Deep Learning-based GPU Simulation for Agile Architecture-Algorithm Co-design

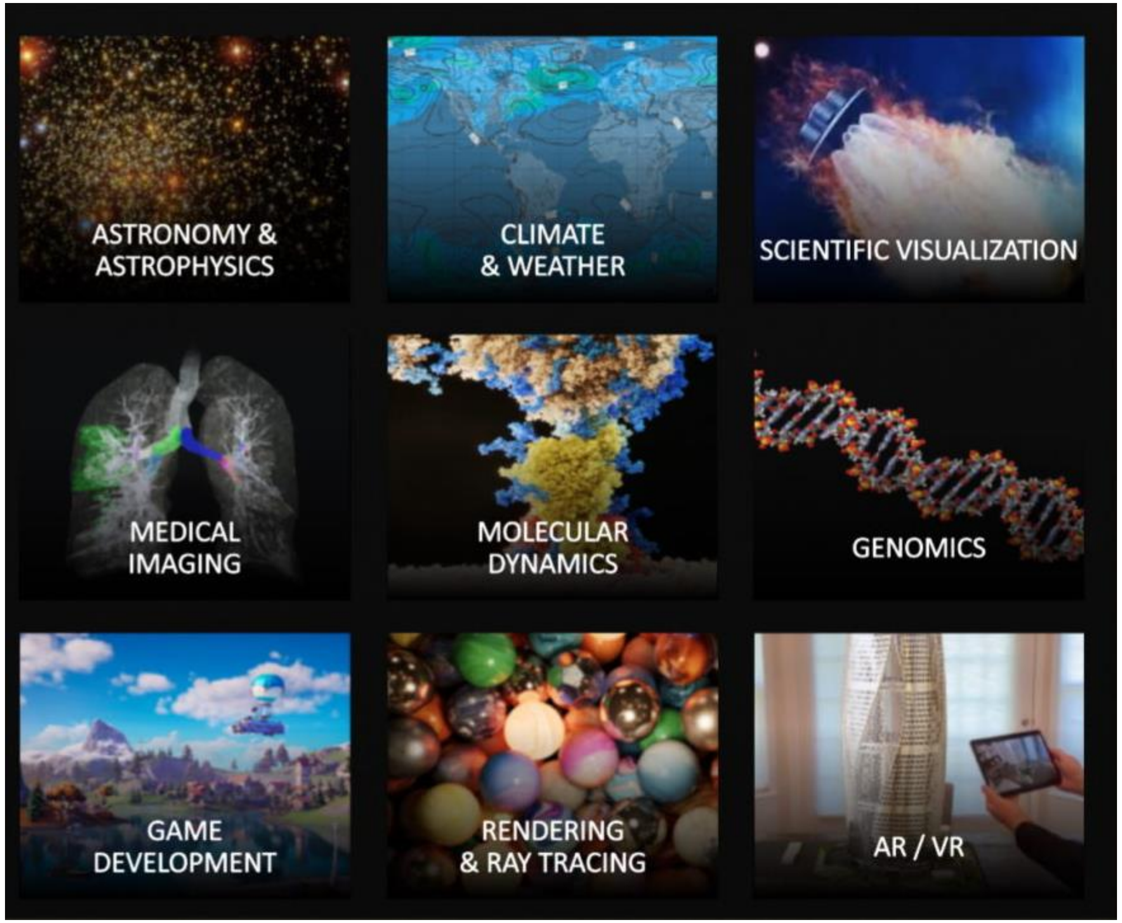
Junyu Yin[*1], Lingda Li[2], Hai Duong[1], Keren Zhou[1]

[1]George Mason University

[2]Brookhaven National Laboratory

Contents

The Expanding Role of GPUs in Modern Computing



The industrial HPC revolution by GPUs

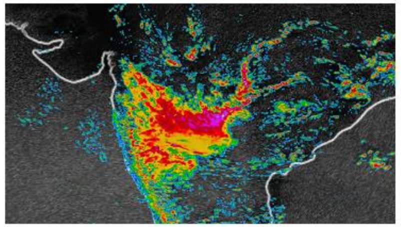
Rank (previous)	Rmax Rpeak (PetaFLOPS)	Name	Model	CPU cores	Accelerator (e.g. GPU) cores	Total Cores (CPUs + Accelerators)
1 –	1,206.00 1,714.81	Frontier	HPE Cray EX235a	561,664 (8,776 × 64-core Optimized 3rd Generation EPYC 64C @2.0 GHz)	36,992 × 220 AMD Instinct MI250X	8,699,904
2 ▲	1,012.00 1,980.01	Aurora	HPE Cray EX	1,104,896 (21,248 × 52-core Intel Xeon Max 9470 @2.4 GHz)	63,744 × 128 Intel Max 1550	9,264,128
3 –	561.20 846.84	Eagle	Microsoft NDv5	172,800 (3,600 × 48-core Intel Xeon Platinum 8480C @2.0 GHz)	14,400 × 132 Nvidia Hopper H100	2,073,600

TOP 500 supercomputers in June 2024

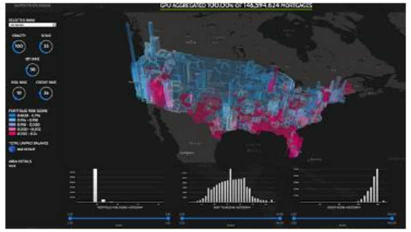
https://en.wikipedia.org/wiki/TOP500
https://blogs.nvidia.com/blog/industrial-hpc-revolution/

Efficient S/H Co-Design with Modeling and Simulation

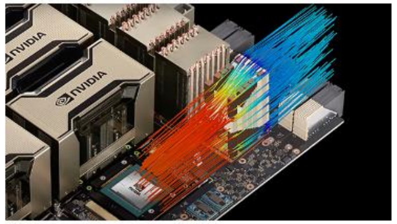
The architecture designs are complicated



Predict Weather Patterns



Accelerate Financial Models



Speed Up Engineering Simulations

……

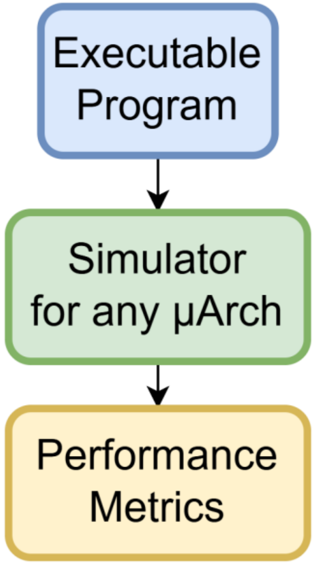
The computation tasks are complicated

Modeling and Simulation are textbook standards in architecture research

https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf
https://www.nvidia.com/en-us/high-performance-computing/simulation-and-modeling/

Modeling Approach1: Execution-driven Simulation

- What is it?
  - A technique where the actual instructions of a program are executed in a simulated environment



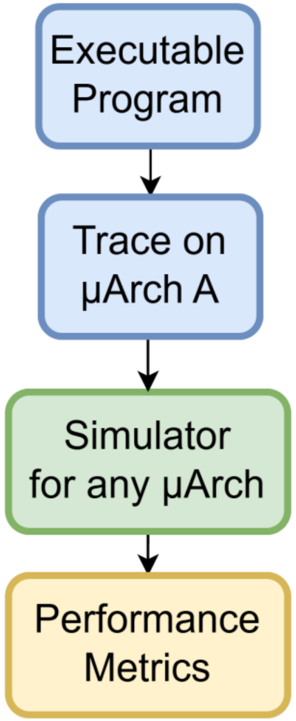
- Characteristics
  - **Dynamic**: it models the effects of each instruction by executing them
  - **Accurate**: it captures detailed impact of every instruction on the system
  - **Time-consuming**: it can take days to years for simulating large programs

- Examples: GPGPU-Sim [1], MGPUSim [2]

[1] Bakhoda, Ali, et al. "Analyzing CUDA workloads using a detailed GPU simulator." *2009 IEEE international symposium on performance analysis of systems and software*.
[2] Sun, Yifan, et al. "MGPUSim: Enabling multi-GPU performance modeling and optimization." *2019 ACM/IEEE 46th ISCA*.

Modeling Approach2: Trace-driven Simulation

- What is it?
  - A technique where a pre-recorded trace from a program's execution is used as the input for the simulation
  - A trace can be obtained from another simulator or by instrumenting a program
  - Trace: a log of executed instructions with additional running information like memory address



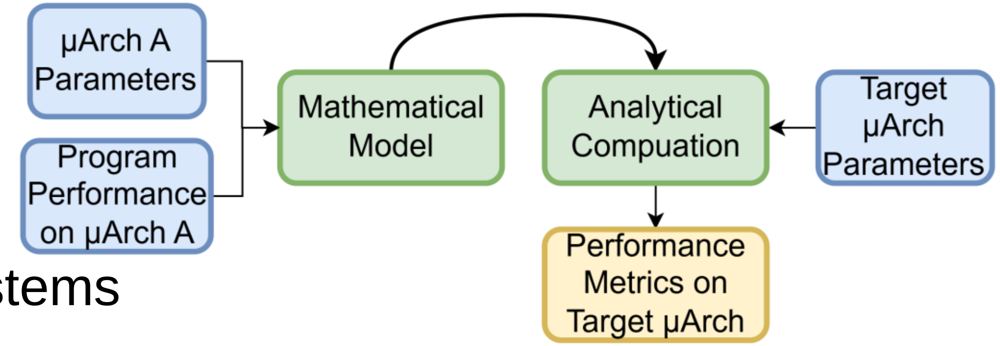
- Characteristics
  - **Static**: the trace is captured once and is fixed during the simulation
  - **Replay-based**: it only replays the trace without executing the actual program
  - **Time-efficient**: it avoids re-executing each instruction during simulation
  - **Accuracy-limited**: it reuses the same trace for different microarchitectures

- Examples: Accel-Sim [1]

[1] Khairy, Mahmoud, et al. "Accel-Sim: An extensible simulation framework for validated GPU modeling." *2020 ACM/IEEE 47th ISCA*.

Modeling Approach3: Analytical Modeling

- What is it?
  - A technique that uses mathematical formulas to estimate system performance based on certain assumptions



- Characteristics
  - **Abstract**: it uses simplified assumptions to model systems
  - **Fast**: it is much faster than simulations
  - **Accuracy-limited**: heuristic assumptions can be wrong and detailed behaviors are ignored

- Examples: GPUMech [1], GCoM [2]

[1] Huang, Jen-Cheng, et al. "GPUMech: GPU performance modeling technique based on interval analysis." *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*.
[2] Lee, Jounghoo, et al. "GCoM: a detailed GPU core model for accurate analytical modeling of modern GPUs." *2022 ACM/IEEE 49th ISCA*.

Summary Table of GPU Modeling

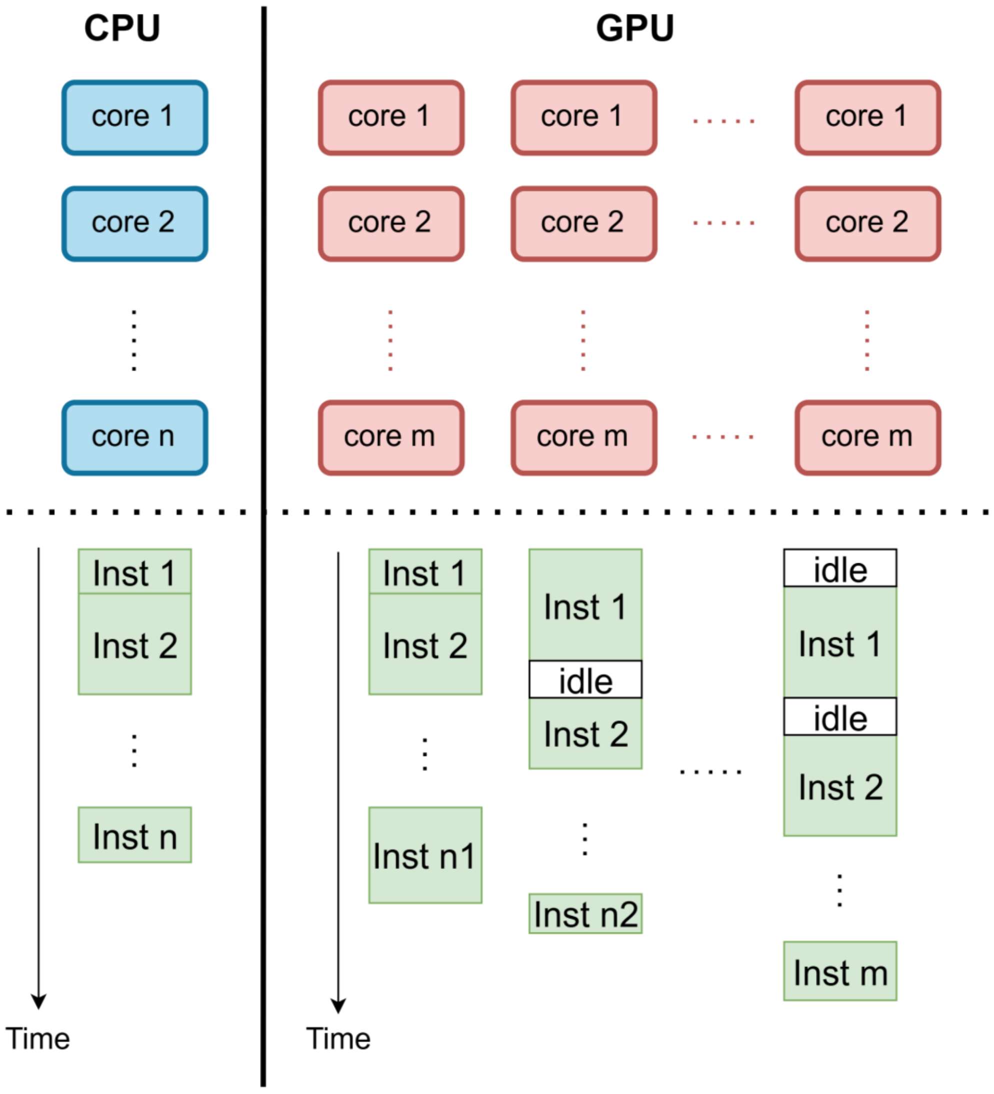
Modeling Methods		Running Time	Accuracy	Complexity of Implementation
Cycle-level Simulator	Execution-driven	Long	High	High
	Trace-driven	Medium	Medium	High
Analytical Modeling		Fast	Low to Medium	Medium to High
Ideal GPU Simulator		Fast	High	Medium

Contents

Challenge1: Highly Parallel Execution Patterns

- CPU Architecture
  - Few high-performance cores (16 or 32)
  - Optimized for tasks that require irregular control flow
- GPU Architecture
  - Many simple cores (more than 10k)
  - Optimized for tasks with high-degree of parallelism

- CPU Execution
  - Typically, only a single sequential instruction flow is considered
- GPU Execution
  - Single Instruction Multiple Threads (SIMT) Model
  - Multiple parallel instruction flows are considered
  - Branch divergence, load imbalance, etc.

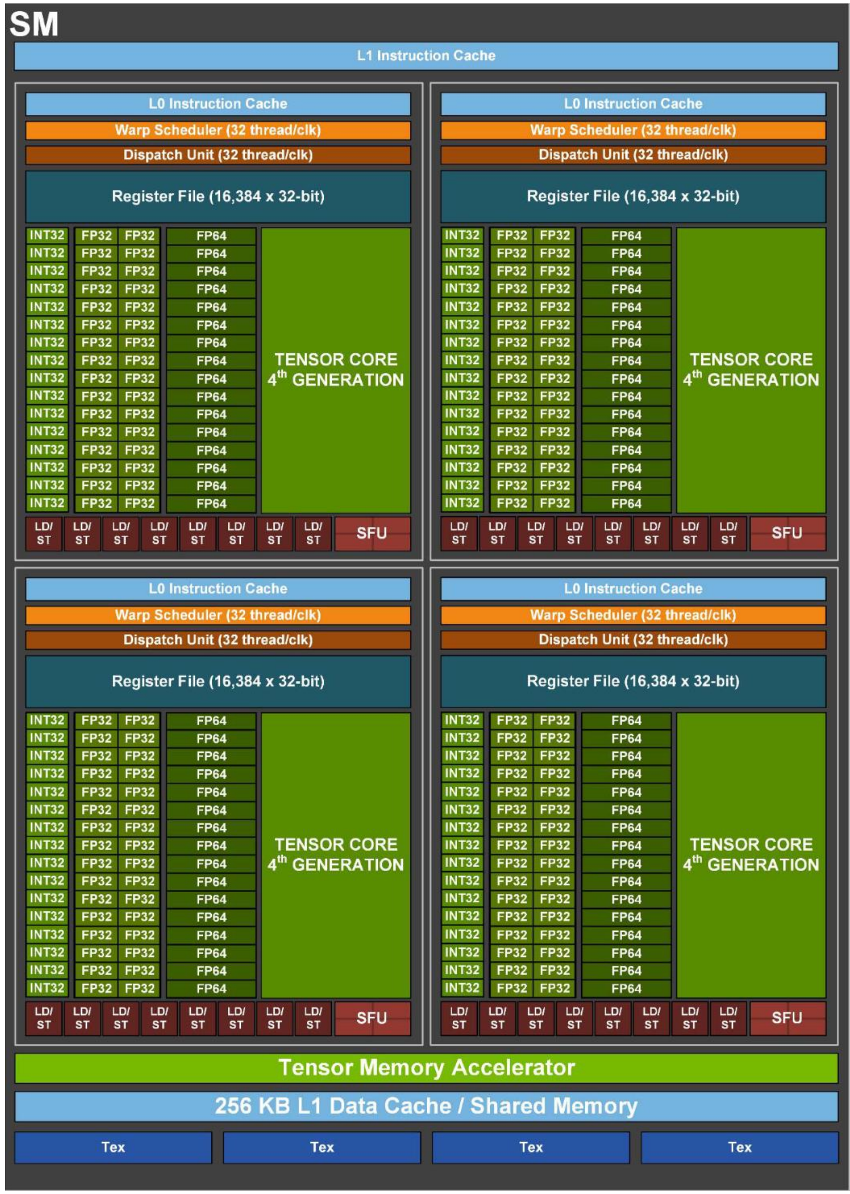


Challenge2: Intricacies of Modern GPU Architectures

- Dedicated Hardware Modules
  - Sub-core models, Tensor Cores, Texture Units, etc.

- Complex Memory Hierarchy
  - L0 Instruction cache, L1 data cache, shared memory, global memory, Tensor Memory Accelerator, etc.

- Synchronization & Scheduling
  - Thousands of threads need to be coordinated
  - Dedicated hardware schedulers for resource allocation, workload distribution, etc.



GH100 Streaming Multiprocessor (SM)

https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper

Challenges & Opportunity

- For GPU simulation, we need to tackle with highly parallel instruction flows

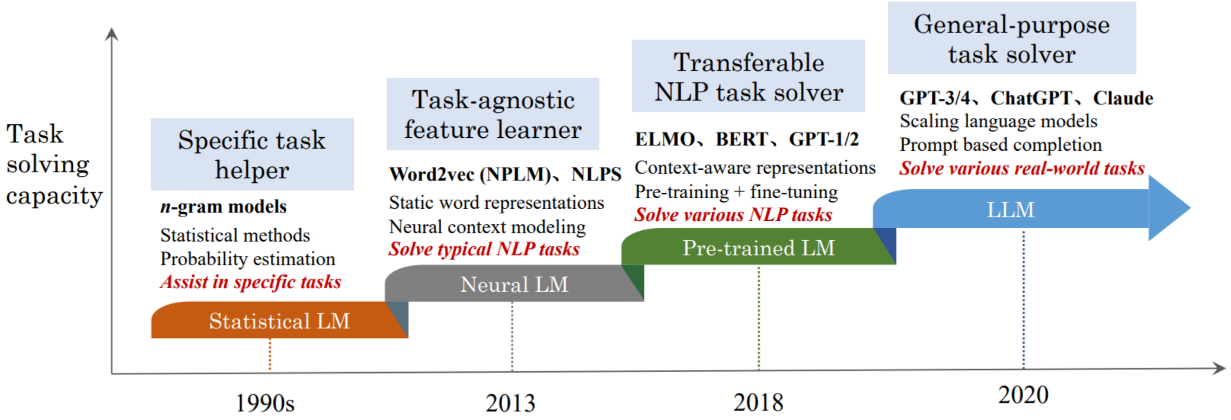
- Microarchitectures of current GPU are extremely complicated
  - Some hardware details are kept as top secrets by major companies
  - It would be impossible to implement a 100% accurate simulator

Can we get an accurate performance model
without knowing the hidden hardware details?

It is the time to leverage the power of deep learning!

Deep Learning for Hardware Simulation

- The powerful learning ability of deep neural networks
  - Neural networks have the potential to recognize any complex patterns [1]
  - Large language models (LLMs) are powerful general-purpose task solvers [2]



- Enabling Agile Architecture-Algorithm Co-design
  - Remove the excessive simulation time and complexity of traditional simulators
  - Identify promising design choices much faster

[1] Hornik, Kurt, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators." Neural networks 2.5 (1989): 359-366.
[2] Zhao, Wayne Xin, et al. "A survey of large language models." *arXiv preprint arXiv:2303.18223* (2023).

Deep Learning for Hardware Simulation (Cont'd)

- Generalized Modeling Across Configurations
  - By training on a sufficient number of architecture-workload combinations, it can generalize easily to unseen architectures and workloads
  - It avoids the painstaking modeling process required by traditional methods

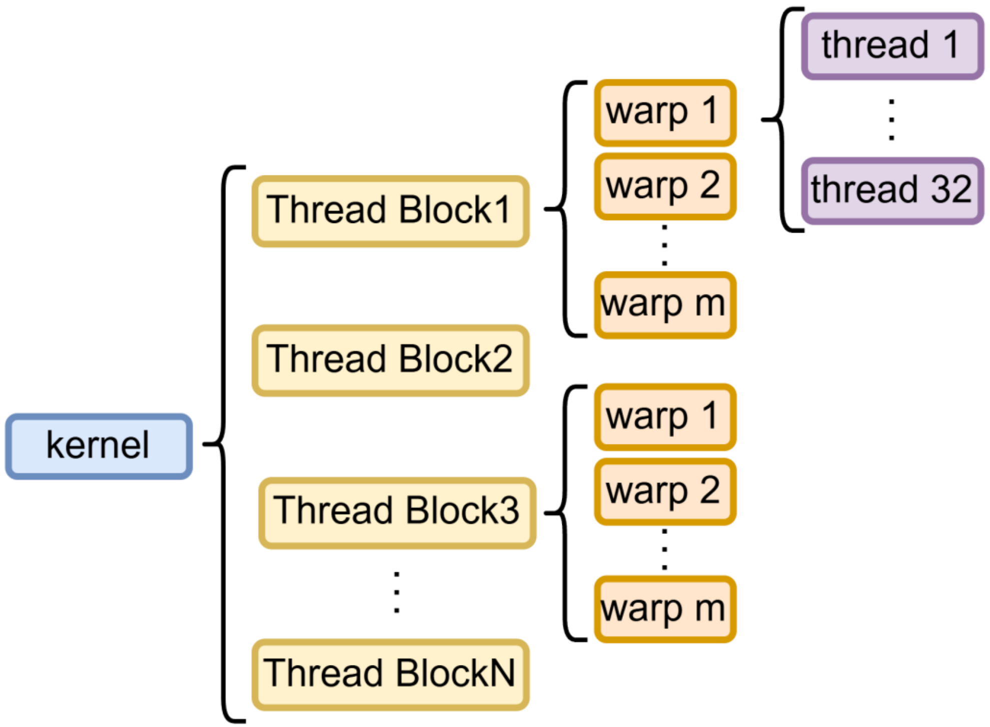
- Successful DL-based performance models for CPU
  - Basic-block-level prediction: Ithemal [1]
  - Instruction-level prediction: PerfVec [2]

[1] Mendis, Charith, et al. "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks." *International Conference on machine learning*. PMLR, 2019.
[2] Li, Lingda, Thomas Flynn, and Adolfy Hoisie. "Learning Independent Program and Architecture Representations for Generalizable Performance Modeling." *arXiv preprint arXiv:2310.16792* (2023).

Contents

Introduction: GPU Execution Model

- Our design is based on NVIDIA GPUs and CUDA programs
  - *Kernel*: a function that runs on a GPU
  - *Thread Block*: a group of threads that execute together
  - *Warp*: a group of typically 32 threads that execute in a SIMT and lockstep manner
  - In the context of GPU, “instructions” = “warp-level instructions”



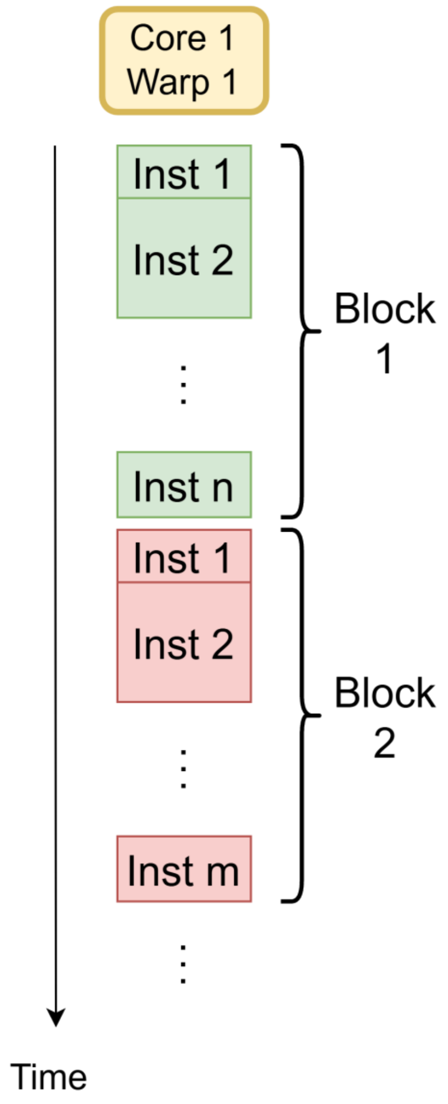
Introduction (Cont'd)

- The instruction traces are collected by instrumenting a simulator
  - Easier to get performance metrics of interest than binary instrumentation
  - Trace: a log of executed instructions with additional information like memory address

- We choose the PTX as our instruction set architecture (ISA)
  - PTX: a low-level parallel-thread execution virtual machine and ISA
  - An intermediate representation compatible with different NVIDIA microarchitectures

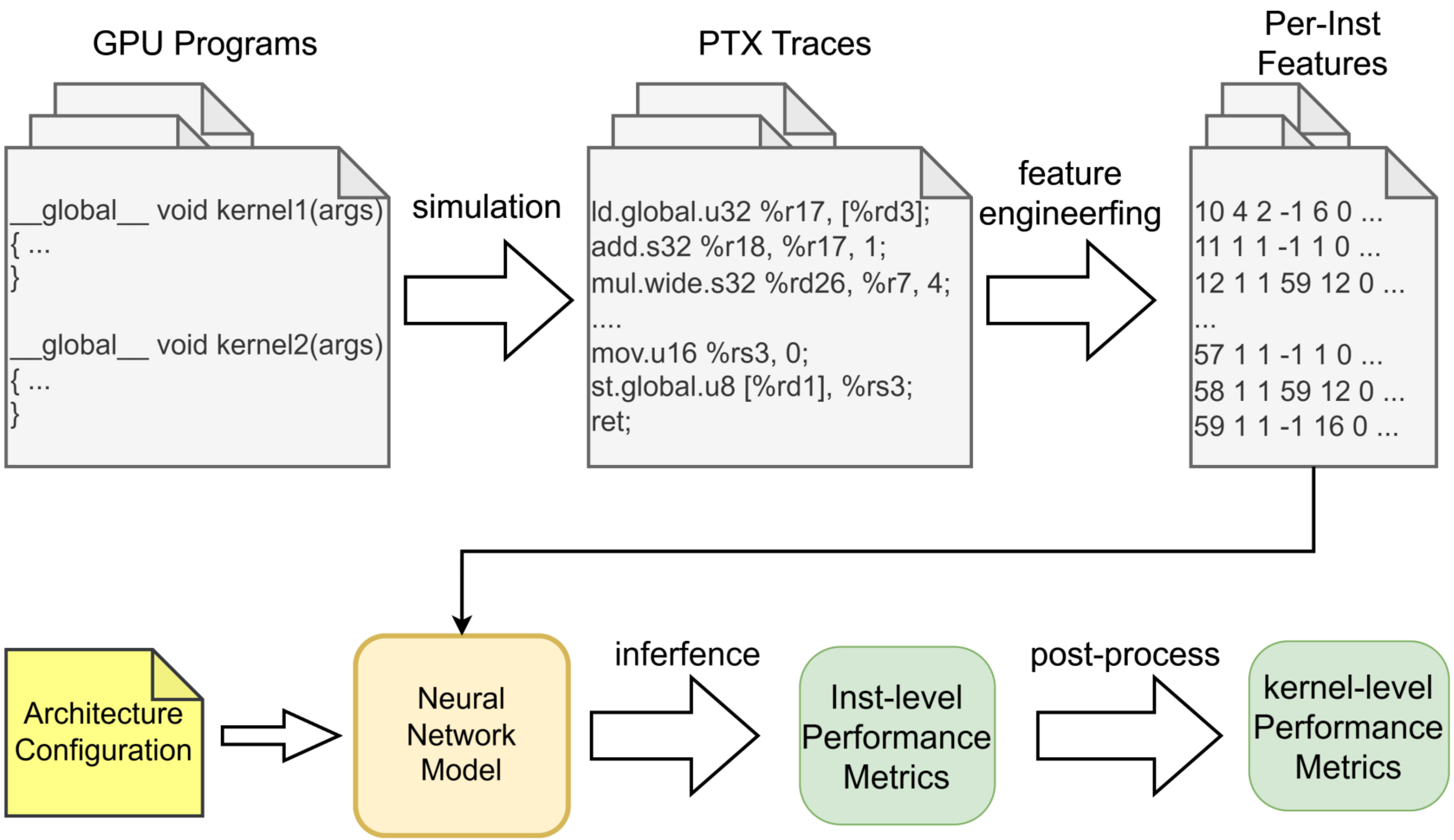
Introduction (Cont'd)

- The following slides assume the single-warp scenario
  - There is only 1 SM (core) for hardware configuration
  - Thread blocks of an executed kernel contain only 1 warp
  - The simplest execution scenario of GPU, just like a sequential execution
  - The very first step to validate our proposed method

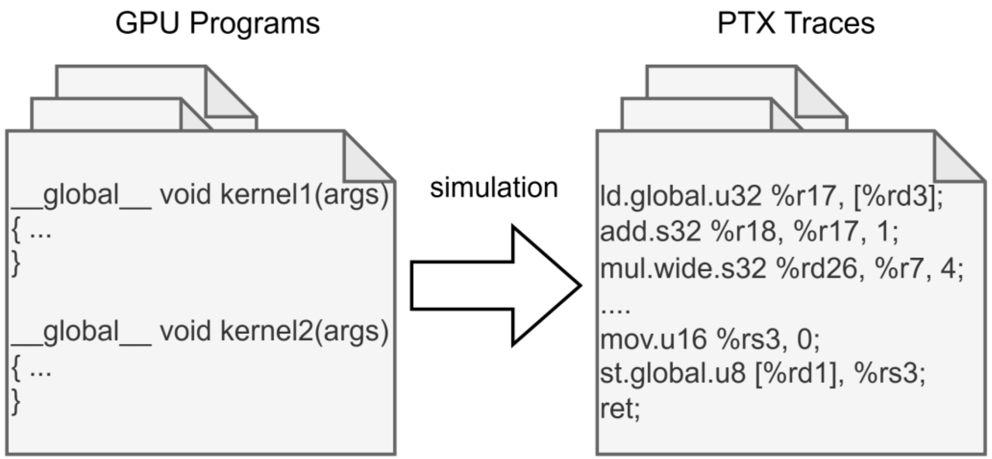


We want to keep the details of parallel scenario for now since it is an ongoing work ^_^

Overview of the Pipeline

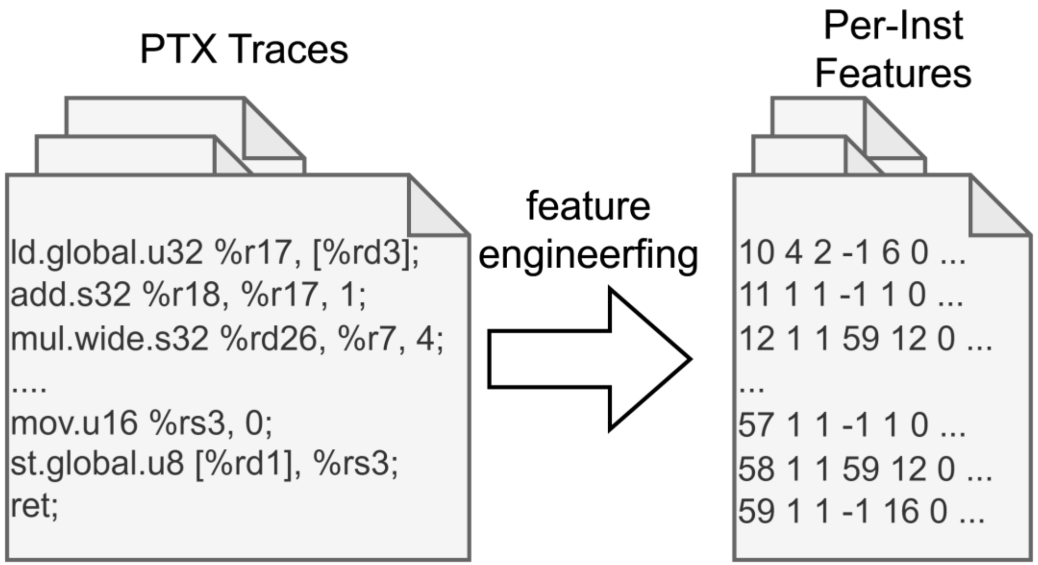


Data Collection



- We collect data by running PTX simulation with an instrumented GPGPU-SIM
- Theoretically, any detailed simulator supporting PTX is ok

Feature Engineering

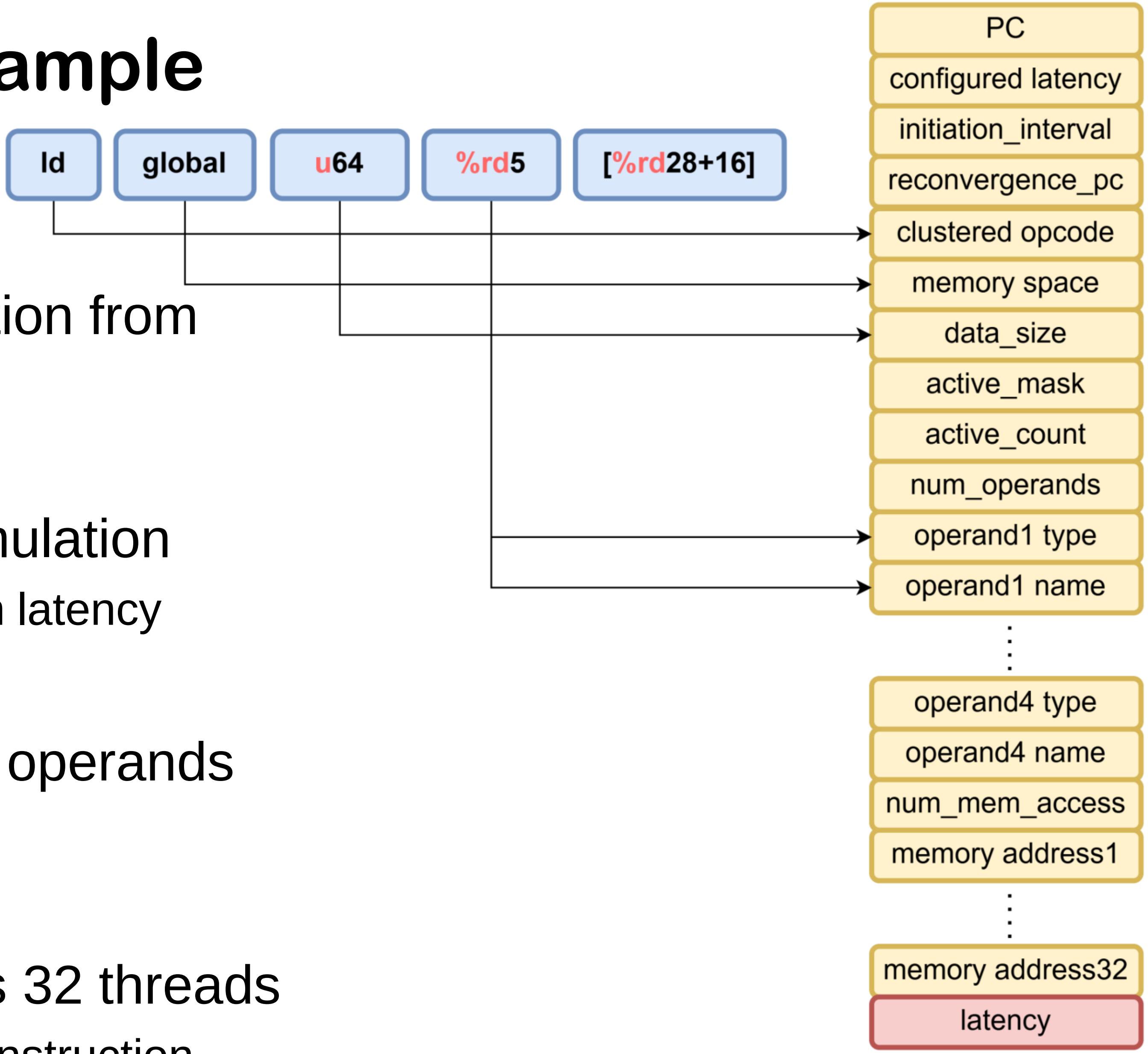


- We include the static information of an instruction itself and the dynamic information from its execution trace

- See an example in next slides

Feature Engineering: An Example

- The arrows link to the static information from instruction itself
- The rest fields are obtained from simulation
  - The red box is the cycle-level instruction latency
- We assume that there are at most 4 operands
  - 1 destination and 3 source
- Note that a warp instruction contains 32 threads
  - 32 memory addresses for any memory instruction



Feature Engineering: Register Dependency

- Note that PTX ISA uses virtual registers rather than physical registers
  - It can't capture the actual register dependency situation

- We use dependency distance to rename the register
  - Dependency distance: the number of instructions between a register and its previous occurrence
  - Example: the dependency distance of %f19 is 4 and which of %rd11 is 1. Then %f19 is converted to (float register + 4) and %rd11 to (double register + 1)

```
ld.shared.f32   %f19,     [%r34];
mad.lo.s32      %r35,     %r1,      %r11, %r2;
mul.wide.s32    %rd11,    %r35,     4;
add.s64         %rd12,    %rd10,    %rd11;
st.global.f32   [%rd12],  %f19;
```

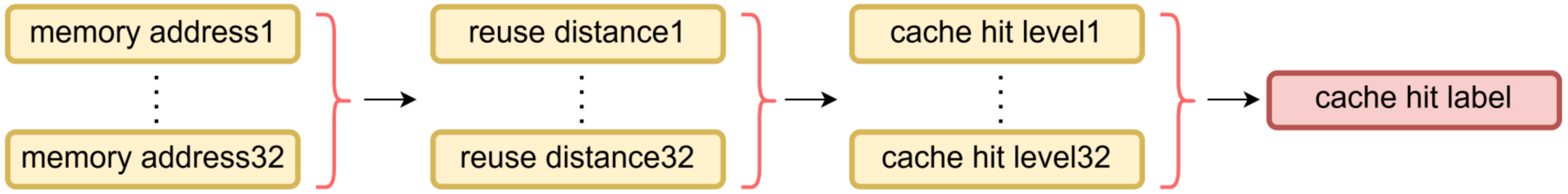
  - We can remove the destination register in this encoding format

Feature Engineering: Memory Dependency

- For neural networks, raw memory addresses are hard to learn
  - They are just some super large value (3221356576, 3221362720, …)
  - They have no functional information for prediction

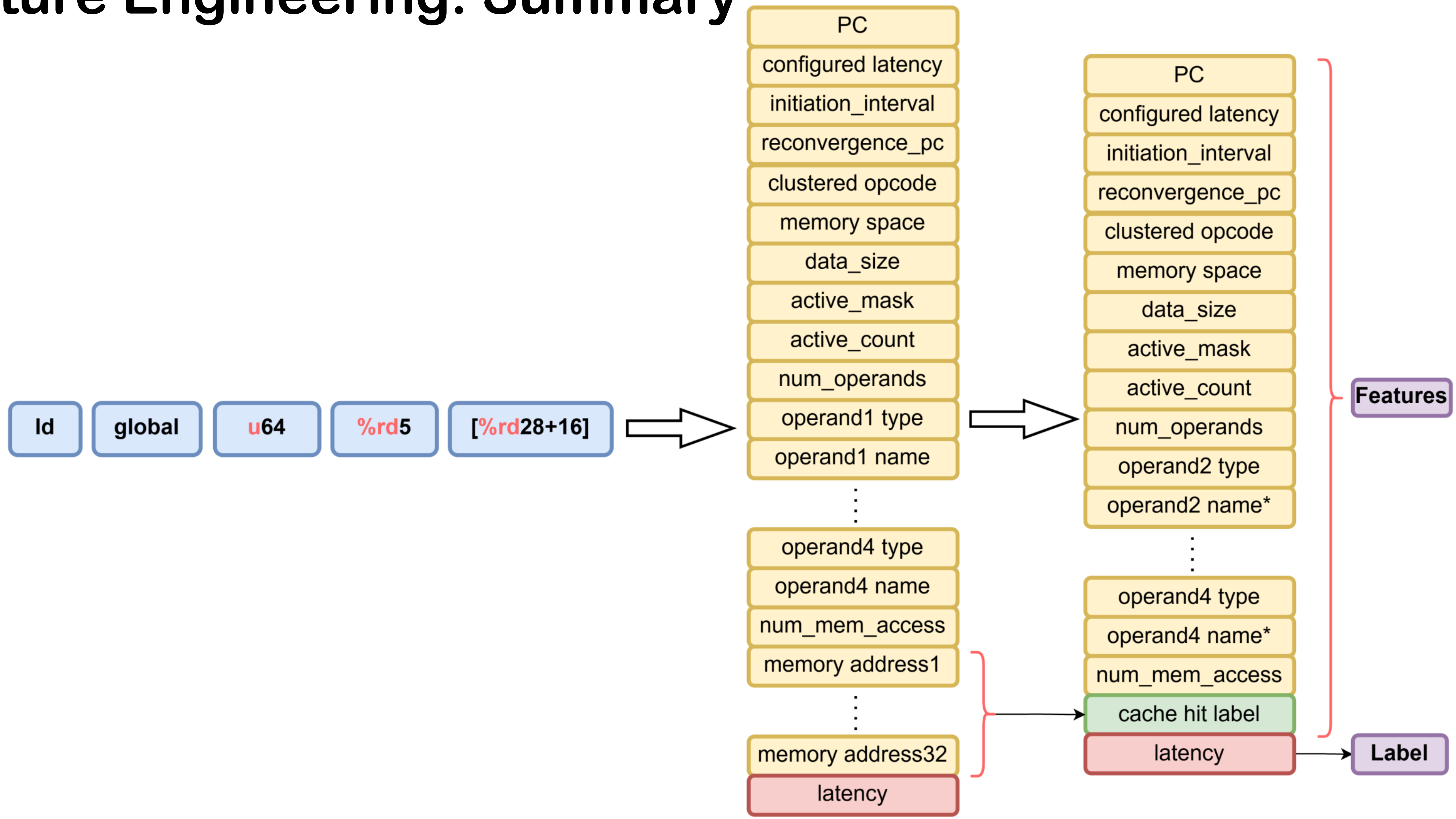
- Reuse distance is a well-known means to model cache behavior
  - The number of different memory addresses between two identical addresses

Feature Engineering: Memory Dependency

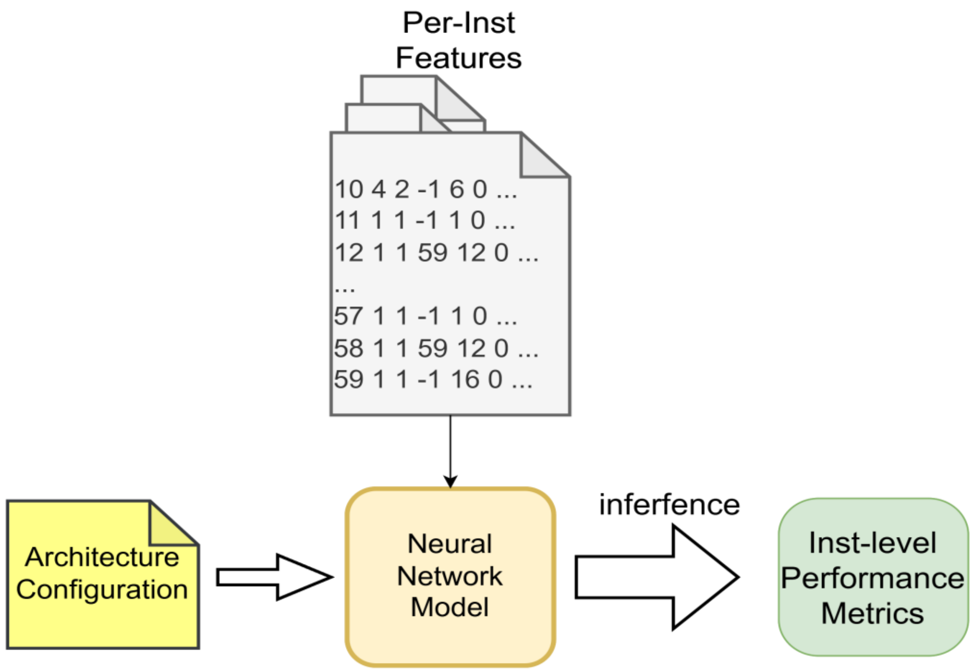


- We use reuse distance of each accessed address to predict its cache behavior
  - It is easy to calculate under the single-warp scenario
  - Roughly, small distance means hitting in L1 cache, the medium one means hitting in L2 cache and the large one means missing in all caches
  - Warp memory instruction is dominated by the memory access with the longest latency

Feature Engineering: Summary



Framework

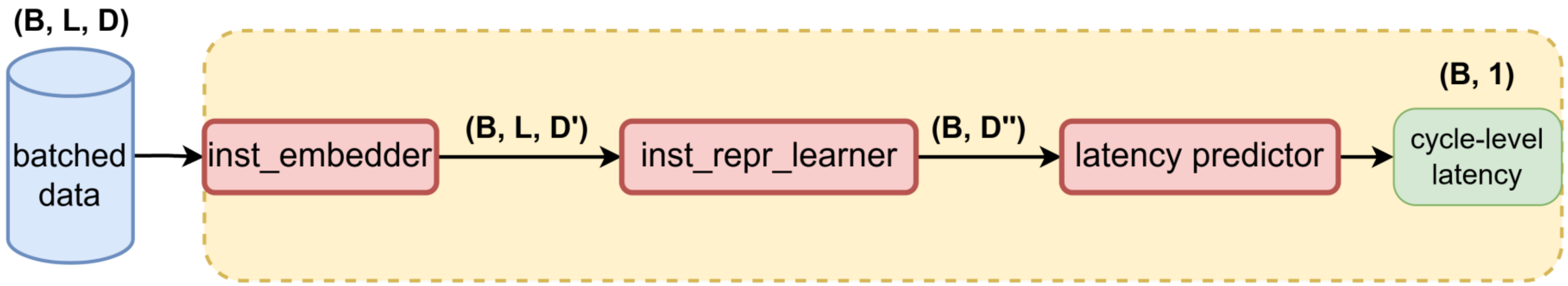


- The latency of an instruction is not only related to itself, but also to the instructions executed before it

- We need a sequence model to accurately predict instruction latency

- There are some categorical values in the instruction features

Neural Network Design

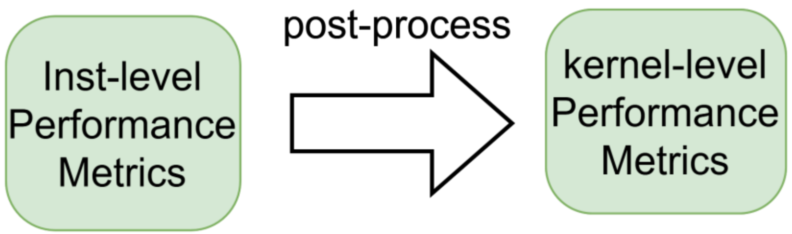


- Instruction Embedder: handling categorical values in instruction features
  - It can be one-hot encoding or embedding tables

- Instruction Representation Learner: learning the vector representation of an Inst
  - $F(I_i, I_{i-1}, \dots, I_{i-L+1}) = R_i,\ I_i, I_{i-1}, \dots, I_{i-L+1} \in \mathbb{R}^{D'},\ R_i \in \mathbb{R}^{D''}$
  - F can be any sequence model: LSTM, Transformer, GPT, etc.

- Latency Predictor: predicting the cycle-level latency of an Inst
  - $P(R_i) = C_i,\ C_i \in \mathbb{R}$ is cycle-level latency for Instruction I_i
  - P can be any regression model: MLP, Linear Regressor, etc.

Post Process



- In the context of GPU, we care more about the kernel-level performance
- We predict incremental latency for each instruction
  - The end cycle of an instruction minus that of its previous one
- We can sum them up for kernel-level latency
  - Assume a kernel with N instructions, then its latency is $\sum_{i=1}^{N} C_i$, C_i is the incremental latency for each instruction
  - It only works for the single-warp scenario

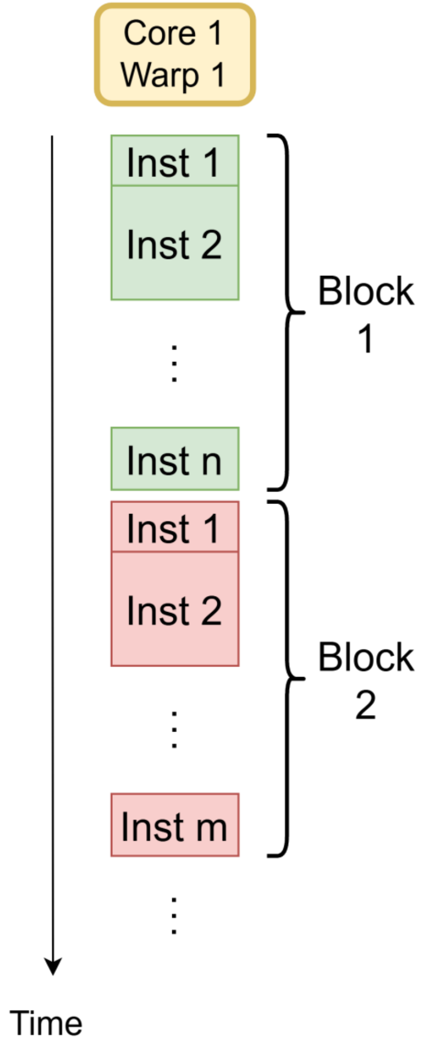
Contents

- Background
- Motivations
- Method
- Preliminary Results
- Ongoing Work
- Future Work

Experiment Setting: Execution Configuration

- All the reported experiments were conducted under the single-warp scenario
  - Modified configuration files to have only 1 SM (Core)
  - Modified CUDA programs so that each kernel uses with only 1 warp

- It just like a sequential execution without any contention
  - The simplest execution scenario for GPU
  - The first step to validate our method



Experiment Setting: Datasets

- We collected data by instrumenting GPGPU-SIM [1] (version 4.0)
- 48 programs under RTX3070 configuration from 6 benchmark suites have been simulated
  - 1046 executed kernels, 99 different kernels, ~400k instructions

Benchmark Name	Number of Programs
Rodinia 3.1	11
Polybench 1.0	15
Microbenchmark	9
CUDA SDK	3
Pannotia	8
Lonestargpu 2.0	3

- Use 39 programs for training and validation; Use the rest for test

[1] Khairy, Mahmoud, et al. "Accel-Sim: An extensible simulation framework for validated GPU modeling." *2020 ACM/IEEE 47th ISCA*.

Preliminary Results: Cache Predication

	Precision / Recall (%)			Accuracy
True Label	Hit in L1$	Hit in L2$	Miss	
Train set	100 / 89	89 / 100	100 / 100	99.6%
Test set	100 / 71	56 / 100	100 / 100	98.3%

Preliminary Results: Latency Predication

Instruction-level	MAPE / MAE / RMSE		
# of parameters / Model	Train	Validation	Test
411,685 / LSTM	- / 5 / 37	8.8% / 5 / 37	78% / 15 / 54
446,245 / Transformer	- / 4.8 / 36	24% / 4.9 / 36	118% / 16 / 64

Kernel-level	MAE / MAPE	
# of parameters / Model	Train	Test
411,685 / LSTM	1116 / 4%	1466 / 10%
446,245 / Transformer	1054 / 4%	2218 / 12%

Contents

Extending Reuse Distance to Parallel Execution

- When switching to parallel execution, we don't know the exact order of instructions

- We should also consider the followings when extending it to GPUs
  - Instructions within an SM share L1 and L2 memories
  - Instruction between different SMs and kernels only share L2 memory
  - Adaptive L1D cache
  - Some Instructions inherently bypass L1 (atomic operations, ld.global.cg, etc.)

- Ideas
  - Align all the instructions into one timeline by some estimated order
  - Maintain per-SM reuse stacks and include more execution information

Extending Framework to Parallel Execution

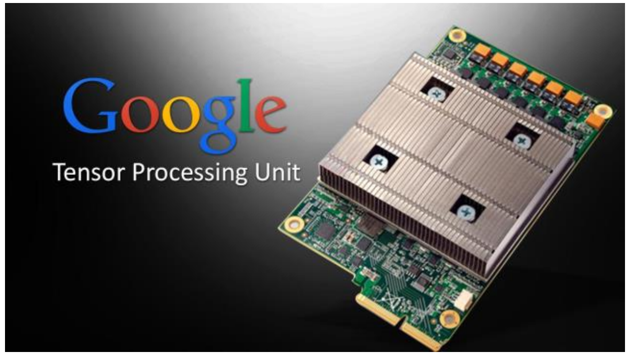
- Including hardware resource limitation
  - The number of execution units, operand collectors, etc.
- Including concurrent pattern
  - The number of waves
  - The size of each wave
- Implementing post-process under parallel situation
  - Run a statistical analysis on Inst-level performance to estimate the kernel-level one
  - Selecting some representative warps by warp profiling techniques

Contents

Cover More Architectures

- In this work, we only model NVIDIA GPUs and CUDA programs

- In the future, we can extend our model to more Architectures
  - AMD, Intel, etc.
  - Even to other parallel platforms like TPU, FPGA, etc.



The Data Bottleneck

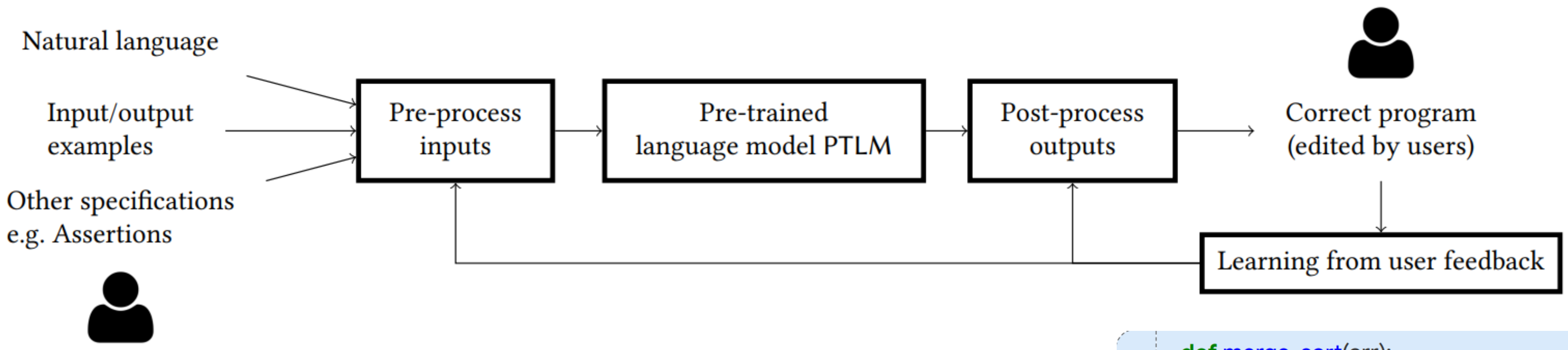
- Deep learning models need millions of programs for effective learning
  - Based on the scaling law [1], we need a dataset size of $5 \times 10^3 \times (7 \times 10^9)^{0.74}$ to fully utilize a small 7b model
  - However, there are only a few benchmarks, each with about a dozen of programs
    - Not all of them can be simulated (prohibitively large trace size)

- The current GPU benchmark suites don’t meet our requirement
  - Most of GPU benchmark suites are outdated
    - Rodinia (2009), Polybench (2012), Parboil (2012), etc.
  - We need to capture latest GPU features
    - Tensor Cores, Tensor Memory Accelerator (TMA), etc.

[1] Kaplan, Jared, et al. "Scaling laws for neural language models." *arXiv preprint arXiv:2001.08361* (2020).

Deep Learning for Program Synthesis

- Deep neural networks have the potential to generate a program
  - sketch, input/output examples, natural language descriptions, etc.



  - Copilot, GPT-4o, Cursor, etc.

prompt

Write a Python function to sort the given array by using merge sort. Your code should satisfy these tests:

```
assert merge_sort([3, 4, 2, 6, 5, 7, 1, 9]) == [1, 2, 3, 4, 5, 6, 7, 9]
assert merge_sort([7, 25, 45, 78, 11, 33, 19]) == [7, 11, 19, 25, 33, 45, 78]
assert merge_sort([3, 1, 4, 9, 8]) == [1, 3, 4, 8, 9]
```

model

```
def merge_sort(arr):
  if len(arr) < 2:
    return arr;
  mid = len(arr) // 2;
  left = arr[:mid];
  right = arr[mid:];
  left = merge_sort(left);
  right = merge_sort(right);
  merged = [];
  i = j = 0;
  while i < len(left) and j < len(right):
    if left[i] < right[j]:
      merged.append(left[i]);
      i += 1;
    else:
      merged.append(right[j]);
      j += 1;
  merged.extend(left[i:]);
  merged.extend(right[j:]);
  return merged;
```


Austin, Jacob, et al. "Program synthesis with large language models." *arXiv preprint arXiv:2108.07732* (2021).
Jain, Naman, et al. "Jigsaw: Large language models meet program synthesis." *Proceedings of the 44th International Conference on Software Engineering*. 2022.

Q & A