handled before and after this loop. The nested loops for the other NPU-typed dimensions are wrapped around.

compute tile rows, and the stride is set to that of the output tensor’s last NPU-typed M dimension. The first size of each operation matches the second-to-last NPU-typed dimension. The other NPU-typed dimensions are realised through the explicit unrolling of `npu_dma_memcpy_nd` operations.

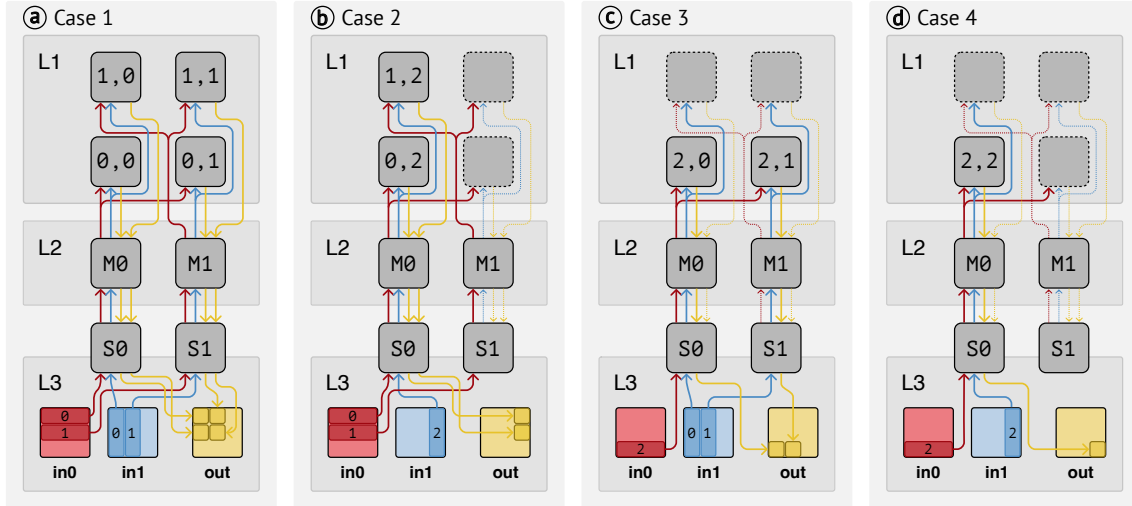
All data transfers must be finished before the NPU process can return. Due to the dependency between output and input transfers, it is sufficient to wait only for the output transfers, since the input transfers finish before them.

A TEC can require more `npu_dma_memcpy_nd` operations than BDs available on a shim tile. Therefore a reuse strategy for BDs is required. Within the first unrolled NPU-typed dimension, a ping-pong scheme is employed as the reuse strategy. The BDs of each shim tile are divided into two groups of the same size. Initially, operations are issued for all BDs in both groups. Then, wait operations on the output BDs of the first group are performed and all BDs from the first group are reused directly. This process is repeated, alternating between the two groups, until all the `npu_dma_memcpy_nd` operations in the first unrolled NPU-typed dimension have been issued. Finally, wait operations are executed for all remaining output transfers.

## 5.4 Uneven Distributed Dimensions

It is possible that the innermost NPU-typed M or N dimension is not divisible by the number of rows or columns of compute tiles on the NPU. Fig. 8 shows a simplified example in which only  $2 \times 2$  compute tiles are used, and the output tensor is  $3 \times 3$ -tiled in the M and N dimension. There are four different cases to consider:

Ⓐ **Case 1.** In the standard case, all compute tiles receive both input tensor tiles. This is already handled by the implementation and does not require further attention.



**Fig. 8: Data movement through the object\_fifos for a  $3 \times 3$  tiled output tensor (C) on a  $2 \times 2$  grid of compute tiles. The compute tiles (L1) are labelled with the currently computed output tensor tile ids. Inactive object\_fifos and compute tiles are dotted.**

ⓑ **Case 2.** If the size of the distributed N dimension is not divisible by the number of compute tile columns, the last iteration in this dimension must be handled specially. Some `npu_dma_memcpy_nd` operations that load the second input tensor are modified so that fewer tiles are loaded than through the other operations of the second input tensor, i.e. some compute tile columns receive fewer second input tiles than others. As `object_fifos` cannot be changed dynamically, the tiles of the first input tensor are distributed to all compute tiles. When broadcasting, `object_fifos` only advance when all the receiving tiles have consumed a tensor tile. To advance the `object_fifos` of the first input tensor, the code that consumes the send tiles is wrapped in a loop whose size is set to the product of all inner loop sizes of this N dimension. This new loop is executed right after the loop of this N dimension on all compute tiles that do not always receive the second input tiles. Since the dimension size is not divisible by the number of compute tile columns, the loop size matching this dimension has to be adapted accordingly. The loop size on compute tiles that always receive a second input tile is rounded up, while the loop size on compute tiles that do not always receive a second input tile is rounded down when divided by the number of compute tile columns. This results in fewer output tiles being produced in columns that do not always receive second input tiles. Therefore, the `npu_dma_memcpy_nd` operations for the output tensor are likewise adapted.

ⓒ **Case 3.** Special handling is also required when the distributed M dimension is not divisible by the number of compute tile rows. The handling of the input tiles is the same as the previous case, but the handling of the output tiles is different. As with broadcasting, joining `object_fifos` only advances when all the sending tiles produce a tensor tile. In this case, the number of output tiles produced differs in one column, therefore, all `object_fifos` that send output tiles from a memory tile to a shim tile are split into two `object_fifos`. In each column, one is linked with

the compute tiles that always receive tiles of the first input tensor, and the other is linked with the remaining compute tiles. The `npu_dma_memcpy_nd` operations for the output tensor are also split accordingly. .

④ **Case 4.** This case occurs when neither of the distributed dimensions can be evenly distributed across the compute tiles. No further changes are needed, as the interplay of the previous two cases handles this.

## 6 Evaluation

In this section, I analyse the performance of the implemented GEMM kernel and TEIR lowering. I address four key questions:

- *Question:* Does the hand-optimised assembly kernel deliver the expected performance? *Answer:* Section 6.2 shows that slightly fewer bank conflicts occur than expected and that there are 25 unaccounted cycles, likely due to the overhead associated with the kernel call.
- *Question:* Which GEMM kernel sizes and configurations should be avoided? *Answer:* Section 6.3 reveals that kernels with small dimensions, a transposed output data layout, and edge treatment where  $1\times 4$  or  $4\times 1$  blocking is applied should be avoided.
- *Question:* What data layout should be used for the tensors to achieve a high-performing tensor contraction? *Answer:* On the lowest power setting, the data layout has little influence on performance. At the highest power setting, however, the tensors should have a data layout that allows for linear memory access. Section 6.4 demonstrates that this can increase performance by more than 25 percentage points of the TPP compared to the standard GEMM layout.
- *Question:* What level of performance can be achieved with this implementation of a binary tensor contraction? *Answer:* The highest performance was achieved in the grid search in Section 6.4 with a performance, at 5,790 GFLOPS.

### 6.1 Hardware and Software

All evaluations are performed on the same testbed. This is equipped with an AMD Ryzen 7 8700G SoC and 60 GiB of DDR5-5200 memory. The Ryzen SoC, codenamed *Phoenix*, contains the NPU. The NPU runs firmware version 1.5.5.391 and uses the XDNA driver and XRT version 2.20.0 with commits 10111bb and 1542983. The *xchesscc* compiler version is U-2023.06. *peano* uses *clang* 20.0.0 with commit 7f4f732 of llvm-aie repository. *xchesscc* is used to compile the *AIE API* kernels because it achieves higher performance, and *peano* is used to compile the assembly kernels. The XRT can control the NPU’s clock frequency and offers three power modes: `powersaver`, `balanced`, and `turbo`. Each power mode corresponds to a clock frequency. A microbenchmark revealed that the clock frequencies of the NPU on the testbed are 0.79 GHz, 1.02 GHz, and 1.79 GHz. The power mode is set to `balanced` unless stated otherwise. All experiments comprise three warm-up iterations, 20 time measurements. The mean of these measurements is then used for evaluation. The NPU incurs a call overhead of several microseconds. To mitigate the impact of this on the kernel experiments, the kernel call is looped 10,000 times on the compute tile and the measured time is divided by the loop count. There was no significant difference in the performance of the GEMM kernels when executed on one or multiple

compute tiles. The results presented here show the performance of the experiments using one compute tile.

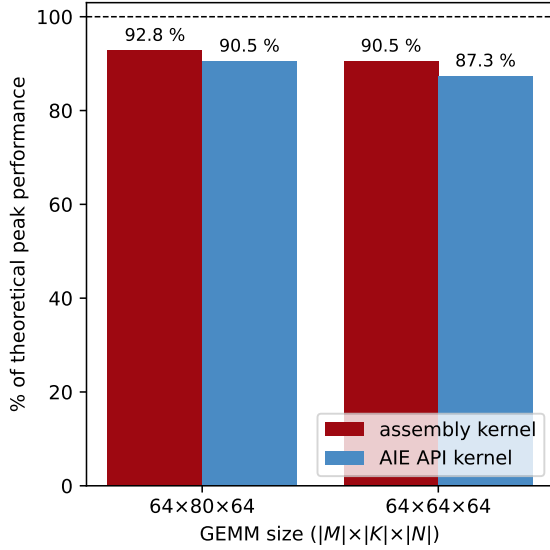
## 6.2 GEMM Assembly Kernels

First, we will look at the predictions made for the assembly kernels. As mentioned in Section 4.2, when bank conflicts are ignored, the performance of a kernel written in the VLIW ISA can be determined by counting the executed instructions. We perform this calculation for the assembly kernel with GEMM sizes  $64 \times 80 \times 64$ . The kernel has 13 instructions in the warm-up phase, 160 ( $4 \times 4 \cdot 10$ ) instructions with a `vmac.f` operation and 5 without in the unrolled K-loop, and 6 instructions for the return statement. The instructions in the unrolled K-loop are executed 16 times. A total of 2,659 ( $13 + 16 \cdot (160 + 5) + 6$ ) instructions are executed in this kernel. When the assumed bank conflicts occur, the time taken to execute the instructions should be equivalent to 2,739 ( $13 + 16 \cdot (160 + 5 + 5) + 6$ ) cycles. The minimum number of instructions required for GEMM size  $64 \times 80 \times 64$  is 2,560 ( $(64/4) \cdot (80/8) \cdot (64/4)$ ). Therefore, the kernel has an efficiency of 96.3% based on the number of executed instructions and 93.4% when accounting for the assumed bank conflicts. With the power mode set to `balanced`, these instructions should take 2.607 or 2.685  $\mu s$  to execute. The measured time for one kernel execution is 2.703  $\mu s$ . The difference between the calculated time without bank conflicts and the measured time is only 0.096  $\mu s$ . As one cycle is about 0.001  $\mu s$  long, a time of 96 cycles is unaccounted for. This measurement is repeated with each data movement operation replaced by the corresponding NOP operation. The result is a kernel execution time of 2.632  $\mu s$ . This indicates that about 71 cycles are lost due to bank conflicts, with an additional 25 cycles consumed by the call overhead. There are 9 fewer bank conflicts than the expected 5 in each M-N-loop iteration, totalling 80.

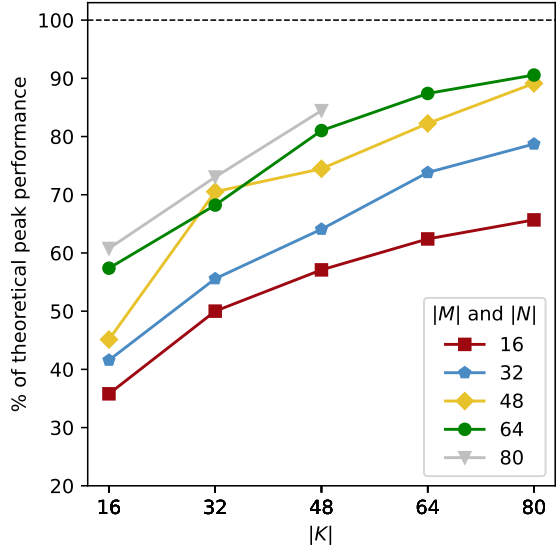
## 6.3 GEMM AIE API Kernels

We will now shift our focus to the performance evaluation of *AIE API* kernels. First, we will compare the assembly kernels with the *AIE API* kernels of the same GEMM size. Fig. 9 shows the hardware utilisation of these four kernels. Both assembly kernels perform better than their corresponding *AIE API* kernel. The differences are 2.3% and 3.2% of TPP for GEMM sizes of  $64 \times 80 \times 64$  and  $64 \times 64 \times 64$ , respectively. The smaller assembly kernel has the same performance as the larger *AIE API* kernel.

*AIE API* kernels are used for comparing kernels with varying GEMM sizes, since writing all the assembly kernels by hand is not feasible. Fig. 10 plots the hardware utilisation of 23 *AIE API* kernels with different GEMM sizes. The sizes  $80 \times 64 \times 80$  and  $80 \times 80 \times 80$  are missing because the tile sizes are too large to fit into the DMEM of the compute tile with double buffering. The performance of the kernels follows a trend. For kernels with the same M and N dimension size performance increases with a larger K-dimension but starts to plateau towards the end. Performance increases with larger M and N dimensions. One outlier is the kernel with size  $48 \times 32 \times 48$ , which has a higher performance than the kernel with the next larger M and N dimension. This increase in performance with dimension size is expected, as compute intensity



**Fig. 9:** The hardware utilisation is shown of the two handcrafted assembly kernels and the two GEMM *AIE API* kernels with the same GEMM sizes. The utilisation is given as a percentage of the TPP.

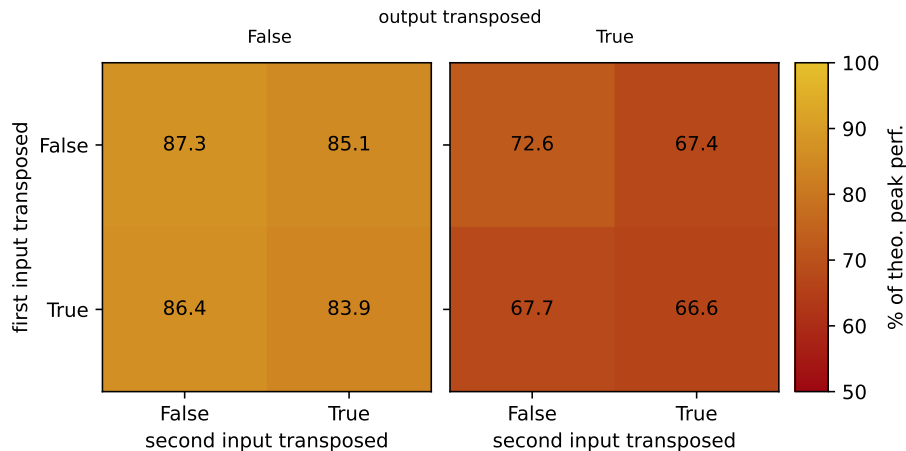


**Fig. 10:** The hardware utilisation of GEMM *AIE API* kernels of different sizes is shown.  $M$  and  $N$  have the same size in each benchmarked kernel and are plotted as different lines.  $|K|$  is plotted on the X-axis. The utilisation is given as a percentage of the TPP.

increases with larger dimension sizes, while the overhead becomes smaller in relation to the total computation.

The data layout of a tensor does not always follow the standard GEMM layout, in which the  $K$ -dimension has a unit stride in the first input and the  $N$  dimension has a unit stride in the second input and output. If the unit stride is not in the same dimension of an input or output as in the standard GEMM, then the input or output matrix is transposed. The *AIE API* kernels can handle all combinations of non-transposed and transposed inputs and output. The performance of each combination for a  $64 \times 64 \times 64$  GEMM kernel is shown in Fig. 11. The kernel with no transposed layouts achieves the highest performance, while the kernel where all layouts are transposed achieves the lowest. The output layout has the largest effect on performance, with a drop of up to 18.7% of the TPP. It is possible to convert a GEMM with a transposed output to a GEMM with a standard output layout by switching the inputs. This switches the input layouts from standard to transposed, or vice versa. This should always be done, since all combinations with a standard output layout perform better than those with a transposed output layout.

We will now examine the performance of *AIE API* kernels with edge treatment around the  $4 \times 4$  blocking of the output matrix. Fig. 12 visualises the hardware utilisation of *AIE API* kernels where the  $K$ -dimension size is set to 64, while the  $M$  and  $N$  dimension sizes are increased from 64 to 76 in increments of 4. The kernel with  $M$  and  $N$  dimension sizes set to 64 has no edge treatment. Kernels with  $M$

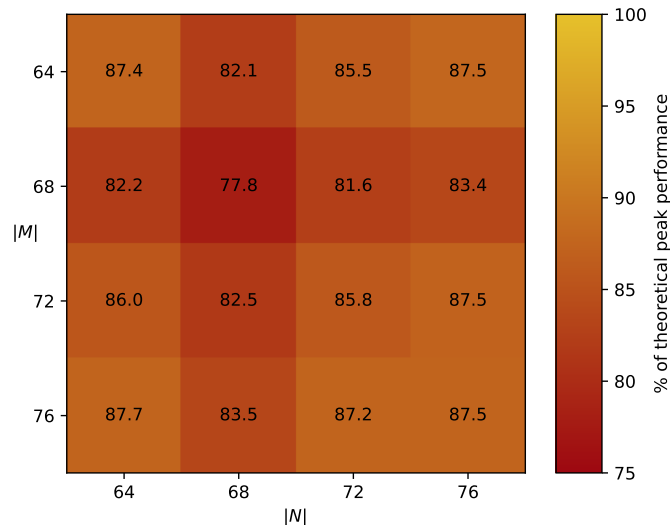


**Fig. 11:** This shows hardware utilisation of GEMM *AIE API* kernels with a GEMM size of  $64 \times 64 \times 64$  and different transpose configurations. This illustrates how efficiently the compiler can minimise the impact of the *vshuffle* operations. The utilisation is given as a percentage of the TPP.

or N dimension size set to 64 have an edge treatment with  $i \times 4$  or  $4 \times i$  blocking, where  $i$  ranges from 1 to 3, corresponding to the dimension which does not have a size of 64. This blocking is applied 16 times. Kernels where neither the M nor the N dimension has a size of 64 have an edge treatment with blocking sizes of  $i \times j$ ,  $i \times 4$  and  $4 \times j$ , where  $i$  and  $j$  range from 1 to 3, corresponding to the sizes of the M and N dimensions. In this case, the  $i \times j$  blocking is applied once, while the other two are applied 16 times each. The kernel with the lowest performance has the M and N dimension sizes set to 68. This kernel has an edge treatment with sizes  $1 \times 1$ ,  $1 \times 4$ , and  $4 \times 1$ . This makes it the kernel with the smallest blocking sizes in the edge treatment. The performance of all kernels with  $1 \times 4$  or  $4 \times 1$  blocking is lower than that of all other kernels. The kernel with the highest performance has M and N dimension sizes of 76 and 64, respectively, indicating that computation of the output matrix can be performed efficiently using a  $3 \times 4$  blocking.

## 6.4 Binary Tensor Contractions

This section focuses on a complete binary tensor contractions and their data layouts. The data layout of tensors is given as an *einsum* expression, where all tensors are stored contiguously using general row-major storage, i.e. the dimension of the last label has a unit stride and the strides of the dimension increase, with the dimension of the first label having the largest stride. The sizes of the dimensions are listed in alphabetical order, corresponding to their dimension labels. All compute tiles are

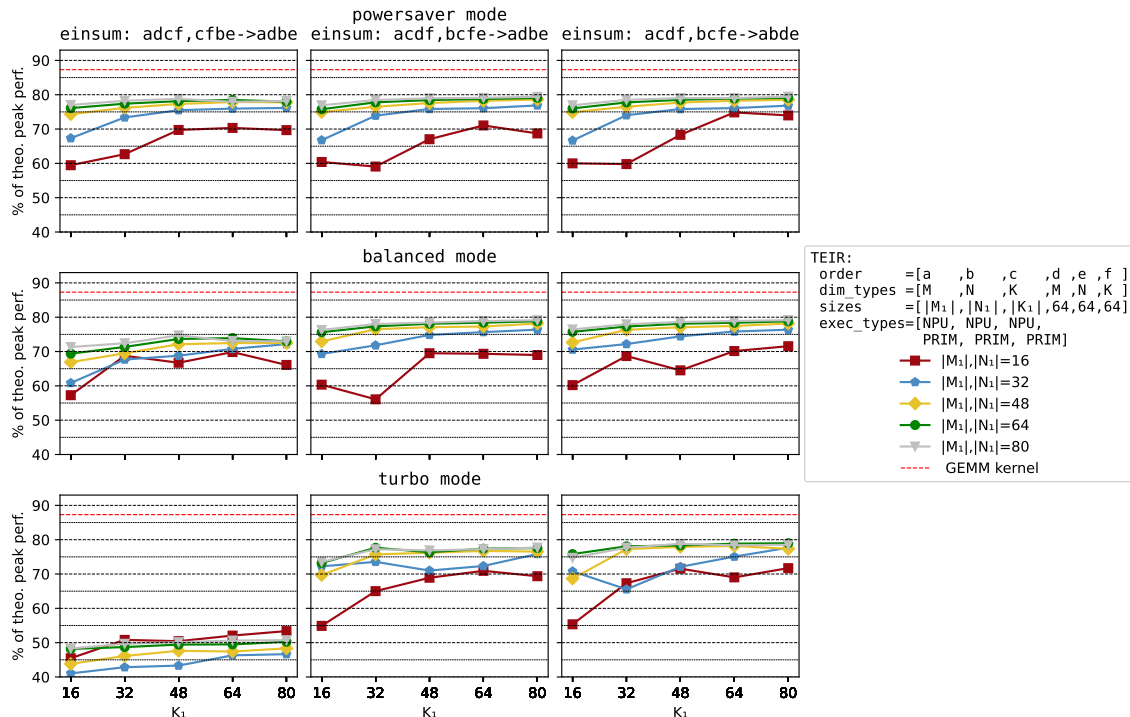


**Fig. 12: The hardware utilisation of GEMM *AIE API* kernels of sizes  $|M| \times 64 \times |N|$  is shown. This depicts the efficiency of the edge treatment. The utilisation is given as a percentage of the theoretical peak performance.**

used, and data movement is performed as described in Section 5.

A small grid search was performed with three different *einsum* expressions and the three different power modes to see how the hardware behaves under different settings and workloads. Fig. 13 shows the results of this search. All dimensions with the same label have the same execution type. Dimensions labeled d, f, and e have the execution type set to **prim**, while the other three have the execution type set to **npu**. The order of the dimensions in all three TECs is equal to the alphabetical order of their labels. The three *einsum* expressions describe three different data layouts. The first expression (**adcf,cfbe** $\rightarrow$ **adbe**) has the same layout as standard matrix-matrix multiplication. The second expression (**acdf,bcfe** $\rightarrow$ **adbe**) has a layout in which the input tensor tiles for a GEMM are ordered by access. This gives it, in theory, a linear memory access. The output tensor is tiled as in the first expression. The last expression (**acdf,bcfe** $\rightarrow$ **abde**) has the same input layout as the second expression, and the output tensor follows the same layout as the input.

Contractions with dimensions a and b of size 16 and 32 have volatile performance measurements, especially when they also have a small c dimension. This indicates that call overhead or memory latencies can drastically vary. Therefore, these contractions will not be included in the evaluations. In most cases, performance increases with larger dimension sizes, but this may be due to smaller call overheads compared to the contraction. At the lowest power setting, the performance of all three data layouts is very close. The difference in performance is less than 1.3 percentage points of the TPP, which could be due to system noise. Switching to the **balanced** mode results in a decrease in performance of over 4 percentage points for the first data layout and under 1 percentage points for the other two, except for a decrease of around 2.2 percentage points for both in the smallest con-

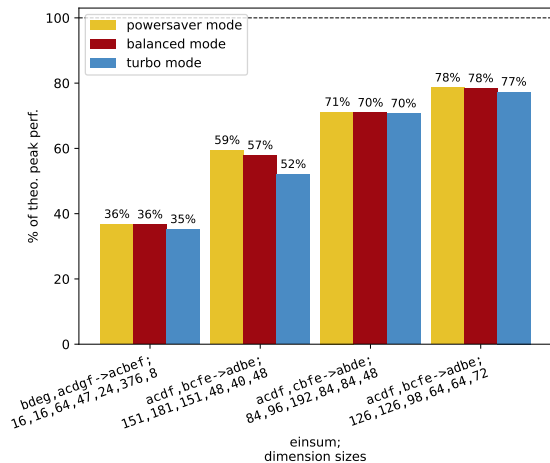


**Fig. 13: Hardware utilisation for three different binary tensor contractions with different dimension sizes is shown. Dimensions labeled a, b, and c have size  $|M_1|, |N_1|$  and  $|K_1|$ , respectively. Dimensions  $M_1$  and  $N_1$  have the same size in each benchmarked contraction and are plotted as different lines.  $|K_1|$  is plotted on the X-axis. Dimensions labeled e, d, and f have a size of 64 and are used for the GEMM kernel. The tensor data layouts are given as their corresponding *einsum* expression. The same GEMM kernel is used for all contractions and its hardware utilisation is marked in red. The utilisation is given as a percentage of the TPP.**

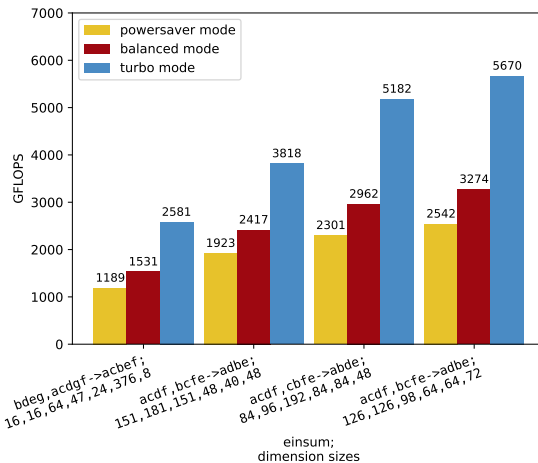
traction. In the highest power mode, performance decreases by over 22 percentage points compared to the second power mode for the first data layout. For the second and third data layout, there is no clear trend in the change of performance. The highest performance of 5,790 GFLOPS is achieved in the grid when the dimensions a, b, and c have a size of 64, 64 and 80, respectively. The computation performed for the first data layout and the resulting tensor contraction is equal to that performed by the `matrix multiplication example` in the `mlir-ai` repository [5] when the GEMM kernel size is set to  $64 \times 64 \times 64$ .

The final experiment measures the performance of four binary tensor contractions. These four contractions come from a set of binary tensor contractions introduced by Springer and Bientinesi [27], and they have the highest arithmetic intensity of them. Dimension reordering, fusion and splitting are performed to generate high-performing TECs. The layout of the input tensors is changed so that the tiled tensors for the GEMM are ordered by access, i.e. the input tensors have the second layout type from the previous experiment. This is done for all contractions except

the third, due to a stride size restriction in the `npu_dma_memcpy_nd` instruction. This is done without altering the actual layout of the output tensor. The alphabetical order of the dimension labels is the same as their order in the TECs. The last two dimensions of each tensor have the execution type `prim` and all others have the type `npu`.



**Fig. 14: Hardware utilisation of the four benchmarked binary tensor contractions.** The dimension sizes are ordered after the alphabetical order of their corresponding dimension labels. The utilisation is given as a percentage of the TPP.



**Fig. 15: Performance of the four benchmarked binary tensor contractions.** The dimension sizes are ordered after the alphabetical order of their corresponding dimension labels. Performance is expressed in `bfloat16` GFLOPS.

The hardware utilisation of the tensor contractions are shown in Fig. 14. The first contraction achieves around 36% of the TPP. The second contraction achieves an average of 56%. Both contractions have an arithmetic intensity in their GEMM kernels of 5.9 and 15 operations per byte. This is lower than the arithmetic intensity of the other two contractions, which have an arithmetic intensity above 22 operations per byte in their GEMM kernels. The third and fourth contractions reach 70% and 78% of TPP, respectively. The achieved `bfloat16` performance of the four binary tensor contractions is shown in Fig. 15. The fourth tensor contraction achieves the highest total performance, with an average of 5,670 GFLOPS in the turbo mode.

## 7 Discussion and Outlook

This work provided a binary tensor contraction framework in the form of a TEIR lowering algorithm with a creation scheme for high-performance assembly and *AIE API* GEMM kernels. The TEIR is lowered to efficient MLIR-AIE code, which uses an optimised GEMM kernel at its center. Using this approach, binary tensor contractions can utilise up to 78.5% of the TPP and achieve up to 5,670 GFLOPS on the NPU.

There are still restrictions on the strides, sizes, and loop order within the TEIR. Some of these limits are hardware-dictated, such as the 32-bit memory access, while others are software-imposed. For instance, the requirement that the K dimension of the GEMM kernel must be a multiple of eight could be relaxed to a multiple of four by utilising only the lower half of the vector registers and setting the higher half to zero. While this wastes half of the computations in the instruction, it can enable better GEMM kernel sizes with higher operations per byte. If the size of the K dimension in the GEMM kernel is only divisible by four, the layout of the input tiles must also be changed to a  $4 \times 4$ -blocked layout. The size restrictions on all GEMM dimensions could be reduced to a multiple of two by applying filter operations to the vector and accumulation registers, this would also result in performance reduction.

In the current implementation, it is not possible to specify which M and N npu-typed dimensions are broadcast along the compute tile rows and columns. Furthermore, the current implementation always broadcasts an M dimension along the rows and an N dimension along the columns. On the current hardware, this makes no difference; however, on the next-generation NPU, there is a  $4 \times 8$  grid of compute tiles.

Further improvements can be made to the BD usage. Currently, the ping-pong reuse scheme only employed in the first unrolled loop. At the end of this loop, the NPU waits for all BDs to finish before starting the next one. The ping-pong reuse scheme should be applied to all unrolled loops.

The assembly kernels have 19 instructions outside and five instructions inside the hardware loop that do not perform a `vmac.f`. The number of wasted instructions can be reduced by moving the first and last iterations of the fused M-N-loop outside of the hardware loop. This allows the instructions in the warm-up phase and the jump operation to be packed more efficiently. This also enables the five wasted instructions inside the M-N-loop to be packed more efficiently, as they can be moved to the start of the next iteration.

As the NPU incurs significant call overhead, consideration should be given to which TECs the CPU should be used for instead of the NPU. Breuer et al. [21] demonstrated that the same CPU used in this testbed can achieve a performance of over 1,000 GFLOPS for certain tensor contractions. Rösti and Franz [9] used the NPU on their testbed with an AMD Ryzen 9 7940HS for only five matrix multiplications in inference and seven in training a GPT-2 model. The smallest GEMM size for which the NPU was used is  $768 \times 256 \times 768$ . All GEMMs smaller than this were computed using the CPU.

## 8 References

- [1] AMD. “AMD Ryzen AI Software,” Accessed: Oct. 17, 2025. [Online]. Available: <https://www.amd.com/en/developer/resources/ryzen-ai-software.html>
- [2] Intel. “Quick overview of Intel’s Neural Processing Unit (NPU),” Accessed: Oct. 17, 2025. [Online]. Available: <https://intel.github.io/intel-npu-acceleration-library/npu.html>
- [3] Qualcomm. “The Qualcomm AI Engine,” Accessed: Oct. 17, 2025. [Online]. Available: <https://www.qualcomm.com/products/technology/processors/ai-engine>
- [4] A. Rico et al., “AMD XDNA NPU in Ryzen AI Processors,” *IEEE Micro*, vol. 44, no. 6, pp. 73–82, 2024. DOI: 10.1109/MM.2024.3423692
- [5] AMD. “mlir-ai: MLIR-based AI Engine toolchain,” Accessed: Oct. 17, 2025. [Online]. Available: <https://github.com/Xilinx/mlir-ai>
- [6] A. Breuer. “Tield Execution IR,” Accessed: Oct. 17, 2025. [Online]. Available: <https://scalable.uni-jena.de/opt/pbtc/chapters/teir.html>
- [7] AMD. “AMD Completes Acquisition of Xilinx,” Accessed: Oct. 17, 2025. [Online]. Available: <https://web.archive.org/web/20241007212550/https://www.amd.com/en/newsroom/press-releases/2022-2-14-amd-completes-acquisition-of-xilinx.html>
- [8] AMD. “Versal Adaptive SoC AIE-ML Architecture Manual (AM020),” Accessed: Oct. 17, 2025. [Online]. Available: <https://docs.amd.com/r/en-US/am020-versal-ai-ml/Overview>
- [9] A. Rösti and M. Franz, *Unlocking the amd neural processing unit for ml training on the client using bare-metal-programming tools*, 2025. arXiv: 2504.03083 `\mkbibbrackets {\thefield {eprintclass}}`. [Online]. Available: <https://arxiv.org/abs/2504.03083>
- [10] AMD. “Versal Adaptive SoC AIE-ML Register Reference (AM025),” Accessed: Oct. 17, 2025. [Online]. Available: <https://docs.amd.com/r/en-US/am025-versal-ai-ml-register-reference/Overview>
- [11] AMD. “Ryzen AI Software,” Accessed: Oct. 17, 2025. [Online]. Available: <https://ryzenai.docs.amd.com/en/latest/>
- [12] AMD. “AI Engine API User Guid,” Accessed: Oct. 17, 2025. [Online]. Available: [https://download.amd.com/docnav/aiengine/xilinx2025\\_1/aiengine\\_api/ai\\_api/doc/index.html](https://download.amd.com/docnav/aiengine/xilinx2025_1/aiengine_api/ai_api/doc/index.html)
- [13] AMD. “AMD XDNA Driver for Linux,” Accessed: Oct. 17, 2025. [Online]. Available: <https://github.com/amd/xdna-driver>
- [14] AMD. “XRT: Xilinx Runtime,” Accessed: Oct. 17, 2025. [Online]. Available: <https://github.com/Xilinx/XRT>

- 
- [15] A. Einstein, “The foundation of the general theory of relativity,” *Annalen Phys.*, vol. 49, no. 7, J.-P. Hsu and D. Fine, Eds., pp. 769–822, 1916. DOI: 10.1002/andp.19163540702
- [16] N. Developers. “numpy.einsum,” Accessed: Oct. 17, 2025. [Online]. Available: <https://numpy.org/doc/stable/reference/generated/numpy.einsum.html>
- [17] T. J. authors. “jax.numpy.einsum,” Accessed: Oct. 17, 2025. [Online]. Available: [https://docs.jax.dev/en/latest/\\_autosummary/jax.numpy.einsum.html](https://docs.jax.dev/en/latest/_autosummary/jax.numpy.einsum.html)
- [18] TensorFlow. “tf.einsum,” Accessed: Oct. 17, 2025. [Online]. Available: [https://www.tensorflow.org/api\\_docs/python/tf/einsum](https://www.tensorflow.org/api_docs/python/tf/einsum)
- [19] PaddleNLP. “PaddleNLP einsum,” Accessed: Oct. 17, 2025. [Online]. Available: <https://paddlenlp-en.readthedocs.io/en/stable/source/paddlenlp.ops.einsum.html>
- [20] I. Preferred Networks and I. Preferred Infrastructure. “cupy.einsum,” Accessed: Oct. 17, 2025. [Online]. Available: <https://docs.cupy.dev/en/stable/reference/generated/cupy.einsum.html>
- [21] A. Breuer et al., “Einsum trees: An abstraction for optimizing the execution of tensor expressions,” in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. New York, NY, USA: Association for Computing Machinery, 2025, pp. 275–292, ISBN: 9798400710797. [Online]. Available: <https://doi.org/10.1145/3676641.3716254>
- [22] R. N. C. Pfeifer, J. Haegeman, and F. Verstraete, “Faster identification of optimal contraction sequences for tensor networks,” *Phys. Rev. E*, vol. 90, p. 033315, 3 Sep. 2014. DOI: 10.1103/PhysRevE.90.033315 [Online]. Available: <https://link.aps.org/doi/10.1103/PhysRevE.90.033315>
- [23] C.-C. Lam, P. Sadayappan, and R. Wenger, “On optimizing a class of multi-dimensional loops with reductions for parallel execution,” *Parallel Process. Lett.*, vol. 7, pp. 157–168, 1997. [Online]. Available: <https://api.semanticscholar.org/CorpusID:9440379>
- [24] D. G. a. Smith and J. Gray, “Opt\_einsum - a python package for optimizing contraction order for einsum-like expressions,” *Journal of Open Source Software*, vol. 3, no. 26, p. 753, 2018. DOI: 10.21105/joss.00753 [Online]. Available: <https://doi.org/10.21105/joss.00753>
- [25] J. Gray and S. Kourtis, “Hyper-optimized tensor network contraction,” *Quantum*, vol. 5, p. 410, Mar. 2021, ISSN: 2521-327X. DOI: 10.22331/q-2021-03-15-410 [Online]. Available: <https://doi.org/10.22331/q-2021-03-15-410>

- [26] C. Staudt, M. Blacher, J. Klaus, F. Lippmann, and J. Giesen, “Improved Cut Strategy for Tensor Network Contraction Orders,” in *22nd International Symposium on Experimental Algorithms (SEA 2024)*, L. Liberti, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 301, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 27:1–27:19, ISBN: 978-3-95977-325-6. DOI: 10.4230/LIPIcs.SEA.2024.27 [Online]. Available: <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SEA.2024.27>
- [27] P. Springer and P. Bientinesi, “Design of a high-performance gemm-like tensor–tensor multiplication,” *ACM Trans. Math. Softw.*, vol. 44, no. 3, Jan. 2018, ISSN: 0098-3500. DOI: 10.1145/3157733 [Online]. Available: <https://doi.org/10.1145/3157733>

## 9 Appendix

### Git

#### source code

[https://github.com/scalable-analyses/tensor-aie/tree/master\\_thesis](https://github.com/scalable-analyses/tensor-aie/tree/master_thesis)

### Acronyms

<b>AGU</b>	address generator unit
<b>AIE</b>	AI Engine
<b>AIE-ML</b>	AI Engine-Machine Learning
<b>ASIC</b>	application-specific integrated circuit
<b>ALU</b>	arithmetic logic unit
<b>BD</b>	buffer descriptor
<b>DMA</b>	direct memory access
<b>DMEM</b>	data memory
<b>FLOPS</b>	floating-point operations per second
<b>FLOP</b>	floating-point operation
<b>GEMM</b>	general matrix-matrix multiplication
<b>ILP</b>	instruction-level parallelism
<b>IMEM</b>	instruction memory
<b>ISA</b>	Instruction Set Architecture
<b>MAC</b>	multiply-accumulate
<b>MLIR</b>	multi-level intermediate representation
<b>ML</b>	machine learning
<b>NOP</b>	no operation
<b>NPU</b>	neural processing unit
<b>NoC</b>	network on a chip
<b>PL</b>	programmable logic
<b>PU</b>	processing unit
<b>SCF</b>	structured control flow
<b>SIMD</b>	single instruction multiple data

**SoC** system on a chip

**TEIR** tiled execution intermediate representation

**TEC** tiled execution configuration

**TPP** theoretical peak performance

**VLIW** Very Long Instruction Word

**XRT** Xilinx Runtime

## List of Figures

1	Overview of the XDNA Architecture on <i>Phoenix</i> , showing <b>compute tiles (L1)</b> , <b>memory tiles (L2)</b> , <b>shim tiles</b> and a separate <b>command processor</b> . The tiles are connected via AXI4 streams, which are configured by the switch boxes. Dotted components are not longer supported by the XDNA driver. (adapted from [9]) . . . . .	6
2	Overview of functional units on a compute tile. [8] . . . . .	7
3	Original TEIR domain and notation specification [6]. . . . .	10
4	Overview of lowering and compilation process of a TEC. Files and intermediate representations are shown in <b>yellow</b> and used tools are shown in <b>blue</b> . (adapted from [9]). . . . .	12
5	Data layout and access pattern of tensors in DMEM. The first input, second input and output is colored <b>red</b> , <b>blue</b> and <b>yellow</b> , respectively. The register are colored <b>gray</b> . The sub-matrix sizes $4 \times 8 \times 4$ for the <code>vmac.f</code> operation are annotated as <i>r</i> , <i>s</i> and <i>t</i> , respectively. The labels <i>m</i> , <i>k</i> , and <i>n</i> correspond to the other parts of the split dimensions <i>M</i> , <i>K</i> , and <i>N</i> of the GEMM tiles. The access in first <i>M-N</i> iteration and last <i>K</i> iteration is highlighted. Four sub-matrices of the first input are multiplied pairwise with four sub-matrices to compute $4 \times 4$ -blocked sub-matrices of the output. . . . .	16
6	Supported TEIR domain. . . . .	21
7	Standard data movement of tiled input and output tensors. Dimension annotations refer to the dimension types, with the index indicating the position in the TEC starting from the end. This only applies to <code>prim</code> and <code>npu</code> -typed dimensions. The asterisk (*) refers to all other dimensions. . . . .	23
8	Data movement through the <code>object_fifos</code> for a $3 \times 3$ tiled output tensor ( <b>C</b> ) on a $2 \times 2$ grid of compute tiles. The compute tiles (L1) are labelled with the currently computed output tensor tile ids. Inactive <code>object_fifos</code> and compute tiles are dotted. . . . .	25
9	The hardware utilisation is shown of the two handcrafted assembly kernels and the two GEMM <i>AIE API</i> kernels with the same GEMM sizes. The utilisation is given as a percentage of the TPP. . . . .	29

10	The hardware utilisation of GEMM <i>AIE API</i> kernels of different sizes is shown. $M$ and $N$ have the same size in each benchmarked kernel and are plotted as different lines. $ K $ is plotted on the X-axis. The utilisation is given as a percentage of the TPP. . . . .	29
11	This shows hardware utilisation of GEMM <i>AIE API</i> kernels with a GEMM size of $64 \times 64 \times 64$ and different transpose configurations. This illustrates how efficiently the compiler can minimise the impact of the <code>vshuffle</code> operations. The utilisation is given as a percentage of the TPP. . . . .	30
12	The hardware utilisation of GEMM <i>AIE API</i> kernels of sizes $ M  \times 64 \times  N $ is shown. This depicts the efficiency of the edge treatment. The utilisation is given as a percentage of the theoretical peak performance. . . . .	31
13	Hardware utilisation for three different binary tensor contractions with different dimension sizes is shown. Dimensions labeled a, b, and c have size $ M_1 ,  N_1 $ and $ K_1 $ , respectively. Dimensions $M_1$ and $N_1$ have the same size in each benchmarked contraction and are plotted as different lines. $ K_1 $ is plotted on the X-axis. Dimensions labeled e, d, and f have a size of 64 and are used for the GEMM kernel. The tensor data layouts are given as their corresponding <i>einsum</i> expression. The same GEMM kernel is used for all contractions and its hardware utilisation is marked in red. The utilisation is given as a percentage of the TPP. . . . .	32
14	Hardware utilisation of the four benchmarked binary tensor contractions. The dimension sizes are ordered after the alphabetical order of their corresponding dimension labels. The utilisation is given as a percentage of the TPP. . . . .	33
15	Performance of the four benchmarked binary tensor contractions. The dimension sizes are ordered after the alphabetical order of their corresponding dimension labels. Performance is expressed in <code>bfloat_16</code> GFLOPS. . . . .	33

## List of Tables

1	<i>einsum</i> Examples (adapted from [21]) . . . . .	9
2	The rules for inferring the dimension type from the strides of the tensors. A dimension exists in a tensor if the strides of the dimension in the tensor are non-zero. . . . .	11
3	Example operations of the AIE-ML ISA. The texts in angle brackets (<>) are placeholders for specific registers, unit suffixes or numbers. The bar ( ) indicates an option. . . . .	14

## Listings

1	C++ Example of Nested M-N-Loops . . . . .	18
---	---	----

2	C++ Example of Fused M-N-Loop with Branching Pointer Updates .	18
3	C++ Example of Fused M-N-Loop with Branchless Pointer Updates .	18
4	Python code with <i>IRON</i> API for nested loops on a the compute tile with index (2,3). The GEMM kernel call with the input tile handling is inside a the loop for the NPU-typed K-dimension. The data movement and initialisation of the output tile are directly handled before and after this loop. The nested loops for the other NPU-typed dimensions are wrapped around. . . . .	24