BetterTogether: An Interference-Aware Framework for Fine-grained Software Pipelining on Heterogeneous SoCs

Yanwen Xu[1], Rithik Sharma[1], Zheyuan Chen[1], Shaan Mistry[1], Tyler Sorensen[2,1]
[1]*University of California, Santa Cruz*, Santa Cruz, CA, USA
[2]*Microsoft Research*, Redmond, WA, USA
{yxu83, riksharm, zchen406, sdmistry, tyler.sorensen}@ucsc.edu

***Abstract*—Data processing on the edge offers advantages over cloud-based solutions, including reduced latency and lower energy consumption. However, to fully utilize edge SoCs, applications must be efficiently mapped onto these devices' constituent heterogeneous processing units. This mapping is challenging, as modern SoCs incorporate diverse compute units, such as big.LITTLE architectures and GPUs with distinct performance characteristics. Furthermore, due to edge SoCs' integration and resource constraints, execution on one processing unit can interfere with the runtime of others, complicating the construction of modular and composable performance models.**

To address these challenges, we present BetterTogether, a flexible scheduling framework that enables fine-grained software pipelining on heterogeneous SoCs. Applications are provided as a sequence of stages, each with a CPU and GPU implementation. These stages can then be pipelined across the various processing units on the SoC. The novel component of BetterTogether is its ability to generate accurate and efficient pipeline schedules using a profile-guided performance model that captures execution time under representative intra-application interference. We demonstrate the portability of BetterTogether by evaluating it on three SoCs with GPUs from different vendors (NVIDIA, Arm, and Qualcomm) and using three computer vision edge workloads with different computational characteristics. Our performance model yields predictions that correlate strongly with measured results. Using these models, we construct efficient pipeline schedules specialized to each workload-platform combination, outperforming homogeneous GPU baselines in nearly all cases, with a geomean speedup of 2.14x and a maximum of 7.59x. Finally, we present a preliminary exploration with Google's EdgeTPU, highlighting BetterTogether's extensibility to incorporate an even wider variety of processing units into its backend.

***Index Terms*—edge computing, heterogeneous computing, graphics processing units, multiprocessing systems, scheduling algorithms, performance modeling**

I. Introduction

While large-scale data processing is typically performed in the cloud on powerful GPU clusters, modern mobile and edge devices are becoming increasingly capable, incorporating high-performance SoC processors. Meanwhile, new applications and models are being designed for these resource-constrained devices, reducing memory and compute demands while maintaining sufficient fidelity. Edge computing offers several benefits, including lower latency [18], [30], reduced energy consumption [13], and availability even without internet connectivity. As a result, edge processing is already seeing widespread adoption and is expected to grow rapidly in both deployment and demand [21].

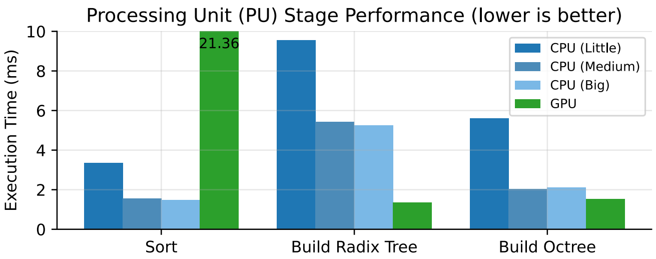


Fig. 1. The execution time of three pipeline stages on different processing units (PUs) on a Google Pixel shows significant variation. E.g., sorting is faster on big/medium CPU cores, while building a radix tree is faster on the GPU. This heterogeneity highlights the need to map stages to PUs to utilize compute resources effectively.

Realizing the benefits of edge processing requires overcoming challenges, particularly in efficiently utilizing the underlying hardware. These devices typically integrate a range of *Processing Units* (PUs), including heterogeneous CPU cores (e.g., in big.LITTLE architectures), GPUs, and AI accelerators (e.g., TPU). Each PU has different computational characteristics that impact the efficiency with which it executes different tasks. For example, Fig. 1 illustrates the execution time of three tasks on four types of PUs within a Google Pixel 7a. This device features eight CPU cores across three tiers: 2 Cortex-X1 (big), 2 Cortex-A78 (medium), and 4 Cortex-A55 (little), along with an Arm Mali-G710 GPU. The evaluated tasks are a subset of the stages from a 3D octree construction pipeline commonly used in computer vision and robotics (discussed in Sec. IV-A) [25]. The results reveal significant performance differences: for the sorting task, the GPU performs poorly; for building the radix tree, the GPU is fastest; and for octree construction, the big and medium CPU cores perform comparably to the GPU.

a) Heterogeneous Parallelism: To efficiently process the application stages shown in Fig. 1, one might consider a

This material is based upon work supported by the National Science Foundation under Award No. 2239400. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

data-parallel approach, assigning different amounts of data to each PU based on its relative efficiency [8], [41]. However, this strategy remains suboptimal, as it forces PUs to perform poorly suited tasks; e.g., the GPU would still handle some portion of sorting despite being inefficient. Fortunately, many edge applications have two properties that can be exploited for efficient execution: (1) they can be decomposed into stages, and (2) they process a *streaming* input, i.e., where small, independent inputs, such as frames, arrive continuously over time. Given these properties, edge platforms and applications are well-suited to *pipeline parallelism*, where stages can be mapped to the PU that provides the most efficient computation, e.g., considering Fig. 1, the GPU can build the radix tree while the CPU performs sorting. The octree construction stage can be assigned to CPU or GPU, providing flexibility in the pipeline generation.

However, a given *pipeline schedule* (i.e., a mapping from stages to PUs) is not portable across devices. The landscape of edge hardware is highly diverse; for instance, in contrast to the Google Pixel described earlier, the NVIDIA Jetson Nano features a larger GPU but lacks heterogeneous CPU cores. Because of this, the optimal pipeline schedule for the Nano will differ from the optimal schedule for the Pixel. Therefore, a pipelining framework must be capable of creating schedules for a wide range of devices.

b) Performance Models: Developing effective pipeline schedulers requires accurate performance models for how each application stage performs across different PUs. Prior work often builds performance models using isolated benchmarks, i.e., profiling each PU independently and then composing the results to predict system-wide behavior [5], [6]. While this modular approach is convenient, it breaks down on edge SoCs, where performance on one PU can be affected by activity on others [27]. As a result, models built in isolation often fail to capture intra-application interference, leading to inaccurate performance predictions [22].

To illustrate these difficulties, we evaluated an edge image classification task: a pruned (sparsified) version of AlexNet [28], [35] where each DNN layer is a pipeline stage. Following prior approaches, we build isolated performance models for each type of PU (e.g., big, medium, little CPUs, and GPU) on the Google Pixel. While the model predicted an optimal pipeline with a latency of 4.95 ms, the actual measured latency was 7.77 ms, nearly 57% slower than predicted. Our observations align with prior work, which has reported discrepancies of up to 60% between predicted and actual latency on Android devices [22]. Extending this analysis across multiple schedules, we observed low correlation between predicted and observed latencies (discussed in Sec. V). These findings highlight the challenge of isolated performance models on edge devices, where system variability can undermine their accuracy when composed.

A. BetterTogether

We present *BetterTogether*, a framework for software pipelining on heterogeneous SoCs. Its key contribution is a performance modeling technique that provides accurate schedule prediction on a variety of edge devices, allowing *BetterTogether* to generate static pipeline schedules that outperform homogeneous implementations (both CPU and GPU).

The *BetterTogether* framework consists of two core components: a performance profiler and a pipeline optimizer, both integrated into an automated C++ implementation that enables end-to-end execution.

a) BetterTogether Performance Profiler (BT-Profiler): To address inaccurate performance models for edge devices, we introduce the *BT-Profiler*: a profiling method that captures intra-application interference. Like traditional models, *BT-Profiler* profiles each stage on each PU and builds a profiling table. However, to better reflect system conditions during application execution, it adds a controlled background load by assigning other application stages to the other PUs.

This background load is critical for performance prediction accuracy, as we observed execution time differences of up to $2.25\times$ for certain stages when profiled with and without background load on the Google Pixel.

b) BetterTogether Pipeline Optimizer (BT-Optimizer): Using the profiling table from the previous step, *BT-Optimizer* formulates the schedule optimization problem as a set of linear constraints and solves it using an off-the-shelf solver. The result is a pipeline schedule that maps application stages to specific PUs on the target device. When generating pipeline schedules, *BT-Optimizer* incorporates an additional constraint: it not only seeks to minimize latency but also filters out schedules that underutilize the device. These schedules yield better performance and preserve the conditions under which the performance models were constructed.

c) BetterTogether Implementer (BT-Implementer): Finally, the *BT-Implementer* executes the pipeline on the target system, managing concurrent execution and synchronization between PUs. Since *BetterTogether* targets shared-memory SoCs, *BT-Implementer* does not perform explicit memory transfers. Instead, PUs communicate through shared main memory using concurrent queues. Beyond producing the final application deployment, *BT-Implementer* is a rigorous empirical tool for exploring and evaluating pipeline schedules across diverse edge platforms.

d) Evaluation: We evaluate *BetterTogether* on three computer vision applications with varying computational characteristics: a dense DNN, a sparse DNN, and a 3D octree pipeline. To highlight the portability of *BetterTogether*, our evaluation systems span three widely deployed edge SoCs across three vendors: a Google Pixel 7a (Arm-based), OnePlus 11 (Qualcomm), and NVIDIA Jetson Nano.

We evaluate the accuracy of our performance models, showing that the *BetterTogether* performance modeling is strongly correlated with actual execution time and, thus, can effectively guide the construction of efficient schedules. In contrast, prior performance modeling techniques show lower correlation with actual execution time, especially for the ARM and NVIDIA SoCs for sparse applications. Using these accurate and efficient schedules, we show that *BetterTogether* produces efficient static schedules, with average speedups of 2.14× over GPU-only and 7.59× over CPU-only implementations. Beyond CPUs and GPUs, we further demonstrate the extensibility of *BetterTogether* by incorporating a mobile AI accelerator (Google's EdgeTPU). Although preliminary, this exploration highlights the flexibility of *BetterTogether* and the potential benefits of leveraging emerging AI accelerators.

Contributions: We propose *BetterTogether*:

- A performance modeling approach that accounts for intra-application interference and accurately predicts the runtime of pipeline schedules (Sec. III-C).
- A static pipeline generator for heterogeneous edge SoCs that can efficiently implement pipeline schedules across a variety of devices (Sec. III-E).
- An evaluation showing that *BetterTogether* is effective across different edge devices and applications, all with differing characteristics, with a geomean speedup of 2.14× over homogeneous pipelines (Sec. V).

Our implementation of *BetterTogether* is open source and available at https://github.com/ucsc-redwood/better-together.

II. Background

A. Heterogeneous System-on-Chips (SoCs)

Heterogeneous SoCs integrate CPUs, GPUs, and accelerators on a single silicon die and are connected to a single pool of DRAM. These SoCs frequently adopt a *Unified Memory Architectures* (UMAs), where all PUs share a common pool of DRAM through a centralized memory controller and a unified physical address space [19], [39], [51]. Some, like NVIDIA Jetson [32], also support shared last-level caches between CPU and GPU cores. This contrasts with larger heterogeneous systems found in the data center systems, which use discrete GPUs (dGPUs) with dedicated VRAM connected via PCIe or NVLink. Mobile SoCs commonly feature heterogeneous CPU designs, such as big. LITTLE architecture, combining high-performance *big* cores with energy-efficient *little* cores. Some devices, such as the Google Pixel (Sec. I), also include *medium* cores for balanced performance. The OS dynamically schedules tasks across these cores to optimize power and performance.

a) Big cores (CPU): are designed for high single-threaded performance, often serving as the primary processor for latency-sensitive tasks. They feature out-of-order execution, superscalar pipelines, large L2 caches (256–1024 KiB for Cortex-X1), and high clock speeds (up to 3.2 GHz for Cortex-X3), along with advanced branch prediction and prefetching. Given these characteristics, they are ideal for tasks with unbalanced parallelism or irregular memory access, such as sparse linear algebra and graph traversal.

b) Little cores (CPU): prioritize energy efficiency with simpler in-order pipelines, smaller L2 caches (256–512 KiB), and lower clock speeds (1.7–2.0 GHz). While they have lower IPC, their power efficiency makes them well-suited for lightweight tasks (e.g., max-pooling in CNNs), though less effective for compute- or memory-intensive tasks.

c) Graphics Processing Units (GPU): Our work focuses on integrated GPUs (iGPU) on edge platforms, which operate within a UMA and thus, enable efficient, fine-grained collaboration with other PUs. Architecturally, GPUs consist of multiple *Streaming Multiprocessors* (SMs), each with many lightweight SIMT cores that execute threads in lockstep groups (e.g., warps of size 32 in NVIDIA architectures, but varying sizes in other GPUs). This execution model excels on regular workloads with uniform control flow and coalesced memory accesses, such as dense linear algebra. However, their performance suffers from thread divergence or non-coalesced access, making GPUs less effective for sparse or irregular tasks like graph traversals.

d) AI Accelerators: Modern SoCs also integrate domain-specific processing units optimized for machine learning workloads. Unlike the general SIMT execution model used by GPUs, these accelerators adopt architectures specialized for tensor operations on dense linear algebra kernels, such as *General Matrix–Matrix Multiplication* (GEMM) and convolution. They often natively support reduced-precision arithmetic (e.g., INT8, BF16, FP16) for improved efficiency. E.g., Google's Tensor Processing Unit (TPU) [29], Apple's Neural Engine, and other vendor-specific accelerators. Compared to CPUs and GPUs, AI accelerators are less flexible but deliver significantly higher performance-per-watt on GEMM and convolution machine learning tasks [46].

B. Heterogeneous Programming Models

Heterogeneous SoCs are typically programmed using a host-device model, where the CPU (host) coordinates execution on the GPU/Accelerator (device). A *kernel* is a small, self-contained unit of computation. To exploit many-core systems, developers use *data parallelism*, executing the same kernel across multiple data subsets simultaneously.

a) Host-side: Host-side parallelism typically involves spawning multiple threads, each processing a data subset in parallel. Typically, the OS dynamically assigns threads to cores; however, developers can improve performance stability by pinning threads to specific cores. On Linux, this can be done using interfaces such as `sched_setaffinity()` or `pthread_setaffinity_np()`. Android inherits these Linux kernel features and extends them with `cpuset` and `cgroup` mechanisms. However, not all platforms support explicit thread affinity, and some older kernels or embedded systems may restrict certain core pinning. E.g., macOS uses Apple's heterogeneous M-series chips but lacks APIs to assign threads to specific cores, which limits fine-grained scheduling.

b) Device-side: On the GPU device side, parallel computations are executed using *General-Purpose GPU* (GPGPU) frameworks such as CUDA [43] and Vulkan Compute [49]. These systems provide APIs for device management and rely on compute languages like CUDA/C++ or SPIR-V to program thread-level parallelism under a hierarchical *Single Instruction, Multiple Threads* (SIMT) model. A GPU kernel (written in a GPU programming language) is launched across a large number of lightweight threads organized into *thread blocks* (in

CUDA) or *workgroups* (in Vulkan). These groups can execute cooperatively, sharing local memory and allowing for efficient built-in synchronization. This hierarchy maps naturally onto the GPU hardware, which executes threads in the same group on the same SM. The host launches device kernels asynchronously using *streams* (for CUDA) or *Command Queues* in Vulkan. This enables overlapping execution between CPU and GPU tasks.

On the accelerator device side, each vendor typically provides specific APIs, but it is often a declarative, graph-based programming model. An example for mobile SoCs is the Android *Neural Networks API* (NNAPI) [47], which is a C API designed for executing ML tensor operations. With NNAPI, applications construct a computation graph composed of predefined tensor operators (e.g., matrix multiplications and convolutions) and their operands. The API implementation then ships the computation to its accelerators. The developer is still responsible for binding input/output memory buffers and for invoking and synchronizing execution. The predefined operator set heavily relies on vendor-specific system, including both hardware and software availability.

III. *BetterTogether* FRAMEWORK

We now present *BetterTogether*, a framework for software pipelining on heterogeneous SoCs. *BetterTogether* provides accurate performance models on edge devices by emulating realistic intra-application background loads and thus, can accurately provide efficient pipeline schedules across a variety of applications and devices.

A. BetterTogether Overview

BetterTogether operates on streaming applications that consist of a sequence of smaller computational tasks, referred to as *compute kernels* (see Sec. III-B; e.g., sorting, prefix sum, convolution). Thus, *BetterTogether* can create software pipelines where kernels are mapped to different PUs on the device. Figure 2 shows an overview of *BetterTogether*, organized into three stages: *Inputs* (1–2), *Schedule Generation* (3–4), and *Pipeline Execution* (5).

1) **Host/Device Code**: The *BetterTogether* input requires kernel implementations on both the host and the device side (e.g., as in Fig. 4) for the application.
2) **Target System Specification**: The *BetterTogether* input also requires PU (e.g., CPU and GPU) information for the target device, including an *affinity map* of threads to CPU type (e.g., big or little).
3) **Performance Profiler (*BT-Profiler*)**: Using the inputs, the *BT-Profiler* profiles kernels across the device PUs. It constructs a 2D *profiling table* of execution times, where the rows are the kernels and the columns are the PUs. To improve accuracy, each kernel is profiled within a simulated pipeline environment, where other PUs concurrently execute kernels from the same application. This setup mimics realistic intra-application interference, and thus, more accurate performance models can be built from the profiling table.
4) ***BetterTogether* Optimizer (*BT-Optimizer*)**: Using the profiling table, the *BT-Optimizer* formulates and solves constraints to minimize latency while maximizing system utilization. This produces candidate *schedule*, i.e., a mapping of stages to PUs.
5) **Pipeline Implementer (*BT-Implementer*)**: The *BT-Implementer* takes the schedule and generates a pipelined C++ implementation that spawns threads, manages concurrent queues, and dispatches kernels to the appropriate PUs. To account for small modeling inaccuracies, the *BT-Implementer* can be used to execute and evaluate several of the top schedules to select the best-performing schedule.

The result is an efficient software pipeline tailored to the target system, minimizing latency while maintaining high system utilization. The entire process in components (3–5) is fully automated: *BetterTogether* delivers end-to-end optimization by constructing efficient pipeline schedules from user-provided input code, with minimal human intervention.

B. BetterTogether Terminology

We now describe the terminology used in this paper. A *Stage* represents a unit of computation and has a well-defined input and output specification. A *Compute Kernel* (kernel) implements a stage. Kernels are implemented across multiple backends (OpenMP, CUDA, Vulkan) to support cross-platform execution (e.g., in Fig. 3). Example codes are shown in Fig. 4, where `morton32` is a function that computes the Morton encoding of a given 3D point cloud data. Because we target UMA systems, kernels operate directly on raw pointers to inputs, outputs, and constants. A *Chunk* refers to one or more contiguous stages and serves as the basic unit of scheduling. An *Application* is a sequence of chunks, where the output of one chunk serves as the input to the next.

At the system level: The *UsmBuffer* is a unified memory buffer allocated in the shared DRAM of the SoC. The UsmBuffer provides pointers in both host-side code and device-side code. We implemented this using C++ `std::pmr::vector` as the front-end and supplied it with backend-specific allocators, such as `cudaMallocManaged` in CUDA, or `VkBuffer` in Vulkan. This design creates a unified view of data for both the host and device and ensures zero-copy memory access, eliminating data movement overhead.

C. The BetterTogether Performance Profiler

A limitation of prior works is that they failed to capture performance accurately when the system is fully occupied, leading to unreliable predictions, especially for edge devices. Our experiments showed that the little core on Android devices can degrade performance by $\approx 1.38\times$ when the system is fully utilized. Surprisingly, GPU can achieve $\approx 2\times$ speedup under heavy load[1]. These inaccuracies propagate to the optimization stage, resulting in suboptimal schedules. To address this problem, we present the *BT-Profiler*, which incorporates interference-aware profiling by executing each stage under

[1] We hypothesize the speedup is due to GPU frequency boosting under heavy CPU load by vendor-specific firmware. See Sec. V-C for details.

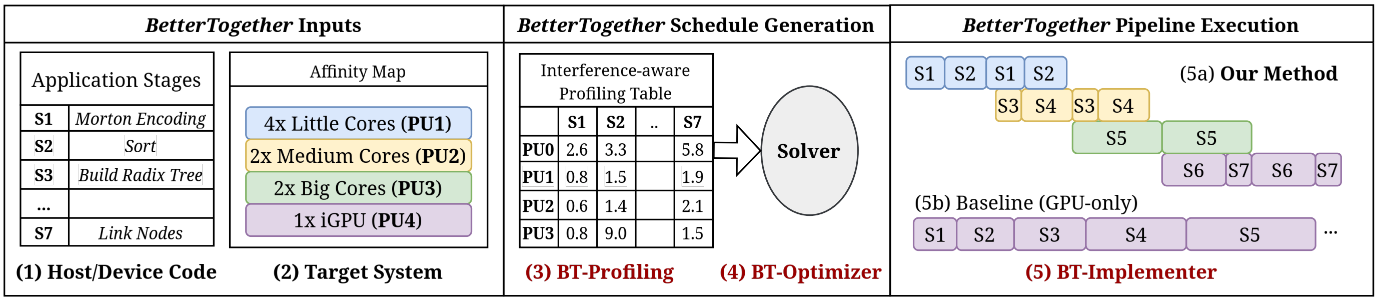
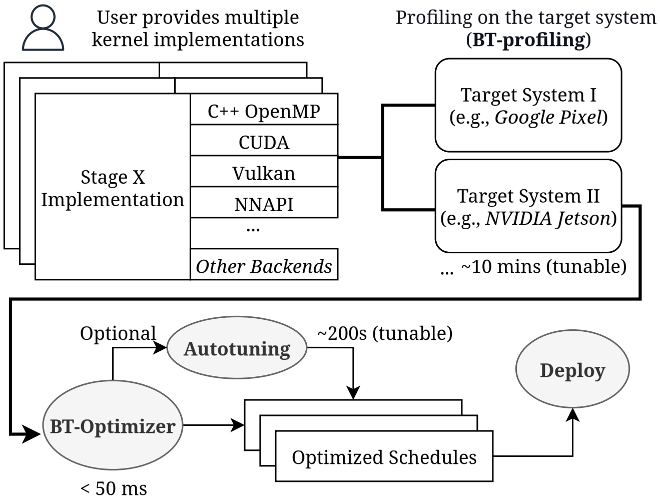


Fig. 2. *BetterTogether* overview. (1) The input application is provided as a sequence of stages. (2) The target system describes multiple types of PUs. (3) An interference-aware profiling table is constructed. (4) A multi-stage solver generates efficient pipeline schedules that meet a utilization threshold. (5) *BetterTogether* efficiently executes the schedule.



User provides multiple kernel implementations
Profiling on the target system (BT-profiling)
Stage X Implementation
C++ OpenMP
CUDA
Vulkan
NNAPI
...
Other Backends
Target System I (e.g., *Google Pixel*)
Target System II (e.g., *NVIDIA Jetson*)
... ~10 mins (tunable)
Optional
Autotuning
~200s (tunable)
Deploy
BT-Optimizer
Optimized Schedules
< 50 ms

Fig. 3. End-to-end workflow diagram of *BetterTogether*.

two execution modes: (1) *isolated*, where the stage runs alone on its assigned PU without external interference, and (2) *interference-heavy*, where synthetic stages' computations are co-scheduled on other PUs to simulate intra-application interference. Specifically, we introduce controlled interference by having all other PUs run the same computation as the measuring PU during profiling. During this period, we record only the throughput processed by the measuring PU. This simulation ensures that the measured performance accounts for contention and synchronization overhead commonly encountered in real pipeline processing.

Finally, the *BT-Profiler* characterizes the performance of each pipeline stage on different PUs using a black-box profiling methodology. We directly measure the observable execution time and throughput of each stage and do not use any source-level inspection or micro-architectural modeling. Latency is measured using high-resolution hardware timers, utilizing the *ARM Generic Timer Framework* on the host, and CUDA event pairs or Vulkan timestamp queries on the device. Throughput is measured by counting how many times the stage completes execution within a fixed time duration. Each measurement is executed 30 times, and the mean execution time is recorded to reduce measurement noise. The resulting latency values are aggregated into a *profiling table*; this table has a row for every stage s and a column for every PU p. Entry s, p denotes the average latency of stage s when executed on PU type p. In this work, collecting a single profiling table takes on average $\approx$ 6 minutes per device per application.

D. The BetterTogether Optimizer

Given a profiling table, the next task is to produce an efficient pipeline schedule. This search must efficiently navigate a combinatorially large space. E.g., given a 9-stage AlexNet pipeline running on a Google Pixel with four little, two medium, two big cores, and one GPU, there are $4^9 \approx 262K$ possible schedule instances. To navigate this space, we propose a three-step optimization approach: first, using linear programming to optimize for a utilization threshold; then, a manual search to minimize latency; and then an optional autotuning step that executes a small number of pipelines on the actual device to account for small modeling inaccuracies. Our optimization routine requires two inputs: an *affinity map*, including PU types, the list of available processors per type, and their core IDs; and the *interference-aware* profiling table generated from *BT-Profiler*.

a) Step One: Optimizing Utilization: Similar to prior work [17], we first use linear programming constraints; however, rather than minimizing pipeline latency, we first filter pipeline schedules that maximize utilization. To do this, we introduce a new objective called *Gapness*, defined as the difference between the execution time of the longest and shortest chunk. The key insight is that optimizing for gapness yields schedules whose predicted runtimes more accurately reflect the conditions under which interference-aware profiling data were collected. We implement our constraints in the SMT solver, which includes support for minimizing and maximizing constraints. The formulation is as follows:

b) Step Two: Optimizing Latency: The previous optimization produces a single schedule that minimizes gapness and thereby maximizes utilization. However, high utilization does not always translate to low latency, as stages may be assigned to PUs that compute inefficiently for certain tasks (e.g., assigning sorting to the GPU, as illustrated in Fig. 1). To address this limitation, we perform a second optimization

Listing (1) CPU kernel (OpenMP)

```
void run_stage_1_cpu(in, out, N) {
  #pragma omp parallel for
  for (int i = 0; i < N; ++i)
    out[i] = morton32(in[i]);
}
```

Listing (2) GPU kernel (CUDA)

```
__global__ void run_stage_1_gpu(in, out, N) {
  int idx = threadIdx.x + blockDim.x * blockIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = idx; i < N; i += stride)
    out[i] = morton32(in[i]);
}
```

Fig. 4. Parallel Morton-key stage implementation in the Octree application. The CPU version uses `OpenMP` for work distribution across all cores. The GPU version maps iterations across threads in `CUDA`. The `Vulkan` version follows a similar logic using `gl_LocalInvocationID`, `gl_WorkGroupSize`, and `gl_NumWorkGroups` (implemented in OpenCL).

that explicitly targets latency. Instead of generating just one schedule, we produce a set of diverse candidates ($\mathcal{K} = 20$ in this work), each with a different assignment of stages to PUs. To ensure diversity, we iteratively block previously found solutions by adding a constraint that prevents the solver from returning the same schedule:

$$\sum_{i \in \mathcal{N}} x_{i,\sigma_i^{(\ell)}} \;\le\; |\mathcal{N}| - 1. \tag{C5$_\ell$}$$

Here, $\sigma^{(\ell)}$ denotes the ℓ-th schedule returned by the solver, where $\sigma_i^{(\ell)} \in \mathcal{C}$ indicates the PU assigned to stage i. We can then straightforwardly sort the candidate schedules by latency (i.e., by $T_{\max}$) to obtain schedules that have both low latency and high utilization.

Interestingly, we found that the resulting top-ranked schedules often show contiguous groups of similar performance according to the model. This is likely because they preserve the same *critical* assignments, and mapped the most expensive stages to high-performance PUs, while varying only the placement of cheaper stages across less capable PUs. As a result, schedules tend to cluster into *performance tiers*. E.g., the top five schedules may all have similar latencies (e.g., around 11.2 ms, within $\pm 6\%$). In comparison, the next group of ten schedules drops significantly, forming a second tier with latencies around 17 ms ($\pm 5\%$).

c) Optional Step Three: Autotuning: Given the sorted schedules from the previous stage, we now perform the final stage, which consists of running concrete schedules on the actual device. While overall, we find a high correlation between our performance estimates and actual execution time (see Sec. V-B), there may still be small inaccuracies, especially within the same performance tier. Since each evaluation is brief and only a few pipelines are executed, this step completes quickly relative to the cost of the initial optimization. By measuring the actual latency of the top schedules, the best-performing schedule is selected.

d) Notation and Decision Variables:

N	Total number of pipeline stages
N_i	The pipeline stage i, with $i \in \mathcal{N} = \{0, \ldots, N{-}1\}$
$\mathcal{C}$	PU classes: $\mathcal{C} = \{c_1, \ldots, c_M\}$
$t_{i,c}$	Profiled latency of stage i on PU c
$x_{i,c}$	Decision variable: $x_{i,c} \in \{0, 1\}$; $x_{i,c} = 1 \Leftrightarrow$ stage i runs on PU c
$T_{\min}, T_{\max}$	Lower and upper bounds of chunk runtimes

e) Constraints 1: Exactly one PU per stage:

$$\forall i \in \mathcal{N}: \quad \sum_{c \in \mathcal{C}} x_{i,c} = 1 \tag{C1}$$

f) Constraints 2: Contiguity; let $i < j < k$. Stages mapped to the same PU must form a single chunk:

$$(x_{i,c} \wedge x_{k,c}) \;\Rightarrow\; x_{j,c} \tag{C2}$$

g) Constraints 3a, 3b: Per-Chunk Runtime Bounds: Let $y_{c,i,j} \;=\; \bigwedge_{k=i}^{j} x_{k,c}$, which evaluates to 1 if and only if all stages from index i to j are assigned to PU c. That is, if $[i, j]$ forms a maximal chunk on PU c. For every such chunk, we impose upper and lower bounds on its total runtime:

$$y_{c,i,j} \Rightarrow \sum_{k=i}^{j} t_{k,c} \;\le\; T_{\max} \tag{C3a}$$

$$y_{c,i,j} \Rightarrow \sum_{k=i}^{j} t_{k,c} \;\ge\; T_{\min} \tag{C3b}$$

h) Objective: We can now optimize for device utilization by minimizing *Gapness*, i.e., the difference between the longest executing chunk and the shortest executing chunk.

$$\min(T_{\max} - T_{\min}) \tag{O1}$$

i) Solver Implementation: We encode constraints (C1)–(C5$_\ell$) and objective (O1) to *z3* via its Python API [15]. For our Pixel 7a case study (with $N = 9$ stages and $M = 4$ PUs), each solver invocation completes in <50 ms on a commodity laptop. During the autotuning phase, we evaluate a set of $\mathcal{K} = 20$ candidate schedules produced by the solver. Each candidate is executed for a fixed interval of 10 seconds to measure its throughput, and the best one is then selected.

E. BetterTogether Pipeline Implementer

We now describe how the static pipeline schedules generated by *BT-Optimizer* are implemented and executed on the target system. We describe how the application data is encapsulated (in a *TaskObject*) and how the PU kernels are launched and managed (*dispatcher threads*), describing both the CPU and GPU cases.

a) TaskObject: A *TaskObject* is a container that holds all memory buffers and metadata required to execute an application from the first to the final stage. It primarily consists of *UsmBuffers* (defined in Sec. III-B), which include both persistent data and intermediate scratchpad memory used by individual stages. A *TaskObject* may also contain constants or scalar parameters, e.g., input dimensions. In addition to unified buffers, a *TaskObject* can also include host- and device-only memory, depending on stage requirements. For example, a GPU radix sort kernel may contain device-local memory for temporary histograms that are not needed by the host kernels. To avoid memory allocation overhead during execution, we pre-allocate scratchpad regions.

b) Dispatcher Threads: The *BT-Implementer* uses a multi-buffering approach by allocating multiple *TaskObject*s to enable overlapping execution of pipeline stages. It executes pipeline schedules using a fixed number of long-lived *dispatcher threads*, one for each pipeline chunk. These dispatcher threads communicate through lightweight, lock-free *single-producer, single-consumer (SPSC) queues*, which pass pointers to *TaskObject*s between pipeline chunks.

Recall that each chunk consists of one or more stages. Each dispatcher thread repeatedly performs the following:

1) Pop a *TaskObject* pointer from the previous queue.
2) Synchronize all memory buffers required by this chunk to be up-to-date and visible to the designated PU.
3) Dispatches the *compute kernel* for each stage within that chunk in sequence.
4) Yield until all dispatched compute kernels in the chunk have completed execution.
5) Push the pointer of the *TaskObject* to the next queue.

The dispatcher thread itself does not participate in the computation. This setup allows multiple chunks to be executed concurrently, enabling efficient overlapping of the CPUs/GPU computations. Once all chunks have processed a *TaskObject*, it is reset and pushed back to the first queue, effectively recycling it for subsequent use.

c) CPU Executions: For a CPU chunk (e.g., big or little cores), the dispatcher thread sequentially invokes the OpenMP compute kernels for each stage in that chunk. To ensure each chunk runs on the desired CPU core type, the dispatcher passes affinity information to the worker thread, which then calls `sched_setaffinity()` to bind itself to the specified core. Because OpenMP includes implicit synchronization, the dispatcher thread yields after invoking the compute kernels, allowing the OpenMP worker threads to execute the stage. This behavior prevents the dispatcher from occupying CPU resources during compute kernel execution. Moreover, OpenMP internally uses a thread pool, which helps avoid the overhead of frequent thread creation and teardown.

d) GPU Executions: On CUDA-enabled platforms, the dispatcher thread first issues prefetching hints for all the UsmBuffers associated with this chunk by calling `cudaStreamAttachMemAsync()`, indicating that the GPU will access the data. The driver can use these prefetching hints to optimize the coherence operations. Next, the dispatcher sequentially submits each compute kernel to the CUDA stream associated with the chunk using `cuLaunchKernel()`. These kernel launches are non-blocking; execution proceeds asynchronously on the GPU stream. Finally, we use `cudaStreamSynchronize()` to ensure all memory and compute operations in the stream complete before proceeding.

Similarly, on Vulkan platforms, `VkFence` objects are used to track kernel completion and enforce execution order. Through `vkCmdPipelineBarrier` [48], Vulkan enables the user to precisely define when a memory operation (e.g., CPU write or GPU read) becomes visible. These memory regions are then recorded into a `VkCommandBuffer`. Each compute kernel (implemented as shaders for Vulkan) is recorded sequentially into the command buffer. After recording, the dispatcher submits the command buffer to a dedicated `VkQueue` associated with the chunk, along with a `VkFence` object to track completion.

IV. Experimental Setup

We evaluated both performance model accuracy and pipeline execution performance across four heterogeneous edge platforms: two distinct mobile SoCs and an embedded SoC evaluated in both normal and low-power modes, treated as separate devices. All host-side code, including OpenMP and Vulkan dispatch logic, is compiled using `clang`. Device-side kernel code is compiled separately per backend: CUDA kernels are compiled with `nvcc`. Vulkan compute shaders are written in GLSL and compiled offline to SPIR-V (v1.3) using `glslc`, targeting Vulkan 1.3.251.

We use a custom profiling framework built on top of *Google Benchmark* [20], extended to support multi-stage pipelines and configurable hardware affinity. Each run consists of 30 tasks, and we report the total end-to-end latency across all stages. For accurate intra-stage measurements, each pipeline stage is instrumented with a cycle-accurate logger that reads the `cntvct_el0` register on ARM64. On Jetson platforms, we use NVIDIA Nsight Systems to verify core utilization and confirm that execution respects affinity and PU mappings. All reported results are measured after warmup, excluding GPU initialization overhead.

A. Evaluated Applications

We evaluated three representative workloads with distinct computational patterns: (1) *AlexNet-dense*, (2) *AlexNet-sparse*, and (3) *Octree*; each representing regular, irregular, and mixed computational patterns, respectively. These workloads are commonly used in computer vision tasks, particularly in resource-constrained environments, due to their modest computational and memory demands. Table I summarizes the key characteristics of the evaluated applications, highlighting the diversity in input types and computational patterns across dense and sparse models.

a) AlexNet-dense: AlexNet-dense performed image classification using a standard dense *Convolutional Neural Network* (CNN) architecture. It represented regular, structured computation dominated by dense linear algebra operations, primarily convolutions. The network consisted of nine sequential stages: four convolutional layers, each followed by a max-pooling operation, and a final fully connected (linear) layer. The convolutional stages dominated the computation, while the final linear layer was relatively lightweight. We used the standard CIFAR-10 dataset [34], with AlexNet-dense processing one image per task.

b) AlexNet-sparse: AlexNet-sparse shared the same network structure as the dense variant but applied structured pruning to reduce model size and computation. We used Condensa [28] to prune the convolutional layers, resulting in weight tensors stored in *Compressed Sparse Row* (CSR) format. This introduced irregular memory access patterns and sparse computation, making AlexNet-sparse representative of workloads with irregular sparsity. We also used the standard CIFAR-10 dataset; however, since the sparse variant has a significantly lower per-image compute cost, we process batches of 128 images per task, as shown in Tab. tab:alexnet-layout.

c) Octree: Octree (e.g., Octomap [25]) is a 3D vision workload that constructs spatial hierarchies from point cloud inputs, commonly used in applications such as 3D reconstruction and scene representation. We followed the implementation by Karras et al. [31], which consists of seven stages of various computing patterns: The pipeline begins with (1) *Morton Encoding*, which converts 3D coordinates into 32-bit Morton codes, followed by (2) *Sort*, which performs a radix sort on the encoded keys. (3) *Duplicate Removal* eliminates redundant Morton codes to ensure unique spatial entries. Next, (4) *Build Radix Tree* constructs a hierarchical binary tree structure over the sorted keys. (5) *Edge Counting* traverses the tree to count outgoing edges per node. (6) *Prefix Sum* computes child offsets from edge counts, and (7) *Build Octree* allocates and writes the final octree to memory. Some stages, such as *Morton Encoding*, are highly regular and parallelizable using simple `DOALL` loops. Others, like *Sort* and *Prefix Sum*, are parallelizable but nontrivial to implement efficiently on GPUs. Later stages like *Build Radix Tree* and *Edge Counting* involve irregular control flow and are especially difficult to map to GPUs, which lack support for dynamic memory allocation and pointer chasing.

B. Evaluated Platforms

Table III summarizes key architectural specifications of the evaluated platforms, highlighting the diversity in CPU and GPU across mobile and embedded domains.

TABLE I
CHARACTERISTICS OF EVALUATED APPLICATIONS. PC STANDS FOR POINT CLOUD IN THE OCTREE APPLICATION

Application	**Input**	**Stages**	**Characteristics**
AlexNet-Dense	Image	9	Dense Linear Algebra
AlexNet-Sparse	Image	9	Sparse Linear Algebra
Octree	PC	7	Mixed Sparse & Dense

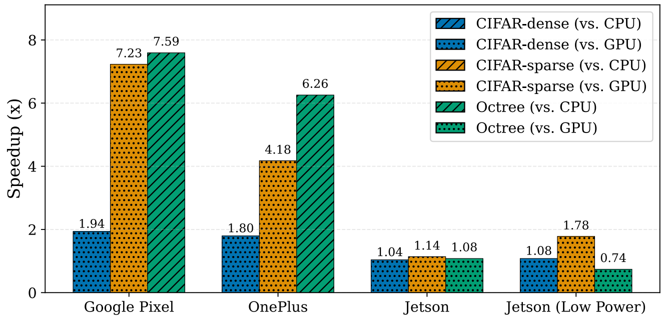


Fig. 5. Speedup of *BetterTogether* over the best CPU or GPU baseline for each task and device (see Tab. IV).

a) Android Platforms: We evaluate two Android devices, covering a range of CPU/GPU configurations. The code is compiled with the *Android NDK r29.0.13113456*, targeting 64-bit ARM architectures. The Google Pixel runs *Linux 6.1.99*, and the OnePlus device runs *Linux 5.15.149*. For Google's TPU, we used *NNAPI* from the *Android NDK* by linking with the *-lneuralnetworks* flag.

b) Jetson Platform: The Jetson Orin Nano 8GB board runs *Ubuntu 22.04.5 LTS (aarch64)* with a custom NVIDIA kernel *5.15.148-tegra*, CUDA *12.6.68*, and NVIDIA driver version *540.4.0*. The Jetson has a low-power model, where two of the cores are shut off, and the core frequency is reduced to 729 MHz; in this model, power consumption is reduced from 25W to 7W. This platform is representative of robotics and autonomous vehicle workloads.

V. RESULTS

We now evaluate *BetterTogether* across the three applications and the four heterogeneous hardware platforms described in Sec. IV. We begin by showing the speedup of our heterogeneous scheduling approach over the best-performing homogeneous baselines (Sec. V-A). We then assess the accuracy of our performance model by comparing its predicted latencies against real-world execution, and evaluate how our auto-tuner selects schedules under such contention (Sec. V-B). Finally,

TABLE II
CIFAR-10 ALEXNET VARIANT. ALL CONV LAYERS USE 3 × 3 KERNELS WITH STRIDE 1 AND PADDING 1. POOLING USES 2 × 2 WITH STRIDE 2.

Layer	**Shape**	**Kernel**	**Act.**
Input	(128, 3, 32, 32)	–	–
Conv1	(128, 16, 32, 32)	3×3, 3→16	ReLU
Pool1	(128, 16, 16, 16)	2×2 pool	–
Conv2	(128, 32, 16, 16)	3×3, 16→32	ReLU
Pool2	(128, 32, 8, 8)	2×2 pool	–
Conv3	(128, 64, 8, 8)	3×3, 32→64	ReLU
Conv4	(128, 64, 8, 8)	3×3, 64→64	ReLU
Conv5	(128, 64, 8, 8)	3×3, 64→64	ReLU
Pool3	(128, 64, 4, 4)	2×2 pool	–
FC	(128, 10)	1024→10	Softmax

we examine how intra-application interference impacts both performance and prediction accuracy (Sec. V-C).

A. Overall Heterogeneous Performance

Figure 5 shows an overall geometric mean speedup of 2.14× across all workloads and platforms, with the highest observed speedup 7.59× on the Octree application running on the Google Pixel. This substantial gain is partly due to Octree's unique mix of regular and irregular compute patterns, which makes it well-suited for approachname's fine-grained heterogeneous pipeline scheduling.

a) Baselines: The baselines used are the same implementations as in *BetterTogether*, except that tasks only run on a single type of PU. We performed benchmarks on all stages, using `DOALL` parallelism. For the GPU baseline, we offload the entire workload to the GPU; this is an accelerator-oriented approach. For the CPU baselines, we use only the big cores, as they consistently deliver the best performance across our datasets. Mixing big and little cores led to degraded performance due to load imbalance. We compared our best schedules generated by the best-performing baselines, shown in Tab. IV, and presented visually in Fig. 5 for each application on each device. As shown, AlexNet-dense performs poorly on CPUs in mobile devices like those from Google/OnePlus, primarily due to limited CPU parallelism.

b) Mobile Platforms: Among all platforms, *BetterTogether* achieved its highest geometric mean speedup on the Google Pixel: 4.75×, with a peak of 7.59×. The OnePlus showed slightly lower gains, with a geometric mean speedup of 3.61× and a maximum of 6.26×. We attribute this to two factors: First, the baseline performances on the Pixel are relatively low due to its lower CPU frequency compared to OnePlus (shown in Tab. III); Second, the Pixel offers full CPU affinity control, allowing all eight cores to be pinned, whereas OnePlus only allows 5 out of 8 cores to be pinned. This enables *BetterTogether* to exploit both compute specialization and parallelism across stages.

c) Jetson Orin Platforms: The Jetson platform (regular configuration) showed the most modest improvements, with a geometric mean of 1.09× and a maximum of 1.14×. On the other hand, the Jetson (low-power configuration) exhibited a slightly higher geometric mean of 1.15× and a maximum of 1.78×. However, one instance of the Octree application produced a slight slowdown (1 out of 12 configurations) despite selecting the best schedule produced by *BT-Optimizer*. In low-power mode, the Jetson platform operates with only four CPUs running at approximately half the frequency of the regular configuration, resulting in even less heterogeneity. This contrasts with the other two mobile phones, which provide three to four distinct CPU types at varying performance levels. This was the only instance of negative speedup in our evaluation.

TABLE III
HARDWARE SPECIFICATIONS OF TESTED EDGE PLATFORMS. IN OUR PRELIMINARY EXPLORATION OF MOBILE AI ACCELERATORS, WE ALSO EVALUATE THE INTEGRATED TPU IN THE GOOGLE TENSOR G2 SOC

Device	CPU (Cores @ Frequency)	GPU
Google Pixel 7A	2× Cortex-X1 @ 2.85 GHz 2× Cortex-A78 @ 2.35 GHz 4× Cortex-A55 @ 1.80 GHz	Mali-G710 MP7
OnePlus 11	1× Cortex-X3 @ 3.2 GHz 2× Cortex-A715 @ 2.8 GHz 2× Cortex-A710 @ 2.8 GHz 3× Cortex-A510 @ 2.0 GHz	Adreno 740
Jetson Orin Nano	6× Cortex-A78AE @ 1.7 GHz	Ampere GPU
Jetson Orin Nano (low-power mode)	4× Cortex-A78AE @ ˜0.85 GHz	Ampere GPU

B. Model vs. Real-World Measurement

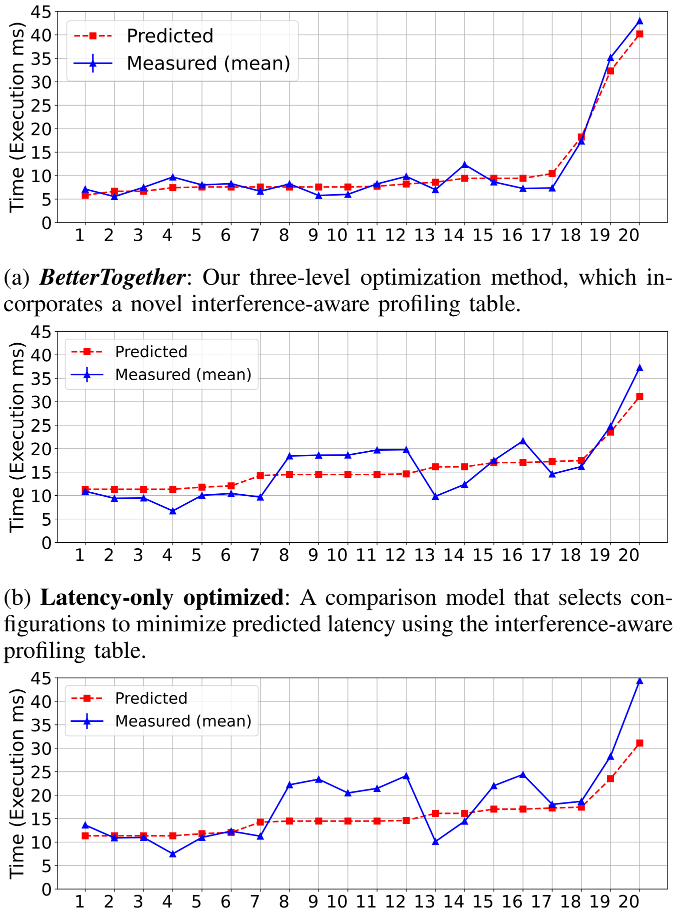
To evaluate the accuracy of our scheduling model, we first compared the predicted latencies with the real-world execution measurement of the top 20 filtered schedules, for each application on each device. We then compute the Pearson correlation between predicted and measured latency across all candidate schedules. A correlation near 1.0 indicates strong agreement, while values closer to 0 indicate weak or no correlation. Overall, our three-level optimization with the interference-aware profiling table achieves a high arithmetic mean correlation of 0.92 across all devices and applications, with a maximum of 0.99, in Fig. 7a. We present the overall correlation heatmap of the standard approach, which uses the isolated profiling table and optimizes only for predicted latency, in Fig. 7b.

Among the workloads, AlexNet-dense yields the highest correlation (geomean of 0.97), likely due to its regular, predictable execution pattern. In contrast, Octree exhibits the lowest correlation (0.87), reflecting its irregular behavior. While lower, this still reflects a strong positive correlation. Across platforms, the OnePlus device achieves the highest average correlation (0.97), while Jetson (in its normal-power configuration) shows the lowest (0.88).

a) Autotuning Solutions: In Sec. III-D0b, we detailed how optimal schedules are selected from candidates generated by *BT-Optimizer*. E.g., Tab. V presents the measured and predicted execution times for the top 10 of these candidates on the AlexNet-Sparse workload running on a Google Pixel device. Index 1 (grey column) corresponds to the schedule with the lowest predicted latency, selected as the default output by *BT-Optimizer*. Its measured execution time is 5.34 ms. However, index 4 (blue column) achieves the best-observed latency of 3.96 ms, which yields a 1.35× speedup over the default predicted-best. This result suggests that users of

TABLE IV
RAW BASELINE PERFORMANCE (IN MILLISECONDS) FOR EACH DEVICE ON CPU AND GPU. ENTRIES ARE FORMATTED AS CPU — GPU, WITH THE FASTER RESULT IN BOLD. LOWER IS BETTER.

Device	AlexNet (Dense)	AlexNet (Sparse)	Octree
Google	940.0 — **11.4**	45.8 — **44.9**	**14.2** — 58.7
OnePlus	730.0 — **12.1**	53.2 — **27.9**	**12.7** — 47.2
Jetson	23.5 — **5.5**	486.0 — **27.2**	16.2 — **5.4**
Jetson (LP)	58.5 — **23.6**	1042.0 — **101.0**	39.7 — **7.3**



(a) ***BetterTogether***: Our three-level optimization method, which incorporates a novel interference-aware profiling table.

(b) **Latency-only optimized**: A comparison model that selects configurations to minimize predicted latency using the interference-aware profiling table.

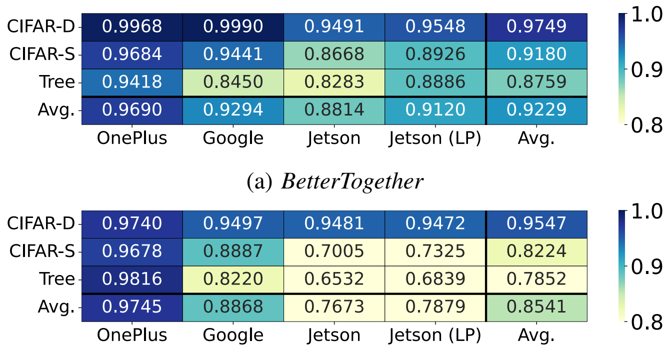
(c) **Isolated Table and Latency-only optimized**: A comparison model that uses an isolated profiling table, and minimizes latency. This strategy is commonly used in prior work.

Fig. 6. Comparison of predicted and measured execution times for the top 20 schedules across different schedules on AlexNet (sparse) on Google Pixel. (a) demonstrates that *BetterTogether* produces predictions that closely match measured execution time. Prior approaches (b) and (c) show discrepancies between predictions and actual performance.

BetterTogether could obtain an additional 35% speedup on top of the existing 1.56× improvement over the best homogeneous baseline by applying the third optimization level described in Sec. III-D, which evaluates a small set of top candidates via runtime benchmarking. In our experiments, with 20 candidate schedules ($\mathcal{K} = 20$), the entire autotuning phase takes approximately ≈ 200 seconds to complete per device per application.

TABLE V
MEASURED AND PREDICTED LATENCY (MS) FOR THE TOP 10 SCHEDULES ON GOOGLE PIXEL, RUNNING ALEXNET (SPARSE). SPEEDUP IS MEASURED AGAINST SCHEDULE NUMBER 1.

	1	2	3	**4**	5	6	7	8	9	10
Measured	5.34	5.38	4.23	**3.96**	7.67	5.35	6.99	5.48	5.86	7.37
Predicted	5.65	5.86	5.86	**5.86**	7.95	7.95	7.95	7.95	7.95	7.95
Speedup	1.00	0.99	1.26	**1.35**	0.70	1.00	0.76	0.97	0.91	0.72



(a) *BetterTogether*

(b) Using isolated performance profiles and optimizing only for latency.

Fig. 7. Correlation (0.0–1.0) between predicted and actual times across all applications and platforms. Higher is better.

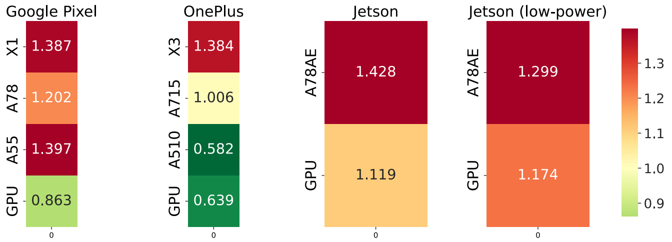


Fig. 8. Average ratio of interference-heavy to isolated execution times for all applications on each device. Red indicates slowdown under contention; green indicates speedup; yellow indicates negligible impact.

C. Impact of Interference

To explore why previous approaches fail to produce accurate performance models (as demonstrated in Fig. 6b and Fig. 6c), we now explore the impact of interference on each PU across our devices. To do this, we ran our profiling in both an isolated environment and an interference-heavy environment, following the steps in Sec. III-C. Figure 8 shows the average ratio of interference vs isolated profiling across all three applications for each device.

a) On CPUs: CPUs show diverse behavior under interference, with slowdown ratios varying across core types and devices, generally above 1.0. On the Pixel, all CPU clusters are affected: little cores experience a 1.39× slowdown, medium cores 1.20×, and big cores 1.40×. Jetson follows a similar trend, with its little cores slowing down by 1.43× and 1.29× in low-power mode. However, the OnePlus device shows unusual behavior. While its big cores show the expected slowdown (1.38×) and medium cores remain unaffected (1.00×), the little cores surprisingly experience a significant speedup (0.63×). We found no official documentation explaining this effect. However, runtime monitoring showed that the system increases the frequency of the A510 cores during interference. This suggests OnePlus may enter a high-performance mode in response to system load, improving the performance of the little cores.

b) On GPUs: On Pixel and OnePlus (implemented in Vulkan), GPUs often speed up significantly under load, e.g., 0.86× and 0.639×, respectively. In contrast, the Jetson device (implemented in CUDA) shows moderate slowdowns of 1.19× and 1.74× but remains relatively stable compared to the mobile platforms. This behavior is likely caused by dynamic system management at the OS level: clock boosting, thermal limits, etc. We found no official documentation explaining this effect. Notably, mobile phones run unrooted Android systems, where low-level GPU behavior is influenced by vendor-specific firmware and closed-source drivers. We consulted with engineers from a major mobile vendor, whose insights were consistent with our observations.

c) Summary: The same benchmark exhibits diverse performance patterns across platforms, operating systems, and GPU backends. This underscores the need for *BT-Profiler* to accurately model and predict system behavior under representative intra-application interference.

D. AI Accelerators

```
model = Model.create();
input, ..., output = model.addOperands();

model.setOperandValue(filter, filterData);
model.setOperandValue(bias, biasData);
model.setOperandValue(pad, PADDING_SAME,
              stride, 1, act, FUSED_NONE);

model.addOperation(CONV_2D, [input, filter,
 bias, pad, stride, stride, act], [output]);
model.identifyIO([input], [output]);
model.finish();

comp = Compilation.create(model).finish();
Execution.create(comp)
          .setInput(inputData)
          .setOutput(outputData).compute();
```

Listing 3. NNAPI Conv2D kernel (Pseudocode)

Although this work primarily targets CPUs and integrated GPUs, the modular design of *BetterTogether* makes it straightforward to extend the framework to additional PUs. To illustrate this extensibility, we conducted a preliminary exploration with Google's EdgeTPU (see Sec. II-A0d). The goal of this experiment is not to provide a fully optimized TPU implementation, but rather to demonstrate the feasibility of incorporating more PUs (e.g., accelerators). By doing so, we highlight how *BetterTogether* can accommodate the growing diversity of heterogeneous SoCs, where AI accelerators are now increasingly deployed alongside CPUs and GPUs.

We implemented the AlexNet-dense kernels using the *Android Neural Networks API* (NNAPI) [47]. The model is constructed by declaring quantized tensor operands for inputs, filters, biases, and outputs, adding CONV_2D, MAXPOOL operations with parameters (e.g., padding, stride, and activation), and then binding input/output buffers at runtime, as shown in List. 3. After compilation, the execution runs directly on the device's supported accelerator, i.e., on DEVICE 0: GOOGLE-EDGETPU. Although the OnePlus device integrates an AI accelerator (Qualcomm Hexagon Tensor Accelerator), it was not exposed through NNAPI and thus could not be utilized in our implementation.

When profiled in isolation, the TPU kernels ran in 9.85 ms, but under full GPU contention (*BT-Profiler*) the latency improved to 8.64 ms, corresponding to a 12.3% speedup. We ran the experiments across multiple runs and system states, and this trend remains the same. This indicates that even dedicated AI accelerators are subject to interference effects. Using these measurements, we re-ran the scheduler with CPUs, the GPU, and the additional TPU included. The resulting optimal schedule achieved an end-to-end latency of 6.90 ms, representing a 1.25× improvement over the configuration without the TPU. In this schedule, the TPU was assigned to execute Stage 1 (6.86 ms), while the GPU processed all subsequent stages.

Note that these results are preliminary, as TPU support is not yet fully integrated into *BetterTogether*. These results demonstrate the flexibility of *BetterTogether*, and they highlight the potential benefits of integrating additional PUs. We leave a more robust implementation and comprehensive evaluation across a variety of AI accelerators for future work.

VI. Related Work

a) Utilizing Heterogeneous PUs: Redwood [50] introduces heterogeneous decomposition strategies for tree traversal workloads across CPU, GPU, and FPGA on UMA systems. However, it targets a narrow workload class compared to *BetterTogether*, which supports a wider range of applications. StarPU [6] provides dynamic scheduling for task graphs on heterogeneous architectures using cost models. MOSCOA [4] employs a metaheuristic-based static scheduling approach. However, these frameworks do not specifically address edge devices with big.LITTLE architectures.

b) Pipeline Parallel Execution: Benoit et al. [12] offer a comprehensive survey of pipeline scheduling methods. Agrawal et al. [3] investigate scheduling for linear pipelined workflows on parallel machines, showing that latency minimization is easy. However, most techniques assume a known optimal mapping of pipeline stages to PUs in advance. Cordes et al. [14] map tasks across homogeneous CPU cores without considering accelerators like GPUs. Emeretlis et al. [17] address heterogeneous CPU clusters using *Integer Linear Programming* (ILP). Hu et al. [26] statically schedule DNN inference across GPUs and CPUs. While these approaches differ in their platforms and scheduling formulations, they all rely on static cost models that fail to capture interference effects. As a result, predicted performance often diverges from actual execution, limiting its applicability in dynamic, resource-constrained environments like mobile SoCs.

c) Interference-aware Benchmarking: Interference effects are common in data center environments [36]: Makroo et al. [38] uses active resource monitoring to address noisy neighbors in cloud infrastructure; Hochenbaum et al. [24] uses a statistical learning method to detect anomalies in clouds.

For edge systems: Bechtel et al. [10] demonstrate that resource contention on Jetson Orin increases latency and reduces accuracy. Afonso et al. [1] identify a range of systematic error sources on hardware factors on mobile platforms, including DVFS, thermal throttling, and power management on mobile platforms, proposing mitigation strategies primarily for external and OS-level noise. Aviv et al. [7] highlight performance variability factors such as GPU driver overhead and context switching, proposing kernel fusion and workload shaping to improve isolated benchmark reliability. Barve et al. [9] recognized that in heterogeneous computing environments, application performance can degrade due to interference from shared, non-partitionable resources and unpredictable co-located workloads. Iorga et al. [27] specifically investigated interference on mobile and embedded ARM platforms, identified shared resources such as last-level caches and memory buses as primary sources of contention. However, existing work does not address intra-application interference directly on mobile SoCs, a key focus of our work.

VII. Discussion and Future Works

A. Static vs. Dynamic Pipeline Scheduling

Prior work on heterogeneous scheduling algorithms generally falls into two categories: *static* and *dynamic*. Static scheduling determines the optimal stage-PU mapping at compile time or prior to execution (via *BT-Profiler* and *BT-Optimizer*). In contrast, dynamic scheduling can adapt task assignments at runtime [33], [45], e.g., via work-stealing [2], [23] or live profiling. Static pipelines are less flexible than dynamic pipelines in that they cannot adapt to unexpected runtime changes after deployment, due to situations like power throttling or environmental changes [14], [17]. However, dynamic pipelines also have drawbacks: e.g., [44] shows that the added complexity in dynamic scheduling approaches can introduce up to 13% runtime overhead. Also, dynamic approaches may suffer from local minima, as the work-stealing approach can assign tasks to available PUs regardless of suitability [42].

Future works can explore more efficient dynamic scheduling: For example, utilizing information from our profiling tables to avoid poor work distribution choices and falling back to efficient static schedules if a stable environment is detected.

B. Generalization to Broader Applications

In *BetterTogether*, the input workload relies on (1) streaming input that can be pipelined, and (2) can be decomposed into distinct stages. This is a natural fit for computer vision tasks, which motivated our evaluation. We showed three representative classes for edge computing in this paper (sparse, dense, and hybrid). However, any application that has this characteristic could potentially benefit from *BetterTogether*. Given the modular design of *BetterTogether*, it is straightforward to add new kernels as long as C/C++ bindings are available. We assume users provide optimized device-specific kernels or leverage existing libraries and ML frameworks (e.g., TFLite, cuBLAS, CUB) for common operations.

Future work could apply *BetterTogether* beyond vision to other edge workloads, such as ML-based audio processing [40], autonomous systems [11], drone navigation [16], [37], and related domains.

C. Integrated vs. Discrete Accelerators

This work primarily focuses on mobile SoCs with UMA architectures, where the CPU and integrated accelerators share the same physical memories. Thus, communication between PUs is more efficient than communication in traditional discrete accelerator systems, i.e., communication does not require expensive data transfers across PCIe (as shown in [19], [39], [51]). However, the scheduling algorithm in *BetterTogether* can translate to discrete accelerators or even multiple GPUs.

Future works can explore extending *BetterTogether* to discrete accelerators like GPU or multi-GPU architecture, by profiling and modeling the memory transfer cost *BT-Profiler* and integrating that additional data transfer cost in the solver, to extend *BetterTogether* to a wider range of HW architectures beyond mobile SoCs.

VIII. Conclusion

In this work, we proposed *BetterTogether*, a framework for generating highly efficient static pipeline schedules on heterogeneous edge SoCs. We proposed a novel performance modeling approach that takes into account intra-application interference and can accurately predict the execution time of pipeline schedules. Our evaluation demonstrates that *BetterTogether* is effective across diverse edge devices and application workloads, achieving up to $7.59\times$ speedup over homogeneous baselines, with a geometric mean improvement of $2.14\times$, and a high prediction accuracy with an average correlation of 92.3%.

References

[1] S. Afonso and F. Almeida, "Rancid: Reliable benchmarking on Android platforms," *IEEE Access*, 2020.

[2] M. Agostini, F. O'Brien, and T. Abdelrahman, "Balancing graph processing workloads using work stealing on heterogeneous CPU-FPGA systems," in *Proceedings of the 49th International Conference on Parallel Processing*, 2020.

[3] K. Agrawal, A. Benoit, L. Magnan, and Y. Robert, "Scheduling algorithms for linear workflow optimization," in *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010.

[4] M. Akbari and H. Rashidi, "A multi-objectives scheduling algorithm based on cuckoo optimization for task allocation problem at compile time in heterogeneous systems," *Expert Systems with Applications*, 2016.

[5] J. A. Ambrose, Y. Yachide, K. Batra, J. Peddersen, and S. Parameswaran, "Sequential c-code to distributed pipelined heterogeneous MPSoC synthesis for streaming applications," in *IEEE International Conference on Computer Design (ICCD)*, 2015.

[6] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," in *Euro-Par Parallel Processing*, 2009.

[7] R. Aviv and G. Wang, "OpenCL-based mobile GPGPU benchmarking: Methods and challenges," in *Proceedings of the 4th International Workshop on OpenCL*, 2016.

[8] E. Ayguade, R. M. Badia, D. Cabrera, A. Duran, M. Gonzalez, F. Igual, D. Jimenez, J. Labarta, X. Martorell, R. Mayo *et al.*, "A proposal to extend the OpenMP tasking model for heterogeneous architectures," in *International Workshop on OpenMP*, 2009.

[9] Y. D. Barve, S. Shekhar, A. Chhokra, S. Khare, A. Bhattacharjee, Z. Kang, H. Sun, and A. Gokhale, "FECBench: A holistic interference-aware approach for application performance modeling," in *IEEE International Conference on Cloud Engineering (IC2E)*, 2019.

[10] M. Bechtel and H. Yun, "Analysis and mitigation of shared resource contention on heterogeneous multicore: An industrial case study," *IEEE Transactions on Computers*, 2024.

[11] A. J. Ben Ali, M. Kouroshli, S. Semenova, Z. S. Hashemifar, S. Y. Ko, and K. Dantu, "Edge-SLAM: Edge-assisted visual simultaneous localization and mapping," *ACM Transactions on Embedded Computing Systems*, 2022.

[12] A. Benoit, Ü. V. Çatalyürek, Y. Robert, and E. Saule, "A survey of pipelined workflow scheduling: Models and algorithms," *ACM Computing Surveys (CSUR)*, 2013.

[13] K. Cao, Y. Liu, G. Meng, and Q. Sun, "An overview on edge computing research," *IEEE Access*, 2020.

[14] D. Cordes, M. Engel, O. Neugebauer, and P. Marwedel, "Automatic extraction of pipeline parallelism for embedded heterogeneous multi-core platforms," in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2013.

[15] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[16] C. Deng, X. Fang, and X. Wang, "UAV-enabled mobile-edge computing for AI applications: Joint model decision, resource allocation, and trajectory optimization," *IEEE Internet of Things Journal*, 2022.

[17] A. Emeretlis, G. Theodoridis, P. Alefragis, and N. Voros, "Static mapping of applications on heterogeneous multi-core platforms combining logic-based benders decomposition with integer linear programming," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2017.

[18] A. E. Eshratifar and M. Pedram, "Energy and performance efficient computation offloading for deep neural networks in a mobile cloud computing environment," in *Proceedings of the 2018 Great Lakes Symposium on VLSI*, 2018.

[19] D. Giri, P. Mantovani, and L. P. Carloni, "Accelerators and coherence: An SoC perspective," *IEEE Micro*, 2018.

[20] Google, "Google Benchmark," 2025, online; accessed 2025-06-16.

[21] Grand View Research, "Edge Computing Market Size, Share & Trends Analysis Report, 2024," 2024, online; accessed 2025-04-24.

[22] Y. Guo, Y. Xu, and X. Chen, "Freeze it if you can: Challenges and future directions in benchmarking smartphone performance," in *Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications*, 2017.

[23] S. Heldens, A. L. Varbanescu, and A. Iosup, "Dynamic load balancing for high-performance graph processing on hybrid cpu-gpu platforms," in *6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*, 2016.

[24] J. Hochenbaum, O. S. Vallis, and A. Kejariwal, "Automatic anomaly detection in the cloud via statistical learning," *arXiv preprint arXiv:1704.07706*, 2017.

[25] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An efficient probabilistic 3d mapping framework based on octrees," *Autonomous Robots*, 2013.

[26] Y. Hu, C. Imes, X. Zhao, S. Kundu, P. A. Beerel, S. P. Crago, and J. P. N. Walters, "Pipeline parallelism for inference on heterogeneous edge computing," *arXiv preprint arXiv:2110.14895*, 2021.

[27] D. Iorga, T. Sorensen, J. Wickerson, and A. F. Donaldson, "Slow and steady: Measuring and tuning multicore interference," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.

[28] V. Joseph, G. L. Gopalakrishnan, S. Muralidharan, M. Garland, and A. Garg, "A programmable approach to neural network compression," *IEEE Micro*, 2020.

[29] 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.

[30] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, 2017.

[31] T. Karras, "Maximizing parallelism in the construction of BVHs, octrees, and k-d trees," in *Proceedings of the Fourth ACM SIGGRAPH/Eurographics Conference on High-Performance Graphics*, 2012.

[32] L. S. N. Karumbunathan, "Nvidia Jetson AGX Orin Series," 2022, technical Brief.

[33] J. Y. Kim and C. Batten, "Accelerating irregular algorithms on GPGPUs using fine-grain hardware worklists," in *47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[34] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[35] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Advances in Neural Information Processing Systems*, 2012.

[36] T. Lorido-Botran, S. Huerta, L. Tomás, J. Tordsson, and B. Sanz, "An unsupervised approach to online noisy-neighbor detection in cloud data centers," *Expert Systems with Applications*, 2017.

[37] Y. Lu, Z. Xue, G.-S. Xia, and L. Zhang, "A survey on vision-based UAV navigation," *Geo-spatial information science*, 2018.

[38] A. Makroo and D. Dahiya, "A systematic approach to deal with noisy neighbour in cloud infrastructure," *Indian journal of science and technology*, 2016.

[39] P. Mantovani, E. G. Cota, C. Pilato, G. Di Guglielmo, and L. P. Carloni, "Handling large data sets for high-performance embedded applications in heterogeneous systems-on-chip," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2016.

[40] A. Mehrish, N. Majumder, R. Bharadwaj, R. Mihalcea, and S. Poria, "A review of deep learning techniques for speech processing," *Information Fusion*, 2023.

[41] S. Mittal and J. S. Vetter, "CPU-GPU heterogeneous computing techniques: A survey," *ACM Computing Surveys (CSUR)*, 2015.

[42] A. Navarro, R. Asenjo, S. Tabik, and C. Caşcaval, "Load balancing using work-stealing for pipeline parallelism in emerging applications," in *Proceedings of the 23rd international conference on Supercomputing*, 2009.

[43] NVIDIA Corporation, *CUDA C++ Programming Guide*, release 12.6 ed., https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2025, accessed: 2025-09-03.

[44] D. Sanchez, D. Lo, R. M. Yoo, J. Sugerman, and C. Kozyrakis, "Dynamic fine-grain scheduling of pipeline parallelism," in *2011 IEEE International Conference on Parallel Architectures and Compilation Techniques*, 2011.

[45] F. Song, A. YarKhan, and J. Dongarra, "Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems," in *Proceedings of the conference on high performance computing networking, storage and analysis*, 2009.

[46] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, 2017.

[47] The Android Developers, "Android neural networks API (NNAPI)," https://developer.android.com/ndk/guides/neuralnetworks.

[48] The Khronos Group, "Vulkan specification - pipelines," https://docs.vulkan.org/spec/latest/chapters/pipelines.html, 2024, accessed: 2025-04-24.

[49] ——, "Vulkan API specification," https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html, 2025, accessed: 2025-09-03.

[50] Y. Xu, A. Li, and T. Sorensen, "Redwood: Flexible and portable heterogeneous tree traversal workloads," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023.

[51] J. Zuckerman, D. Giri, J. Kwon, P. Mantovani, and L. P. Carloni, "Cohmeleon: Learning-based orchestration of accelerator coherence in heterogeneous SoCs," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021.