From Threads to Tiles: T2T, a Compiler for CUDA-to-NPU Translation via 2D Vectorization

Shuaijiang Li
SKLP, ICT, CAS
UCAS
Beijing, China
lishuaijiang19b@ict.ac.cn

Jiacheng Zhao*
SKLP, ICT, CAS
UCAS
Beijing, China
zhaojiacheng@ict.ac.cn

Ying Liu
SKLP, ICT, CAS
UCAS
Beijing, China
liuying2007@ict.ac.cn

Shuoming Zhang
SKLP, ICT, CAS
UCAS
Beijing, China
zhangshuoming21s@ict.ac.cn

Lei Chen
UCAS
Beijing, China
chenlei2014@mails.ucas.ac.cn

Yijin Li
SKLP, ICT, CAS
Beijing, China
liyijin@ict.ac.cn

Yangyu Zhang
SKLP, ICT, CAS
UCAS
Beijing, China
zhangyangyu19b@ict.ac.cn

Zhicheng Li
SKLP, ICT, CAS
UCAS
Beijing, China
lizhicheng21s@ict.ac.cn

Runyu Zhou
SKLP, ICT, CAS
UCAS
Beijing, China
zhourunyu22z@ict.ac.cn

Xiyu Shi
SKLP, ICT, CAS
Beijing, China
shixiyu@ict.ac.cn

Chunwei Xia
University of Leeds
Leeds, UK
C.Xia@leeds.ac.uk

Yuan Wen
University of Aberdeen
Aberdeen, UK
yuan.wen@abdn.ac.uk

Xiaobing Feng
SKLP, ICT, CAS
UCAS
Beijing, China
fxb@ict.ac.cn

Huimin Cui
SKLP, ICT, CAS
UCAS
XCORESIGMA CO.,LTD.
Beijing, China
cuihm@ict.ac.cn

***Abstract*—CUDA's programming model, exposing massive parallelism via fine-grained scalar threads, has become the de facto standard for GPU computing. Concurrently, NPUs are emerging as highly efficient accelerators, but their architecture is fundamentally different, relying on coarse-grained, explicit 2-D tile-based instructions. This creates a critical challenge: bridging the semantic gap "*From Threads to Tiles*". A direct translation is infeasible, as it requires lifting the implicit parallelism of CUDA's scalar model into the explicit, multi-dimensional vector space of NPUs, a problem we formalize as a lifting challenge.**

This paper introduces T2T, a compiler framework that automates this "Threads to Tiles" translation via the *2-D Vectorization* technique. T2T first transforms a CUDA kernel's implicit SIMT parallelism into a structured, explicit loop nest via our *Unified Parallelism Abstraction (UPA)*, making the parallelism analyzable. From this representation, T2T's core vectorization engine systematically selects optimal pairs of loops and maps them onto the NPU's 2-D tile instructions to maximize hardware utilization. To ensure correctness and handle performance-critical CUDA features, a final set of semantics-preserving optimizations is applied, including efficient control-flow management and vectorization of warp-level intrinsics.

We implement T2T based on Polygeist and evaluate representative NPU architectures. On a diverse set of benchmarks, kernels translated by T2T achieve up to 73% of native CUDA performance on an A100 GPU and outperform baseline translation approaches by up to 6.9×. Our work demonstrates that a systematic, compiler-driven approach to 2-D vectorization is a principled and high-performance path for porting the rich CUDA ecosystem to the evolving landscape of NPU accelerators.

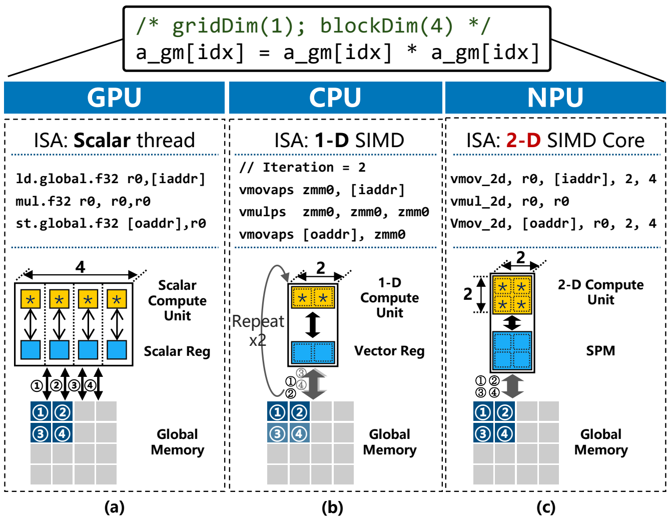
***Index Terms*—MLIR, CUDA, auto-vectorization, heterogeneous computing, AI accelerators**

* Corresponding author.

I. Introduction

General-purpose Graphics Processing Units (GPGPUs), primarily programmed via the CUDA framework [1], have become the dominant compute substrate for highly parallel workloads spanning domains such as scientific simulation [2]–[5], data analytics [6]–[8], and artificial intelligence [9], [10]. CUDA exposes a *scalar-thread* programming abstraction within the Single Instruction, Multiple Thread (SIMT) execution model, enabling developers to express massive parallelism using familiar sequential code constructs. Each thread is mapped to a lightweight scalar core and organized into warps that execute in lockstep, facilitating efficient hardware-level scheduling and instruction-level parallelism. This combination of expressive power and scalable execution has positioned CUDA as the *de facto* standard for accelerator programming in modern high-performance computing.

Concurrently, the hardware landscape is undergoing rapid diversification. *Neural Processing Units (NPUs)*, including architectures such as Google's TPU [11], Huawei Ascend [12], Cambricon [13], and Graphcore [14], are becoming increasingly pervasive across data centers, edge devices, and mobile System-on-Chips (SoCs). These domain-specific accelerators are designed to deliver significantly higher energy efficiency and performance for compute-intensive kernels, particularly those involving convolutions, matrix multiplications, and reductions. This evolution raises a fundamental question: *Can the rich ecosystem of CUDA applications be efficiently executed on NPUs without requiring extensive manual rewrites or hardware-*



```
/* gridDim(1); blockDim(4) */
a_gm[idx] = a_gm[idx] * a_gm[idx]
```

```
ld.global.f32 r0,[iaddr]
mul.f32 r0, r0,r0
st.global.f32 [oaddr],r0
```

```
// Iteration = 2
vmovaps zmm0, [iaddr]
vmulps  zmm0, zmm0, zmm0
vmovaps [oaddr], zmm0
```

```
vmov_2d, r0, [iaddr], 2, 4
vmul_2d, r0, r0
Vmov_2d, [oaddr], r0, 2, 4
```

Fig. 1: Execute the same CUDA program on GPU, CPU, and NPU: one block is launched, transferring four strided numbers from global memory to registers. The program's inherent parallelism is materialized differently across platforms.

specific reengineering?

The Challenge: Lifting Scalar CUDA Threads to 2-D Tile Instructions. Directly mapping CUDA kernels onto NPUs is challenging because the two programming models assume fundamentally different *vectorization semantics*. CUDA presents the abstraction of scalar threads (Level 0), where each thread executes a single iteration, and parallelism arises from launching a large number of blocks of threads. This illusion is powerful for programmability, but it means that vector structure is *implicit*, scattered across barriers, memory coalescing, and warp intrinsics. As shown in Figure 1(a), GPU launching 1 block with 4 threads each enables parallel processing of a 2x2 matrix, where each thread loads, computes, and stores 1 element. CPUs, in contrast, rely on *1-D SIMD vectors* (Level 1), where vector lanes are explicit and compiler auto-vectorization can systematically pack iterations. As shown in Figure 1(b), each CPU core vectorizes 2 consecutive elements and processes them in parallel. Using 1 core, a 2x2 matrix is computed by looping over the instructions twice.

NPUs go further by exposing *2-D tile instructions* (Level 2). As shown in Figure 1(c), a single NPU core can utilize 2-D memory and compute instructions to process a 2x2 matrix by loading 4 non-contiguous elements, processing them and storing the results. These instructions operate on multi-dimensional blocks of data with configurable strides and repeat counts, and they require explicit DMA transfers into on-chip SRAM and bank-aware alignment to achieve performance. To efficiently utilize these instructions, several requirements need to be met, such as loops must be restructured into two dimensions, arrays must be tiled, and overlap between DMA and compute must be generated. Therefore, migrating CUDA's implicit scalar abstraction to NPU's explicit 2-D tiles is more than simply "widening" loops; it involves *lifting and restructuring* the program's execution and memory model into a higher-dimensional vector space.

Based on the above observation, we present T2T, a compiler framework that lifts CUDA kernels to NPUs via *2-D Vectorization*, which consists of three key steps.

First, we restructure the parallelism exposed by CUDA's SIMT programming model into explicit loop nests. Since SIMT execution guarantees that all threads in a block or grid run in parallel, these loops are free of loop-carried dependencies. This transformation is captured in the *Unified Parallelism Abstraction (UPA)*, which provides a structured and analyzable form of the kernel.

Second, we perform *2-D vectorization* by fully exploiting the parallel semantics made explicit in UPA. This step formalizes 2-D vectorization as the problem of selecting and combining several loop dimensions from the original loop nest and mapping them onto the NPU's 2-D tile instructions. Our objective is to minimize the number of tile instructions executed while preserving correctness and maximizing hardware utilization.

Finally, we apply a series of *semantics-preserving optimizations* to bridge CUDA's execution model and NPU hardware. These optimizations include: control-flow mask optimizations to reduce branch overhead, and warp-level functions optimizations by leveraging parallel execution across all 32 threads for direct vectorization.

This paper makes the following contributions:

- We formalize the CUDA-to-NPU translation problem as a *lifting problem*, identifying the semantic gap between scalar-thread SIMT execution and 2-D tile instructions.
- We propose *2-D Vectorization*, a novel method built upon the Unified Parallelism Abstraction (UPA), which selectively and collaboratively maps several levels of the loop nest to NPU tile instructions.
- We design a series of *semantics-preserving optimizations*, including control-flow mask reduction and warp-level intrinsic vectorization.
- We implement a prototype system supporting two representative NPUs (Huawei Ascend [12] and Cambricon [13]). Evaluation on the Rodinia benchmark suite [15] and AI-CUDA-Engineer kernels [16] shows that our approach achieves up to 73% of native performance while outperforming baseline translations by 6.9×.

T2T is available at https://github.com/onlyoh/T2T.

II. Background

A. Classical Neural Processing Units (NPUs) Architecture

Neural Processing Units (NPUs) are specialized accelerators designed to efficiently run deep learning tasks. They excel at AI by using 2-D tile-based instructions and direct memory access to perform matrix operations on-chip. While NPUs are supported by high-level, AI-specific toolchains like PyTorch [9] and vendor SDKs [17], [18], their programming model is not suited for general-purpose workloads. A key architectural trade-off is that their powerful 2-D instructions impose stricter memory alignment constraints than typical CPU

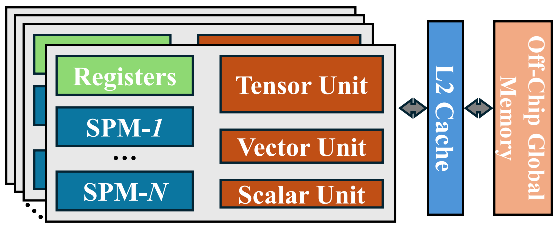


Fig. 2: NPU Architectures Abstraction

SIMD units—a design choice made to simplify hardware and improve efficiency.

This significantly increases the difficulty of porting CUDA programs to NPUs:

- SegSize Constraint: the size of each contiguous segment loaded per repeat must be a multiple of 32 B (e.g., the size of each blue row in GM in Figure 1(c)).
- SegStride Constraint: the stride between repeats must be a multiple of 32 B (e.g., the size of each gray row in GM in Figure 1(c)).

Multi-core Hierarchical Design. Modern Neural Processing Unit (NPU) architectures, while heterogeneous, are typically built on a hierarchical template of multiple Processing Engines (PEs), as shown in Figure 2. Each PE integrates specialized scalar (for control flow and address calculations), vector (for element-wise and reductions operations) and tensor (for dense computations such as matrix multiplication and convolution) compute units designed not around register files, but around direct access to software-managed scratchpad memories (SPMs). This architectural template is realized in prominent NPUs like Google TPU [11], Cambricon MLU [17], and Huawei Ascend [12].

Heterogeneous Compute and Memory Units. This paradigm presents fundamental challenges for porting CUDA programs. The NPU's reliance on explicitly managed SPMs contrasts sharply with the GPU model of hardware-managed caches and large register files, shifting the burden of data locality and scheduling entirely to the compiler. Furthermore, NPU memory systems are optimized for high throughput rather than low latency, demanding aggressive compiler techniques to hide memory access costs. While their hardware offers powerful, tensor-like semantics (e.g., long vectors and large strides), these dedicated instructions with intricate parameterization create a complex compilation target. These architectural departures from the familiar GPU model are what make direct translation of CUDA programs exceptionally difficult.

B. CUDA Programming Model

CUDA [1], the de facto programming standard for AI applications, employs a Single Instruction Multiple Thread (SIMT) model. Its programming model is built on a four-tier hierarchy of **Grids**, **Blocks**, **Warps**, and **Threads**, which is fundamental for compilers to analyze a CUDA kernel.

In this hierarchy, a grid is made of thread blocks, each of which has up to 1024 threads arranged in 1D, 2D, or 3D layout. Within each SM, threads are grouped into warps of 32 threads. Warps work in a SIMT (Single Instruction, Multiple Thread) manner, meaning all threads in a warp run on separate scalar CUDA cores, process the same instruction at the same time on different data. Each thread processes data like a single unit, which simplifies coding. Threads in a block use shared memory to share data, and the GPU schedules blocks to SMs based on available resources to maximize performance.

Semantically, threads within a block execute the same instructions in parallel, but there are often divergent control flow branches in the program, and different architectures handle branches differently. On GPUs with implicit parallelism, branches are executed sequentially while masking inactive cores. In contrast, SIMD architectures with explicit parallelism require if-conversion to transform control flow into data flow, enabling masked computation and memory instructions. When hardware lacks support for masked memory, blend instructions are used to emulate the semantics (as shown in Figure 4b line 6) [19] but introduce substantial memory overhead (line 5).

C. Polygeist/MLIR Compiler Infrastructure

The Multi-Level Intermediate Representation (MLIR) [20] is a versatile compiler infrastructure that supports flexible and modular code transformations through its dialect-based framework. This system allows multiple abstraction levels to coexist, enabling progressive lowering from high-level source code to target-specific representations [21]–[25]. Polygeist [26] provides a C/C++/CUDA front-end for MLIR, converting them into MLIR dialects while preserving critical high-level information, which is essential for our CUDA-to-NPU transformation pipeline.

III. Motivation

Transpiling CUDA programs, originally designed for SIMT execution, to non-SIMT processors has been an active line of research, explored by pioneering works such as Polygeist [26] and COX [27]. These efforts demonstrate that it is possible to retarget CUDA to CPUs, but the underlying challenge remains: the translation requires *lifting* from scalar-thread computation to either 1-D vector computation (the CPU case, the focus of most prior work) or to 2-D tile computation (the NPU case, our focus). We find that techniques developed for 1-D vectorization are insufficient when extended to the 2-D setting.

Figure 3 illustrates this point using a CUDA kernel from the Rodinia [15] benchmark suite, namely the `bpnn_adjust_weights_cuda` kernel in the `backprop` application. To fully exploit the 2-D tile instructions exposed by NPUs, this kernel requires 2-D vectorization.

Consider the memory access pattern at line 6 of Figure 3a. We first rewrite the CUDA kernel into a fully parallel loop-nest representation, shown in Figure 6(b). The parallelism is expressed explicitly as four nested loops with induction variables by, wid, txg and tx (from outermost to innermost), whose iteration ranges are ([1,4096]), ([1,8]), ([1,2]) and ([1,16]), respectively.

Examining the innermost loop tx, we observe that both `index` and `index_x` vary linearly with tx: each increment of

```
// gridDim(1, 4096), blockDim(16, 16)
__global__ void bpnn_adjust_weights_cuda(
  float *delta, int hid, float *w, float *oldw){
  int index = hid*16*by + hid*ty + tx + 1 + hid;
  int index_x = tx + 1;
  w[index] += delta[index_x] + oldw[index];
  __syncthreads();
  ...
}
```

(a) CUDA kernel snippet

```
//gridDim(1, 4096), blockDim(1, 16)
__global__ void bpnn_adjust_weights_cuda(..){
  int index = hid*16*by + hid*ty + 1 + hid;
  w[index:index + 16] += delta[1: 1 + 16] +
  ↪  oldw[index: index + 16];
  __syncthreads();
  ...
}
```

(b) 1-D Vectorization along tx (factor 16) as adopted by LLVM

```
//gridDim(1, 256), blockDim(1, 16)
__global__ void bpnn_adjust_weights_cuda(..){
// (addr, SegSize, SegStride, SegNum) 2D params
  int index = hid*16*by*16 + hid*ty + 1 + hid;
  Load (delta[1], 64, 0, 16) -> (delta_vec, 64,
  ↪  64, 16)
  Load (oldw[index], 64, hid*16*4, 16) ->
  ↪  (oldw_vec, 64, 64, 16)
  Load (w[index], 64, hid*16*4, 16) -> (w_vec,
  ↪  64, 64, 16)
  Compute (oldw_vec[0], 64, 64, 16) +
  ↪  (delta_vec[0], 64, 64, 16) + (w_vec, 64,
  ↪  64, 16) -> (w_vec, 64, 64, 16)
  Store (w_vec, 64, 64, 16) -> (w[index], 64, hid
  ↪  * 16 * 4, 16)
  __syncthreads();
  ...
}
```

(c) 2-D Vectorization along tx and by (with a factor of (16, 16))

Fig. 3: Part of *bpnn_adjust_weights_cuda* kernel in Rodinia suite *backprop* case[1]. 1-D vectorization is feasible along tx; it cannot be further extended to 2D with ty, but tx can be combined with by for 2-D vectorization.

```
__global__ void f(int n, float *A) {
  int tid = threadIdx.x;
  if (tid < n) A[tid] = tid;
  return;
}
```

(a) original scalar codes

```
__global__ void f_vec(int n, float *A) {
  int tidVec[128] = {0, 1,..., 127};
  int mask = VSCmpLt(tidVec, n);
  int oldVal[128], newVal[128];
  vecRead(oldVal, A, 0);
  vecSelect(newVal, tidVec, oldVal, mask);
  vecWrite(A, newVal, 0);
  return;
}
```

(b) vectorized mask execute codes

Fig. 4: Diverging control flow mask execution.

tx increases both values by 1. This regularity allows standard 1-D vectorization to be applied, as illustrated in Figure 3b. However, beyond this inner loop no further optimization is possible. For example, `index` includes the term $hid \cdot ty$, so each increment of ty advances the index by a stride that depends on the value of `hid`. This irregularity may violate the NPU's SegStride constraint, preventing ty from being combined with tx in a 2-D vectorization.

Instead, if we change our perspective from 1-D to 2-D vectorization and jointly analyze loops tx and by, a different opportunity emerges. In the by loop, the term in `index` is $hid \cdot 16 \cdot by$. Because the data type is `float` (4 bytes), each increment of by advances the index by a multiple of 64 bytes, which satisfies the SegStride constraint. Thus, tx and by can be combined to enable valid 2-D vectorization, as illustrated in Figure 3c (with a vector factor (VF) of 16 as an example).

This case motivates us that 2-D vectorization cannot be obtained simply by applying 1-D vectorization level by level from innermost loops. It requires more sophisticated analysis that cooperatively reason across multiple loop nests in order to satisfy NPU hardware constraints and unlock tile-level instructions.

IV. Methodology

A. Overview

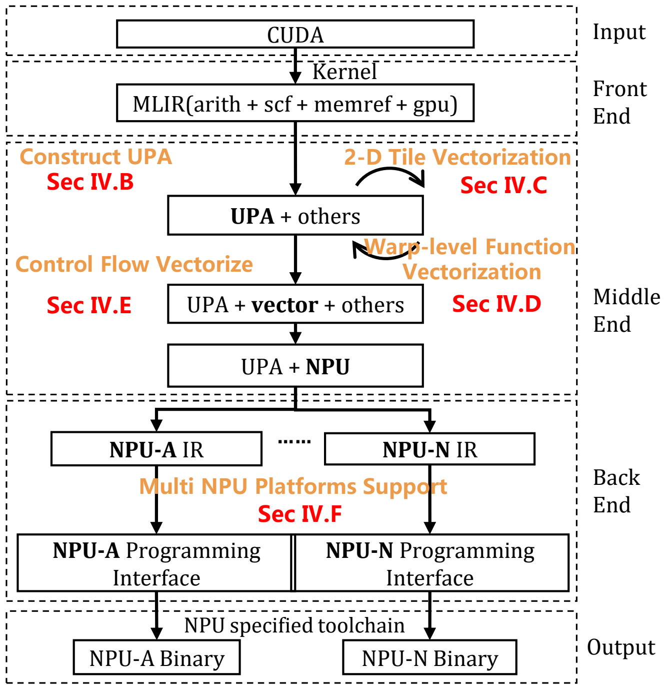


Fig. 5: Overview of our compiler pipeline.

The overall workflow of T2T is depicted in Figure 5. First, at the frontend, Polygeist parses CUDA programs into MLIR dialect representations. Subsequently, in the midend, we construct our platform-independent UPA (§§ IV-B), based on

[1]For clarity, we applied simplifications that do not affect the analytical complexity.

which multi-stage vectorization (§§ IV-C), control-flow mask optimizations (§§ IV-D), and semantics-driven translation of warp-level functions §§ IV-E are carried out. The program is then lowered to our proposed NPU IR, where NPU-generic optimizations are applied. Finally, in the backend (§§ IV-F), the NPU IR is further lowered for NPU-specific optimizations, and platform-specific code is generated and compiled into binary files using the respective platform toolchains.

In the following section, we describe our T2T designs in detail, highlighting each stage of the compilation process.

B. Unified Parallelism Abstraction (UPA)

We introduce the *Unified Parallelism Abstraction (UPA)*, a unified representation of CUDA kernels that makes implicit parallel semantics explicit in a form amenable to vectorization. Following the design principles of Polygeist [26] and related loop-based IRs, we build UPA on *loop nests*, motivated by the inherent thread-level parallelism of CUDA grids, blocks, and warps.

a) UPA function: We represent a CUDA kernel (i.e., a `__global__` function) as a **UPA function**:

$$\mathcal{F} = \mathcal{L}_{\texttt{grid}} \circ \mathcal{L}_{\texttt{block}} \circ \mathcal{L}_{\texttt{warp}} \circ \mathcal{L}_{\texttt{thread}} \circ \mathcal{B},$$

where $\mathcal{L}_d$ denotes a loop over dimension $d \in \{\texttt{grid}, \texttt{block}, \texttt{warp}, \texttt{thread}\}$, and $\mathcal{B}$ is the kernel body. This construction translates all levels of CUDA parallelism (grids, blocks and warps loops) into a structured nest of loops, thereby capturing the kernel's global parallel behavior in a uniform representation. Note that we explicitly model the `warp` level, as it is crucial for enabling warp-semantics optimizations (see §§ IV-E for details).

b) UPA Units: A UPA function is partitioned into one or more **UPA Units**. Each UPA Unit is defined as the *outermost perfect nest of loops* that can be vectorized as a whole. UPA Units serve as the fundamental units of vectorization: each Unit must adopt exactly one vectorization strategy (e.g., scalarization, 1-D SIMD, or 2-D tiling), while different Units may employ different strategies. We denote a UPA Unit as

$$\mathcal{U} = \mathcal{L}_1^O \circ \mathcal{L}_2^O \circ \cdots \circ \mathcal{L}_{k-1}^I \circ \mathcal{L}_k^I \circ \mathcal{B}',$$

where $\mathcal{L}_i^O$ are inter-block loop levels inherited from the UPA function, $\mathcal{L}_i^I$ are intra-block counterparts and $\mathcal{B}'$ is the Unit's loop body. We refer to the loop nests within a UPA Unit $\mathcal{U}$ as its *Parallel Regions (PRs)*.

c) Loop fission by inter-thread operations: UPA Units are constructed from the UPA function through *loop fission* at *inter-thread operations*. Inter-thread operations are CUDA intrinsics involving communication across threads: (i) *synchronization* (e.g., `__syncthreads` between threads in a block, `__syncwarp` between threads in a warp), or (ii) *data exchange* (e.g., `__shfl_sync`). Formally, given a loop nest $\mathcal{L}_1^O \circ \cdots \circ \mathcal{L}_k^I$ with body $\mathcal{B}$, if $\mathcal{B}$ contains an inter-thread operation *op*, we split $\mathcal{B} = \mathcal{B}_1 \cdot op \cdot \mathcal{B}_2$ and produce two Units:

$$\mathcal{U}_1 = \mathcal{L}_1^O \circ \cdots \circ \mathcal{L}_k^{I1} \circ \mathcal{B}_1, \quad \mathcal{U}_2 = \mathcal{L}_1^O \circ \cdots \circ \mathcal{L}_k^{I2} \circ \mathcal{B}_2.$$

For synchronization operations, *op* is removed after fission, ensuring that $\mathcal{U}_1$ executes completely before $\mathcal{U}_2$, thus preserving CUDA's execution–synchronization–execution semantics. For data-exchange operations, *op* is preserved as an explicit transfer between $\mathcal{U}_1$ and $\mathcal{U}_2$. This mechanism ensures that loop nests faithfully capture CUDA's inter-thread semantics in a barrier-free form suitable for independent vectorization.

d) Example: Figure 6 illustrates this process: a CUDA kernel is first translated into a UPA function (multi-level loop nest), then partitioned into multiple UPA Units by loop fission at `__syncthreads` and warp-level intrinsics. Each Unit can then be mapped to a specific vectorization strategy.

C. 2-D Tile Vectorization

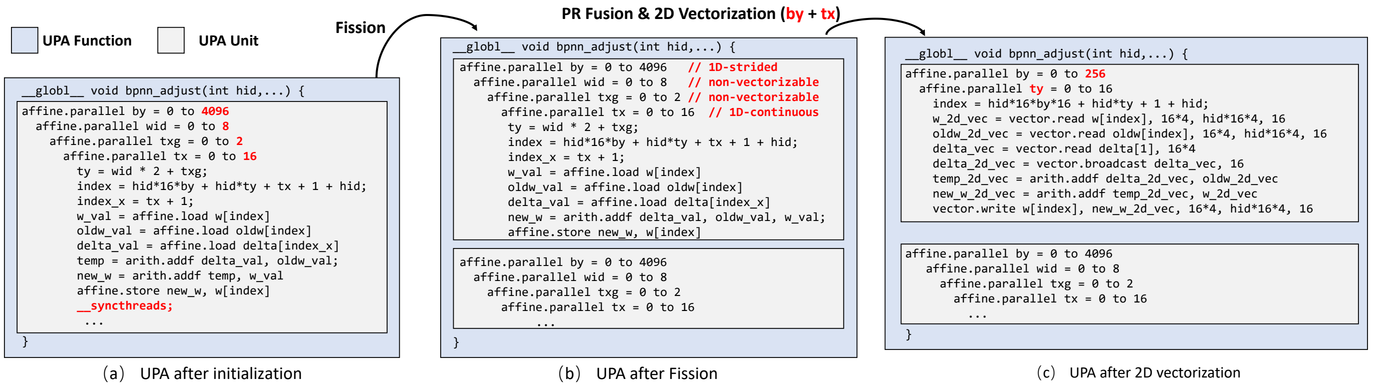
In each UPA unit, 2-D tile vectorization is applied to generate 2-D memory and compute instructions targeting NPUs. As specified in §§ II-A, 2-D NPU instructions must satisfy the SegSize constraint and the SegStride constraint, making 2-D tile vectorization differ from conventional loop vectorizers adopted by existing compilers (e.g., LLVM), in both its legality and profitability.

For legality, the fundamental assumption of conventional loop vectorizers lies in that loops may present complicated loop-carried dependencies, which no longer holds true in UPA units, where all loop nests (i.e., parallel regions) are fully permutable according to CUDA semantics, as shown in Figure 6 (b). Thereby, data dependence analysis can be absent for 2-D tile vectorization.

For profitability, conventional loop vectorizers first pick a single loop nest (the innermost one in most cases) and then exploit various cost models to evaluate the static cycles of the instruction sequence in its body, to determine whether to transform the original scalar instruction sequence into vector. Obviously, such linear approach is not applicable to 2-D instructions in NPUs, which may require more than one loop nests (i.e., parallel regions to be selected (from up to 7, representing blk.xyz, warp, tid.xyz respectively) and transformed for better hardware utilization. We adopt a two-step strategy that first analyzes the vectorization pattern of each PR in each UPA unit, and then decides the optimal way to combine such vectorization patterns so that the number of 2-D instructions executed by this UPA unit can be minimized.

1) Parallel Region Vectorizability Analysis : Given a PR, it presents one of the following five vectorization patterns:

- *Transparent*, if all its loop iterations access the same set of memory locations.
- *1D-continuous-vectorizable*, if each loop iteration accesses a consecutive memory region occupying a constant len bytes, two memory regions accessed by any two adjacent loop iterations are adjacent, and the total footprint $len \times tc$ satisfies the SegSize constraint.
- *1D-strided-vectorizable*, if each loop iteration accesses one data element with the stride between the two elements accessed by adjacent loop iterations staying constant across all iterations (thus denoted as $stride$), and $stride$ satisfies the SegStride constraint.


UPA Function
UPA Unit
Fission
PR Fusion & 2D Vectorization (by + tx)
(a) UPA after initialization
(b) UPA after Fission
(c) UPA after 2D vectorization

```
__globl__ void bpnn_adjust(int hid,...) {
affine.parallel by = 0 to 4096
  affine.parallel wid = 0 to 8
    affine.parallel txg = 0 to 2
      affine.parallel tx = 0 to 16
        ty = wid * 2 + txg;
        index = hid*16*by + hid*ty + tx + 1 + hid;
        index_x = tx + 1;
        w_val = affine.load w[index]
        oldw_val = affine.load oldw[index]
        delta_val = affine.load delta[index_x]
        temp = arith.addf delta_val, oldw_val;
        new_w = arith.addf temp, w_val
        affine.store new_w, w[index]
        __syncthreads;
         ...
}
```

```
__globl__ void bpnn_adjust(int hid,...) {
affine.parallel by = 0 to 4096   // 1D-strided
  affine.parallel wid = 0 to 8   // non-vectorizable
    affine.parallel txg = 0 to 2 // non-vectorizable
      affine.parallel tx = 0 to 16  // 1D-continuous
       ty = wid * 2 + txg;
       index = hid*16*by + hid*ty + tx + 1 + hid;
       index_x = tx + 1;
       w_val = affine.load w[index]
       oldw_val = affine.load oldw[index]
       delta_val = affine.load delta[index_x]
       new_w = arith.addf delta_val, oldw_val, w_val;
       affine.store new_w, w[index]

affine.parallel by = 0 to 4096
  affine.parallel wid = 0 to 8
    affine.parallel txg = 0 to 2
      affine.parallel tx = 0 to 16
          ...
}
```

```
__globl__ void bpnn_adjust(int hid,...) {
affine.parallel by = 0 to 256
  affine.parallel ty = 0 to 16
    index = hid*16*by*16 + hid*ty + 1 + hid;
    w_2d_vec = vector.read w[index], 16*4, hid*16*4, 16
    oldw_2d_vec = vector.read oldw[index], 16*4, hid*16*4, 16
    delta_vec = vector.read delta[1], 16*4
    delta_2d_vec = vector.broadcast delta_vec, 16
    temp_2d_vec = arith.addf delta_2d_vec, oldw_2d_vec
    new_w_2d_vec = arith.addf temp_2d_vec, w_2d_vec
    vector.write w[index], new_w_2d_vec, 16*4, hid*16*4, 16

affine.parallel by = 0 to 4096
   affine.parallel wid = 0 to 8
     affine.parallel txg = 0 to 2
       affine.parallel tx = 0 to 16
         ...
}
```

Fig. 6: Examples of Unified Parallelism Abstraction(UPA) for CUDA code in Figure 3(a).

- *2D-tile-vectorizable*, if all its loop iterations can be evenly divided into several strips, satisfying: 1) all loop iterations in any strip access the same consecutive memory region whose length stays invariant across all strips (thus denoted as len), and len satisfies the SegSize constraint, and 2) the stride between the two memory regions accessed by two adjacent strips stays invariant across all strips (thus denoted as $stride$), and $stride$ satisfies the SegStride constraint.
- *Non-vectorizable*, otherwise.

In this way, 2-D tile vectorization is eligible for a single *2D-tile-vectorizable* PR or a combination of one 1D-continuous-vectorizable PR and one *1D-strided-vectorizable* PR, and in either case, one or more *transparent* PRs may be included, but any *non-vectorizable* PR must be excluded.

Vectorizability analysis for a given PR, is performed by analyzing and synthesizing all its memory access operations.

To achieve this, we have first extended the MLIR affine dialect – which is traditionally limited to linear expressions $\mathrm{A}[a*i+b]$ (with i the loop index, a an integer constant and b an expression invariant with i) – to further support non-linear operations such as division, enabling analysis of complex non-linear CUDA memory accesses expressed as $\mathrm{A}[a*i/c+b]$ (with c an integer constant).

Based on this, the pattern of each memory access w.r.t. a given PR indexed by i (a read from or a write to, say, $\mathrm{A}[a*i/c+b]$, with indices from other PRs, say, j independent from i thus expressed in b) is further examined, which are classified into 1) invariant when $a=0$, 2) continuous when $a=c=1$, 3) strided when $a>1, c=1$, 4) tiled when $a>\frac{32}{\mathrm{sizeof(A)}}, c/a>\frac{32}{\mathrm{sizeof(A)}}$, and 5) irregular otherwise. This PR is recognized as *transparent* if all its memory accesses are invariant, *1D-continuous-vectorizable* if all its memory accesses are invariant or continuous, *1D-strided-vectorizable* if all its memory accesses are invariant or strided, *2D-tile-vectorizable* if all its memory accesses are invariant or tiled, and *non-vectorizable* otherwise.

We take the analysis of all PRs in Figure 6(b) as an example. It contains 4 memory access instructions: load_w, load_oldw, load_delta and store_w. The analysis result of all PRs are shown in Table I.

TABLE I: Analysis results of all PRs in Figure 6(b), $Trans$, $1D-C$, $1D-S$, $2D$ and Non respectively represent Transparent, 1D-continuous-vectorizable, 1D-strided-vectorizable, 2D-tile-vectorizable and Non-vectorizable.

	load_w	load_oldw	load_delta	store_w	PR_final
by	1D-S	1D-S	Trans	1D-S	1D-S
wid	Non	Non	Trans	Non	Non
txg	Non	Non	Trans	Non	Non
tx	1D-C	1D-C	1D-C	1D-C	1D-C

2) 2-D Vectorization Plan Generation: Given a UPA unit, the key idea of generating its 2-D tile vectorization plan is to minimize its dynamic 2-D instruction count, under the constraint that its working size does not exceed the SPM capacity.

Since a UPA unit has at most 7 PRs, it is not challenging to simply first enumerate all possible 2D-tile-vectorizable PRs or combinations of one 1D-continuous-vectorizable PR and one 1D-strided-vectorizable PR (as well as the involvement of transparent PRs), and then pick one with the minimal total 2-D instruction count. However, the situation is complicated by the fact that a PR may have a trip-count that is statically unknown when it represents a dimension of CUDA thread-blocks (PRs w.r.t dimensions of CUDA threads all have statically-known trip-counts). To deal with it, we delay the decision until those unknown trip-counts have been resolved, by generating multiple 2-D tile vectorization plans.

In particular, two or more PRs may be fused into one single PR, so that the total number of PRs can be decreased to reduce the enumerating space above. Only two adjacent PRs can be fused, and following rules apply: 1) adjacent *1D-continuous-vectorizable* PR and *1D-strided-vectorizable* PR can be fused into a *1D-continuous-vectorizable* PR, if the stride of the *1D-strided-vectorizable* PR equals to the footprint of the *1D-continuous-vectorizable* PR, and 2) two adjacent *2D-tile-vectorizable* PRs can be fused into a *2D-tile-vectorizable* PR if they have the same number of strips and two corresponding

TABLE II: Generated 2-D vectorization plans of Figure 6(b). OK indicates that the PRs can be combined for 2-D vectorization, whereas NO indicates the opposite.

	by	ty	tx
by	\	NO	OK
ty	NO	\	NO
tx	OK	NO	\

strips from the two PRs access adjacent memory regions. For example, in Figure 6, wid and txg can be fused and get ty.

For example, if pr_2 is 1D-continuous and pr_1 is 1D-strided, fusing them yields pr_3 as 1D-continuous. This eliminates the need for 2-D vectorization(for pr_2 with pr_1), allowing pr_3 to be vectorized in 1D and can combine with another 1D-strided PR for greater coverage. Fusion also can prune the search space in vectorization plan generation by reducing PRs number.

To further enhance performance, in-place computation has been enabled, which allows two arrays A, B to be allocated to the same SPM address if their live range does not overlap, to eliminate 2-D memory instructions that perform copy between different SPM addresses.

We take the 2-D vectorization plan generation in Figure 6(b)(c) as example. It contains 3 PRs: by, ty, and tx. The generated 2-D vectorization plans are shown in Table II.

D. Control Flow Simplification

Conditionals residing in CUDA kernels may introduce expensive masked 2-D vectorization instructions. To reduce them, CUDA-semantic-related control flows are simplified before 2-D tile vectorization, by first unfolding the expressions in such conditionals and then applying constant propagation and PR splitting.

We exemplify such control flow simplification with the conditional at lines 2 in Figure 7, with 32 threads in the first warp setting the initial values of array $\mathrm{d\_temp}$. With conventional loop vectorizer, first a mask is generated according to whether the expression $\mathrm{tx}/32$ evaluates true, and then followed by a masked vector store instruction that performs store according to the mask, in a vector-lane-wise way.

However, apparently, such mask in this case can be eliminated, simplifying the case to an unconditional vector store instruction. This has been achieved, by first unfolding the conditional $\mathrm{if}\ (\mathrm{tx}/32 == 0)$ into $\mathrm{if}\ (\mathrm{warp\_id} == 0)$ according to CUDA semantics (as line 14), and then propagating the value of $\mathrm{warp\_id}$ so that the PR indexed by $\mathrm{warp\_id}$ can be split for mask-free 2-D vectorization (as line 19).

E. Warp-level Function Vectorization

CUDA features warp-level functions to coordinate all threads in a warp, purely for synchronization or further for inter-thread computation/data exchange.

As elaborated in §§ IV-B, the CUDA semantic of warp-level synchronization has been preserved by loop fission in UPA construction, and when inter-thread computation or data exchange presents, the original warp-level function is kept

```
1  // a: original kernel code:
2  if (tx/32 == 0) {
3    d_temp[col_idx] = 0.0f;
4  }
5  // b: traditional optimized: masked store
6  parallel %tx = 0 to block.x, step 32
7    %mask = icmp ugt i32 %tx_div_32, 0
8    store <32 x float>, i32* %d_temp, %mask !cond
     ↪  !mask
9  // c: after mathematically equivalent
   ↪  simplification:
10 parallel %warp_id = 0 to 8, step 1
11   parallel %lane_id = 0 to 32, step 1
12     if(%warp_id == 0) {
13       d_temp[(blockIdx.x % 8) * 32 + %lane_id] =
         ↪  0.0f;
14     }
15 // d: masked-free vectorization, vector factor is
   ↪  32:
16 vector.transfer_write zero_vec_32,
   ↪  d_temp[(blockIdx.x % 8) * 32]
```

Fig. 7: Examples of mask optimization with UPA.

```
1  // warp vote:
2  res = __any_sync(mask, val);
3  // vectorize:
4  tmp = vector.any(val_vec, mask);
5  res_vec = vector.broadcast(tmp);
6
7  // warp shuffle: reduction sum
8  for (int offset = width / 2; offset > 0; offset
   ↪  /= 2)
9    val += __shfl_down(val, offset);
10 // after optimization, we can use vector
   ↪  reduction sum
11 val = vector_reduction_sum(val_vec);
```

Fig. 8: Examples of warp-level function optimization

as a statement in UPA, introducing complexity to 2-D tile vectorization.

Such warp-level function can be simply expanded into scalar codes that complete the tasks sequentially from threads 0 to 31 in this warp (as in COX [27]), however the yielded codes are typically hard to vectorize, since operations from the 32 threads are not isomorphic. Such scalar execution will introduce notably high costs on NPUs, thereby should be further vectorized.

Leveraging cross-vector-lane vectorization instructions (e.g., broadcasting a value to all vector lanes, reducing values of all vector lanes, etc), major warp-level functions can be vectorized, and we take vote and reduction as example, shown in Figure 8. The warp-level vote function, $\mathrm{\_\_any\_sync}$ at line 2, yields $\mathrm{res} = 1$ if there exist a thread in this warp satisfying $\mathrm{mask} = \mathrm{val} = 1$, thereby can be replaced with two vector instructions in lines 4-5, which checks values of $\mathrm{mask}, \mathrm{val}$ across all vector lanes first, and then broadcast the result to all lanes. The warp-level reduction function, $\mathrm{\_\_shfl\_down}$ enclosed in a loop at lines 8-9, accumulates values in all threads into val, thereby can be replaced with a single vector reduction instruction at line 11.

F. Support Multi NPU Platforms

To ensure portability for T2T, we designed an NPU-specific backend using MLIR's dialect mechanism. This backend is

```
// vector dialect op: vector muls with scalar
%b_vec = vector.broadcast(%b)
%c_vec = arith.mul(%a_vec, %b_vec)
// after optimization, we can use NPU's
↪  vector-scalar instruction.
%c_vec = npu.vmuls(%a_vec, %b)
```

Fig. 9: Examples of NPU-related optimization

organized into a two-layer dialect: a High-Level IR (HIR) that preserves program semantics while exposing common hardware features, and a Low-Level IR (LIR) that captures platform-specific instructions and constraints (e.g., the lack of vector division on Ascend chips [18]).

This design balances semantic completeness with modularity, allows T2T to incorporate platform-specific optimizations and easily extend support for emerging NPU architectures by adding new LIR backends, without modifying the core analysis and optimization pipeline.

After vectorization and above related optimizations, the program is lowered from platform-independent representations to HIR, which unifies memory, computation, and intrinsic operations. A platform-specific legalization pass then converts it into the LIR. At the NPU IR stage, we apply two classes of optimizations: generic optimizations that exploit architectural commonalities across NPUs (as illustrated in Figure 9), and platform-specific optimizations that leverage unique hardware features, such as on-the-fly format conversion on Ascend chips or the rich hardware activation functions supported by Cambricon MLUs.

After NPU-specific optimizations, T2T performs platform-specific code generation to produce the final binary file. On the Ascend platform, a CCE program is generated, providing fine-grained control over memory, compute, and synchronization, and compiled with the ccec compiler. On the Cambricon platform, a BangC program is generated (the lowest-level language available to external developers) and compiled with cncc.

V. Evaluation

A. Experimental Setup

Hardware and Software Environment. We verify the correctness of all translated code on physical hardware. Our evaluation utilizes two distinct NPUs from two major vendors, with the NVIDIA A100 GPU serving as the baseline for correctness and performance comparison. We detail the specific configurations in Table III. We build the T2T framework on top of Polygeist [26](commit ID: 8f8ae725), extending its parallel and affine representation, mid-end optimization capabilities, and back-end support to enable the efficient execution of CUDA programs on NPUs.

Benchmark Selection. We evaluate T2T using two representative CUDA benchmark suites.

[1]All FLOPS refer to non-Tensor-Core performance

TABLE III: Hardware specifications for Ascend 910B1, Cambricon MLU370, and NVIDIA A100. The NVIDIA A100 GPU serves as a well-established baseline.

Metric	A100	910B1	MLU370
SMs/Cores	108	24 cube cores + 48 vector cores	16
TFLOPs (FP32)[1]	19.5	90	24
Memory Bandwidth	1555 GB/s	1600 GB/s	614 GB/s
Global Memory	40 GB	64 GB	24 GB
L2 Cache	40 MB	192 MB	2 MB
L1 / SPM	192 KB	192 KB	768 KB

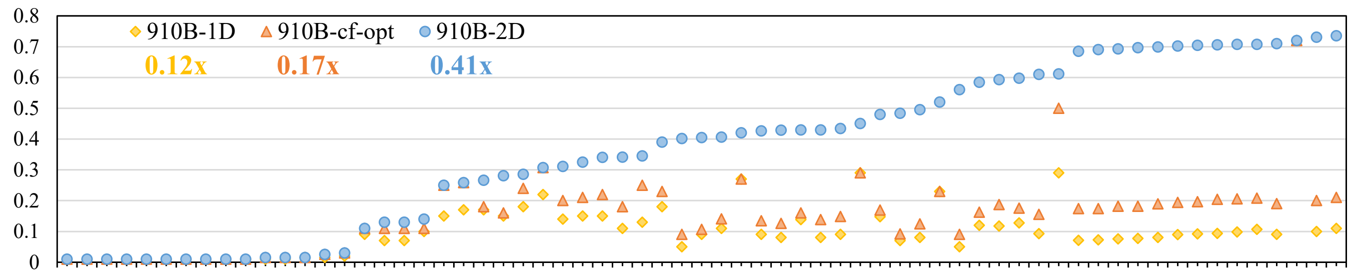
The Rodinia suite [15] contains 24 CUDA benchmarks; Polygeist currently supports 14 of them. Because the target NPUs lack double-precision support, we exclude `particlefilter`, leaving 13 benchmarks comprising 28 kernels. We use these to evaluate T2T's coverage of classical CUDA workloads.

We synthesize a 65-kernel benchmark suite using the AI CUDA Engineer suite [16]. We cover six L1 categories from AI CUDA Engineer—matrix multiplication, activation, normalization, pooling, reduction, and loss—and omit convolution, as modern high-performance implementations are typically provided by vendor libraries such as cuDNN [30]. We also exclude `MSELoss` because the target NPUs lack double-precision support. For operators/tasks with multiple kernel implementations, we select the best-performing variant on NVIDIA A100. Since these only exercise warp shfl, we add `simpleVoteIntrinsics` from CUDA samples to test warp vote support. This process yields 65 kernels in total.

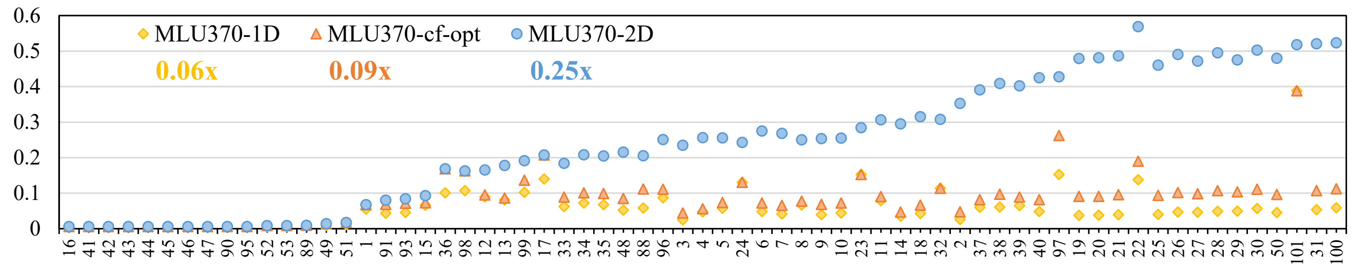
Performance. We verify correctness by comparing output against native GPU execution. Kernel execution time is measured using event-based timers provided by each vendor's library, averaged over 1000 runs. Performance is reported as the ratio of program throughput to the device's theoretical peak, then normalized by the corresponding ratio for CUDA on an NVIDIA A100 GPU. For example, a normalized value of 0.8 means the NPU kernel achieves 80% of the memory-bandwidth or compute utilization attained by the CUDA implementation on A100.

Baselines. To evaluate our proposed 2-D vectorization, we must compare it against a strong 1-D vectorization baseline. The WCCV compiler [31] is a comparable system for GPU-to-CPU translation that employs 1-D vectorization. However, as it targets CPU architectures, a direct performance comparison with our NPU-targeting system is not feasible. We therefore adapt WCCV's core methodology. We have **implemented a 1-D vectorization pass within our own compiler framework**. This baseline pass mirrors the approach of WCCV by identifying and vectorizing a single level of parallelism, effectively representing core 1-D vectorization for CUDA-like workloads.

[2]*hybridsort, kmeans, leukocyte, mummergpu, huffman, heartwall, dwt2d, b+tree, lavaMD, and srad_v2* are excluded according to [28], [29].



(a) Ascend 910B



(b) MLU370

Fig. 10: Normalized Performance on AI CUDA Engineer Bench [16]. `xx-1D` refers to the 1-D vectorization adopted from WCCV [31]; `xx-cf-opt` denotes control-flow simplification; and `xx-2D` represents the 2-D vectorization strategy employed by T2T.

B. Overall Performance

The results of AI CUDA Engineer suite are shown in Figure 10. The horizontal axis denotes the index of each L1 kernel in the suite, where index `101` corresponds to the `simpleVoteIntrinsics` kernel. The vertical axis shows the relative performance metric, as defined in §§ V-A. Overall, T2T successfully executes all 65 kernels on NPUs, achieving up to 74% peak and 41% average performance on the Ascend 910B, and 52% and 25% on the Cambricon MLU370.

To evaluate the effects of control-flow vectorization optimization and 2-D vectorization, we compared them against a baseline with only 1-D vectorization. On the Ascend 910B, control-flow optimization accelerated the 1-D baseline by 42%, and 2-D vectorization on top of it provided an additional 141% speedup. On the MLU370, the corresponding improvements were 50% and 178% respectively.

Except for a few cases (e.g., the `tanh` kernel, where A100 GPUs and Cambricon MLU370 benefit from hardware acceleration unavailable on Ascend chips), performance trends across NPUs are broadly similar. We therefore focus our analysis on the Ascend 910B platform.

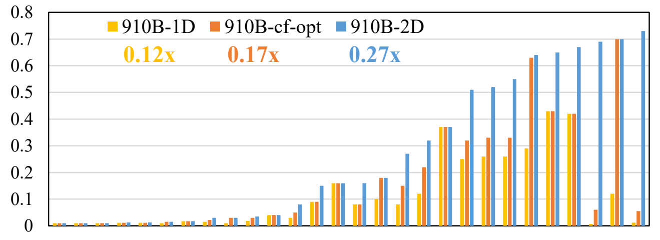
The AI CUDA Engineer benchmark covers diverse operator categories, where operators within the same category exhibit similar performance due to shared computation and memory patterns. Activation operators (e.g., ReLU, GeLU) show the best performance (68%-70%) since their regular patterns enable efficient cross-block 2-D vectorization. Matrix multiplication operators typically launch 2D blocks, where 2-D vectorization exploits both threadIdx.x/y but requires additional broadcast instructions, yielding 40% performance. Pooling operators involve non-contiguous accesses (e.g., $a = A[tid * stride]$) with stride unknown at compile time, making vectorization ineffective, achieving only 0.01× or lower.

Our approach effectively vectorizes most warp-level functions. By combining the Unified Parallelism Abstraction (UPA) with semantic-guided translation, we successfully handle vote and simple shfl operations, achieving 0.72× and 0.45× of native performance on representative kernels. Some workloads that are incompatible with multi-level vectorization still yield around 0.38× performance. However, in rare cases with complex shfl patterns, vectorization is not possible, forcing a fallback to slow scalar simulation that results in only 0.03× performance. These results demonstrate that the final performance is largely dictated by the memory access patterns of the specific workload.

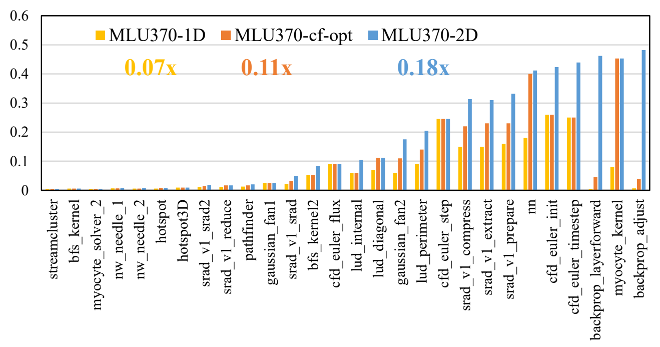
A similar trend is observed in the Rodinia suite, as shown in Figure 11. T2T successfully translates and executes all 28 kernels on the NPU. Overall, T2T achieves 73% peak and 27% average performance on the Ascend 910B, and 48% and 18% on the Cambricon MLU370.

For Rodinia benchmark, on the Ascend 910B, control-flow optimization accelerated the 1-D baseline by 42%, and 2-D vectorization on top of it provided an additional 59% speedup. On the MLU370, the corresponding improvements were 57% and 63%, respectively.

In the `backprop` case, which launches 4096 blocks, performance improvements are dominated by cross-block vectorization, yielding a speedup of about 6.9× over the 1-



(a) Ascend 910B



(b) MLU370

Fig. 11: Normalized Performance on Rodinia Bench [15].

```
1  // gridDim: (1, 4096); blockDim:(16, 16)
2  __global__ void bpnn_layerforward(...) {
3    int by = blockIdx.y;
4    int tx = threadIdx.x, ty = threadIdx.y;
5    int index = (hid + 1) * HEIGHT * by + (hid + 1)
     ↪  * ty + tx + 1 + (hid + 1);
6    int index_in = HEIGHT * by + ty + 1;
7    __shared__ float node[HEIGHT];
8    __shared__ float weights[HEIGHT][WIDTH];
9    if (tx == 0) node[ty] = input[index_in];
10   __syncthreads();
11   weights[ty][tx] = hidden[index];
12   __syncthreads();
13   weights[ty][tx] = weights[ty][tx] * node[ty];
14   __syncthreads();
15   for (int i = 1; i <= __log2f(HEIGHT); i++) {
16     if (ty % pow(2, i) == 0)
17       weights[ty][tx] += weights[ty + pow(2,
         ↪  i-1)][tx];
18     __syncthreads();
19   }
20
21   hidden[index] = weights[ty][tx];
22   __syncthreads();
23
24   if (tx == 0) out[by * hid + ty] =
     ↪  weights[tx][ty];
25 }
```

Fig. 12: Rodinia backprop `bpnn_layerforward` kernel, the highlighted regions correspond to the program's memory access and computation code.

D baseline. By contrast, `myocyte_kernel`, executed with 1 block and 1 thread, the main benefit comes from control-flow mask optimization, which avoids masked execution and achieves GPU-comparable performance, with a speedup of 5.8×. And for `nn`, which launches only 168 one-dimensional blocks, gains little from 2-D vectorization.

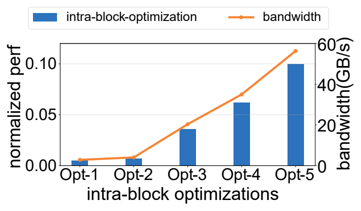
The remaining performance gap can be mainly attributed to three factors. First, hardware architectural differences: NPUs require distinct programming paradigms and data layouts, making it challenging for translated CUDA programs to fully exploit their potential. Moreover, operations that are implicitly and efficiently supported by GPU hardware (e.g., broadcasts) often require explicit instructions on NPUs, incurring additional overhead. Second, hardware parallelism vs. software vectorization: GPUs rely on implicit hardware parallelism and can leverage runtime information for optimizations such as memory coalescing, while NPUs rely on compiler-driven explicit vectorization constrained by compile-time information. Finally, runtime and driver overheads: differences in runtime systems and driver implementations between NPUs and GPUs may further contribute to the observed performance gap.

C. Case Study: Rodinia Backpropagation

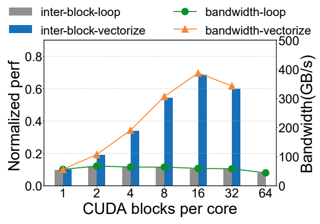
We demonstrate the synergy of our intra-block optimizations on the `bpnn_layerforward` kernel from the Rodinia `backprop` benchmark as a case study - a kernel characterized by complex control flow and multi-dimensional shared memory - using a 4096×(16×16) grid-block configuration, as shown in Figure 13. Our optimizations achieve the following cumulative gains relative to native CUDA performance (CUDA_perf): single-layer vectorization with static analysis improves performance from baseline 0.51% to 0.72%; control flow optimizations on unnecessary mask instructions, redundant memory access and superfluous loops, then provide a 5× speedup to 3.6%; store-to-load forwarding adds a 1.7× gain, reaching 6.2%; and finally, multi-level vectorization on Line 13 in Figure 12 yields another 1.6× boost for a final performance of 9.9%.

We chose not to apply multi-level vectorization to lines 9, 11, 17, 21, and 24 for the following reasons. Lines 9 and 24 were already reduced to a single vector instruction by prior optimizations, rendering further vectorization redundant. For lines 11 and 21, the memory access stride along the `ty` dimension is a variable `(hid+1)`, which prevents our compiler from statically verifying the strict memory alignment required by the NPU hardware. We thus conservatively avoid vectorization for these lines. Finally, vectorizing line 17, which resides within a `ty`-dependent conditional block, would introduce costly mask instructions and redundant memory accesses, negating any potential performance benefit.

The 9.9% of CUDA_perf ceiling for intra-block parallelism, a limit imposed by the Ascend 910B's hardware scheduler and fewer vector cores, necessitates leveraging inter-block parallelism. We explored two approaches: a linear execution of blocks (`inter-block-loop`) and cross-block multi-level vectorization (`inter-block-vectorize`). The linear approach underutilizes NPU resources, yielding a maximum 1.2× speedup. In contrast, our cross-block vectorization, built on UPA, leverages the NPU's multi-repeat and strided memory



(a) Intra-block Optimizations



(b) Inter-block Optimizations

Fig. 13: Optimizations of Rodinia backprop case layerforward kernel. **(a)** Intra-block optimization ablation: options(1-5): 1)baseline with vector length 8; 2)1-D vectorization adopted from WCCV [31]; 3)control-flow simplification; 4)in-place computation; 5)2-D vectorization within a block. **(b)** Further inter-block 2-D vectorization based on 5).

access features. This allows a single instruction to process tasks from multiple CUDA blocks simultaneously, achieving a 6.9× speedup and reaching 68.7% of CUDA_perf with a configuration of 16 CUDA blocks per core (CBPC=16).

We observe a typical performance trend: performance scales near-linearly with the number of CBPC until hardware utilization saturates, after which gains diminish as the execution time of individual NPU blocks increases. The peak performance is achieved at CBPC=16, which maps 256 NPU blocks to 48 vector cores; this suggests that reserving scheduling flexibility for the hardware is more effective than having the compiler handle all vectorization tasks. The maximum CBPC is capped at 64 and 32 for each method, either to avoid underutilizing vector cores or due to hardware constraints. Ultimately, our approach achieves 68.7% of native CUDA performance in `bpnn_layerforward` kernel.

VI. Discussion

Extensibility to other NPU architectures. Beyond the currently tested hardware, our framework exhibits strong portability to other NPUs, provided they expose vendor-specific MLIR dialects, C-like programming interfaces, or LLVM IR. This extensibility stems from our compilation pipeline design, which lowers CUDA to a target-independent High-level IR (HIR) before lowering to a target-dependent Low-level IR (LIR). Since HIR abstracts common NPU hardware features, a significant portion of the translation logic is reusable across targets.

Specifically, for NPUs exposing native MLIR dialects, these can be directly utilized as the LIR layer. For architectures providing C-like interfaces or LLVM IR, LIR construction is achieved by retaining HIR operations mapped to native instructions and specifying architecture-specific parameters, such as segment size and stride constraints for 2-D vectorization. Consequently, our framework can potentially support emerging architectures like AMD XDNA, which provides both MLIR dialects (AIE, AIEVec) and C-like APIs. Furthermore, XDNA's support for 2D memory operations (e.g., 2D DMA) aligns well with our 2-D vectorization pass, promising efficient hardware utilization. However, support for Google TPUs is currently not feasible. To the best of our knowledge, Google TPUs primarily expose a graph-level interface (e.g., StableHLO) without the publicly accessible low-level IR or C-like interface required by our current translation pipeline.

Limitations regarding Tensor Core. The primary limitation of our framework is the current lack of support for Tensor Core operations due to their highly complex usage patterns. The code patterns involved in production-level Tensor Core usage are exceptionally complex, often involving intricate data layouts and memory accesses that are challenging to analyze and translate systematically. We plan to address this in future work by developing more powerful and comprehensive program analysis techniques to correctly identify and map these operations to the corresponding tensor units on NPUs.

VII. Related Work

Prior works on porting CUDA to CPUs [26], [27], [32]–[39] rely on 1-D vectorization, multithreading, or simulation [40]–[43], making them ill-suited for NPUs that depend on efficient 2-D SIMD execution. Work on retargeting CUDA to other GPUs [29], [44], [45] also does not address NPU-specific vectorization. While NPU-specific models like SYCL [46]–[49] and Ascend C [18] exist, they demand extensive manual rewriting, which our automated approach is designed to avoid.

While many compilers [19], [27], [31], [50]–[52] focus on vectorizing SIMT programs, they fall short for NPUs. Prominent techniques like Whole-Function Vectorization [19] are limited by operating on a low-level IR [53] and assuming fixed-width SIMD, rendering them unable to handle the variable-length vector units and complex memory subsystems of NPUs. Other compilers [31], [50] are similarly constrained, as they either rely on generic backends, requiring manual annotation, or are limited to intra-warp vectorization, failing to unlock the hardware's full potential. Even recent LLM-based translators like QiMeng-Xpiler [54] presents challenges in correctness and performance predictability, and it also bypasses mature compiler optimizations. In contrast, our work employs a systematic compiler approach to ensure robust and portable results.

VIII. Conclusion

This paper introduced T2T, a compilation framework for efficiently porting CUDA programs to diverse NPU architectures. By leveraging UPA and 2-D vectorization strategy, T2T effectively maps SIMT parallelism to NPU hardware. Our evaluation shows that T2T achieves up to 73% of native CUDA performance on a diverse set of benchmarks on various NPUs, validating a path to automatically migrate the vast CUDA ecosystem to emerging hardware.

IX. Acknowledgment

We thank all the reviewers for their valuable comments and suggestions. This work was supported in part by the National Key R&D Program of China, Grant No.2023YFB3001502, the Jiangsu Provincial Key R&D Program, Grant No.BG2024028 and the National Natural Science Foundation of China, Grant No.U23B2020, No.62090024, No.62302479, No.62232015.

References

[1] NVIDIA, "CUDA C++ programming guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[2] M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, and E. Lindahl, "GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers," *SoftwareX*, vol. 1, pp. 19–25, 2015, https://doi.org/10.1016/j.softx.2015.06.001.

[3] R. Salomon-Ferrer, A. W. Gotz, D. Poole, S. Le Grand, and R. C. Walker, "Routine microsecond molecular dynamics simulations with AMBER on GPUs. 2. explicit solvent particle mesh ewald," *Journal of chemical theory and computation*, vol. 9, no. 9, pp. 3878–3888, 2013, https://doi.org/10.1021/ct400314y.

[4] B. Acun, D. J. Hardy, L. V. Kale, K. Li, J. C. Phillips, and J. E. Stone, "Scalable molecular dynamics with NAMD on the summit system," *IBM journal of research and development*, vol. 62, no. 6, pp. 4–1, 2018, https://doi.org/10.1147/JRD.2018.2888986.

[5] R. Yokota and L. A. Barba, "Treecode and fast multipole method for N-body simulation with CUDA," in *GPU Computing Gems Emerald Edition*. Elsevier, 2011, pp. 113–132, https://doi.org/10.1016/B978-0-12-384988-5.00009-7.

[6] NVIDIA Corporation, "RAPIDS: Open GPU data science," 2018, https://rapids.ai/.

[7] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," in *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*, 2016, pp. 1–12, https://doi.org/10.1145/2688500.2688538.

[8] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl, "Pump up the volume: Processing large data on GPUs with fast interconnects," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1633–1649, https://doi.org/10.1145/3318464.3389705.

[9] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019, https://dl.acm.org/doi/10.5555/3454287.3455008.

[10] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "{TensorFlow}: a system for {Large-Scale} machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283, https://dl.acm.org/doi/10.5555/3026877.3026899.

[11] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12, https://doi.org/10.1145/3079856.3080246.

[12] H. Liao, J. Tu, J. Xia, H. Liu, X. Zhou, H. Yuan, and Y. Hu, "Ascend: a scalable and unified architecture for ubiquitous deep neural network computing: Industry track paper," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 789–801, https://doi.org/10.1109/HPCA51647.2021.00071.

[13] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, 2014, pp. 269–284, https://doi.org/10.1145/2654822.2541967.

[14] S. Knowles, "Graphcore," in *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE, 2021, pp. 1–25, https://doi.org/10.1109/HCS52781.2021.9567075.

[15] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54, https://doi.org/10.1109/IISWC.2009.5306797.

[16] R. T. Lange, A. Prasad, Q. Sun, M. Faldor, Y. Tang, and D. Ha, "The AI CUDA Engineer: Agentic CUDA kernel discovery, optimization and composition," Sakana AI, Tech. Rep., 2025.

[17] Cambricon, "Cambricon MLU," https://www.cambricon.com/.

[18] Huawei, "Ascend C," https://www.hiascend.com/cann/ascend-c.

[19] R. Karrenberg and S. Hack, "Whole-function vectorization," in *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 2011, pp. 141–150, https://doi.org/10.1109/CGO.2011.5764682.

[20] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 2–14, https://doi.org/10.1109/CGO51591.2021.9370308.

[21] T. Gysi, C. Müller, O. Zinenko, S. Herhut, E. Davis, T. Wicky, O. Fuhrer, T. Hoefler, and T. Grosser, "Domain-specific multi-level ir rewriting for gpu: The open earth compiler for gpu-accelerated climate simulation," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 4, pp. 1–23, 2021, https://doi.org/10.1145/3469030.

[22] N. Vasilache, O. Zinenko, A. J. Bik, M. Ravishankar, T. Raoux, A. Belyaev, M. Springer, T. Gysi, D. Caballero, S. Herhut *et al.*, "Composable and modular code generation in MLIR: A structured and retargetable approach to tensor compiler construction," *arXiv preprint arXiv:2202.03293*, 2022, https://doi.org/10.48550/arXiv.2202.03293.

[23] L. Chelini, A. Drebes, O. Zinenko, A. Cohen, N. Vasilache, T. Grosser, and H. Corporaal, "Progressive raising in multi-level ir," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 15–26, https://doi.org/10.1109/CGO51591.2021.9370332.

[24] M. Essadki, B. Michel, B. Maugars, O. Zinenko, N. Vasilache, and A. Cohen, "Code generation for in-place stencils," in *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, 2023, pp. 2–13, https://doi.org/10.1145/3579990.3580006.

[25] A. Brauckmann, L. Jaulmes, J. W. de Souza Magalhães, E. Polgreen, and M. F. O'Boyle, "Tensorize: Fast synthesis of tensor programs from legacy code using symbolic tracing, sketching and solving," in *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, 2025, pp. 15–30, https://doi.org/10.1145/3696443.3708956.

[26] W. S. Moses, I. R. Ivanov, J. Domke, T. Endo, J. Doerfert, and O. Zinenko, "High-performance gpu-to-cpu transpilation and optimization via high-level parallel constructs," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2023, pp. 119–134, https://doi.org/10.1145/3572848.3577475.

[27] R. Han, J. Lee, J. Sim, and H. Kim, "COX: Exposing CUDA warp-level functions to CPUs," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 4, pp. 1–25, 2022, https://doi.org/10.1145/3554736.

[28] W. S. Moses, L. Chelini, R. Zhao, and O. Zinenko, "Polygeist: Raising C to polyhedral MLIR," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2021, pp. 45–59, https://doi.org/10.1109/PACT52795.2021.00011.

[29] I. R. Ivanov, O. Zinenko, J. Domke, T. Endo, and W. S. Moses, "Retargeting and respecializing GPU workloads for performance portability," in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2024, pp. 119–132, https://doi.org/10.1109/CGO57630.2024.10444828.

[30] NVIDIA Corporation, "NVIDIA cuDNN: CUDA deep neural network library," 2024, https://developer.nvidia.com/cudnn.

[31] H. Sun, F. Fey, J. Zhao, and S. Gorlatch, "WCCV: Improving the vectorization of if-statements with warp-coherent conditions," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 319–329, https://doi.org/10.1145/3330345.3331059.

[32] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu, "MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2008, pp. 16–30, https://doi.org/10.1007/978-3-540-89740-8_2.

[33] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin, "MapCG: Writing parallel program portable between CPU and GPU," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, 2010, pp. 217–226, https://doi.org/10.1145/1854273.1854303.

[34] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2010, pp. 353–364, https://doi.org/10.1145/1854273.1854318.

[35] Z. Guo, E. Z. Zhang, and X. Shen, "Correctly treating synchronizations in compiling fine-grained spmd-threaded programs for cpu," in *2011 International Conference on Parallel Architectures and Compilation*

Techniques. IEEE, 2011, pp. 310–319, https://doi.org/10.1109/PACT.2011.62.

[36] Intel, "DPCT," 2021, https://software.intel.com/en-us/get-started-with-intel-dpcpp-compatibility-tool.

[37] M. Babej and P. Jääskeläinen, "Hipcl: Tool for porting cuda applications to advanced opencl platforms through hip," in *Proceedings of the International Workshop on OpenCL*, 2020, pp. 1–3, https://doi.org/10.1145/3388333.3388641.

[38] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W.-m. W. Hwu, "Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, 2010, pp. 111–119, https://doi.org/10.1145/1772954.1772971.

[39] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "Snucl: an OpenCL framework for heterogeneous CPU/GPU clusters," in *Proceedings of the 26th ACM international conference on Supercomputing*, 2012, pp. 341–352, https://doi.org/10.1145/2304576.2304623.

[40] A. S. Elhelw and S. Pai, "Horus: A modular GPU emulator framework," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2020, pp. 104–106, https://doi.org/10.1109/ISPASS48437.2020.00020.

[41] A. Patel, S. Tian, J. Doerfert, and B. Chapman, "A virtual GPU as developer-friendly OpenMP offload target," in *50th International Conference on Parallel Processing Workshop*, 2021, pp. 1–7, https://doi.org/10.1145/3458744.3473356.

[42] R. Han, B. Tine, J. Lee, J. Sim, and H. Kim, "Supporting CUDA for an extended RISC-V GPU architecture," *arXiv preprint arXiv:2109.00673*, 2021, https://doi.org/10.48550/arXiv.2109.00673.

[43] NVIDIA, "Compiling and executing CUDA programs in emulation mode," https://developer.nvidia.com/cuda-toolkit.

[44] AMD, "HIPIFY: Convert CUDA to portable C++ code," https://github.com/ROCm/HIPIFY.

[45] "CUDA on non-NVIDIA GPUs," https://github.com/vosen/ZLUDA.

[46] The Khronos Group, "SYCL - the C++ single-source heterogeneous programming for acceleration," 2025, https://www.khronos.org/sycl/.

[47] A. Singer, F. Gao, and K.-T. A. Wang, "Syclops: A sycl specific llvm to mlir converter," in *Proceedings of the 10th International Workshop on OpenCL*, 2022, pp. 1–8, https://doi.org/10.1145/3529538.3529992.

[48] W. Feng, S. Yao, K. T. Wang, M. A. Raihan, L. Feng, and C. Xu, "Extending SYCL's programming paradigm with tensor-based SIMD abstractions," in *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, 2022, pp. 59–66, https://doi.org/10.1145/3489525.3511681.

[49] W. Feng, R. Maghareh, and K.-T. A. Wang, "Extending DPC++ with support for Huawei ascend AI chipset," in *Proceedings of the 9th International Workshop on OpenCL*, 2021, pp. 1–4, https://doi.org/10.1145/3456669.3456684.

[50] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "pocl: A performance-portable OpenCL implementation," *International Journal of Parallel Programming*, vol. 43, no. 5, pp. 752–785, 2015, https://doi.org/10.1007/s10766-014-0320-y.

[51] M. Pharr and W. R. Mark, "ispc: A SPMD compiler for high-performance CPU programming," in *2012 Innovative Parallel Computing (InPar)*. IEEE, 2012, pp. 1–13, https://doi.org/10.1109/InPar.2012.6339601.

[52] R. Karrenberg and S. Hack, "Improving performance of OpenCL on CPUs," in *International Conference on Compiler Construction*. Springer, 2012, pp. 1–20, https://doi.org/10.1007/978-3-642-28652-0_1.

[53] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86, https://doi.org/10.1109/CGO.2004.1281665.

[54] S. Dong, Y. Wen, J. Bi, D. Huang, J. Guo, J. Xu, R. Xu, X. Song, Y. Hao, L. Li *et al.*, "QiMeng-Xpiler: Transcompiling tensor programs for deep learning systems with a neural-symbolic approach," in *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2025)*. USENIX Association, 2025, pp. 239–255, https://dl.acm.org/doi/10.5555/3767901.3767915.