Summary

This paper investigates whether the energy consumption of CUDA kernels can be accurately estimated using static analysis of NVIDIA's Parallel Thread Execution (PTX) intermediate representation. The motivation is to enable energy-aware GPU software development without relying on runtime profiling, simulation, or hardware performance counters, which require executing the target application and may introduce significant overhead.

To address this problem, the authors developed a static analysis framework that extracts PTX instruction frequencies from CUDA kernels and combines them with instruction-level energy measurements obtained through external power monitoring. Energy measurements were collected using an ElmorLabs PMD2 power monitor, allowing more accurate measurements than those provided by NVIDIA's built-in sensors. The framework constructs control-flow graphs from PTX code, estimates instruction execution counts, and uses these counts as input for energy prediction models.

The study evaluates three energy estimation approaches. The first two are static instruction-cost models that assign fixed energy weights to PTX instructions, derived using either a baseline-subtraction method or a differential-repeat method. The third approach uses a neural network that learns the relationship between PTX instruction frequencies and kernel energy consumption.

To evaluate the models, the authors created a validation suite consisting of 144 CUDA kernels from a diverse set of workloads, including NAS Parallel Benchmarks, memory-access microbenchmarks, SGEMM matrix multiplication optimizations, transformer-inspired kernels, and synthetic stress tests. Experimental results show that individual PTX instructions vary significantly in energy consumption, with the most expensive instructions consuming up to two orders of magnitude more energy than the least expensive ones.

Despite these differences, the results demonstrate that static PTX analysis alone is insufficient for accurate kernel-level energy prediction. The instruction-cost models achieved mean absolute percentage errors (MAPE) of 90.3% and 91.2%, respectively. The neural-network model improved prediction accuracy, achieving a MAPE of 74.1%, but the errors remained too large for practical use. While some workloads, such as vector addition and certain FFT kernels, were predicted reasonably well, highly optimized matrix multiplication kernels, transformer workloads, and memory-intensive applications exhibited substantial prediction errors.

The findings indicate that GPU energy consumption depends heavily on runtime factors that are not visible in static PTX code, including memory-access patterns, cache utilization, occupancy, instruction scheduling, and hardware resource contention. Consequently, PTX instruction frequencies alone do not provide sufficient information for accurate energy estimation across diverse CUDA workloads.

In conclusion, the paper shows that although static PTX analysis can reveal meaningful differences in instruction-level energy consumption and capture some workload characteristics, accurate GPU energy estimation requires information beyond static instruction counts. Future work should therefore investigate hybrid approaches that combine static analysis with runtime characteristics and hardware performance information.

Static PTX Analysis for CUDA Kernel Energy Estimation

Lasse Fjellanger[♣]
lfjell21@student.aau.dk

Rasmus Madsen[♣]
rma21@student.aau.dk

Sebastian Livoni Larsen[♣]
seblar21@student.aau.dk

[♣]Aalborg University
Department of Computer Science

Abstract

GPUs are central to modern AI and high-performance computing workloads, making energy efficiency an increasingly important concern. This paper investigates whether the energy consumption of CUDA kernels can be estimated statically from NVIDIA's Parallel Thread Execution (PTX) intermediate representation. We develop a static analysis framework that extracts PTX instruction frequencies and combine it with instruction-level energy measurements obtained through external power monitoring. Two instruction-cost models and a neural-network-based instruction-frequency model are evaluated on 144 CUDA kernels spanning scientific computing, memory-intensive workloads, matrix multiplication optimizations, and transformer-inspired applications. The results show that individual PTX instructions vary in energy consumption by up to two orders of magnitude. We find static PTX analysis alone is insufficient for accurate kernel-level energy prediction. The best-performing model achieves a mean absolute percentage error of 74.1%, indicating that runtime factors such as memory behavior, cache utilization, and resource contention play a significant role in GPU energy consumption.

CCS Concepts: • **Computing methodologies → Experimentation; Measurement;** • **Hardware → Power and energy; Performance.**

Keywords: Energy, CUDA, GPU architecture, Power estimation

1 Introduction

The rapid growth of artificial intelligence (AI) and high-performance computing has led to an unprecedented increase in demand for GPU resources, making energy efficiency an increasingly important concern for both economic and environmental reasons [1], [2]. While existing approaches to GPU energy estimation typically rely on runtime profiling, hardware performance counters, or simulation, these methods require execution of the target application and can introduce significant overhead [3]. This paper investigates whether the energy consumption of CUDA kernels can be estimated statically from NVIDIA's Parallel Thread Execution (PTX) intermediate representation. By combining instruction-level energy measurements obtained through external power monitoring with static PTX analysis and machine-learning models, we evaluate the extent to which kernel energy consumption can be predicted prior to execution and assess the potential of PTX-based energy estimation for energy-aware GPU software development.

The **replication package** has been made publicly available on GitHub.

1.1 Overview

This study aims to: (1) determine energy consumption of PTX instructions by experimental measurement and (2) predict energy consumption of CUDA kernels using static analysis. Through the following research questions and their related experiments this work seeks to contribute to a broader understanding of energy-aware CUDA kernel development.

1.1.1 Research Questions

Understanding whether static analysis can replace or reduce the need for runtime profiling is the central question of this study. Accurately predicting energy consumption from PTX code without executing the kernel would dramatically speed up the development cycle and reduce reliance on time-consuming profiling runs. This motivates the following primary research question: *Can energy consumption of CUDA kernels be accurately estimated through static PTX code analysis?* To answer this, we investigate the following subquestions:

RQ1: *What is the energy consumption associated with individual PTX instructions in CUDA kernels?* Understanding the energy footprint of each PTX instruction is fundamental because it allows developers to identify the most power-hungry operations at the instruction level.

RQ2: *How accurately can the energy consumption of CUDA kernels be predicted by assigning static energy weights to PTX instructions?* A well-established approach[4] in static energy analysis is to assign a fixed energy cost to each instruction and estimate total consumption by counting instruction occurrences, accounting for loop iterations. This examines whether that same principle can be effectively applied to PTX.

RQ3: *How does the accuracy of static PTX-based energy estimation vary across different CUDA kernel types and workloads?* A prediction model may perform well on some kernels but poorly on others. Evaluating estimation accuracy across diverse workloads such as memory-bound versus compute-bound kernels reveals the generalizability and limitations of the static analysis approach.

2 Background

This section provides the background needed to follow the rest of the paper. It first describes the GPU architecture and the CUDA programming model that determine how a kernel maps onto hardware. It then covers GPU power modeling

and the tooling used to measure power, including the limitations of NVIDIA's built-in sensors that motivate an external measurement setup. Finally, it introduces Parallel Thread Execution (PTX), the intermediate representation the static analysis operates on.

GPU Architecture & CUDA Programming Model Modern GPU architectures consist of thousands of cores organized into processing units. On NVIDIA GPUs, these units are referred to as Streaming Multiprocessor (SM) [5]. Figure 1 presents an overview of the NVIDIA Ada Lovelace SM architecture. Each SM is equipped with its own compute units (FP32/FP64 cores, integer ALUs, and tensor cores), a memory hierarchy consisting of registers (per-thread, fastest), shared memory/L1 cache (on-chip, low latency, shared within the SM) and global memory backed by L2 cache [5] (off-chip, higher latency but large capacity), warp schedulers for dispatching threads, and Special Function Units (SFUs) that handle transcendental and approximation operations such as sine, cosine, and square root functions. CUDA organizes execution into threads, warps, thread blocks, and grids. A *kernel* is a function that defines the work performed by each thread, executed in parallel across all threads in the grid. A thread is the smallest unit of execution and has its own private registers. A warp consists of 32 threads that execute in lockstep. Multiple warps form a thread block, which shares a pool of shared memory and is assigned to a single SM. L2 cache and global memory are shared across all SMs [5]. A grid is a collection of thread blocks that represents all the threads launched by a kernel. The thread block configuration, which determines the number of threads per block and the number of blocks per grid, influences GPU performance, memory **coalescing** (memory requests from multiple threads within a warp are merged into a single, contiguous memory transaction), SM occupancy (active warps per SM) and energy efficiency. However, this flexibility requires thorough kernel tuning to optimize hardware resource utilization [6]. Suboptimal thread block shapes can degrade performance, presenting multidimensional energy optimization challenges that necessitate specialized solutions [6].

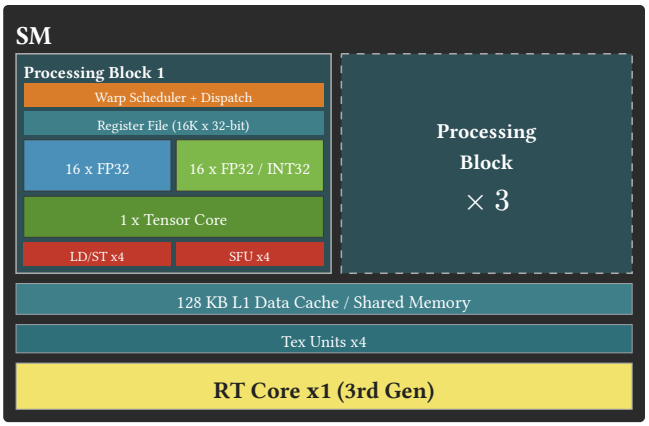


Figure 1: Overview of an NVIDIA Ada Lovelace SM architecture in the GeForce RTX 4080 Super [7].

Figure 2 shows the overall organization of the AD103 GPU as configured in the RTX 4080 Super. At the chip level, the SMs are grouped into Texture Processor Cluster (TPC), which are in turn organized into GPU Processing Clusters (GPC). The AD103 die exposes seven GPCs: six full GPCs of six TPCs each and one reducedGPC of four TPCs, for a total of 80 SMs. All GPCs share a unified L2 cache, which buffers traffic to off-chip global memory across a 256-bit interface served by eight 32-bit memory controllers. The GigaThread Engine distributes thread blocks to the GPCs, while the Peripheral Component Interconnect Express (PCIe) express host interface connects the GPU to the rest of the system. This hierarchical layout SMs within TPCs within GPCs, all backed by a shared L2 determines how a kernel's threads are physically placed on the device and how memory accesses are routed, both of which directly influence the energy consumption.

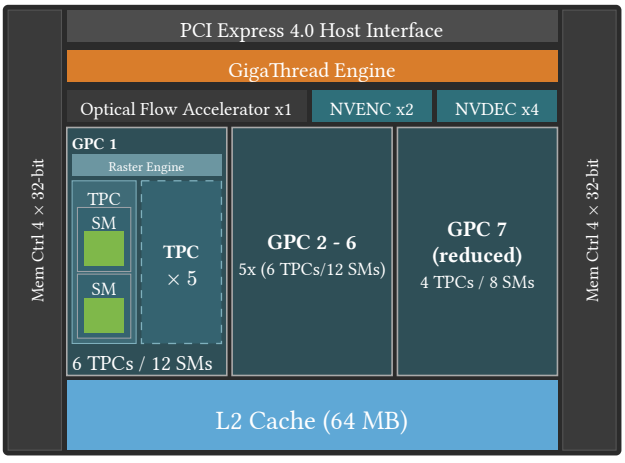


Figure 2: Simplified block diagram of an AD103-class GPU as configured for the RTX 4080 Super, showing 11 enabled GPCs (one cut down to 4 TPCs), 12 memory controllers, and the unified L2 cache [7], [8]. Full size is on Figure 13 in Appendix F.

GPU Power Modeling NVIDIA provides *NVIDIA Nsight Compute* [9], a profiling tool for analyzing and optimizing GPU kernels. It includes an *occupancy calculator* for estimating streaming multiprocessor utilization. However, this tool relies on runtime measurements and is not designed for precise power modeling. For power monitoring, NVIDIA exposes the NVIDIA Management Library (NVML) [10] for querying GPU power consumption programmatically using onboard sensors. Prior work by Z. Yang *et al.* [11] identified significant inaccuracies in NVML-power readings, leading to both underestimation and overestimation of actual energy consumption.

Modern GPUs draw power through two primary sources: the PCI Express (PCIe) x16 slot on the motherboard and dedicated external power connectors supplied directly by the PSU. The PCIe slot can deliver up to 75 W at 12 V, while the majority of power is supplied through external connectors. Starting with the RTX 40 series, NVIDIA GPUs utilize the 16-pin 12VHPWR connector, which supports power delivery of up to 600 W.

The ElmorLabs PMD2 [12], previously used in our premaster study [13], is an external hardware tool for precise measurement of voltage, current, and power across GPU, CPU, and motherboard power cables. For this study's experiments, it is installed inline with power delivery cables, enabling direct measurement of both PCIe slot power and dedicated external power connectors just like we did in our previous work [13]. The manufacturer specifies a maximum voltage error of $\pm 1V$ at $12V$ and a maximum current error of $\pm 0.5A$ at $10A$. The device is used in conjunction with a PCIe slot interposer, which allows signal pass-through while routing slot power through the measurement hardware. In our setup, the PMD PCI-E 4.0 Slot Power Measurement Adapter [14] is used as this interposer between the motherboard PCIe slot and the GPU. The PMD2 micro controller provides the telemetry via USB with a sampling rate of approximately 10 Hz. Both the PMD2 and the PCIe slot adapter are commercially available off the shelf.

Parallel Thread Execution (PTX) is an intermediate representation (IR) produced by NVIDIA CUDA Compiler (NVCC), the NVIDIA CUDA Compiler Driver. Like x86_64 or AArch64, it defines a virtual instruction set targeting an abstract machine rather than concrete hardware. At runtime, the CUDA driver JIT-compiles the embedded PTX into device-specific SASS (Streaming Assembler) instructions for execution on the GPU. Typical PTX instructions are shown in Listing 1.

```
1  fma.rn.f32  %f2, %f0, %f1, %f2;   // d = a*b + c, round-to-nearest, f32
2  ld.global.f32  %f0, [%rd0];       // load f32 from global memory
3  add.s32  %r2, %r0, %r1;           // signed 32-bit integer add
4  bra.uni  $L__BB0_1;               // unconditional branch
```

Listing 1: Typical PTX instructions

3 Preliminaries

This section introduces the graph-theoretic structures the static analysis operates on. The graph definitions in Definition 3.1 are standard and reused from [15], while the Control Flow Graph (CFG) in Definition 3.3 is the established compiler-analysis structures our PTX analysis builds on, drawn from [16] and [17] respectively. The PTX analyzer in Section 4.4 constructs a CFG and a Data Dependency Graph (DDG) directly from a kernel's PTX instructions. The definitions below describes the structures that the analyzer produces.

Definition 3.1 (Graph): A *directed graph* is a pair $G = (V, E)$ with V, a finite vertex set and

$$E \subseteq V \times V$$

a set of ordered edges, where $(u, v) \in E$ denotes an edge from u to v. G is *undirected* if $(u, v) \in E \iff (v, u) \in E$ [15].

The vertices represent the objects of interest, and edges represent the connections between them. In a directed graph, the order of an edge matters, so $u \to v$ does not imply $v \to u$.

Definition 3.2 (Basic Block): A *basic block* is a maximal instruction sequence

$$b = \langle s_1, ..., s_k \rangle$$

with a single entry at s_1, where only the terminator s_k may transfer control (a branch, jump, or return) and every other instruction falls through to its successor [16].

A basic block is a set of instructions that executes sequentially.

Definition 3.3 (Control Flow Graph): A *CFG* is a directed graph $G = (V, E)$ where V is a set of *basic blocks* and $(u, v) \in E \iff$ control may pass from u to v, with a distinguished entry block $v_{\text{in}} \in V$ and exit blocks $V_{\text{out}} \subseteq V$ [16].

Edges capture transfer of control, e.g. a branch, jump, or fall-through; a path from v_{in} corresponds to a possible execution.

4 Methodology

We model the total energy consumption of an NVIDIA GPU executing a kernel as the sum of idle and dynamic power components as shown in Equation 1.

$$P_{\text{total}} = P_{\text{idle}} + P_{\text{dynamic}} \tag{1}$$

Idle power, P_{idle}, is the power consumed even when no useful computation is being performed.

Dynamic power is consumed during kernel execution.

$$P_{\text{dynamic}} = \alpha C V^2 f \tag{2}$$

where α is the switching activity, C is the physical capacitance, V is the supply voltage, and f is the clock frequency [18].

To characterise power consumption, instantaneous power is defined at each sample index n as:

$$W[n] = V[n] \cdot I[n] \tag{3}$$

where $V[n]$ and $I[n]$ where n are the supply voltage and current, respectively, at the n-th sample

To estimate kernel energy consumption we use the PMD2 power monitor, which acquires synchronized voltage and current samples at a nominal sampling frequency of $f_s = 10$ Hz throughout the kernel execution interval T_k, yielding $N = f_s \cdot T_k$ samples. The energy consumption of a kernel spanning the interval is then estimated as:

$$\hat{P} = \frac{1}{N} \sum_{n=0}^{N-1} V[n] \cdot I[n] \tag{4}$$

We utilize CUDA Profiling Tools Interface (CUPTI) to obtain the start and end timestamps of the kernel execution, denoted t_{start} and t_{end}, via callbacks issued after the kernel completes. The kernel execution interval T_k is then:

$$T_k = t_{\text{end}} - t_{\text{start}} \tag{5}$$

4.1 Trapezoidal Rule

The trapezoidal method was used to estimate the integral, approximating the area under the power curve from discrete measurements. Given the sampling frequency and stable workload. The approximation of the energy consumption is thus given by:

$$E_k = \sum_{n=0}^{N-2} \frac{P[n] + P[n+1]}{2} \cdot \Delta t \tag{6}$$

where $\Delta t = 1/f_s$ is the sampling period.

4.2 Clock Synchronisation

CUPTI reports kernel timestamps in the GPU clock domain t_{device}, whereas the PMD2 samples are timestamped in the host CPU clock domain t_{host}. Because the two clocks drift relative to each other, a fixed offset is insufficient. We therefore fit a linear model between the two clock domains, following the approach used in the Tracy sampling profiler [19]. The incremental orthogonal regression used for clock alignment is adapted from TracyCUDA.hpp in the Tracy repository [20].

During program execution, we collect M reference pairs $\left(t_{\text{device}}^{(m)}, t_{\text{host}}^{(m)}\right)$ by recording a CUPTI timestamp alongside the host CPU timestamp. To account for measurement noise in both clock readings, we fit an orthogonal Deming regression with unit variance ratio $\delta = 1$, minimizing residuals in both axes simultaneously. This yields a slope β and intercept γ such that:

$$t_{\text{host}} = \beta \cdot t_{\text{device}} + \gamma \tag{7}$$

Any GPU timestamp is then converted to the host domain via Equation 7, allowing t_{start} and t_{end} from Equation 5 to be converted to the same clock domain as the PMD2 samples. Only PMD2 samples falling within $[t_{\text{start}}, t_{\text{end}}]$ are then included in the trapezoidal summation of Equation 6.

4.3 Kernel Annotation

Since PTX is a Virtual Instruction Set Architecture (V-ISA), loop bounds are not always statically recoverable from the intermediate file, so kernels must communicate their iteration counts to the PTX analyzer explicitly through the `@META Loop` annotation (Section 4.4). We provide this through a C++ macro library, `ptx_meta.h`, that embeds metadata into the PTX output as comments of the form `asm volatile("// @META:<version> <tag> <fields>")`. The `asm volatile` block prevents the compiler from reordering or eliminating the metadata, and as a PTX comment it is stripped before SASS compilation, incurring no runtime overhead.

The analyzer consumes a single tag, `LOOP`, recording a loop with its minimum and maximum iteration counts and an unrolling flag. The training kernel template emits `META_LOOP(<loop_name>, <min_iter>, <max_iter>, <is_unrolled>)`, as shown on line 4 of Listing 2. The `<loop_name>` permits multiple annotations per kernel and labels the emitted comment; annotations are matched to loops by position in the CFG rather than by name. The `<is_unrolled>` flag mirrors the loop's `#pragma unroll` directive, which the developer controls.

The training kernels fix the minimum and maximum bounds to a single compile-time constant, giving an exact trip count by which the analyzer scales the in-loop instruction counts. Variable trip counts, where the bounds differ, are left to future work. Each annotation is associated with its closest enclosing loop region in the CFG.

Any kernel for which a prediction is required must carry the same `META_LOOP` annotation on its loops, not only the

training kernels. Listing 2 shows such an annotation on line 4.

```
1   // Vector add kernel: out[i] = a[i] + b[i]
2   __global__ void vector_add(float *a, float   *b,
    float *out, int _N)
3   {
4       META_LOOP(iterations, ITERATIONS, ITERATIONS,
        false);
5       for(int i = 0; i < ITERATIONS; ++i) {
6           int idx = blockIdx.x * blockDim.x +
            threadIdx.x;
7
8           if (idx < _N) {
9               out[idx] = a[idx] + b[idx];
10          }
11      }
12  }
```

Listing 2: Vector add kernel with loop annotation

The analyzer also requires the launch parameters, the grid and block dimensions, since these give the total thread count by which per-thread instruction counts are scaled to a full launch. Because they are supplied at kernel launch rather than fixed in the source, they cannot be read from the PTX and are instead passed to the analyzer directly. Many of the kernels from both training and validation datasets export the launch configurations at runtime, this is not a requirement for the analyzer but for ease of use since the kernels are ran with different problem sizes which may affect the launch configuration.

4.4 PTX Analyzer

The analyzer constructs a CFG from the kernel's PTX instructions in three passes. The first identifies block boundaries at the kernel entry, every labeled instruction, and every instruction following a branch or return; the second partitions the instruction list at these boundaries; and the third derives successor and predecessor edges from each block's terminating instruction. Block chains with a unique successor and predecessor are then merged, retaining only structurally significant nodes.

Instruction counts are obtained by a depth-first traversal of the CFG. Each block holds a scope iteration count, initialized to 1. On encountering a loop back-edge, the `@META Loop` annotation (Section 4.3) supplies the iteration count, which multiplies the scope count of the loop body. Each instruction's occurrence count is incremented by the current scope count.

The analyzer emits a JSON report of the total instruction count and per-opcode occurrence frequencies, consumed by the feature extraction stage of the prediction pipeline (Section 6). An example report is provided in the replication package.

5 Benchmarks

This section describes the benchmark suite used throughout this work. For the benchmark suite we distinguish two categories: *training kernels*, purpose-built micro benchmarks used to fit the energy models, and *validation kernels*, taken from standard benchmark suites to evaluate prediction accuracy on workloads the model has not seen during training.

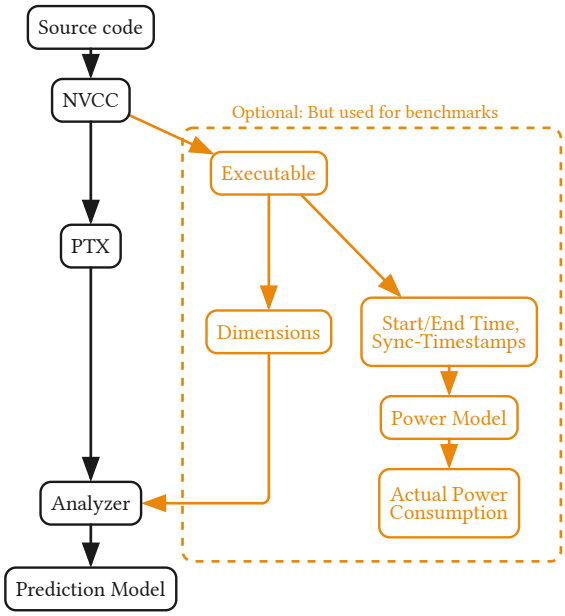


Figure 3: An overview of the model architecture

Figure 3 illustrates the overall structure of our model, outlining its two main paths. The black path is the prediction pipeline applied to every kernel: source code is compiled by NVCC to PTX, which the analyzer converts into per-instruction occurrence counts that the prediction model maps to an energy estimate. The orange path is only exercised when collecting ground-truth measurements for training and validation. Here NVCC also emits an executable, which yields the launch dimensions passed to the analyzer and the start/end synchronization timestamps that bound the kernel execution interval, these timestamps drive the power model that integrates the PMD2 samples into the actual energy consumption used as the measurement label.

5.1 Training Kernels

The kernel used for measuring energy consumption of specific PTX instructions uses the shared kernel template as seen in Listing 3. It contains a fixed loop whose body contains one or more inline PTX `asm volatile` blocks, guarded by `#pragma unroll 1` to prevent the compiler from unrolling the loop and inflating instruction counts. It is also annotated with the `META_LOOP` macro so the PTX analyzer knows the expected iteration count at analysis time, and followed by post-loop sink writes that force the compiler to treat all

computed values as used, preventing dead-code elimination of the measured instructions.

To ensure sufficient coverage of the instructions encountered in practice, the PTX instructions present across all validation kernels were collected and their union of distinct opcodes compiled. For each opcode, a dedicated micro benchmark was generated using the shared kernel template (Listing 3), isolating the instruction of interest within the loop body such that its energy contribution can be attributed independently. The resulting per-instruction measurements form the dataset from which the energy model parameters are estimated.

```
__global__ void ptx_kernel(void* sink) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  META_LOOP(main_loop, ITERATIONS, ITERATIONS,
  true);
  #pragma unroll 1
  for (int i = 0; i < ITERATIONS; ++i) {
    asm volatile("...");  // instruction 1
    asm volatile("...");  // instruction 2, ...
  }
  /* sink writes to prevent dead-code elimination */
}
```

Listing 3: Shared kernel template

Weights are extracted via a differential-repeat method to cancel loop overhead and static baseline power. Two kernel variants are executed back-to-back: a repeat-1 kernel with one instruction copy per loop iteration and a repeat-2 kernel with two copies. Both run the same iteration count, so the repeat-2 kernel performs exactly twice the instruction work. The per-instruction energy is then:

$$E_{\text{inst}} = \frac{E_{\text{r2}} - E_{\text{r1}}}{N_{\text{iters}} \times N_{\text{threads}}}$$

where E_{r1} and E_{r2} are PMD2-measured total energies. Subtraction cancels the loop control instructions, `META_LOOP` annotation overhead, and static GPU idle power, leaving a per-instruction estimate robust to baseline drift. The iteration count is determined by the required count to reach the target of $\approx 10s$ of wall-clock time, ensuring the PMD2 ($\approx$ 10 Hz) collects sufficient samples [11].

5.2 Validation Kernels

Validation kernels are chosen from multiple benchmarks suites to cover a broad spectrum of instructions, memory access patterns, and occupancy regimes.

NAS Parallel Benchmarks (NPB). The NAS Parallel Benchmarks [21] are a set of kernels originally designed to evaluate the performance of parallel supercomputers, derived from Computational Fluid Dynamics (CFD) applications. NPB is a standard baseline in GPU performance and power research [22], [23], providing reproducible workloads with well-understood computational structure. The suite comprises of five kernels, IS (integer sort, random memory access), EP (embarrassingly parallel), CG (conjugate gradient, irregular memory access), MG (multi-grid, memory intensive with long- and short-distance communication), and FT (discrete 3-D FFT, all-to-all communication) and three pseudo-applications: BT (block tri-diagonal solver), SP (scalar penta-diagonal solver), and LU (lower-upper Gauss-Seidel solver), of varying problem classes. We use the CUDA implementation and treat each kernel as an independent benchmark and compiling at problem class S the smallest class, to keep execution time tractable while preserving the full instruction mix of each algorithm. This yields 106 kernels in total.

UBench [24] is a suite of micro benchmarks designed to isolate individual effects arising from thread-block geometry and GPU hardware configuration. We include all 13 kernels that exercise memory access behavior, covering 1-D and 2-D store patterns, column and row reads, diagonal access, single-thread scalar store, and block-level reductions. These kernels perform minimal arithmetic relative to their memory traffic, making them a challenging case for instruction-count models and a useful probe of where static energy estimates break down.

SGEMM Optimisation Variants Following the CUDA matrix multiplication worklog by S. Boehm [25], we include 14 SGEMM kernels, each implementing a distinct optimisation technique: naive global-memory access, coalesced access, shared-memory blocking, 1-D and 2-D thread-block tiling, vectorised loads (`float4`), bank-conflict resolution, autotuned tile sizes, warp tiling, and single- and double-buffered pipelines. Collectively these kernels span a continuous progression from memory-bound to compute-bound behavior, with arithmetic intensity increasing at each step.

Simple Benchmarks

Ten hand-crafted kernels targets specific microarchitectural regimes that random sampling is unlikely to cover densely. Each kernel runs a loop with iterations in interval of 10^7 to 10^8 iterations at a fixed configuration of 256 threads per block and 4 blocks per SM keeping occupancy high. The benchmarks cover pure FP32, FP64, and INT32 compute pipelines in isolation; FP32+INT32 co-issue on the dual datapaths of the NVIDIA Ada Lovelace architecture; 50% warp divergence; a compile-time arithmetic intensity sweep (1 to 64 FP32 operations per global load)); coalesced versus fully strided (512 B stride) global memory access; register-pressure-induced occupancy reduction (32 live FP32 registers, ≈37% SM occupancy on AD103); and shared-memory bank conflicts. The benchmarks and their targets are listed in Table 3 in Appendix C.

FlipFlop Kernels. A FlipFlop Multi-Head Attention (MHA) kernel [26], representative of transformer inference workloads.

Summary The validation set comprises 144 kernels from five sources, summarised in Table 1: 106 NPB kernels spanning CFD-derived algorithms, 13 UBench memory-

access microbenchmarks, 14 SGEMM optimisation variants, 10 hand-crafted microarchitectural benchmarks, and 2 FlipFlop kernels representative of transformer inference workloads.

Suite	Problem class	Kernels
BT — Block Tri-diagonal solver	S (12^3 grid)	19
CG — Conjugate Gradient	S (NA = 1400)	13
EP — Embarrassingly Parallel	S (M = 24)	1
FT — Discrete 3-D FFT	S ($64 \times 64 \times 64$)	14
IS — Integer Sort	S (2^{16} keys)	11
LU — LU factorisation	S (12^3 grid)	23
MG — Multi-Grid solver	S (32^3 grid)	9
SP — Scalar Penta-diagonal	S (12^3 grid)	16
UB — uBench	—	13
SGEMM — SGEMM Optimization Variants	—	14
SB — Simple Benchmarks	—	10
MHA — FlipFlop Multi-Head Attention	-	1

Table 1: NPB CUDA kernels used for validation at problem class S.

5.2.1 Microbenchmarks

The linear model assigns a scalar energy weight to each PTX instruction; these weights are derived from single-instruction calibration kernels run at a fixed launch configuration of a grid with 152 blocks where each block has 1024 threads. This configuration places exactly two blocks on each of the 80 SMs of the target GPU, maximizing SM occupancy. Seventeen instruction templates are calibrated, spanning integer arithmetic (`add.s32`, `add.s64`, `mul.lo.s32`), floating-point arithmetic (`add.f32`, `mul.f32`, `fma.rn.f32`), data movement (`mov.b32`, `mov.u32`, `mov.f32`), type conversion (`cvt.s64.s32`), shift (`shl.b64`), predicate operations (`setp.lt.s32`, `not.pred`, `mov.pred`), global memory (`ld.f32`, `st.f32`), and control flow (`bra`, `bra.uni`). A full list of all the generated instructions can be seen in Table 4 (Appendix G).

5.3 Experimental Setup

Our experiments were conducted on a system equipped with an NVIDIA RTX 4080 Super GPU (16 GB VRAM) and an Intel Core Ultra 7 265KF CPU with 64 GB DDR5-6000 memory. Power consumption of the GPU was monitored via a PMD2 power meter connected through the motherboard's USB-C port on Ubuntu 24.04.4 LTS with CUDA 13.2 and Python 3.12.3. We ensure NVCC compiles without optimizations (`-O0` flag) and retaining debug information (`-G` flag) required by the PTX Analyzer such that the PTX assembly reflects the source code structure. The `--keep` flags outputs the PTX assembly code that the PTX Analyzer uses.

6 Models

We propose three models of increasing complexity. The second model builds on insights from the previous, following an iterative refinement process in which a model is constructed, evaluated, and extended to address its limitations. The predictive accuracy of each model is evaluated in evaluation in Section 7.

6.1 Model A - Static Instruction Cost Model

Building on the instruction cost model of Y.-T. S. Li *et al.* [4], we propose the Static Instruction Cost Model (ICM), which assigns each PTX instruction a scalar energy weight estimated from the training kernels described in Section 5.1. The total kernel energy is then:

$$E_{\text{est}} = \sum_{i}^{N} w_i \cdot c_i \tag{8}$$

where w_i is the energy weight of instruction class i and c_i is the execution count of that instruction. The instruction count c_i was obtained using the PTX analyzer in Section 4.4.

6.1.1 A.BS - Baseline-Subtraction

To obtain a per-instruction energy weight, the target instruction must be isolated from the overhead introduced by the kernel itself. To isolate the instructions, we embed PTX directly via inline assembly blocks within the CUDA C++ kernels. We estimate the energy attributed to a single instruction type by the following subtraction method:

$$w_i = \frac{E_{\text{kernel+instr}} - E_{\text{baseline}}}{N \cdot T} \tag{9}$$

where $E_{\text{kernel+instr}}$ is the measured energy of a kernel containing N copies of the target instruction unrolled in a loop of T iterations, and E_{baseline} is the energy of an structurally identical kernel running the same loop of T iterations with an empty body and same launch configuration. The baseline kernel E_{baseline} is constructed as seen in Listing 4:

```
1  for (int i = 0; i < ITERATIONS; ++i) { /* empty */ }
```

Listing 4: Loop with empty body

The instrumented kernel inserts the target instruction inside the loop body via inline assembly:

```
1  for (int i = 0; i < ITERATIONS; ++i) {
2      asm volatile("mov.u32 %0, %0;\n\t" : "+r"(tmp));
3  }
```

Listing 5: target instruction injected via inline assembly

This method is suited to instructions whose energy contribution is independent of data values and memory state. That is, register-only instructions with deterministic execution cost. Instructions measured via this method include arithmetic families (`add`, `mul`, `fma`, `mad`), type conversions (`cvt`), logical operations (`and`, `or`, `xor`, `not`), shift operations (`shl`, `shr`), and predicate operations (`setp`, `selp`).

6.1.2 A.DR - Differential-Repeat

A challenge when estimating per-instruction energy is that many PTX instructions cannot be measured in complete isolation. Adding an instruction often introduces supporting code, making it difficult to separate the energy of the instruction itself from the energy of its setup. To address this, we compare two otherwise identical kernels: one containing N copies of the target instruction and one containing $N+1$ copies. Because all other instructions and overheads are shared between the kernels, their contributions largely cancel out. In contrast, the Baseline Subtraction method relies on subtracting an empty-loop baseline from an instrumented kernel, which assumes that the baseline accurately captures all non-target energy costs. The per-instruction energy weight reduces to:

$$w_i = \frac{E_{\text{kernel(N+1)}} - E_{\text{kernel(N)}}}{T} \tag{10}$$

In practice we use $N = 1$ meaning the baseline contains a single copy of the target instruction rather than an empty loop, and the instrumented kernel contains exactly one additional copy.

6.2 Model B - Instruction Frequency Network

Rather than isolating individual instructions, we propose a neural network model that takes as input a feature vector characterizing the full PTX instruction composition of each kernel, obtained directly from the PTX analyzer. These instruction occurrence estimates are then aggregated into a feature vector that characterizes the kernel's instructions. For a kernel k, let $\mathcal{I} = \{i_1, i_2, ..., i_n\}$ denote the complete PTX instruction set of size n. We define the feature vector $\boldsymbol{x}^k \in \mathbb{R}^n$ as:

$$x_j^k = \frac{c_j^k}{C^k} \tag{11}$$

where c_j^k is the occurrence count of instruction i_j in kernel k, and C^k is the total instruction count reported by the analyzer.

The training label for kernel k is the measured kernel energy E^k in Joules. To produce a normalized per-thread-instruction energy estimate that is comparable across kernels of varying size and launch configurations, the target is defined as:

$$y^k = \lambda \cdot \frac{E^k}{N^k \cdot C^k} \tag{12}$$

where $N^k = G_x \cdot G_y \cdot G_z \cdot B_x \cdot B_y \cdot B_z$ is the total number of threads launched across the CUDA grid and block dimensions, and $\lambda = 10^{11}$ is a scaling constant to avoid floating-point precision limitations during training. This value was determined empirically through iterative experimentation.

At inference time, the predicted kernel energy consumption is found by inverting the normalization:

$$\hat{E}^k = \frac{\hat{y}^k}{\lambda} \cdot N^k \cdot C^k \tag{13}$$

where $\hat{y}^k$ is the prediction by the neural network for a kernel k.

The model thus learns to predict energy per-thread-instruction, which is then scaled by the total thread-instruction count to recover the absolute energy for the kernel.

Activation Function Initial experiments used the Rectified Linear Unit (ReLU) [27]:

$$\sigma_{\text{ReLU}}(z) = \max(0, z) \tag{14}$$

This proved ineffective for two reasons: (1) difficulty capturing non-linear relationships between instruction mix and power consumption, and (2) the dying ReLU problem [28], where neurons receiving negative pre-activations produce zero gradient and cease to update, stalling optimization.

To address this, Leaky ReLU [29] was adopted instead:

$$\sigma_{\text{LReLU}}(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha z & \text{if } z \leq 0 \end{cases} \tag{15}$$

By introducing a small negative slope α for negative activations, Leaky ReLU preserves a non-zero gradient throughout the input domain, preventing neuron death while retaining the piecewise-linear expressiveness of ReLU [30].

7 Evaluation

This section evaluates the proposed static PTX-based energy estimation approach. The evaluation is structured around the three research questions introduced in Section 1. First, we investigate whether meaningful energy differences can be observed between individual PTX instructions. Second, we evaluate how accurately kernel energy consumption can be predicted using instruction-level energy weights and an instruction-frequency neural network. Finally, we examine how prediction accuracy varies across different kernel types and workloads.

The objective of the evaluation is not to demonstrate a highly accurate predictor, but rather to assess the extent to which information available from static PTX analysis is sufficient to explain and estimate GPU energy consumption. The validation suite consists of 144 kernels spanning scientific computing, memory-access microbenchmarks, matrix multiplication optimizations, transformer-inspired workloads, and synthetic stress tests, providing a diverse set of execution behaviors against which the models can be assessed as referenced in Section 5: Benchmarks.

7.1 RQ1: *What is the energy consumption associated with individual PTX instructions in CUDA kernels?*

Figure 4 presents the 15 lowest and 15 highest energy-consuming PTX instructions measured across our benchmark suite. The least energy-intensive instruction, `mov_u32`, con-

sumes approximately 2.1×10^{-11} Joules, while the most expensive, `div.rn.f64`, consumes approximately 7.2×10^{-9} Joules — a difference of roughly 343 times, or approximately two orders of magnitude.

The results reveal a clear bifurcation in instruction-level energy consumption: the vast majority of instructions exhibit energy on the order of 10^{-8} Joules, whereas a small subset of division and remainder instructions incur energy expenditures several orders of magnitude greater. This disparity is consistent with known GPU microarchitectural behavior [22], [23]. Simple arithmetic instructions such as integer additions and moves map to a single pipeline stage with minimal latency, whereas division and remainder operations require iterative computation across special function units (SFUs), resulting in substantially higher per-instruction energy costs. The complete distribution across all measured instructions is provided in Figure 14 (Appendix H). These per-instruction energy measurements form the static weights used in the analysis presented in RQ2.

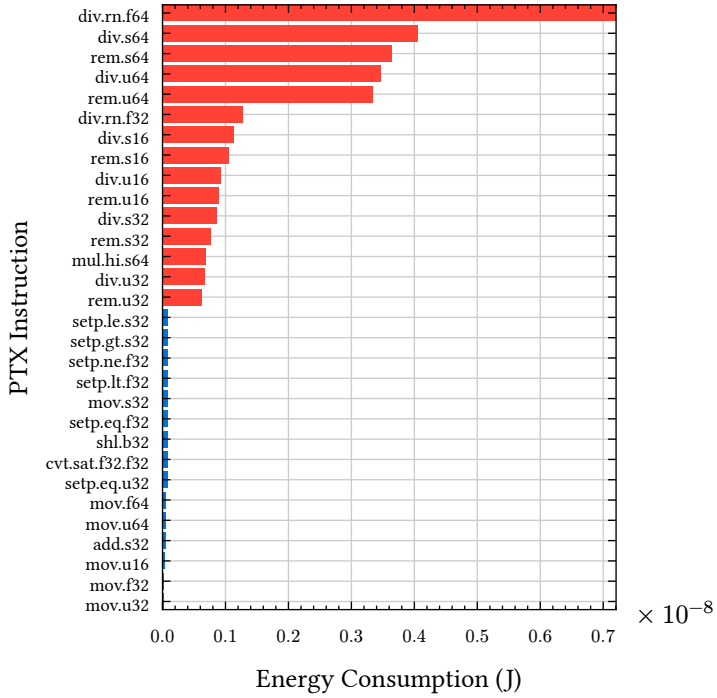


Figure 4: Top 15 highest energy and bottom 15 lowest energy consumption of PTX instructions

7.2 RQ2: *How accurately can the energy consumption of CUDA kernels be predicted by assigning static energy weights to PTX instructions?*

Three models were evaluated against the 144-kernel validation suite described in Section 5: Model A.BS (Baseline-Subtraction), Model A.DR (Differential-Repeat), and Model B (Instruction Frequency Network). Results are reported across MAE (Mean Absolute Error), RMSE (Root Mean Square Error), and MAPE (Mean Absolute Percentage Error). These results are presented in Table 2 and visualized in Figure 8.

Metric	Model A.BS	Model A.DR	Model B
MAE	2028.3 J	2068.6 J	1599.6 J
RMSE	4811.1 J	4932.9 J	4155.2 J
MAPE	90.3%	91.2%	74.1%
Median APE	90.8%	90.7%	90.8%
Within 10%	0%	1.4%	4.8%
Within 25%	7.5%	7.5%	13.7%
Within 50%	12.2%	12.2%	25.3%

Table 2: Comparison of prediction accuracy metrics across all models

7.2.1 Model A.BS - Baseline-Subtraction

Figure 5.A and Figure 5.B show the prediction error of Model A.BS across the validation suite. The model exhibits a strong tendency to under-predict kernel energy consumption, with most kernels producing negative prediction errors and only a limited number of significant over-predictions. This suggests that assigning a single static energy cost to each PTX instruction fails to capture a substantial portion of the energy consumed during execution.

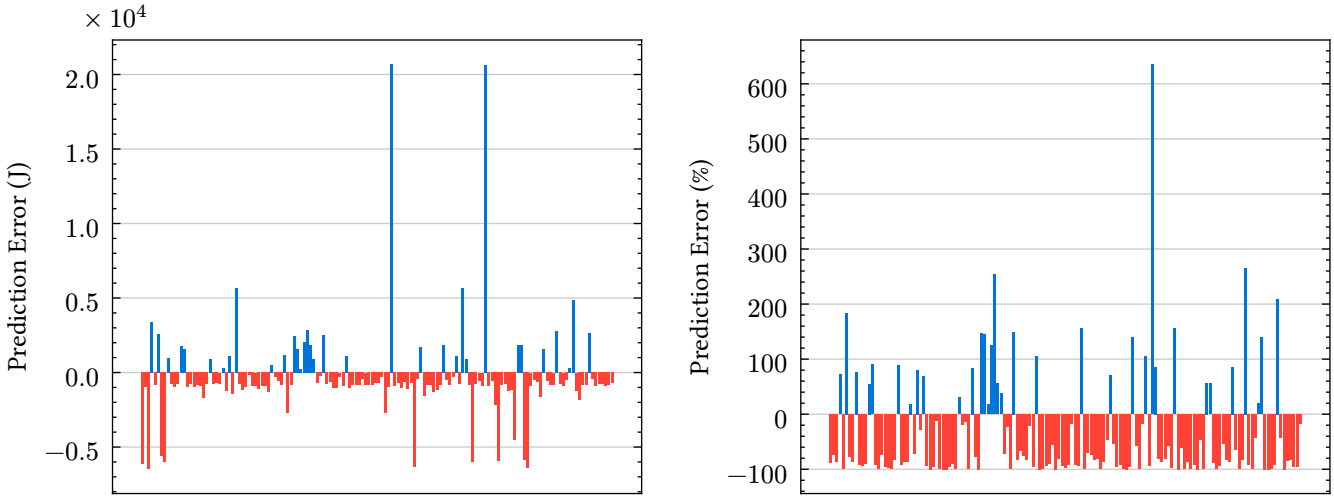
Several kernels exhibit extremely large percentage errors, exceeding 500% in some cases. These outliers indicate that the relationship between PTX instruction counts and measured energy consumption is not purely additive. Factors such as memory-system behavior, cache utilization, occupancy, instruction scheduling, and hardware resource contention are not represented by the model and therefore cannot be reflected in the prediction.

Across the full validation suite, Model A.BS achieves a MAE of 2028.3 J, an RMSE of 4811.1 J, and a MAPE of 90.3% as shown in Table 2. Furthermore, none of the evaluated kernels fall within a 10% error margin, and only 12.2% are predicted within 50% error. These results indicate that although the baseline-subtraction method can derive instruction-level energy weights, the resulting static instruction cost model is insufficient for accurate kernel-level energy estimation.

7.2.2 Model A.DR - Differential-Repeat

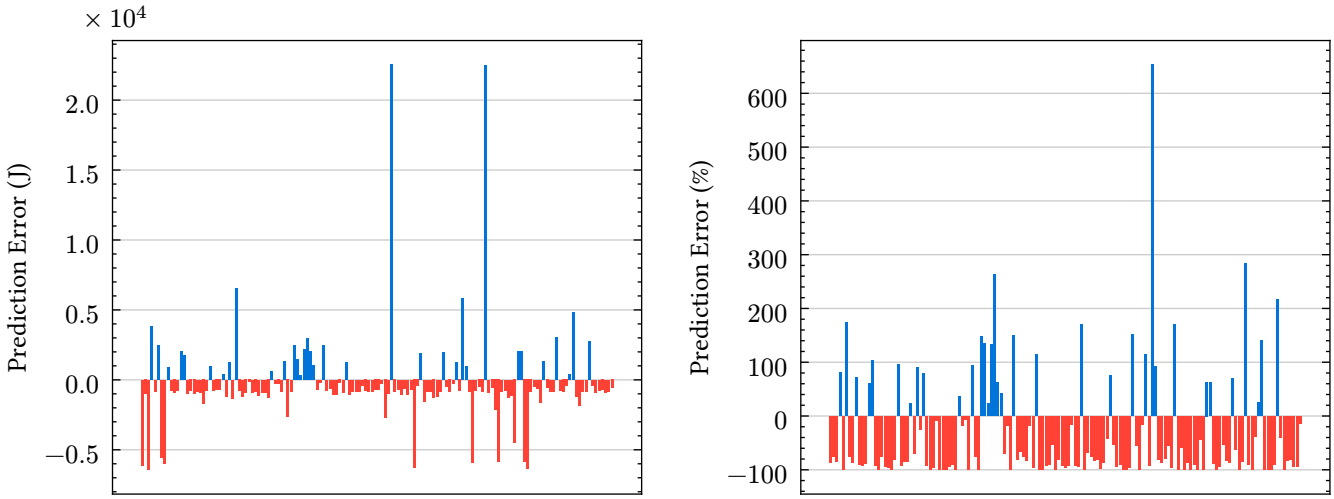
The Differential-Repeat approach was introduced to reduce measurement noise and eliminate kernel overhead that may influence the Baseline-Subtraction method. However, Figure 6.A and Figure 6.B demonstrate that the resulting prediction accuracy remains largely unchanged.

Like Model A.BS, Model A.DR systematically under-predicts energy consumption for most kernels while occa-



(A) Prediction error in Joules. **(B)** Prediction error in percentage.

Figure 5: Prediction error for Model A.BS - Baseline-Subtraction



(A) Prediction error in Joules. **(B)** Prediction error in percentage.

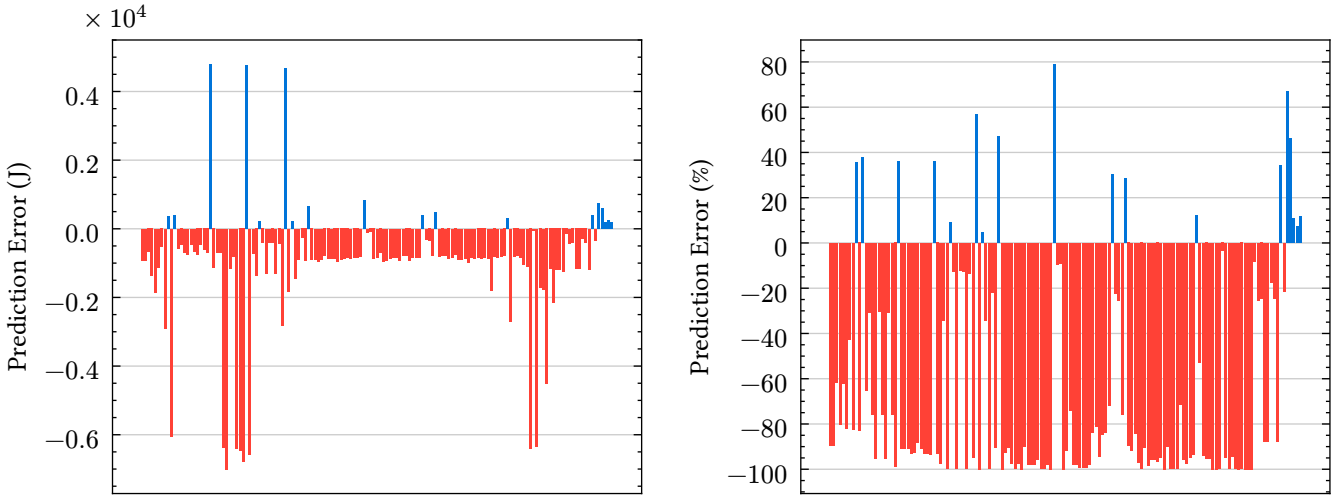
Figure 6: Prediction error for Model A.DR - Differential Repeat

sionally producing very large overestimations. The overall error distribution closely resembles that of the baseline-subtraction model, suggesting that the dominant source of prediction error is not the calibration procedure itself but rather the underlying assumption that kernel energy can be expressed as a linear combination of static instruction costs.

Model A.DR achieves a MAE of 2068.6 J, an RMSE of 4932.9 J, and a MAPE of 91.2%, making it marginally worse than Model A.BS on most metrics as shown in Table 2. The similarity between the two models indicates that improving the estimation of instruction weights alone does not significantly improve prediction accuracy. Instead, the results suggest that important contributors to energy consumption exist beyond instruction frequencies and static per-instruction costs.

7.2.3 Model B — Instruction Frequency Network

Model B consistently outperforms both static instruction-cost models across all reported metrics. In Figure 7 we see that the model still largely under predicts. While it did manage to remove the outliers and even some of the over predictions have now become under predictions. Table 2 also shows the MAE decreases from approximately 2.0 kJ to 1.6 kJ, RMSE decreases from roughly 4.9 kJ to 4.2 kJ, and MAPE improves from around 90% to 74%. While these improvements demonstrate that non-linear relationships between instructions contain predictive information, the absolute prediction errors remain high. The median absolute percentage error remains approximately 91%, and only 25.3% of kernels are predicted within a 50% error margin. Consequently, the neural network improves upon the static



(A) Prediction error in Joules. **(B)** Prediction error in percentage.

Figure 7: Prediction error for Model B - Instruction Frequency Network

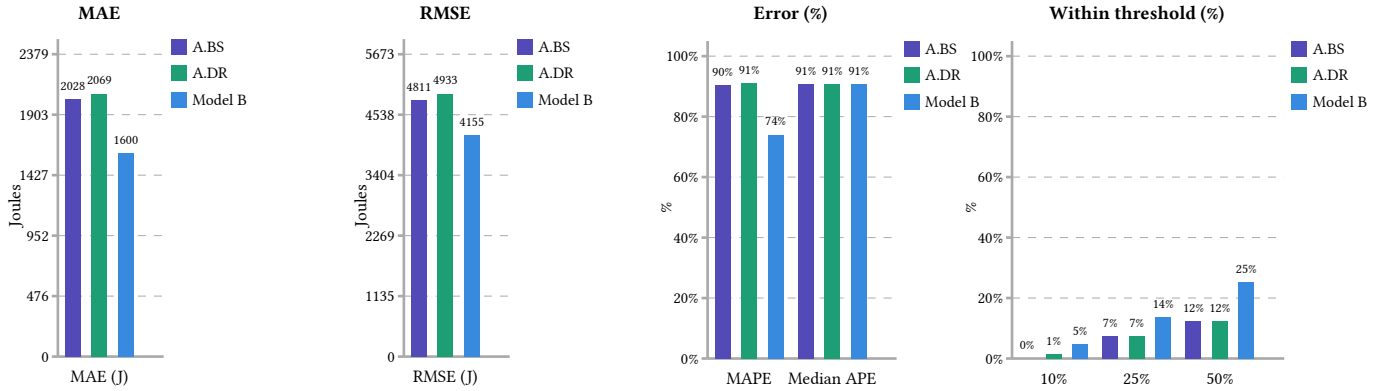


Figure 8: Model comparison across all evaluation metrics

weighting approaches but still fails to provide practically useful energy estimates.

7.2.4 Summary

The results show that prediction accuracy depends strongly on workload type. While the overall prediction accuracy remains poor, the errors are not randomly distributed across kernels. Instead, kernels belonging to the same benchmark family often exhibit similar prediction behavior.

For example, the vector addition kernels are predicted relatively accurately and consistently across all tested input sizes, with errors between approximately 7% and 11%. Similarly, some `FT` (referenced in Table 1) kernels are predicted with comparatively low error, including evolve (≈5%) and both `cffts2_kernel_1` and `cffts3_kernel_1` (≈13%). However, this behavior is not consistent across the entire FT benchmark family, where several kernels still exhibit large prediction errors.

In contrast, many other workloads are predicted poorly. Highly optimized SGEMM kernels, transformer workloads, and several memory-intensive kernels frequently exhibit errors of an order of magnitude or more. These workloads are influenced by runtime effects that are not represented by PTX instruction frequencies alone.

Overall, the results suggest that static PTX analysis captures some workload-specific characteristics and can provide reasonable estimates for certain kernels. However, the large errors observed across many benchmark families indicate that PTX instruction frequencies alone are insufficient for accurate energy prediction across diverse CUDA workloads.

7.3 RQ3: *How does the accuracy of static PTX-based energy estimation vary across different CUDA kernel types and workloads?*

As shown in Figure 9 (Model A.BS) and Figure 10 (Model A.DR), varying the approach used to estimate the weights did not significantly affect the prediction error across different workloads. Furthermore, several kernels within the same benchmark family exhibit comparable error magnitudes. Similarly, in Figure 11, the vector addition kernels and FT show comparable prediction errors across variations for Model B, suggesting that the model captures the scaling behavior of these workloads reasonably well.

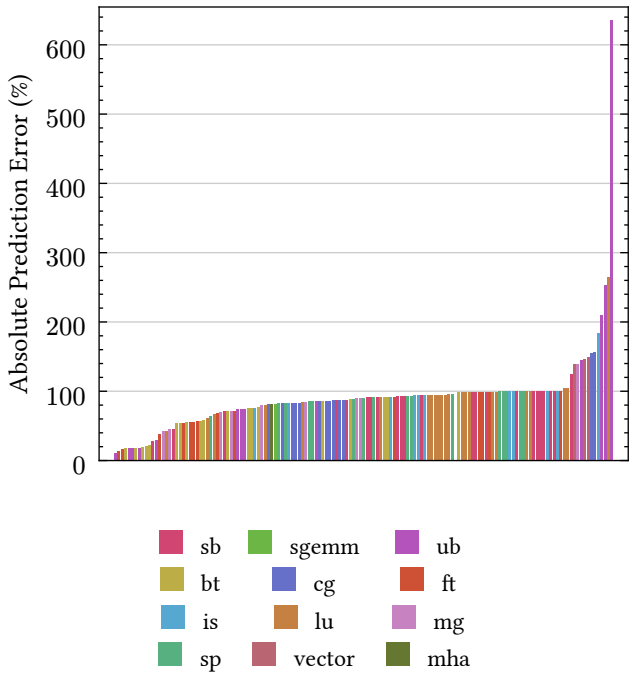


Figure 9: Absolute prediction error (%) for Model A.BS - Baseline Subtraction

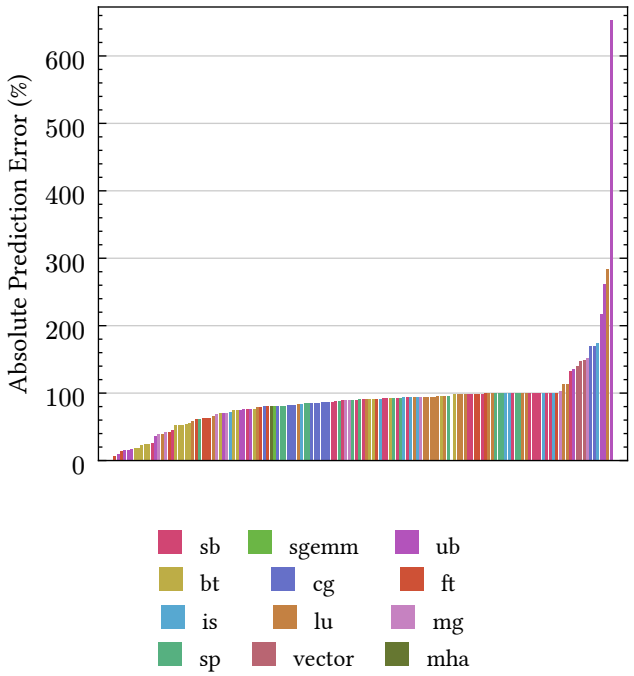


Figure 10: Absolute prediction error (%) for Model A.DR - Differential Repeat

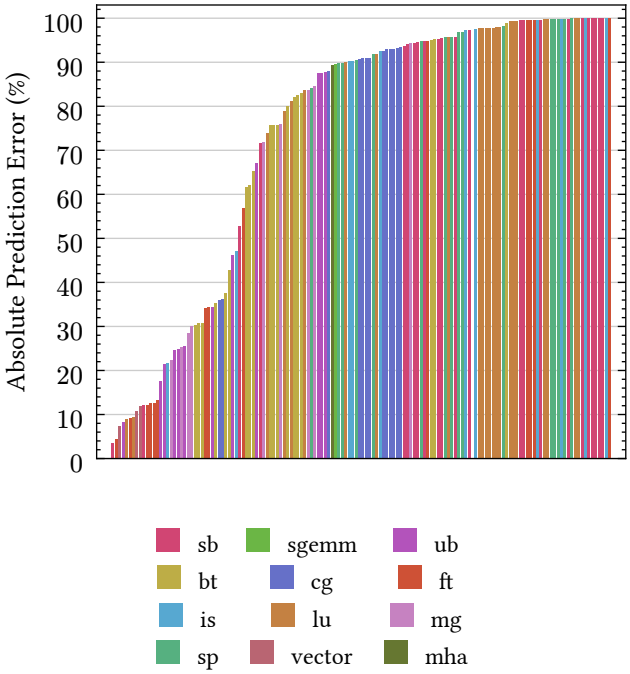


Figure 11: Absolute prediction error (%) for Model B

However, these successes are not representative of the validation suite as a whole. Many benchmark families still exhibit large prediction errors, indicating that PTX instruction frequencies alone do not capture all factors influencing energy consumption. Runtime effects such as memory behavior, cache utilization, occupancy, and instruction scheduling are not visible in the static PTX representation and likely contribute to the observed inaccuracies.

Overall, the results suggest that static PTX analysis contains some predictive information and can provide consistent estimates for certain workload classes, but the approach does not generalize reliably across the diverse set of CUDA kernels evaluated in this study.

8 Related Work

To address the limitations of built-in sensors, the custom-made PowerSensor3 [31] has been introduced as an external measurement solution, which unfortunately is not available off the shelf. It operates using a modified PCIe riser card equipped with a PCIe sensor module for measuring power delivered through external connectors, as well as two additional modules for capturing the 3.3 V and 12 V power supplied via the PCIe slot enabling measurement of total GPU power consumption covering both slot and external inputs. As PowerSensor3 is not commercially available, we instead use the PMD2, which offers a comparable external measurement approach and is available off the shelf.

Early work by S. Hong *et al.* [23] accurately characterized GPU application performance and proposed a power model for the GTX280 (Tesla microarchitecture), achieving highly accurate predictions through analysis of PTX binary and device pipeline at runtime. However, the model could not generalize to different core or memory configurations. Building on this, J. Leng *et al.* [22] integrated S. Hong's model into GPGPU-Sim to create GPUWattch, enabling cycle-level GPU power estimation during execution with support for Tesla and Fermi microarchitectures. Despite

their accuracy, such cycle-level models are too slow for runtime application and require runtime measurements to function.

A second family of approaches relies on hardware performance counters sampled at runtime via tools such as NVML [10] or NVIDIA Nsight Compute [9] to build lightweight power models. While such models achieve high accuracy with lower overhead than cycle-level simulation, prior work by Z. Yang *et al.* [11] has identified significant inaccuracies in NVML power readings, leading to both underestimation and overestimation of actual energy consumption. Counter-based models still require at least one complete execution of the target application and access to hardware instrumentation, making them unsuitable for pre-execution energy estimation.

A third family of approaches uses static analysis of compiled GPU code to avoid runtime measurement altogether. Several recent works tackle this by modeling directly from PTX kernel code: FlipFlop [26] statically analyses PTX assembly to predict energy consumption and recommend Pareto-optimal thread block configurations balancing power and execution time; [32] similarly operates on PTX using recurrent neural networks; and J. Guerreiro *et al.* [33] extracts PTX-level features to train machine learning models for program-level energy estimation. The majority of research on GPU modeling requires at least one execution of the application to obtain predictions. As with the approach presented herein, these works instrument CUDA code to obtain the number of executed instructions in PTX.

GPU Kernel Optimization GPU kernel optimization requires detailed understanding of GPU architectures, execution models, and memory hierarchies [34]. When writing GPU kernels, developers must carefully select execution parameters such as thread block sizes, grid dimensions, loop unrolling factors, and tiling strategies. Kernel Tuner [34] performs runtime profiling to find optimal configurations.

Comparison with existing work While several existing GPU energy modeling approaches rely on runtime hardware counter sampling or repeated kernel execution to characterize workload behavior, *we leverage static PTX analysis to predict energy consumption without kernel execution*, thereby eliminating profiling overhead during feature extraction.

A further distinction concerns the accuracy of power measurement. Prior work [11], [22], [26], [32], [33] predominantly targets data center GPUs and relies on NVML or a hardware-based for power readings. However, Z. Yang *et al.* [11] demonstrated that NVML introduces significant measurement inaccuracies, with errors that are particularly pronounced on consumer-grade GPUs such as the RTX 4080 Super used in this study. To avoid these inaccuracies, we employ the ElmorLabs PMD2 as an external inline power measurement device, capturing both PCIe slot and external connector power directly from the hardware. This yields more accurate ground-truth energy measurements than NVML-based approaches, providing a more reliable basis for model training and evaluation.

Existing Static Analysis Approaches A common static-analysis technique assigns weights to instructions or basic blocks in the Control Flow Graph (CFG) and aggregates them to estimate a target metric [4]. Approaches based on this principle were originally introduced for execution-time estimation and have been studied since at least 1995 [4]. The underlying idea of associating execution costs with program elements and aggregating these costs along feasible execution paths remains a central concept in modern worst-case execution-time (WCET) analysis, where weighted control-flow representations are used to derive safe execution-time bounds P. Wägemann *et al.* [35]

9 Threats to Validity

This section discusses factors that may influence the interpretation of the results and the conclusions drawn from this study. The identified threats outline limitations of the methodology and evaluation and provide context for assessing the scope and applicability of the findings.

9.1 Loop Bound Estimation

The PTX analyzer relies on loop annotations to estimate instruction execution counts. While this approach works well for simple kernels with fixed iteration counts, several kernels in the validation suite contain more complex control-flow structures and nested loops. In these cases, accurately estimating loop bounds was challenging, and the provided annotations may not perfectly reflect the actual runtime behavior. Any inaccuracies in the estimated iteration counts directly affect the instruction frequencies reported by the analyzer, which subsequently propagate to the energy predictions. As a result, a portion of the observed prediction error may stem from inaccuracies in the annotation.

9.2 Hardware Generalizability

The neural network model demonstrated improved prediction accuracy compared to the static instruction-cost models. However, all training and evaluation experiments were conducted on a single GPU architecture, namely the NVIDIA RTX 4080 Super. Consequently, the model may have learned characteristics that are specific to the microarchitectural properties, scheduling behavior, memory hierarchy, and power management mechanisms of that device. It is therefore unclear to what extent the observed performance would generalize to other GPU generations or vendors. Additional experiments across multiple GPU architectures would be required to determine whether the learned relationships between PTX instruction distributions and energy consumption remain stable across hardware platforms, or whether architecture-specific retraining is necessary.

10 Future Works

This section outlines potential directions for extending and improving the work presented in this study. While the proposed approach provides a foundation for static PTX-based energy estimation, several opportunities remain for further investigation and refinement.

10.1 Extending Annotation

Another limitation of the current PTX analyzer is its treatment of control flow. At present, when a conditional branch is encountered, both paths are weighted equally during static analysis. While this simplifies the analysis process, it assumes that each branch is executed with the same frequency, which is rarely the case in real-world workloads. As a result, kernels containing highly biased branches may have their instruction counts and energy estimates misrepresented.

Future work could extend the annotation system to support branch probability metadata. Similar to the existing loop annotations, this could provide an estimated execution distribution for conditional branches. For example, a branch annotation could specify that the true branch is expected to be taken 20% of the time and the false branch 80% of the time.

10.2 Model C.PI - Pairwise Instructions

An additional direction explored during this project was the development of a pairwise instruction model. The motivation behind this approach was to capture interactions between PTX instructions, as the energy consumption of a kernel is unlikely to be solely determined by the independent contribution of each instruction. Instead, certain instruction combinations may exhibit synergistic or antagonistic effects due to shared execution resources, pipeline behavior, or memory subsystem interactions.

Although preliminary experimentation was conducted, the model was not completed within the scope of this project due to time constraints. A major challenge encountered was the rapid growth of the training dataset required to represent instruction pairs. Under the current approach, every possible combination of instructions must be represented in the training data, causing the number of required training kernels to grow combinatorially with the size of the instruction set.

Future work could address this limitation by reducing the size of the pairwise search space. Rather than constructing a single model that considers all possible instruction pairs, a separate model could be trained for scaling different instruction variants or instruction classes, such as u32, u64, f32, and f64. By partitioning the instruction set into smaller groups, the number of pairwise combinations required for training would be significantly reduced, making the approach more tractable.

11 Conclusion

This paper investigated whether the energy consumption of CUDA kernels can be estimated through static PTX code analysis To answer this question, we first measured the energy consumption of individual PTX instructions and then evaluated two instruction-cost models and an instruction-frequency neural network across a validation suite of 144 CUDA kernels, annotated with our tooling. The results demonstrated substantial variation in energy consumption between PTX instructions, confirming that instruction-level characteristics contribute to overall energy usage. However, all evaluated models exhibited high prediction errors, indicating that instruction occurrences alone are insufficient for accurately estimating kernel energy consumption on modern GPUs.

These findings contrast with prior work on CPUs and embedded processors, where instruction-level energy models have achieved useful predictive accuracy[4]. We attribute this difference to the considerably greater complexity of modern GPU architectures, where factors such as memory hierarchy behavior, occupancy, instruction scheduling, parallel execution, and resource contention play a significant role in determining energy consumption. As a result, the relationship between static PTX code and measured energy is not adequately captured by models based solely on instruction frequencies or static instruction costs.

Although the proposed models did not achieve the level of accuracy required for practical energy prediction, the study contributes to understanding how far a purely instruction-occurrence-based approach can be taken for GPU energy estimation, and shows that for some simple kernels our model can predict within 10% indicating that this approach may have some potential. For meaningful GPU energy estimation the model will likely require richer representations of program behavior and hardware interaction than those available from instruction counts alone. We therefore conclude that static PTX analysis contains useful information about energy consumption, but that more sophisticated models are needed to effectively exploit it for accurate GPU energy prediction.

12 Acknowledgements

We would like to thank our supervisors, Bent Thomsen and Søren Kejser Jensen, for their valuable guidance in this research. We also gratefully acknowledge Aalborg University (AAU) for providing the necessary equipment.

This paper was reviewed and refined using generative AI tools to enhance clarity and readability. Generative AI was also utilized to generate code for the experimental studies.

References

[1] International Energy Agency, "Energy and AI: Energy Demand from AI." 2025.

[2] X. Chen *et al.*, "Electricity Demand and Grid Impacts of AI Data Centers: Challenges and Prospects." 2025.

[3] R. A. Bridges *et al.*, "Understanding GPU Power: A Survey of Profiling, Modeling, and Simulation Methods," *ACM Computing Surveys*, vol. 49, no. 3, pp. 1–27, 2016, doi: 10.1145/2962131.

[4] Y.-T. S. Li *et al.*, "Performance analysis of embedded software using implicit path enumeration," in *Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, 1995, pp. 88–98.

[5] Modal Labs, "GPU Glossary." [Online]. Available: https://modal.com/gpu-glossary

[6] NVIDIA Corporation, "CUDA C++ Programming Guide." [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[7] NVIDIA Corporation, "NVIDIA ADA GPU ARCHITECTURE." 2024. [Online]. Available: https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf

[8] W1zzard *et al.*, "NVIDIA GeForce RTX 4080 Super Founders Edition Review - Savings of $200." Accessed: Dec. 17, 2025. [Online]. Available: https://www.techpowerup.com/review/nvidia-geforce-rtx-4080-super-founders-edition/

[9] NVIDIA Corporation, "NVIDIA Nsight Compute." 2024.

[10] NVIDIA Corporation, "NVIDIA Management Library (NVML)." 2024.

[11] Z. Yang *et al.*, "Accurate and Convenient Energy Measurements for GPUs: A Detailed Study of NVIDIA GPU's Built-In Power Sensor," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, Atlanta, GA, USA: IEEE, Nov. 2024, pp. 1–17. doi: 10.1109/SC41406.2024.00028.

[12] ElmorLabs, "PMD2 Power Measurement Device."

[13] L. Fjellanger *et al.*, "Profiling to Power: Mapping CUDA Program Counters to Energy Use," technical report, 2025.

[14] ElmorLabs, "PMD PCI-E 4.0 Slot Power Measurement Adapter."

[15] K. H. Rosen, *Discrete mathematics and its applications*, Eighth edition. New York, NY: McGraw-Hill, 2019.

[16] F. E. Allen, "Control flow analysis," *ACM SIGPLAN Notices*, vol. 5, no. 7, pp. 1–19, Jul. 1970, doi: 10.1145/390013.808479.

[17] D. J. Kuck *et al.*, "Dependence graphs and compiler optimizations," in *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, in POPL '81. New York, NY, USA: Association for Computing Machinery, Jan. 1981, pp. 207–218. doi: 10.1145/567532.567555.

[18] J. M. P. Cardoso *et al.*, *Embedded computing for high performance: Efficient mapping of computations using customization, code transformations and compilation*. Morgan Kaufmann, 2017.

[19] B. Wysocki, "Tracy Profiler." 2024.

[20] B. Wysocki, "Tracy Profiler — TracyCUDA.hpp." 2024.

[21] "NAS Parallel Benchmarks." Accessed: May 18, 2026. [Online]. Available: https://www.nas.nasa.gov/software/npb.html

[22] J. Leng *et al.*, "GPUWattch: Enabling energy optimizations in GPGPUs," *ACM SIGARCH computer architecture news*, vol. 41, no. 3, pp. 487–498, 2013.

[23] S. Hong *et al.*, "An integrated GPU power and performance model," in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010, pp. 280–289.

[24] Y. Torres *et al.*, "uBench: exposing the impact of CUDA block geometry in terms of performance," *The Journal of Supercomputing*, vol. 65, no. 3, pp. 1150–1163, 2013.

[25] S. Boehm, "How to Optimize a CUDA Matmul Kernel for cuBLAS-like Performance: a Worklog." Accessed: Dec. 31, 2022. [Online]. Available: https://siboehm.com/articles/22/CUDA-MMM

[26] S. Rajput *et al.*, "FlipFlop: A Static Analysis-based Energy Optimization Framework for GPU Kernels." [Online]. Available: https://arxiv.org/abs/2601.13345

[27] V. Nair *et al.*, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[28] L. Lu *et al.*, "Dying relu and initialization: Theory and numerical examples," *arXiv preprint arXiv:1903.06733*, 2019.

[29] A. L. Maas *et al.*, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. icml*, 2013, p. 3.

[30] B. Xu *et al.*, "Empirical evaluation of rectified activations in convolutional network," *arXiv preprint arXiv:1505.00853*, 2015.

[31] S. van der Vlugt *et al.*, "PowerSensor3: A Fast and Accurate Open Source Power Measurement Tool." [Online]. Available: https://arxiv.org/abs/2504.17883

[32] G. Alavani *et al.*, "Program analysis and machine learning–based approach to predict power consumption of cuda kernel," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 8, no. 4, pp. 1–24, 2023.

[33] J. Guerreiro *et al.*, "GPU static modeling using PTX and deep structured learning," *IEEE Access*, vol. 7, pp. 159150–159161, 2019.

[34] B. van Werkhoven, "Kernel Tuner: A search-optimizing GPU code auto-tuner," *Future Generation Computer Systems*, vol. 90, pp. 347–358, 2019, doi: https://doi.org/10.1016/j.future.2018.08.004.

[35] P. Wägemann *et al.*, "Worst-case energy consumption analysis for energy-constrained embedded systems," in *2015 27th Euromicro Conference on Real-Time Systems*, 2015, pp. 105–114.

Appendix A: Automatic Parameters Extraction

To obtain the CUDA kernel launch configuration parameters, specifically `gridDim` and `blockDim`, we investigated the use of `lli`, the LLVM JIT execution engine, as a means of dynamically evaluating launch configuration expressions at runtime. This approach was motivated by cases in which the launch parameters are not compile-time constants but are instead determined by variable expressions whose values can only be resolved during execution. By intercepting program execution at the point of the kernel launch, we sought to capture the fully evaluated parameter values as they would be passed to the CUDA runtime.

Appendix B: Pilot

To achieve the target execution time required for efficient measurements, an initial "pilot" model was evaluated. This approach executed each kernel using two different problem sizes and used the resulting execution times to project the problem size required to reach a desired execution duration.

In practice, this method proved ineffective because the execution time of many kernels did not exhibit a linear relationship with problem size. A likely explanation is that kernel launch configurations often change as problem size increases. Such changes can alter the degree of parallelism by increasing the number of concurrently executing threads, thereby affecting execution time in a non-linear manner. As a result, simple linear extrapolation based on two measurements was frequently unable to accurately predict the problem size required to achieve the target execution time.

Appendix C: Simple Benchmarks

Kernel	Targeted effect
`sb_fp32_pure`	FP32 FMAD/FADD/FMUL chain; baseline compute power
`sb_fp64_pure`	FP64 arithmetic; 1:64 throughput ratio on AD103
`sb_int32_pure`	INT32 IMAD/IADD/bitwise; integer pipe isolation
`sb_fp32_int32_mix`	50/50 FP32+INT32 co-issue on separate datapaths
`sb_divergence_50pct`	50% warp divergence; equal work on both branch paths
`sb_arith_intensity`	Arithmetic intensity sweep (`FLOPS_PER_LOAD` = 1–64)
`sb_gmem_coalesced`	Sequential global reads; 4 MB working set above L2
`sb_gmem_strided`	Strided reads (512 B between warp-adjacent threads)
`sb_reg_pressure`	32 live FP32 regs $\approx 37\%$ SM occupancy on AD103
`sb_smem_pressure`	12 KB shared memory/block with alternating bank conflicts

Table 3: Stress benchmarks and the microarchitectural effect each targets.

Appendix D: CUDA-bindings

To help with automation of the experimentation a Python library was created. The library named CUDA-bindings consists of three components, executor, analyzer, power. The executor allows one to execute a program passed as a string or as a path to a CUDA file, and export the variables to Python. Each execution could also optionally be cached, where a hash would be made of the file reducing repetitive workloads. The analyzer consisted of bindings for the PTX Analyzer written in Rust and allowed feature extraction and manipulation to be done in Python. The power component used the exported synchronization timestamps to estimate the power consumption of a kernel.

Appendix E: Machine Setup

The experimental platform was identical to that used in our previous work. It comprised an Intel Core Ultra 7 265KF CPU, an NVIDIA GeForce RTX 4080 SUPER GPU, and 64 GB of DDR5-6000 memory, running Ubuntu 24.04 LTS Server Edition with CUDA 13.2 and NVIDIA driver version 580.105.08[13] For setup of the software/bench-

marks see https://github.com/AAU-P9/aau-p10-ptx-energy/blob/master/README.md
CFG – **Control Flow Graph** 4, 4, 5, 5, 5, 6
CUPTI – **CUDA Profiling Tools Interface** 5
DDG – **Data Dependency Graph** 4
GPC – **GPU Processing Clusters** 3, 3, 3, 3, 3, 3, 3
ICM – **Static Instruction Cost Model** 8
NVCC – **NVIDIA CUDA Compiler** 4, 6, 6, 8
NVML – **NVIDIA Management Library** 4
PCIe – **Peripheral Component Interconnect Express** 3
PTX – **Parallel Thread Execution** 3, 4, 4, 5, 5, 5, 5, 6
SM – **Streaming Multiprocessor** 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 7, 18
TPC – **Texture Processor Cluster** 3, 3, 3, 3
V-ISA – **Virtual Instruction Set Architecture** 5

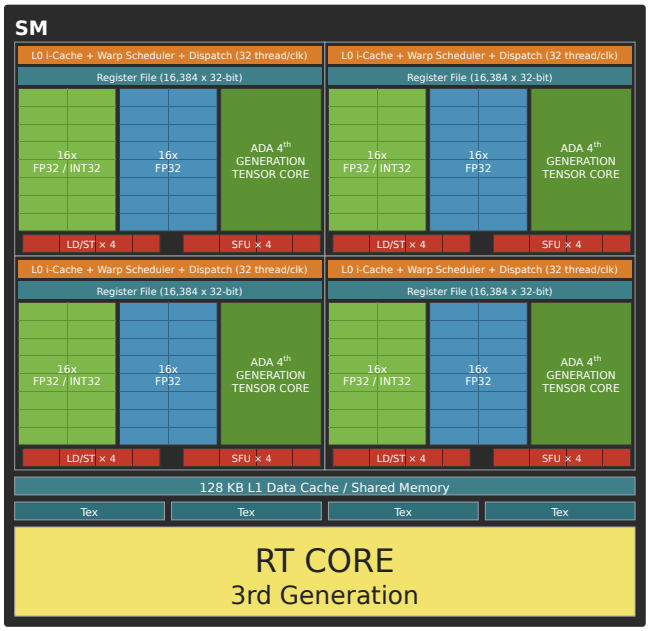


Figure 12: Full overview of an NVIDIA Ada Lovelace SM architecture in the GeForce RTX 4080 Super. [7]“

Appendix F: Full Size Block Diagram of RTX 4080 Super

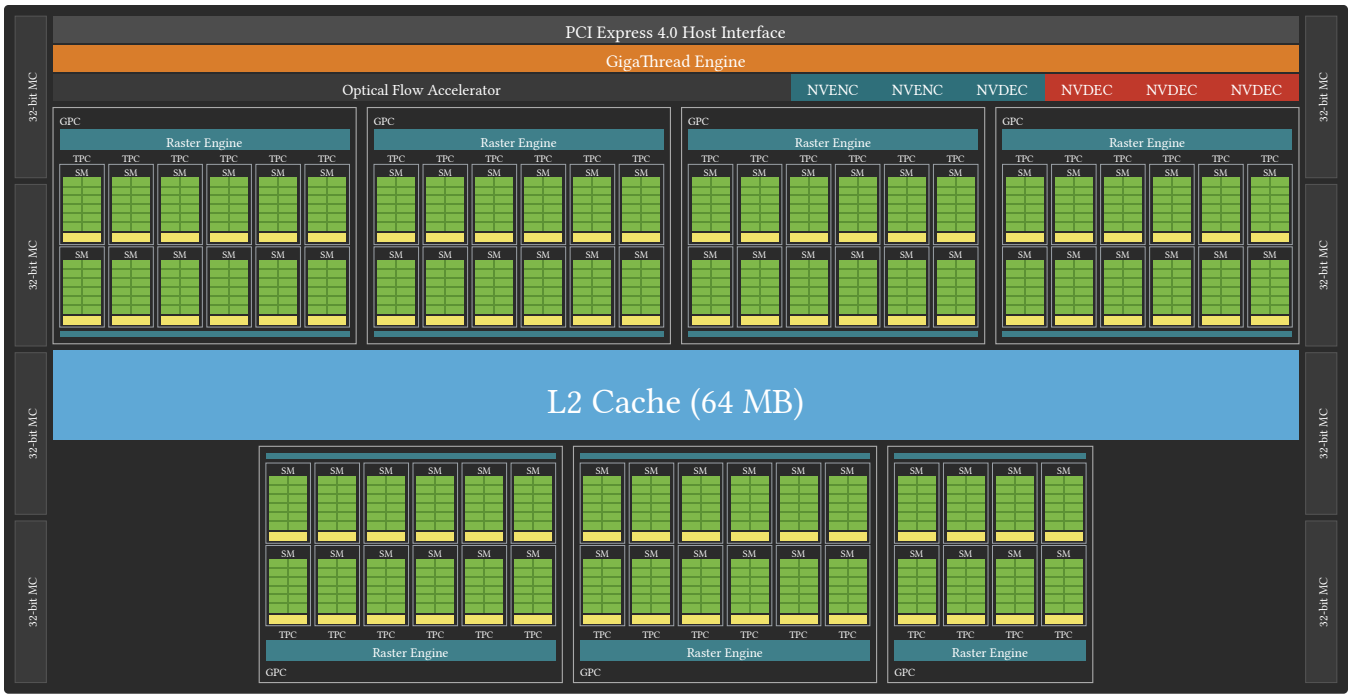


Figure 13: Simplified block diagram of an AD103-class GPU as configured for the RTX 4080 Super, showing 11 enabled GPCs (one cut down to 4 TPCs), 12 memory controllers, and the unified L2 cache [7], [8]

Appendix G: Auto-generated Instruction Kernels

add.f32	add.f64	add.s16	add.s32	add.s64
add.u16	add.u32	add.u64	bar.sync.s32	bra.s32
bra.uni.s32	cvt.s64.s32	cvt.sat.f32.f32	cvt.u64.u32	cvt.widen.s32
cvt.widen.u32	cvta.const.u64	cvta.local.u64	cvta.shared.u64	div.rn.f32
div.rn.f64	div.s16	div.s32	div.s64	div.u16
div.u32	div.u64	ex2.approx.ftz.f32	fma.rm.f32	fma.rm.f64
fma.rn.f32	fma.rn.f64	ld.f32	ld.f64	ld.s16
ld.s32	ld.s64	ld.s8	ld.u16	ld.u32
ld.u64	ld.u8	mov.b16	mov.b32	mov.b64
mov.f32	mov.f64	mov.pred.s32	mov.s16	mov.s32
mov.s64	mov.u16	mov.u32	mov.u64	mul.f32
mul.f64	mul.hi.s32	mul.hi.s64	mul.hi.u32	mul.hi.u64
mul.lo.s32	mul.lo.u32	neg.f32	neg.f64	neg.s16
neg.s32	neg.s64	not.pred.s32	rem.s16	rem.s32
rem.s64	rem.u16	rem.u32	rem.u64	setp.eq.f32
setp.eq.f64	setp.eq.s16	setp.eq.s32	setp.eq.s64	setp.eq.u16
setp.eq.u32	setp.eq.u64	setp.ge.f32	setp.ge.f64	setp.ge.s16
setp.ge.s32	setp.ge.s64	setp.ge.u16	setp.ge.u32	setp.ge.u64
setp.gt.f32	setp.gt.f64	setp.gt.s16	setp.gt.s32	setp.gt.s64
setp.gt.u16	setp.gt.u32	setp.gt.u64	setp.le.f32	setp.le.f64
setp.le.s16	setp.le.s32	setp.le.s64	setp.le.u16	setp.le.u32
setp.le.u64	setp.lt.f32	setp.lt.f64	setp.lt.s16	setp.lt.s32
setp.lt.s64	setp.lt.u16	setp.lt.u32	setp.lt.u64	setp.ne.f32
setp.ne.f64	setp.ne.s16	setp.ne.s32	setp.ne.s64	setp.ne.u16
setp.ne.u32	setp.ne.u64	shfl.sync.down.b32	shl.b16	shl.b32
shl.b64	st.f32	st.f64	st.s16	st.s32
st.s64	st.s8	st.u16	st.u32	st.u64
st.u8	sub.f32	sub.f64	sub.s16	sub.s32
sub.s64	sub.u16	sub.u32	sub.u64	

Table 4: A list of each auto-generated instruction kernel

Appendix H: PTX Instructions Energy Consumption

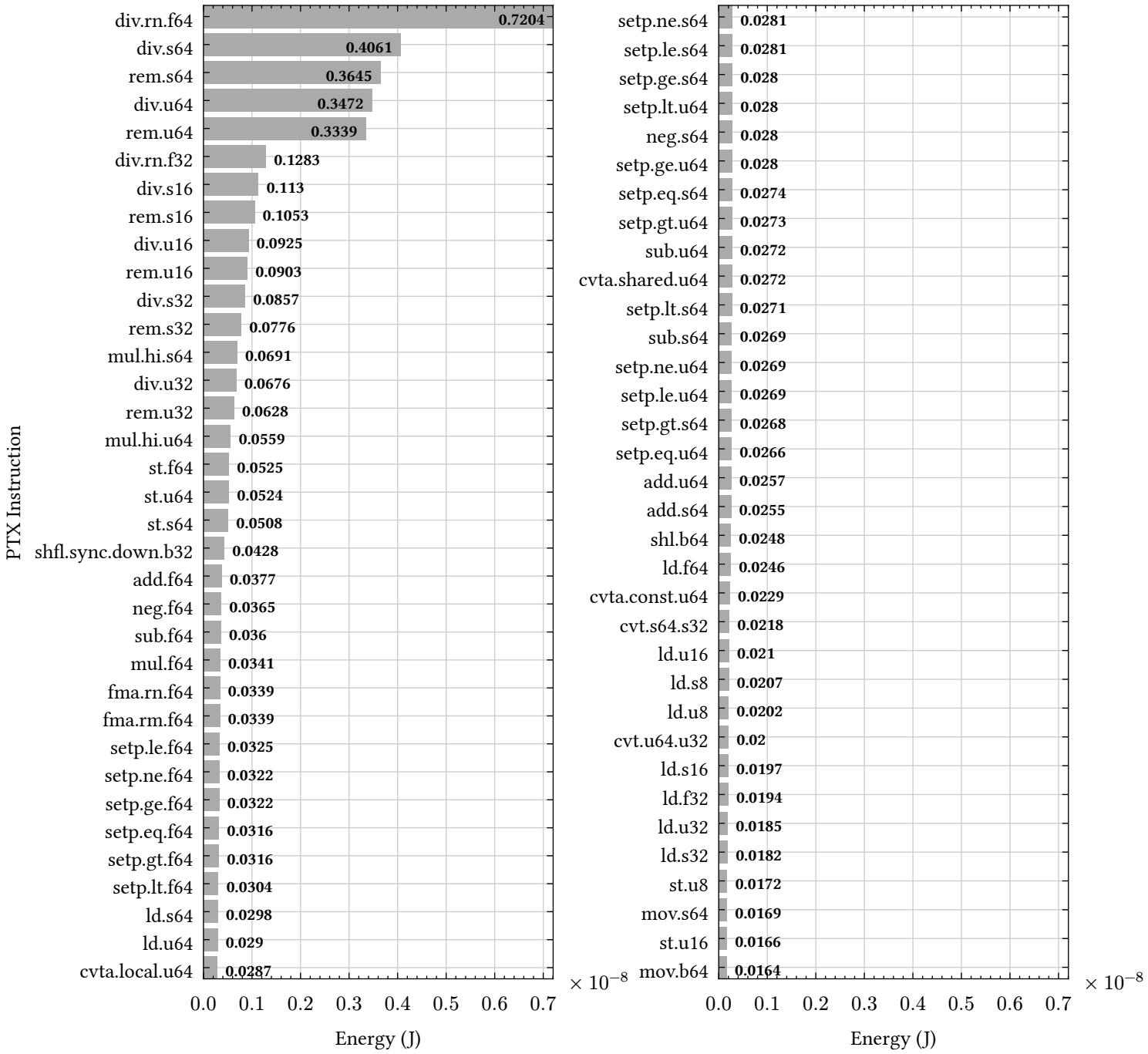


Figure 14: All PTX instructions energy consumption (part 1 of 2)

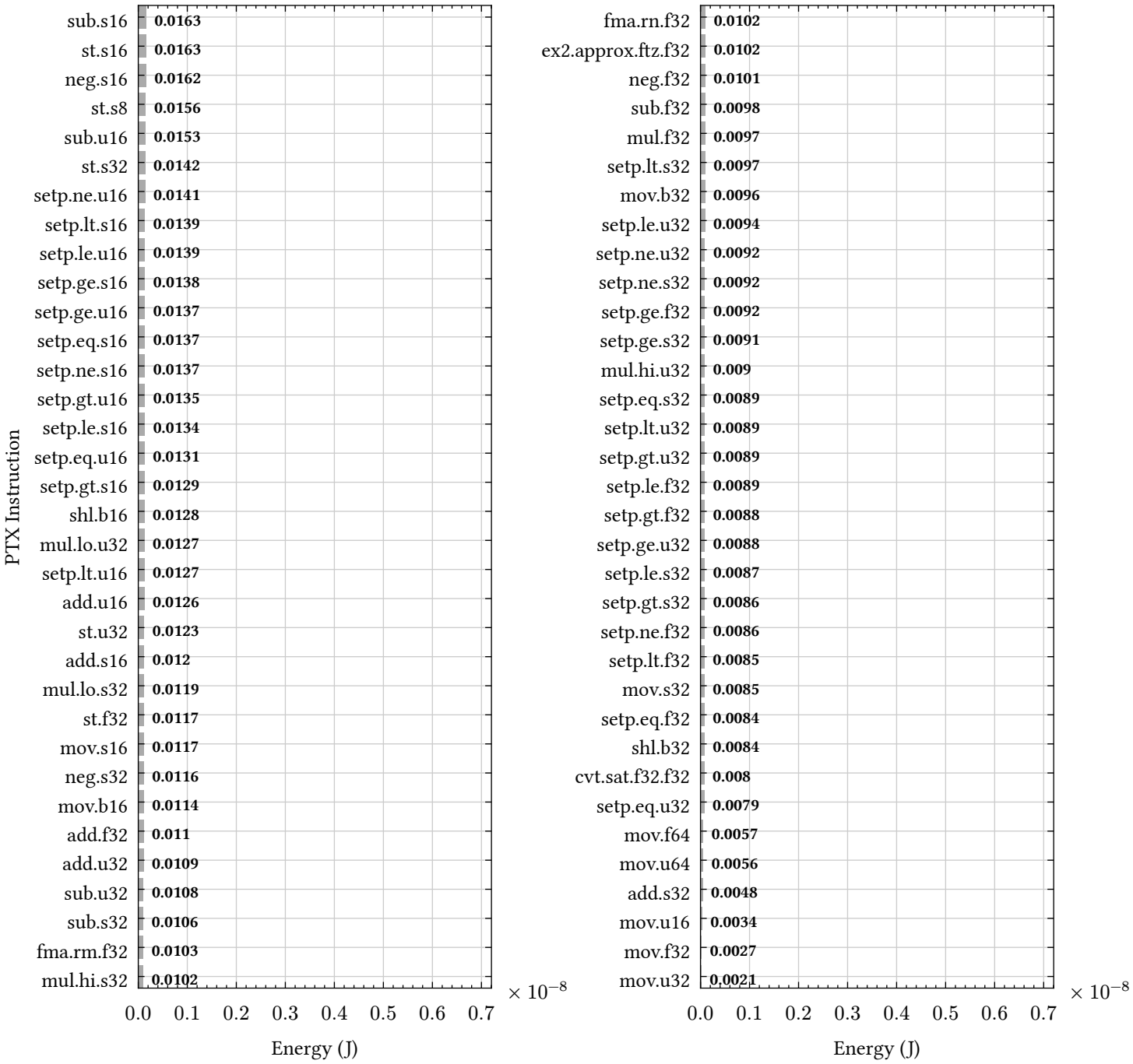


Figure 15: All PTX instructions energy consumption (part 2 of 2)