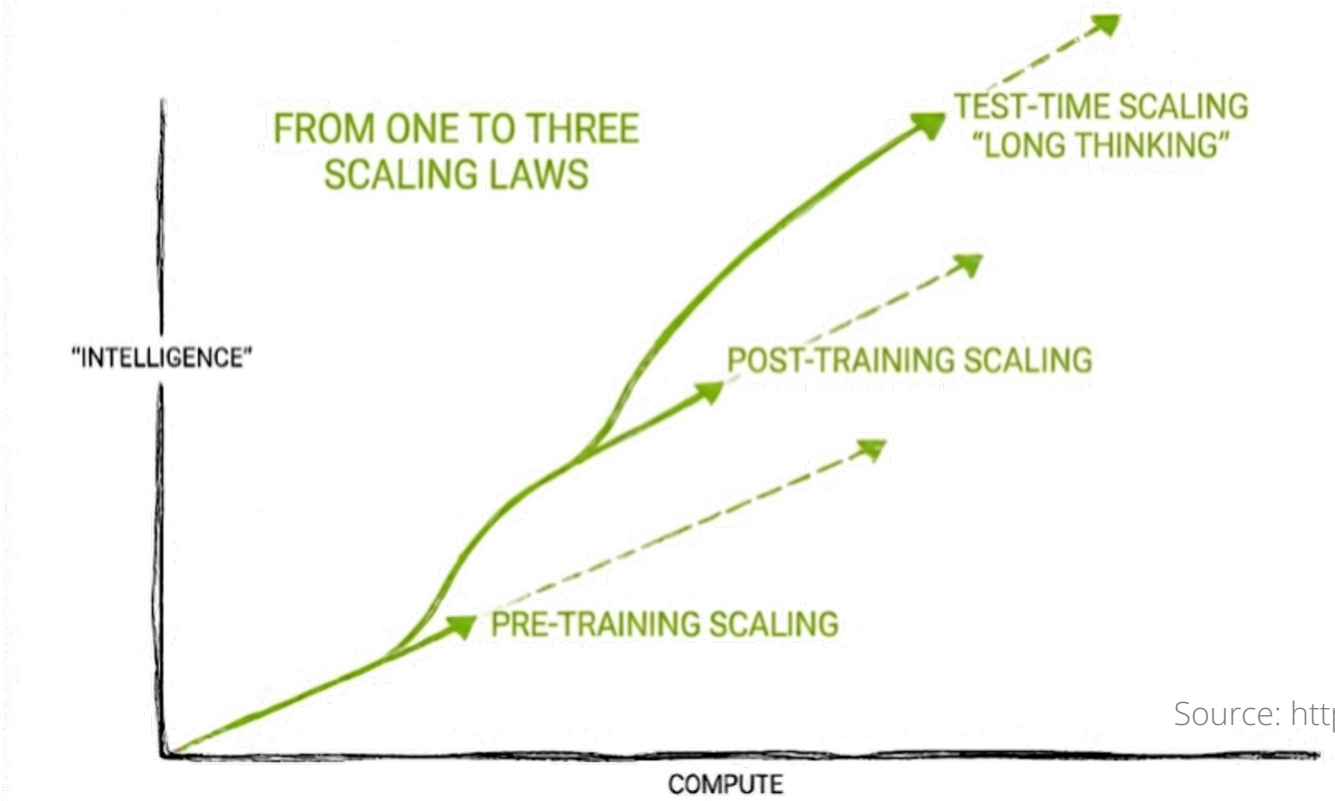
Composable Programming Models for AI Scaling

Hongzheng Chen

Guest Lecture
Rice COMP 468/568
02/25/26

Cornell University

Era of Scaling



High-performance
Large-scale
Heterogeneous

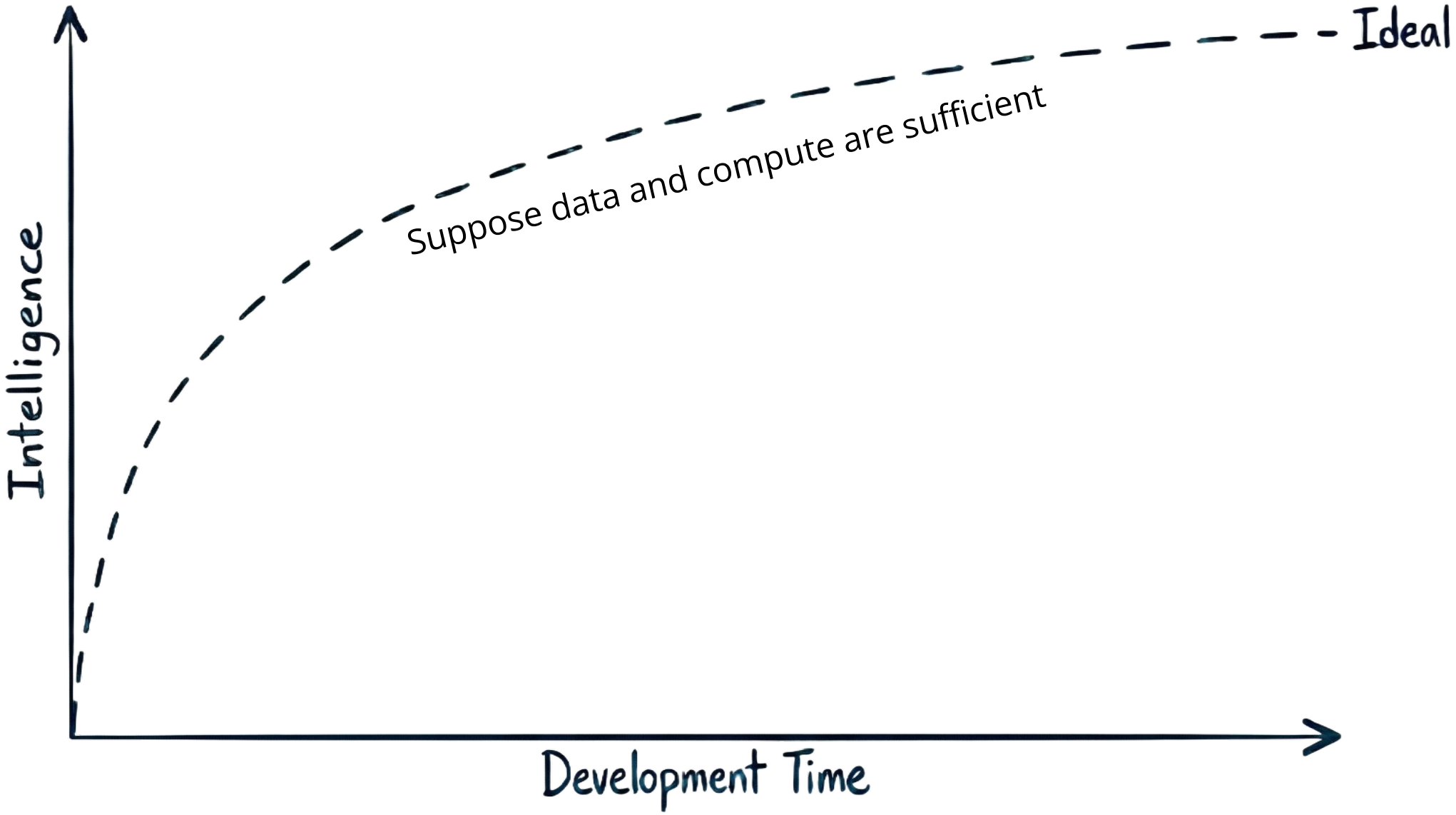
Source: https://blogs.nvidia.com/blog/ai-scaling-laws/

"OpenAI and Cerebras have signed a multi-year agreement to deploy **750 megawatts** of **Cerebras wafer-scale** systems to serve OpenAI customers." – OpenAI, 2026

"Based on **550,000 Nvidia Blackwell AI accelerators**, xAI's Colossus 2 is advertised as the industry's first AI facility that consumes one gigawatt of power for AI inference and training." – xAI, 2026

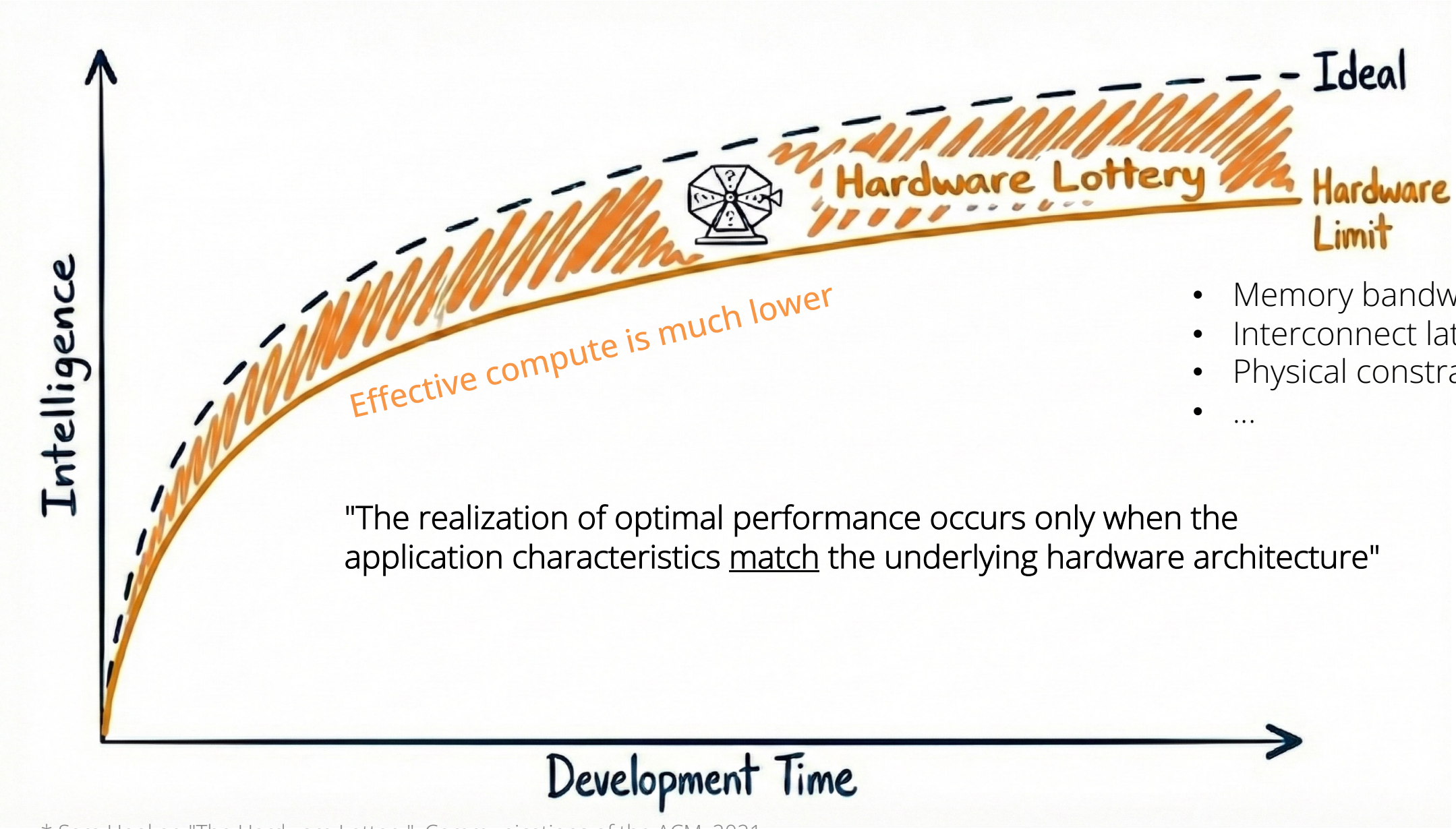
"We are announcing that we plan to expand our use of Google Cloud technologies, including up to **one million TPUs**, dramatically increasing our compute resources." – Anthropic, 2025

The Reality of Scaling



* Jared Kaplan, "Scaling Laws for Neural Language Models", arXiv, 2020.

The Reality of Scaling

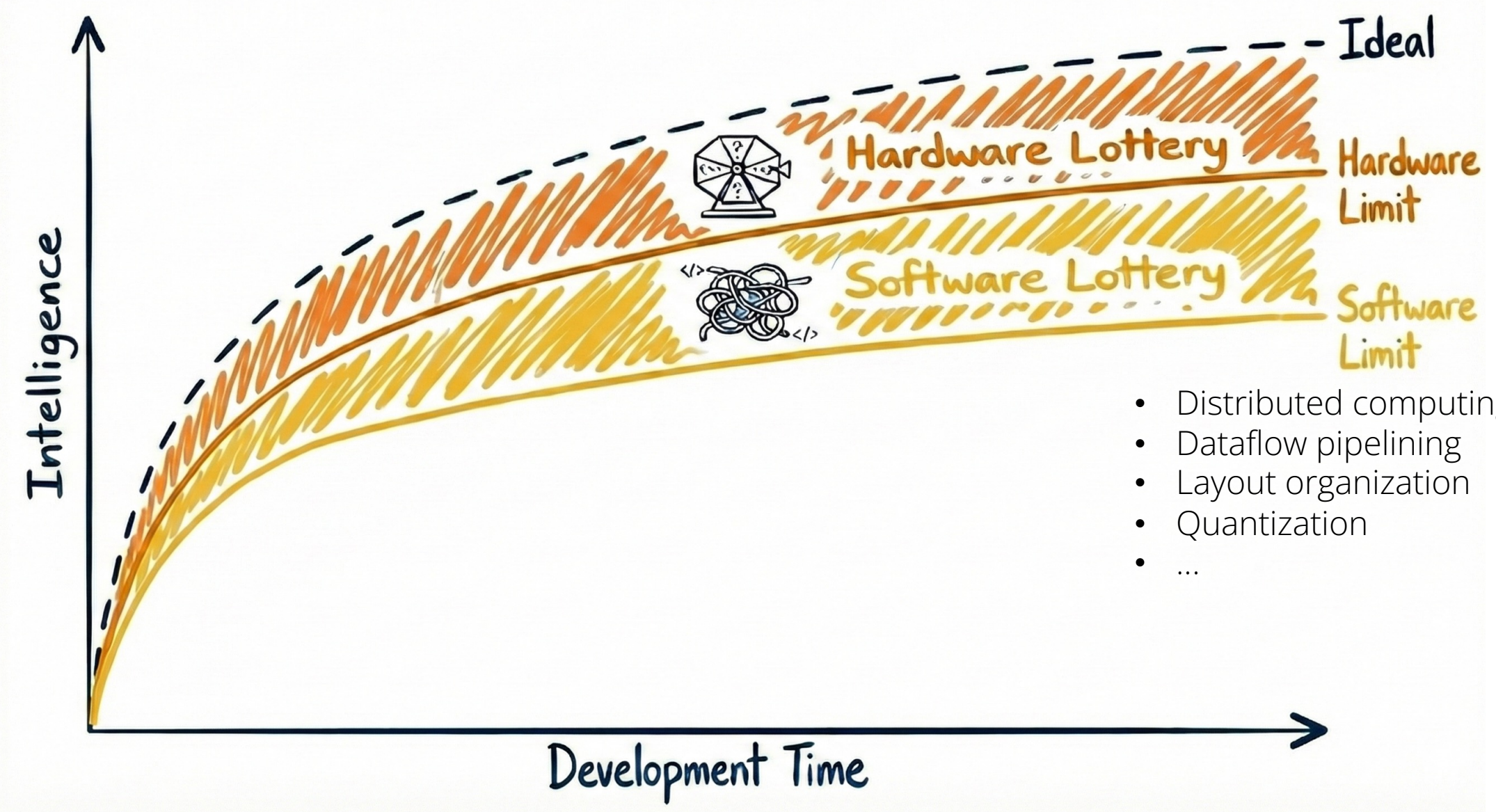


- Memory bandwidth
- Interconnect latency
- Physical constraints
- ...

"The realization of optimal performance occurs only when the application characteristics match the underlying hardware architecture"

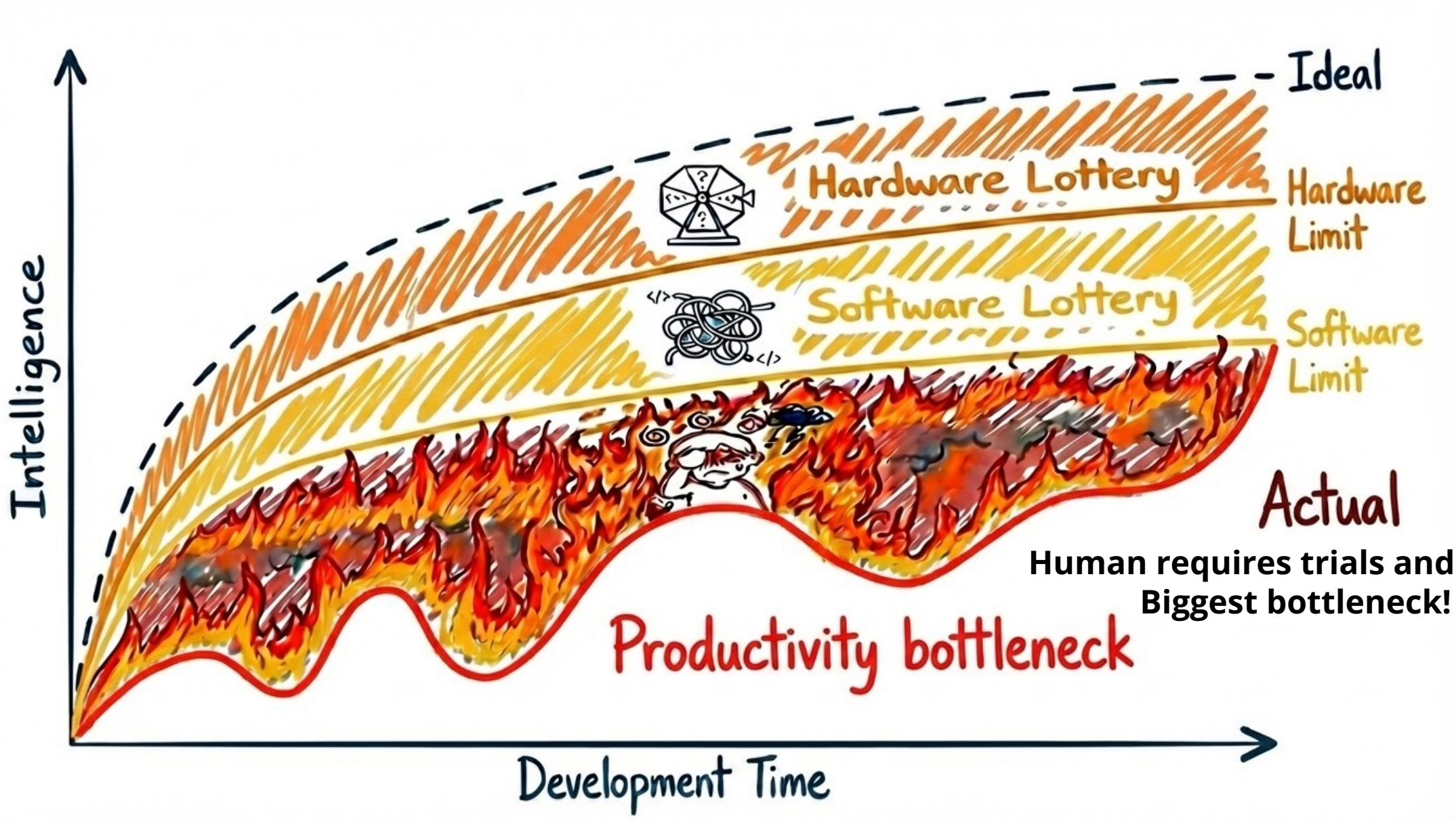
* Sara Hooker, "The Hardware Lottery", Communications of the ACM, 2021.

The Reality of Scaling



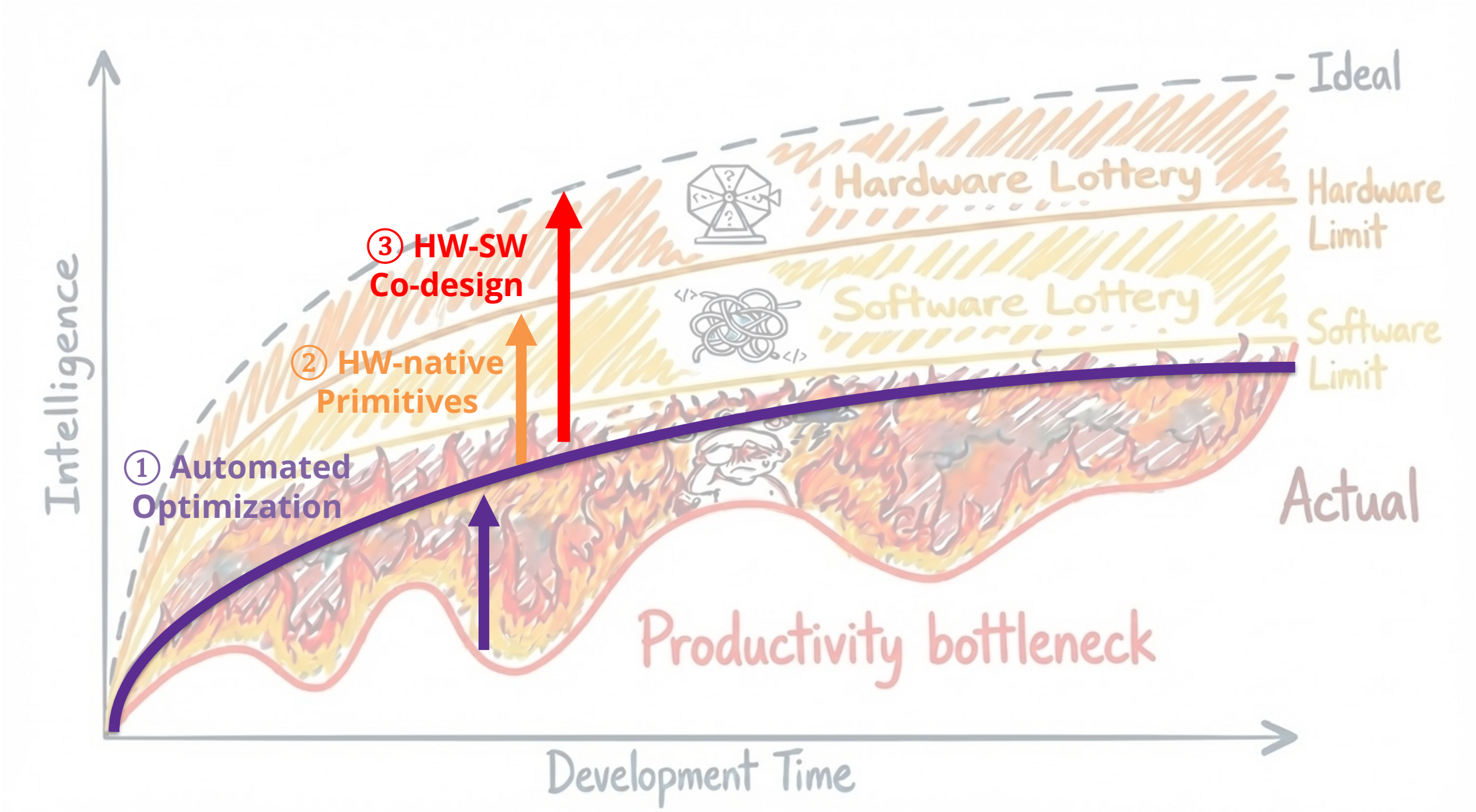
- Distributed computing
- Dataflow pipelining
- Layout organization
- Quantization
- …

The Reality of Scaling



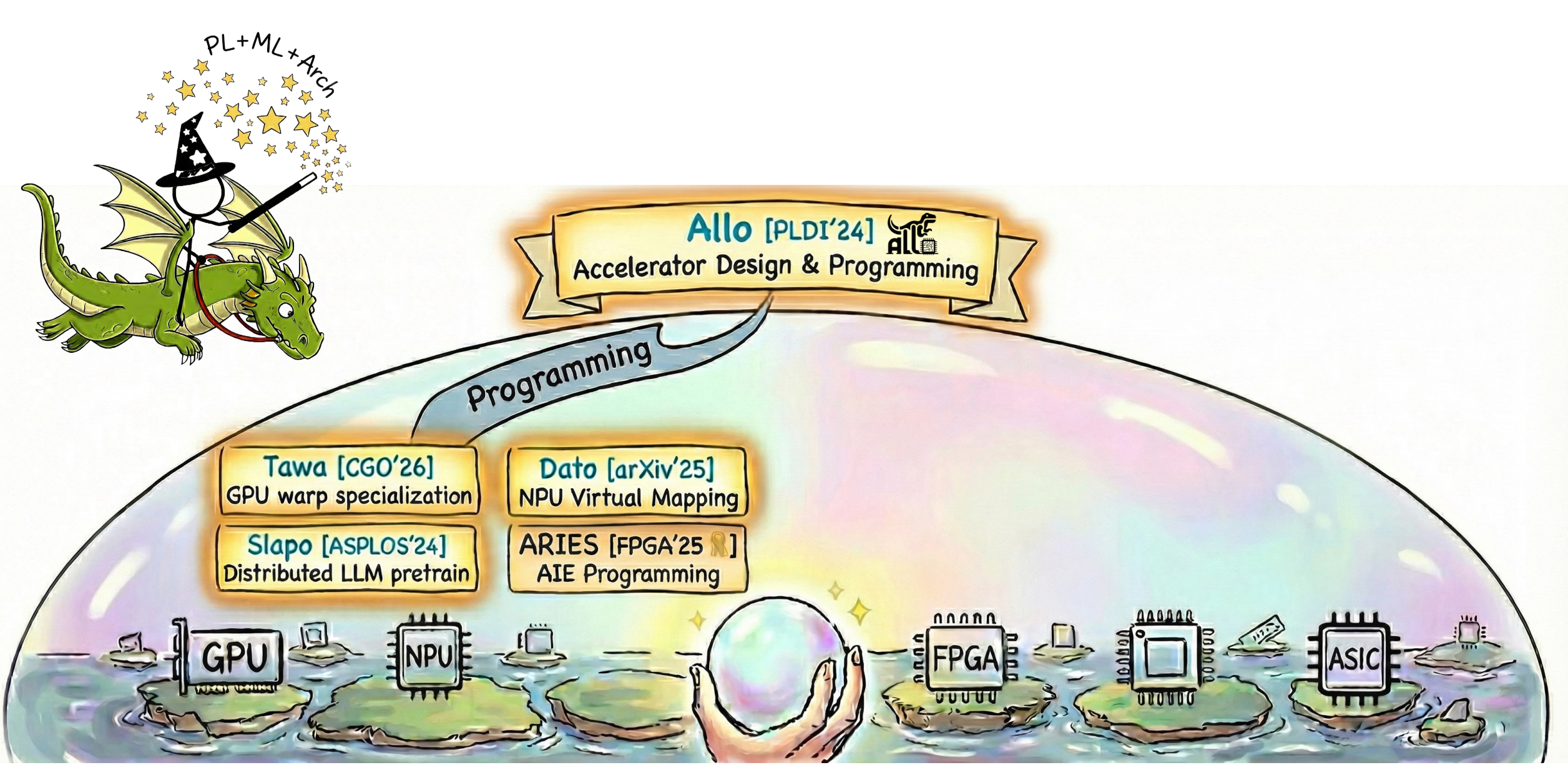
Human requires trials and errors
Biggest bottleneck!

The Reality of Scaling



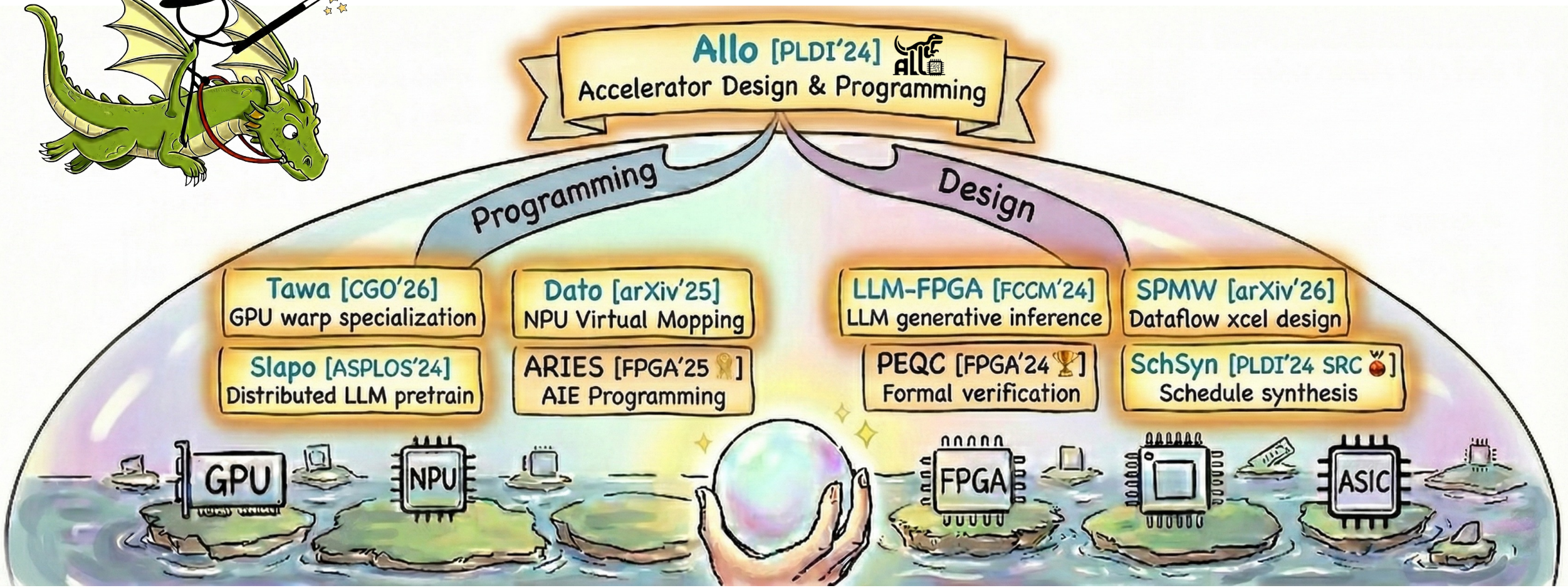
* Highlighted boxes represent first-author works

* Figure generated by Gemini. Source of components : **Dragon**: "Compilers: Principles, Techniques, and Tools"; **Wizard**: "Structure and Interpretation of Computer Programs"; **Sea of accelerators**: "Profiling Hyperscale Big Data Processing"



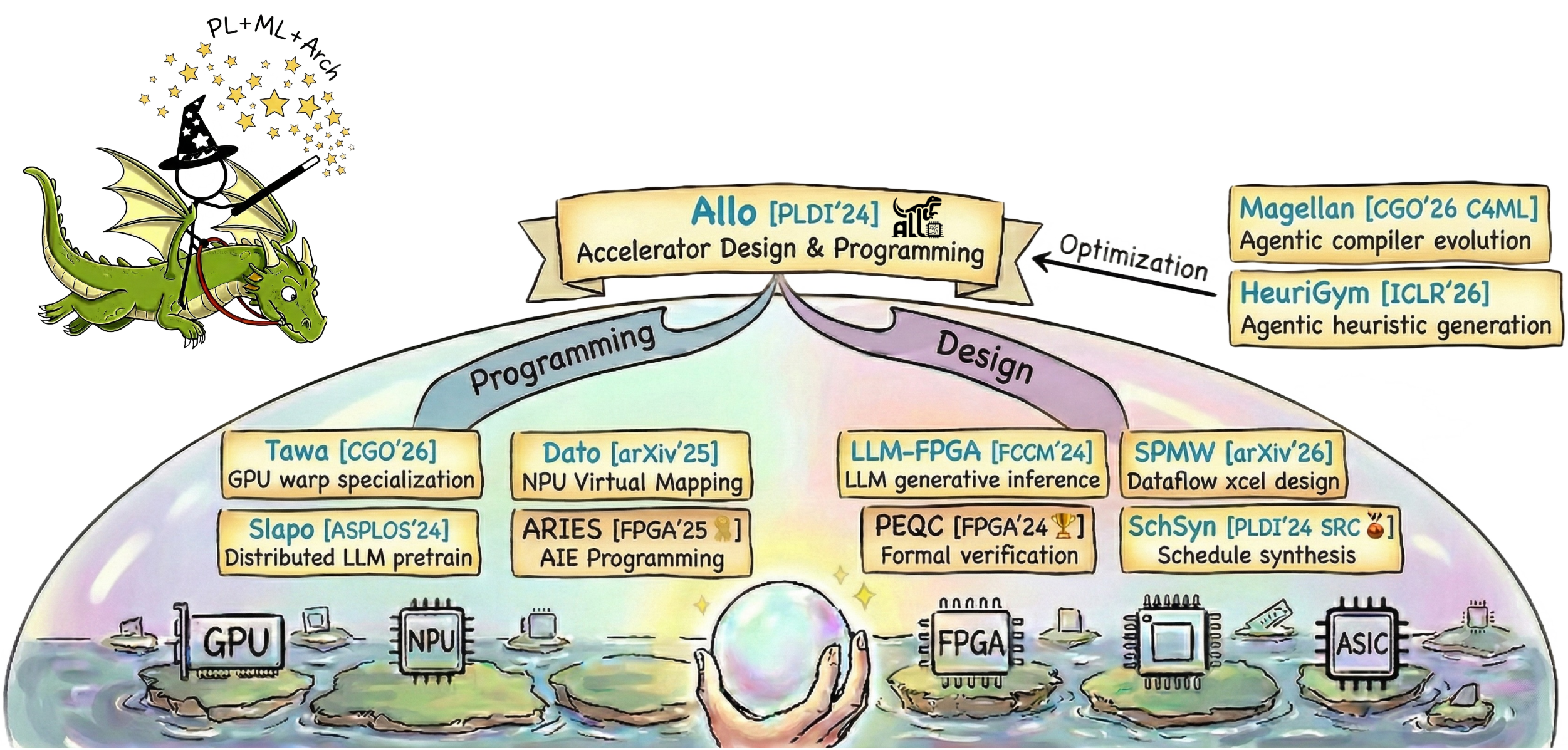
* Highlighted boxes represent first-author works

* Figure generated by Gemini. Source of components : **Dragon**: "Compilers: Principles, Techniques, and Tools"; **Wizard**: "Structure and Interpretation of Computer Programs"; **Sea of accelerators**: "Profiling Hyperscale Big Data Processing"



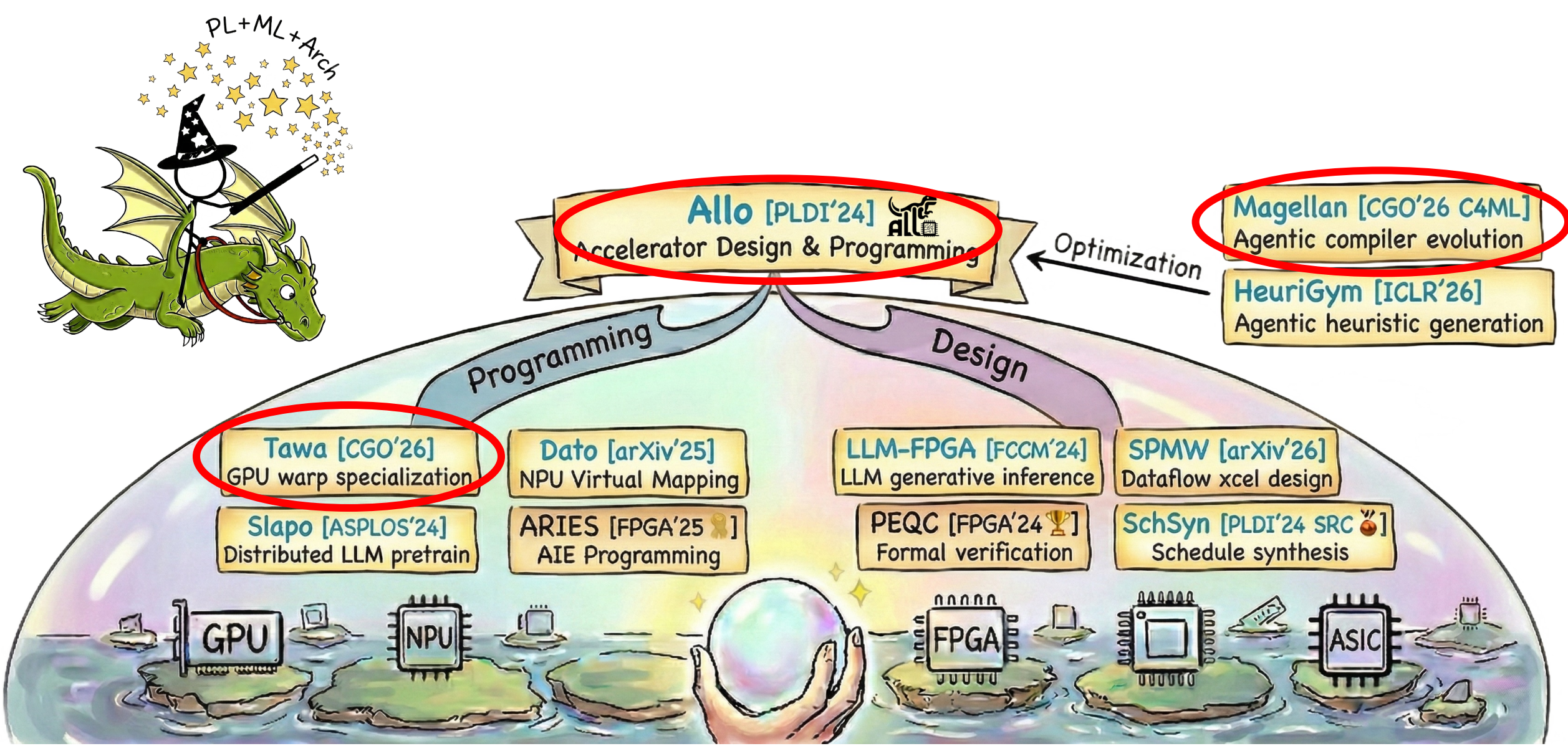
* Highlighted boxes represent first-author works

* Figure generated by Gemini. Source of components : **Dragon**: "Compilers: Principles, Techniques, and Tools"; **Wizard**: "Structure and Interpretation of Computer Programs"; **Sea of accelerators**: "Profiling Hyperscale Big Data Processing"



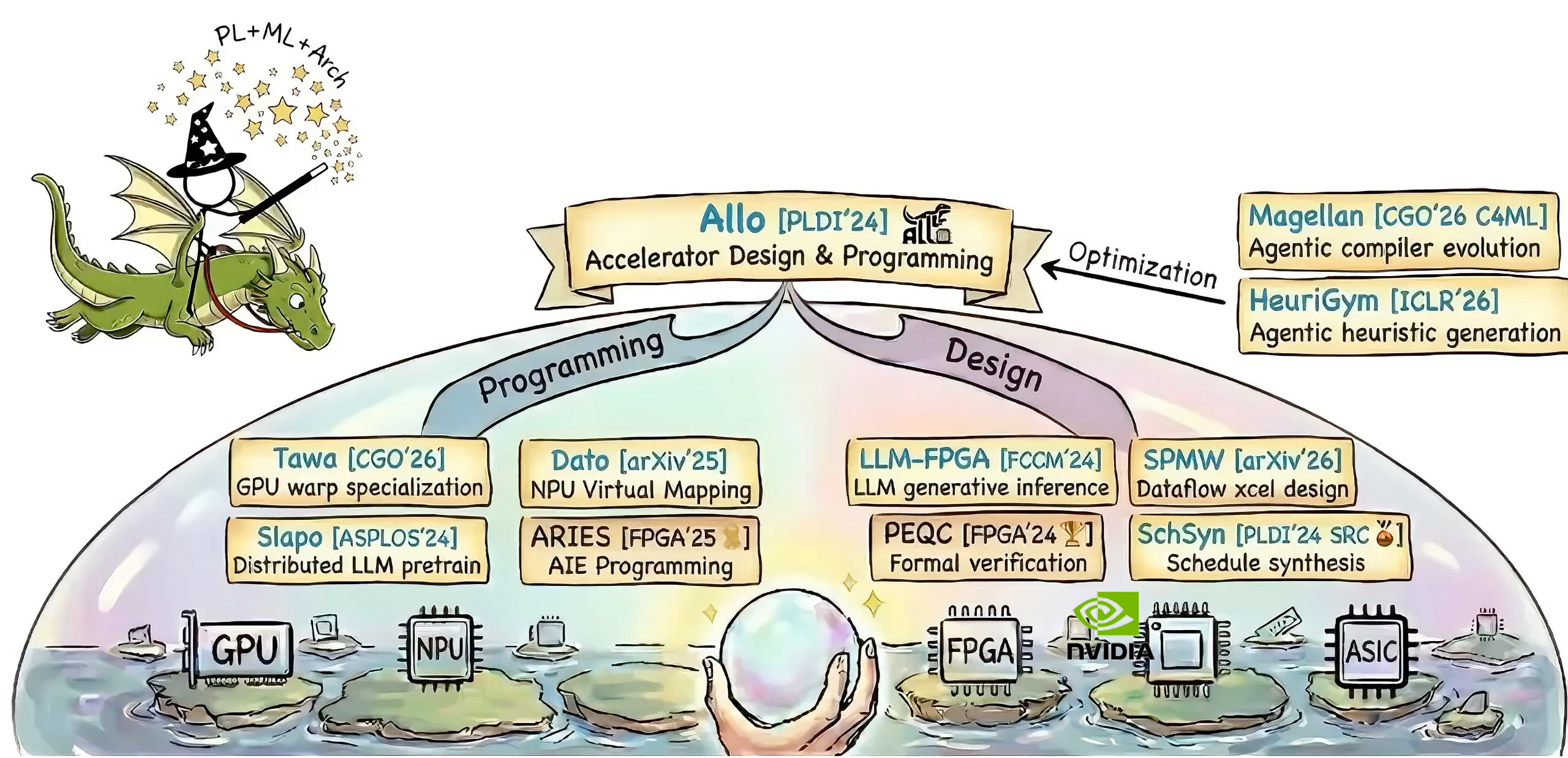
* Highlighted boxes represent first-author works

* Figure generated by Gemini. Source of components : **Dragon**: "Compilers: Principles, Techniques, and Tools"; **Wizard**: "Structure and Interpretation of Computer Programs"; **Sea of accelerators**: "Profiling Hyperscale Big Data Processing"



* Highlighted boxes represent first-author works

* Figure generated by Gemini. Source of components : **Dragon**: "Compilers: Principles, Techniques, and Tools"; **Wizard**: "Structure and Interpretation of Computer Programs"; **Sea of accelerators**: "Profiling Hyperscale Big Data Processing"


PL+ML+Arch
Allo [PLDI'24]
Accelerator Design & Programming
Optimization
Magellan [CGO'26 C4ML]
Agentic compiler evolution
HeuriGym [ICLR'26]
Agentic heuristic generation
Programming
Design
Tawa [CGO'26]
GPU warp specialization
Dato [arXiv'25]
NPU Virtual Mapping
LLM-FPGA [FCCM'24]
LLM generative inference
SPMW [arXiv'26]
Dataflow xcel design
Slapo [ASPLOS'24]
Distributed LLM pretrain
ARIES [FPGA'25]
AIE Programming
PEQC [FPGA'24]
Formal verification
SchSyn [PLDI'24 SRC]
Schedule synthesis
GPU
NPU
FPGA
NVIDIA
ASIC

NVIDIA Hopper and Blackwell GPU Architecture

NVIDIA H100 SXM5 GPU

NVIDIA Blackwell Ultra GPU

Tensor Memory (TMEM)
Storing inputs/outputs of Tensor Cores

Tensor Memory Accelerator (TMA)
<u>Async</u> copy from GMEM to SMEM

Source:
https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/
https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper
https://developer.nvidia.com/blog/inside-nvidia-blackwell-ultra-the-chip-powering-the-ai-factory-era/

NVIDIA Hopper and Blackwell GPU Architecture

NVIDIA H100 SXM5 GPU

NVIDIA Blackwell Ultra GPU

4th Gen Tensor Core
Warp-Group Matrix-Multiply Accumulation (WGMMA)

5th Gen Tensor Core
Unified Matrix-Multiply Accumulation (UMMA)

Source:
https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/
https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper
https://developer.nvidia.com/blog/inside-nvidia-blackwell-ultra-the-chip-powering-the-ai-factory-era/

NVIDIA Hopper and Blackwell GPU Architecture

NVIDIA H100 SXM5 GPU

4th Gen Tensor Core
Warp-Group Matrix-Multiply Accumulation (WGMMA) → **5th Gen Tensor Core**
Unified Matrix-Multiply Accumulation (UMMA)

Performance (TFLOPS)	A100	H100 SXM	HGX B200 (per GPU)
Peak FP16 (Tensor Core, Dense)	312 (80%)	**989.4 (88%)** →	**2250 (94%)**
Peak FP16 (CUDA Core)	78	133.8	150

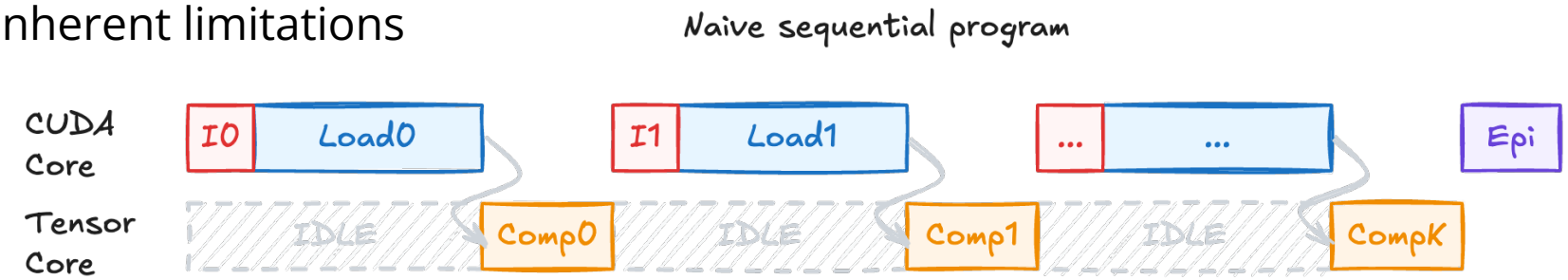
Fully utilize Tensor Core!!!

This work focuses on **Hopper** but also servers as the foundation for Blackwell

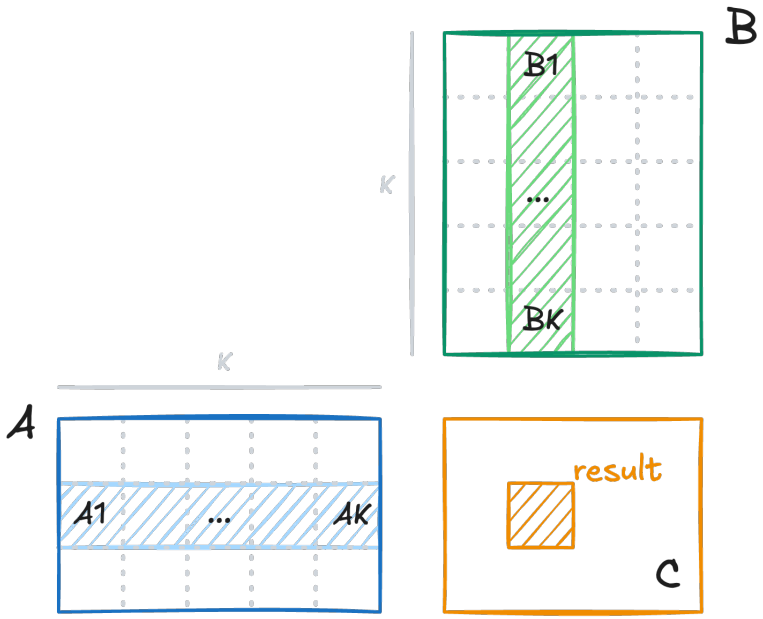
Source:
https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/
https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper
https://developer.nvidia.com/blog/inside-nvidia-blackwell-ultra-the-chip-powering-the-ai-factory-era/

Different Execution Schemes: Sequential

- **Data loading** is time-consuming
- Underutilized Tensor Cores
- Pipelining can help, but has inherent limitations



* Actual latency not drawn to scale




```
result = 0.0
for k in range(K):
  ta = load(A, k)
  tb = load(B, k)
  result += dot(ta, tb)
store(result, C)
```

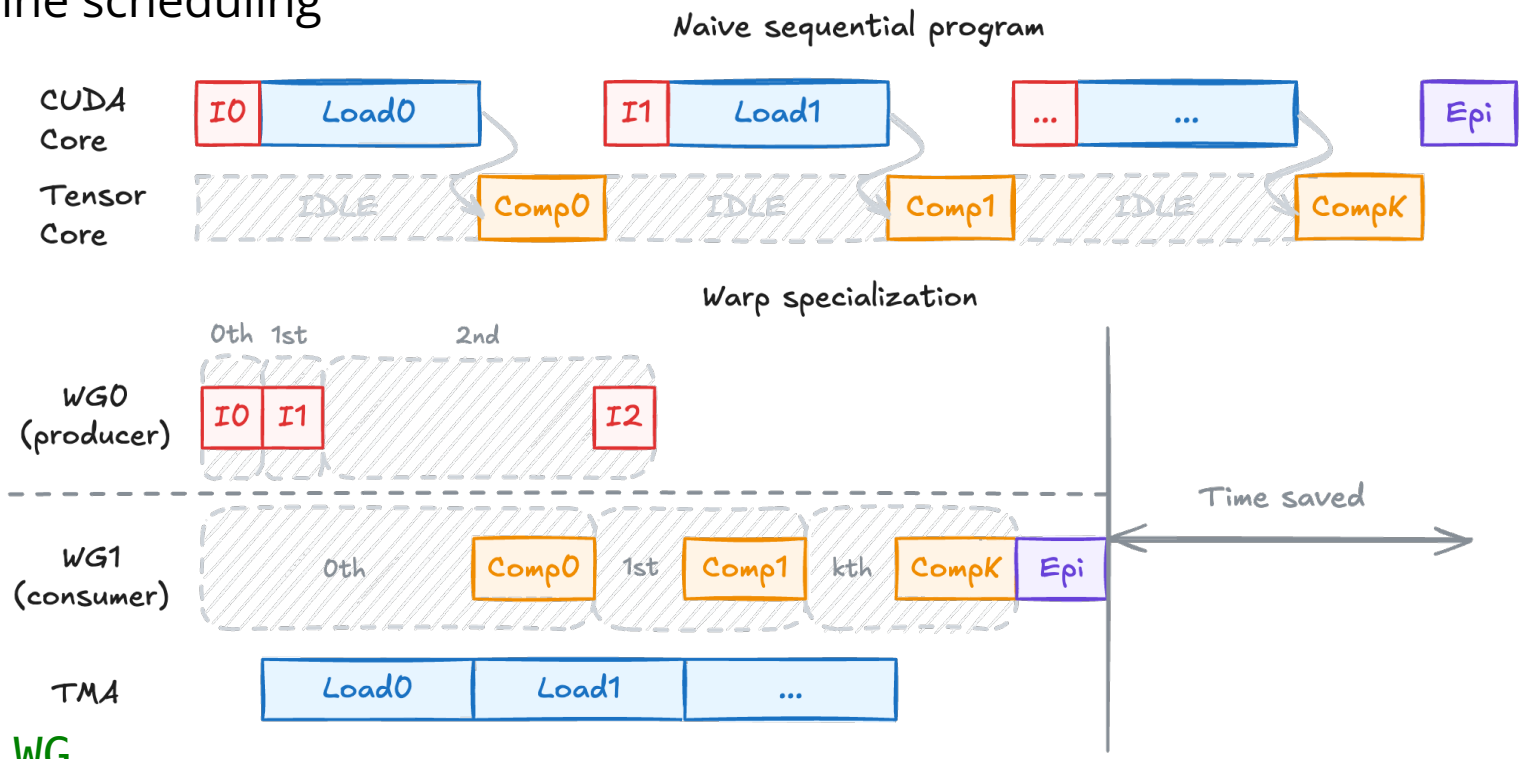
Load

Comp

Epi

Different Execution Schemes: Warp Specialized

- Instructions in different WG are naturally async
  - Significantly simpler than pipeline scheduling
- New communication mechanisms
  - Async TMA
  - Async MMA



* Actual latency not drawn to scale

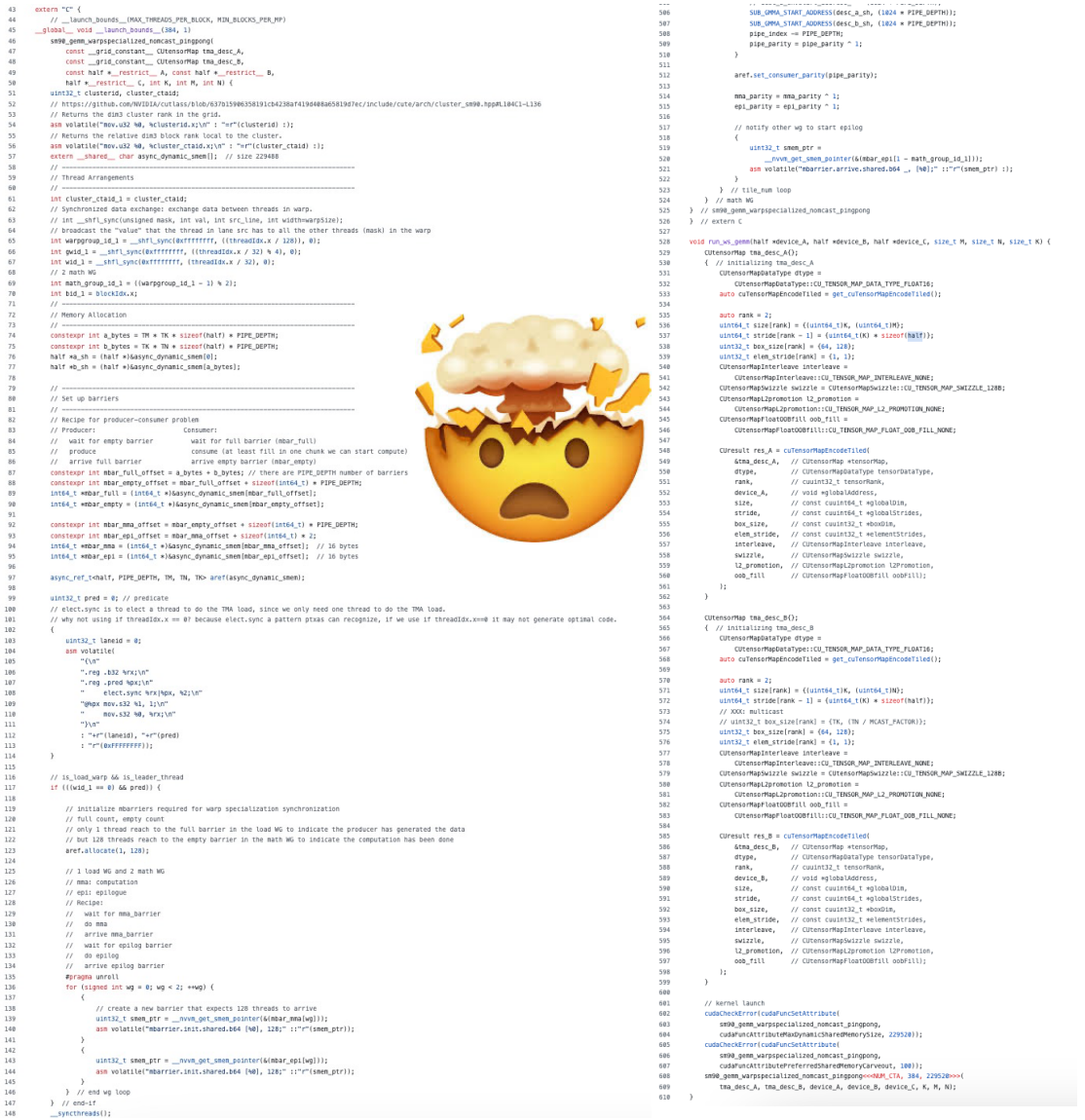
```
# Producer WG
for k in range(K):
  ta = load(A, k)
  tb = load(B, k)
  shmem[...] = ta
  shmem[...] = tb
```

```
# Consumer WG
for k in range(K):
  ta = shmem[...]
  tb = shmem[...]
  result += dot(ta, tb)
  store (result, C)
```

Challenges of Writing Warp Specialized Programs

- ~1k lines of CUDA code
  for a single GEMM kernel on Hopper

- Optimizations
  - TMA asynchronous load
  - Warpgroup partitioning
  - MMA pipelining
  - Layout transformation



Challenge 1: Coordinating Concurrent Warp Roles

- CUDA SIMT
  - Suppose all the threads do the same thing

```
for (int k = 0; k < K; ++k) {
  tma_load((half*)(&shmem[...]),
           tma_desc_A, offA0, offA1);
  tma_load((half*)(&shmem[...]),
           tma_desc_B, offB0, offB1);
  wgmma(...);
}
```

Need to significantly restructure the code to separate the roles

- Warp Specialization
  - Different warps play different roles

```
if (wgid == 0) {
  // producer
  for (int k = 0; k < K; ++k) {
    // TMA load
    // ...
  }
}
```

```
if (wgid == 1) {
  // consumer
  for (int k = 0; k < K; ++k) {
    // WGMMA
    // ...
  }
}
```

Challenge 2: Low-Level Communication Management

- Manually handle synchronization between warps
  - Inlined PTX
  - A set of carefully designed mbarriers

```
#pragma unroll
for (int i = 0; i < N; i++) {
  uint32_t full_ptr =
     __nvvm_get_smem_pointer(&shmem[...]);
  asm volatile("mbarrier.init.shared.b64
                [%0], %1;"::
               "r"(full_ptr),
               "r"(full_count));
  uint32_t empty_ptr =
     __nvvm_get_smem_pointer(&shmem[...]);
  asm volatile("mbarrier.init.shared.b64
                [%0], %1;"::
               "r"(empty_ptr),
               "r"(empty_count));
}
```

Error-prone, low readability,
hard to debug and maintain

```
// producer
for (int k = 0; k < K; k++) {
  mbarrier::try_wait(empty_mbar, prod_parity);
  mbarrier::expect_tx(full_mbar, size);
  // load
  // ...
  mbarrier::arrive(full_mbar);
  // update parity
  // ...
}

// consumer
for (int k = 0; k < K; k++) {
  mbarrier::try_wait(full_mbar, cons_parity);
  // compute
  // ...
  mbarrier::arrive(empty_mbar);
  // update parity
  // ...
}
```

Challenge 3: Resource Allocation and Pipelining

- Manually allocate shared memory and registers for each warp group
- Manually split the MMA pipeline

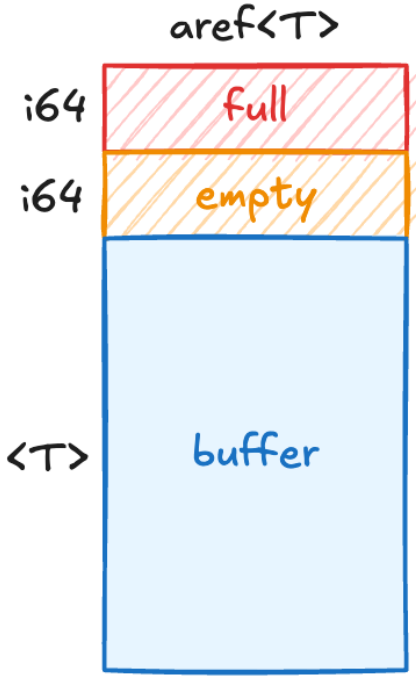
```
constexpr int a_bytes = TM * TK * sizeof(half) * PIPE_DEPTH;
constexpr int b_bytes = TK * TN * sizeof(half) * PIPE_DEPTH;
half *a_sh = (half *)&async_dynamic_smem[0];
half *b_sh = (half *)&async_dynamic_smem[a_bytes];
if (wgid == 0) {
  // producer
  asm volatile("setmaxnreg.dec.sync.aligned.u32 40;" ::);
  // ...
}
if (wgid == 1) {
  // consumer
  asm volatile("setmaxnreg.inc.sync.aligned.u32 232;" ::);
  // ...
}
```

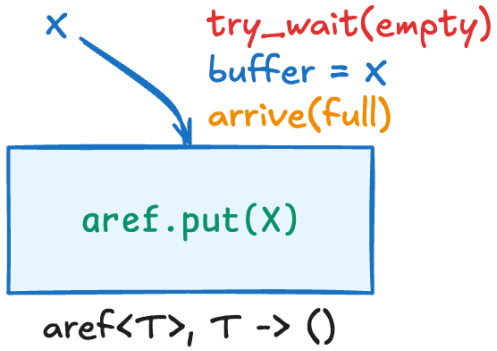
```
// MMA prologue
for (int k0 = 0; k0 < MMA_STAGES - 1; k0++) {
  // issue a few wgmma
  // ...
}
// MMA mainloop
#pragma unroll
for (int k1 = MMA_STAGES - 1; k1 < (K + 63) / 64; k1++) {
  #pragma unroll
  for (int mma_k = 0; mma_k < 4; mma_k++) {
    // wgmma
  }
  asm volatile("wgmma.wait_group.sync.aligned %0;" ::
               "n"((MMA_STAGES - 1)));
  mbarrier::arrive(empty_mbar[...])
}
// MMA epilogue
asm volatile("wgmma.wait_group.sync.aligned %0;" ::"n"(0));
for (int k2 = MMA_STAGES - 1; k2 > 0; k2--) {
  mbarrier::arrive(empty_mbar[...]);
}
```

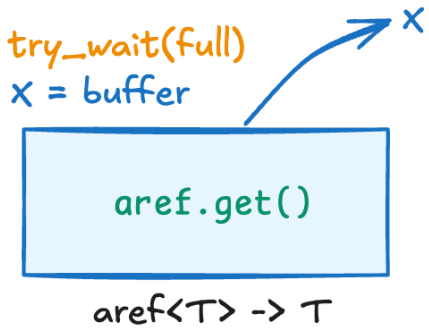
Again, need to significantly refactor the code for each design point

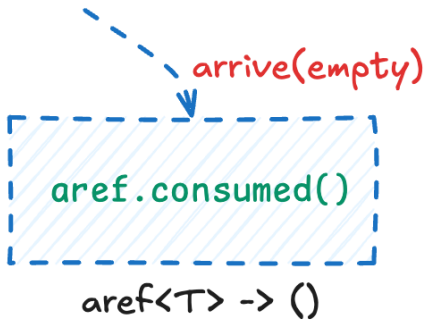
Asynchronous Reference (Aref)

Aref: A set of mbarriers and a shared memory buffer





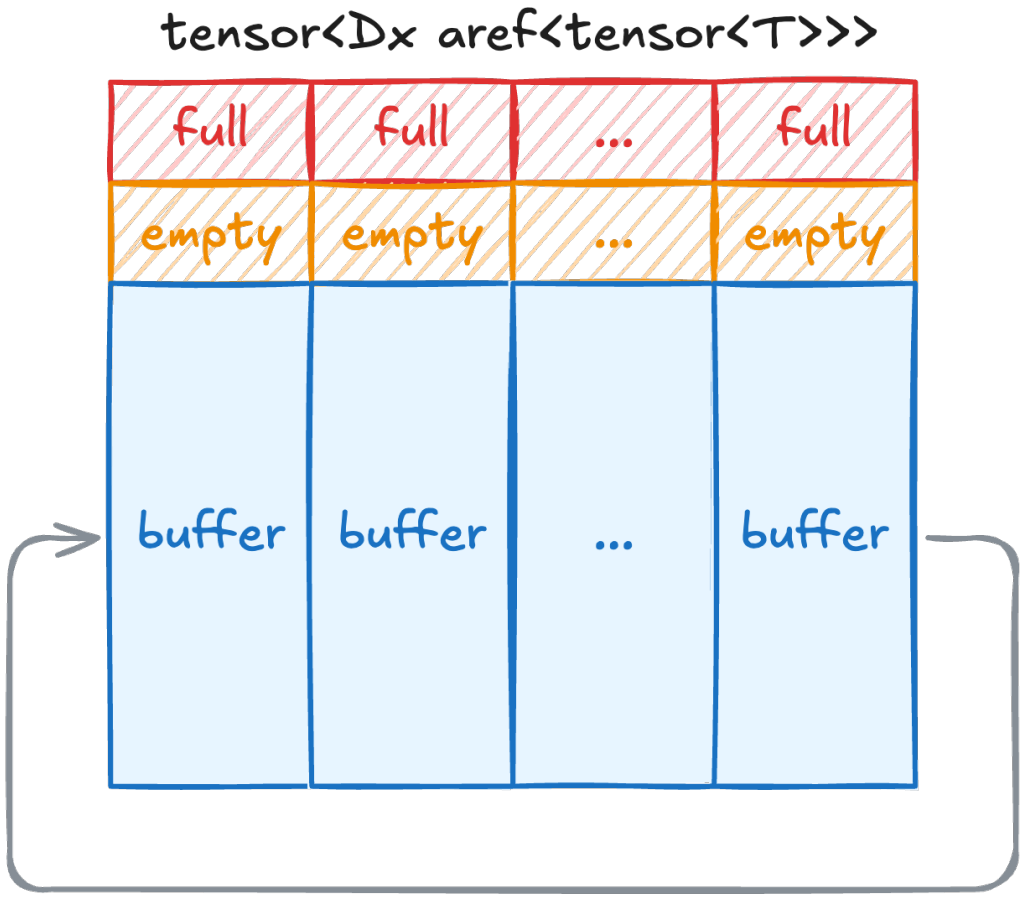


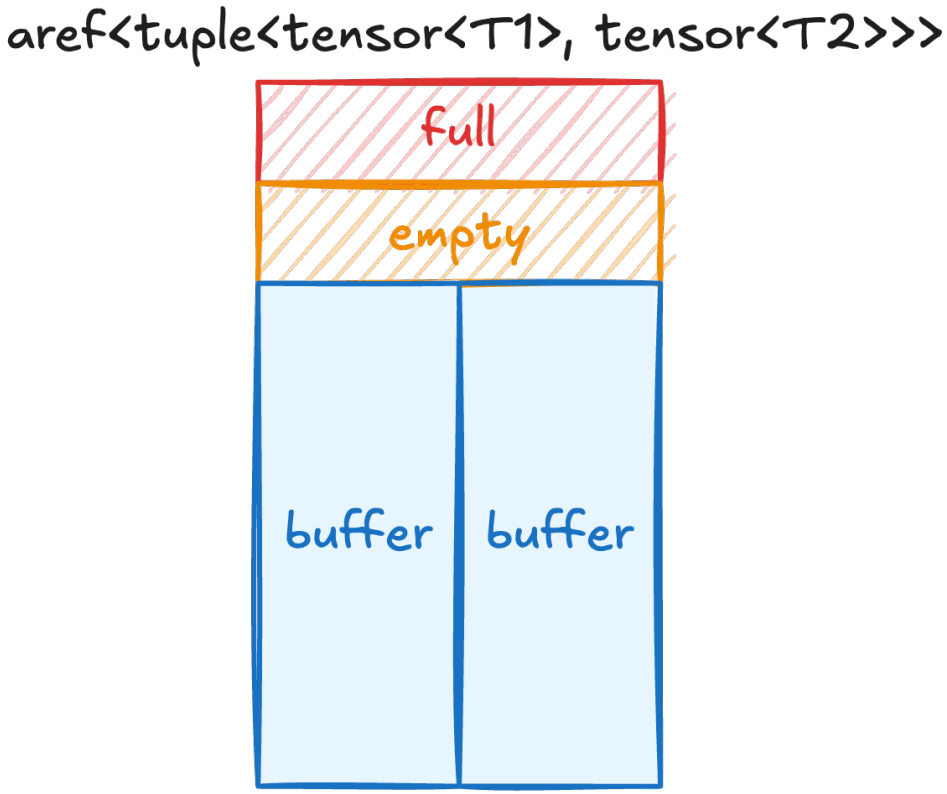


* Operational semantics defined in the paper

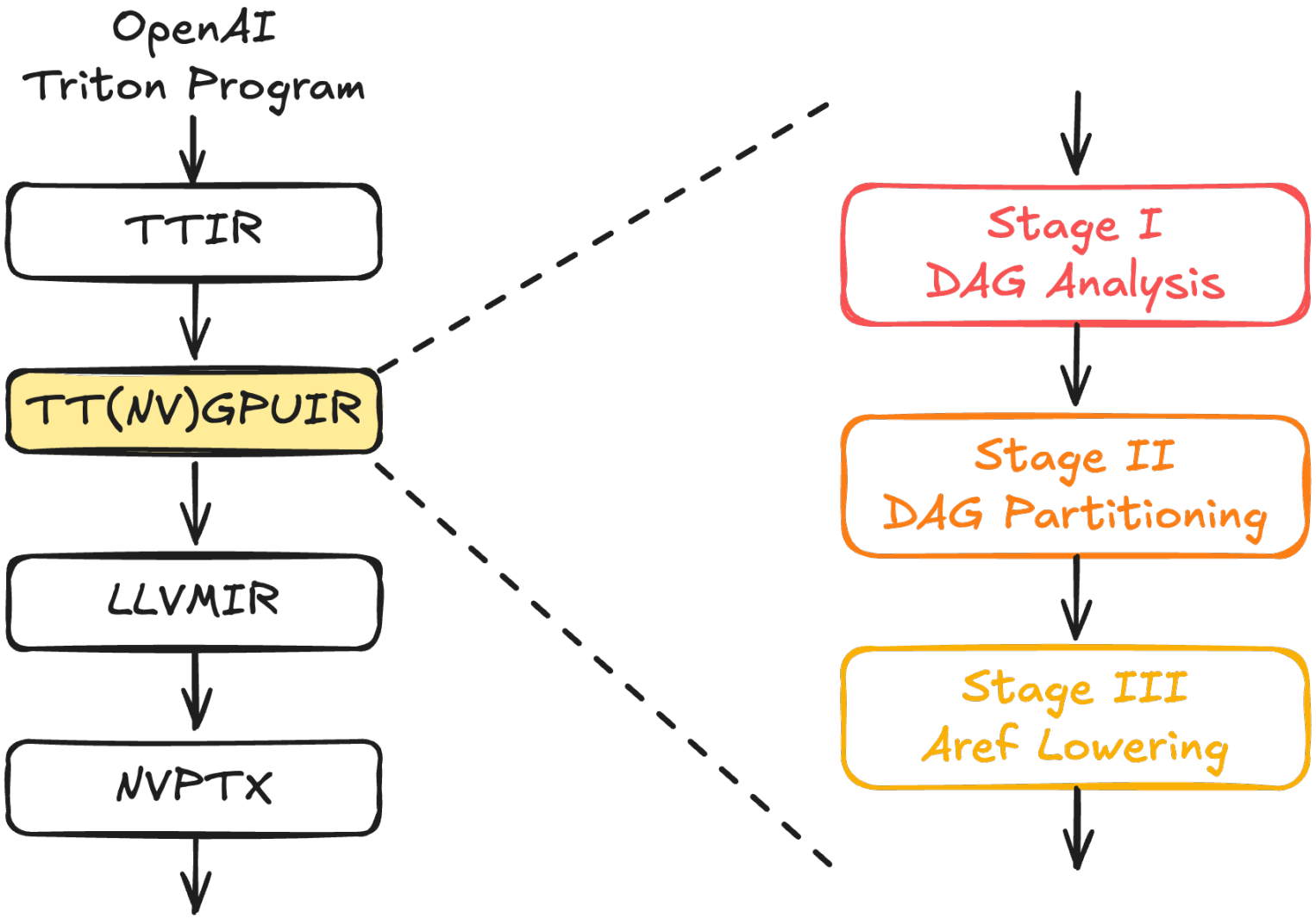
Asynchronous Reference (Aref)

- Cyclic buffer (D: aref size)
  - Limited GPU shared memory
- Shared states
  - Reduce mbarrier overheads



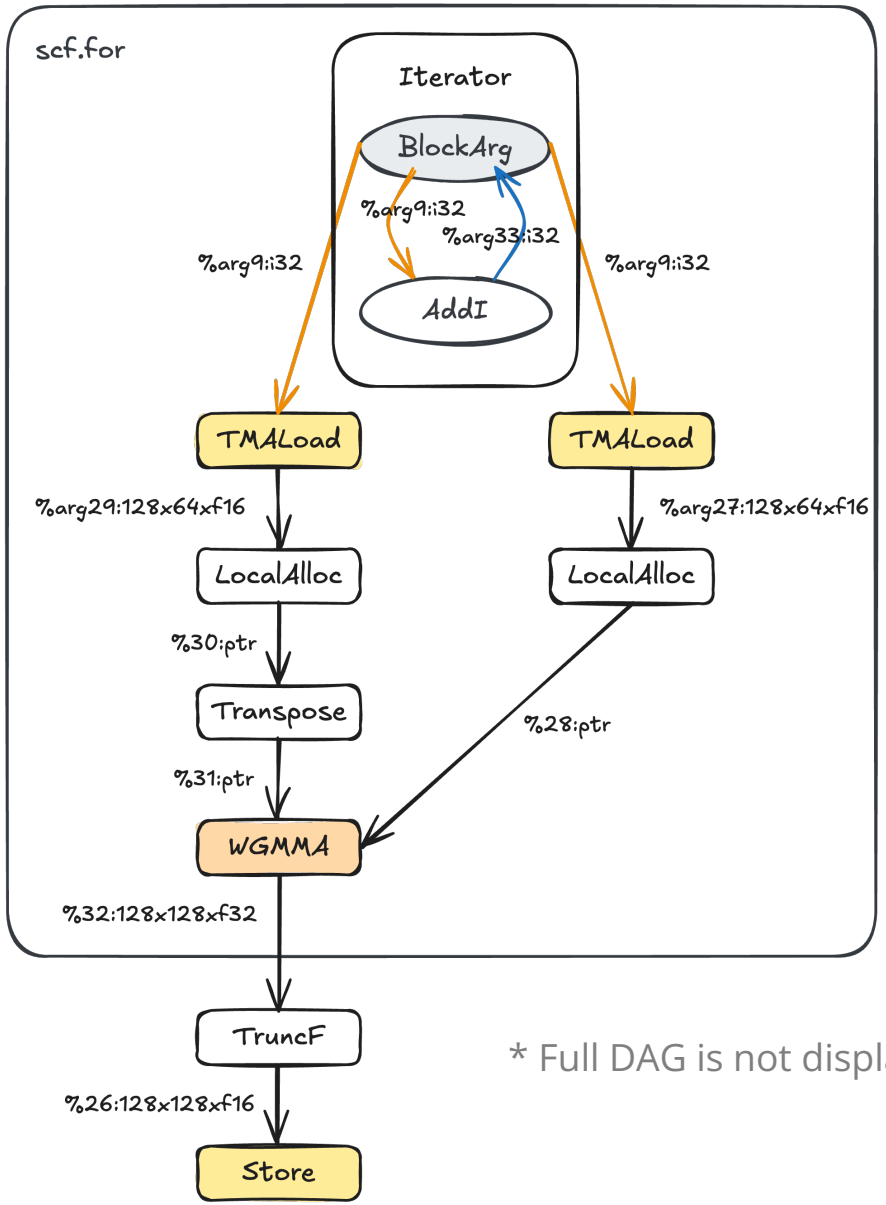


Task-Aware Warp-specialization with Aref (Tawa)



Stage I: DAG Analysis

- Backward traversal through use-def chain
- Tile statement (Black edges)
  - Perform actual computation
- Iteration statement (Orange/Blue edges)
  - Pointer arithmetic
  - Not necessarily consecutive in the IR

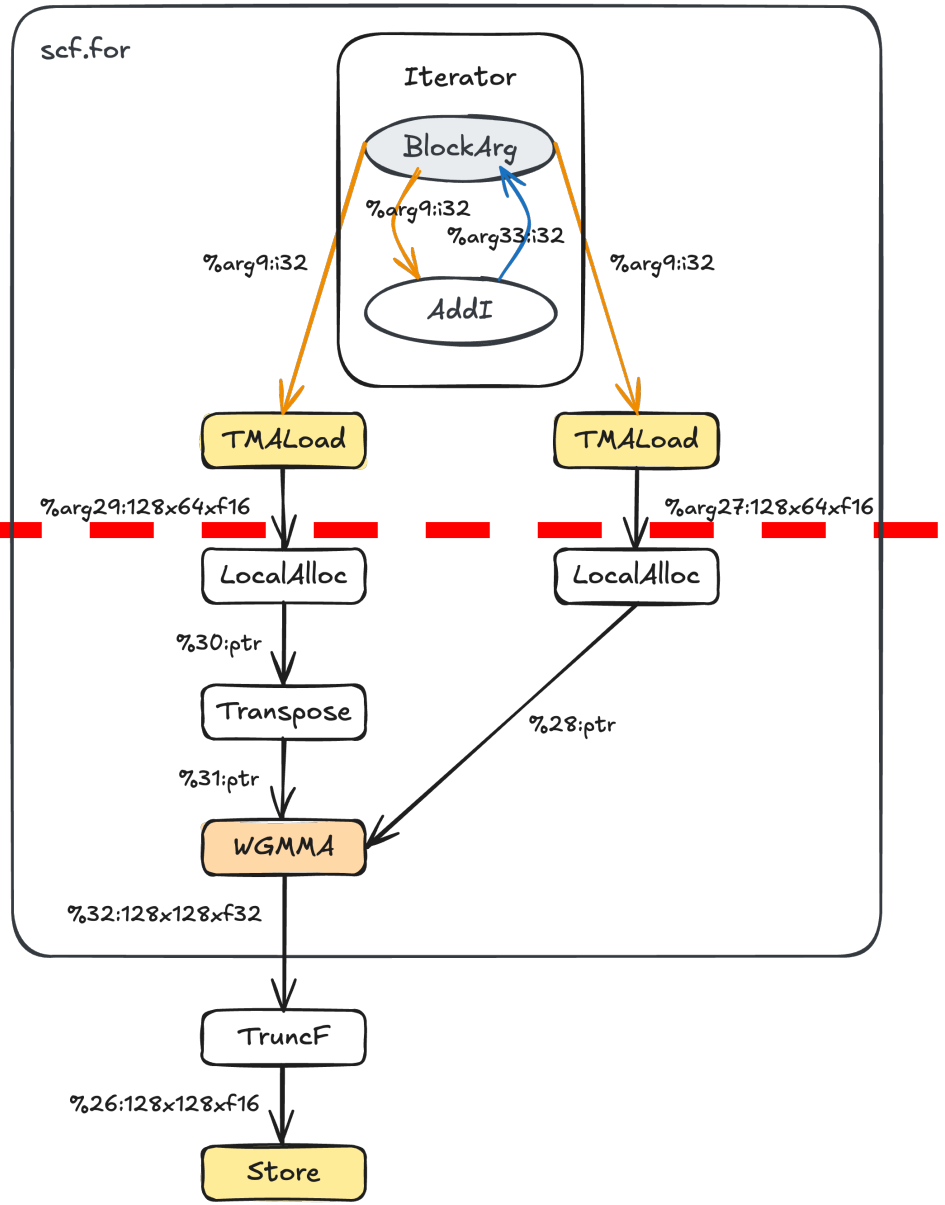


* Full DAG is not displayed here

Stage II: DAG Partitioning & Aref Generation

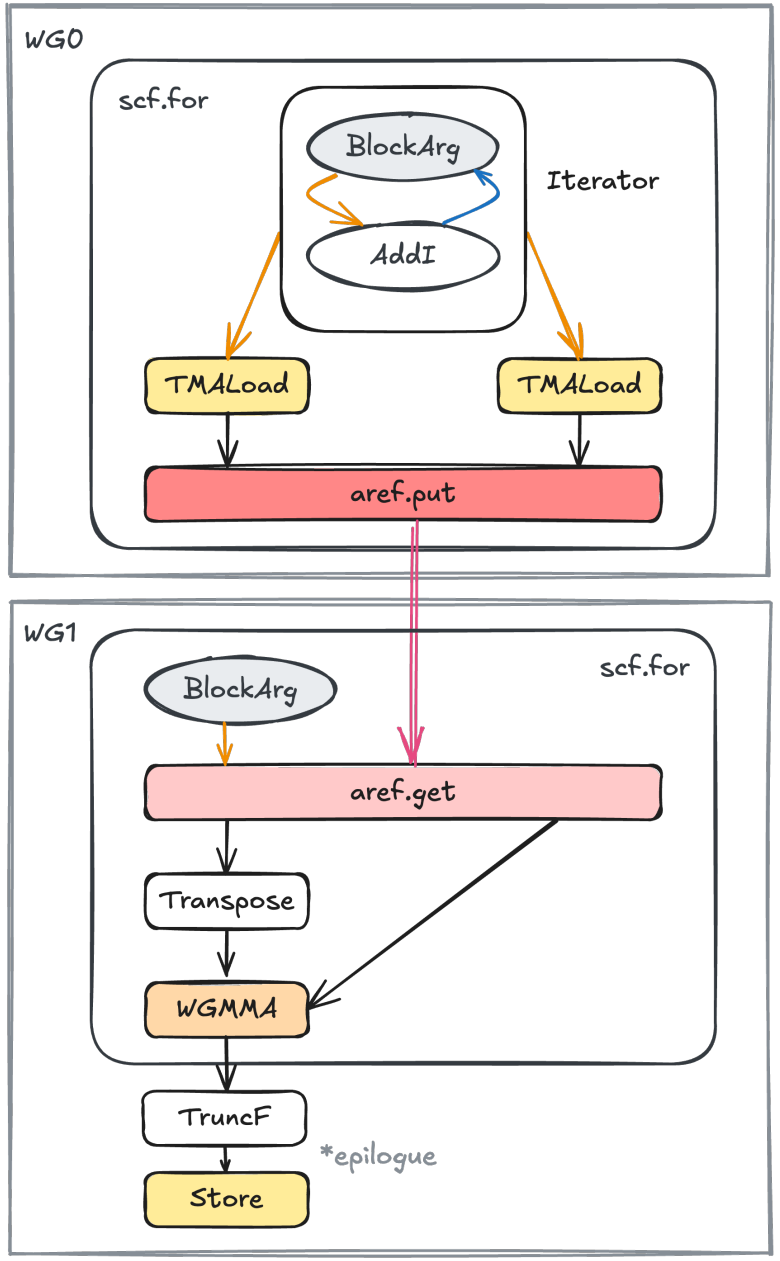
- Tag partitions (# of Partitions=2)
  - Partition 0 (WG0): Load (Iteration statements)
  - Partition 1 (WG1): Compute (Tile statements)
  - Partition 1 also needs to include the epilogue

  - For Blackwell, we need more semantics roles



Stage II: DAG Partitioning & Aref Generation

- Tag partitions (# of Partitions=2)
  - Partition 0 (WG0): Load (Iteration statements)
  - Partition 1 (WG1): Compute (Tile statements)
  - Partition 1 also needs to include the epilogue
- Loop distribution

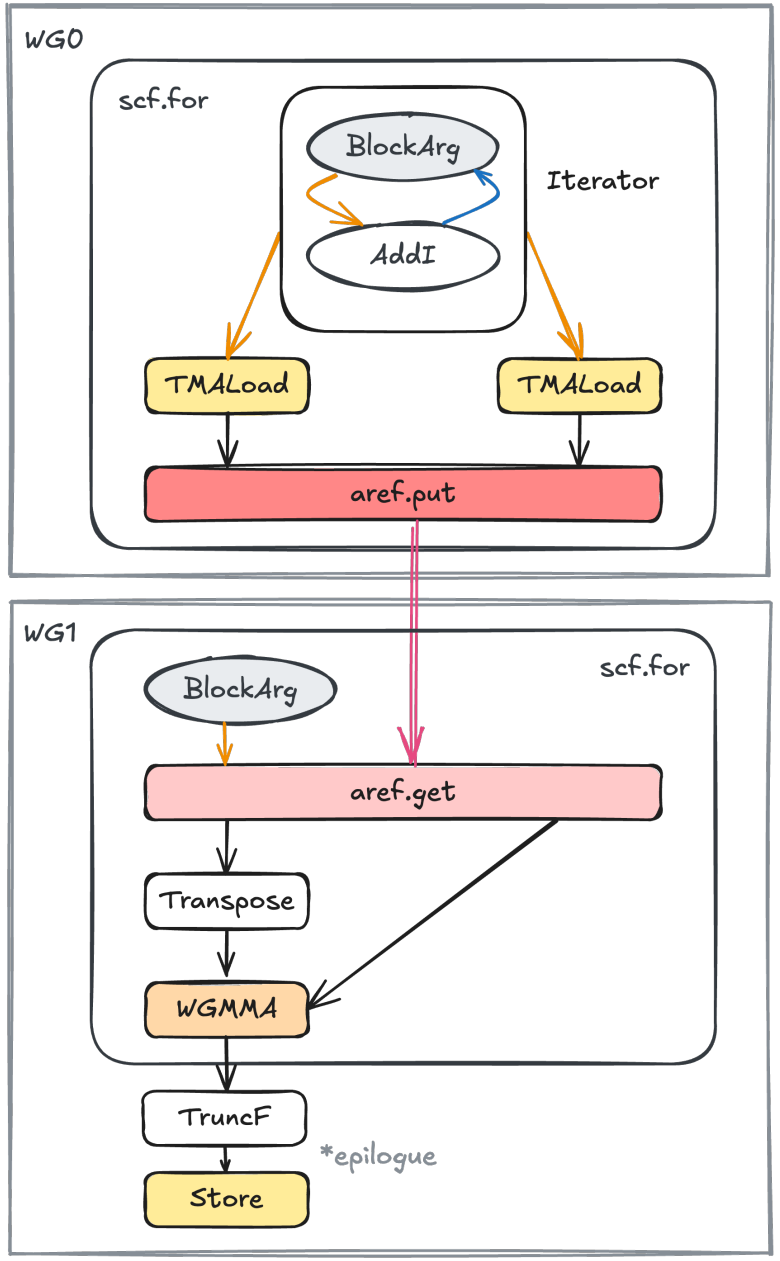


Stage II: DAG Partitioning & Aref Generation

- Tag partitions (# of Partitions=2)
  - Partition 0 (WG0): Load (Iteration statements)
  - Partition 1 (WG1): Compute (Tile statements)
  - Partition 1 also needs to include the epilogue
- Loop distribution
  - Create cyclic aref objects (size D)

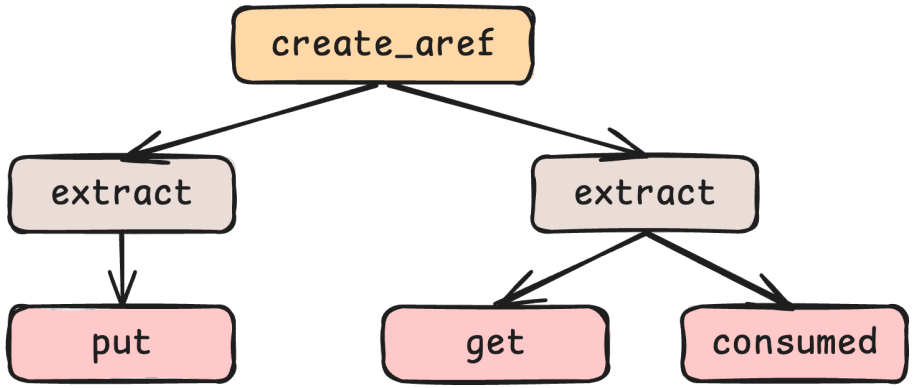
```
taref = aref.create() # Dx aref<tensor<128x128xf16>>
# WG0
for k in range(K):
  ta = tma_load(A, offs_am, offs_k)
  tb = tma_load(B, offs_bn, offs_k)
  taref[k % D].put(ta, tb)
  offs_k += BLOCK_SIZE_K
# WG1
for k in range(K):
  ta, tb = taref[k % D].get()
  acc += wgmma(ta, tb)
  taref[k % D].consumed()
```

* Pseudocode for demonstration



Stage III: Aref Lowering

- Pattern matching and rewrite
  - Only after rewriting the whole subgraph, it can be erased



- Parity
  - A mechanism to avoid deadlock
  - Parity=1, it will skip the waiting

create_aref

```
%buffer = local_alloc : memdesc<Dx51x52xf16>
%full_mbar = local_alloc : memdesc<Dxi64>
%empty_mbar = local_alloc : memdesc<Dxi64>
init_barrier %full_mbar[i], 32*numWarps
init_barrier %empty_mbar[i], 1
```

put

```
wait_barrier %empty_mbar[k % D], parity
barrier_expect %full_mbar[k % D]
async_tma_load %buffer[k % D]
```

get

```
wait_barrier %full_mbar[k % D], parity
```

consumed

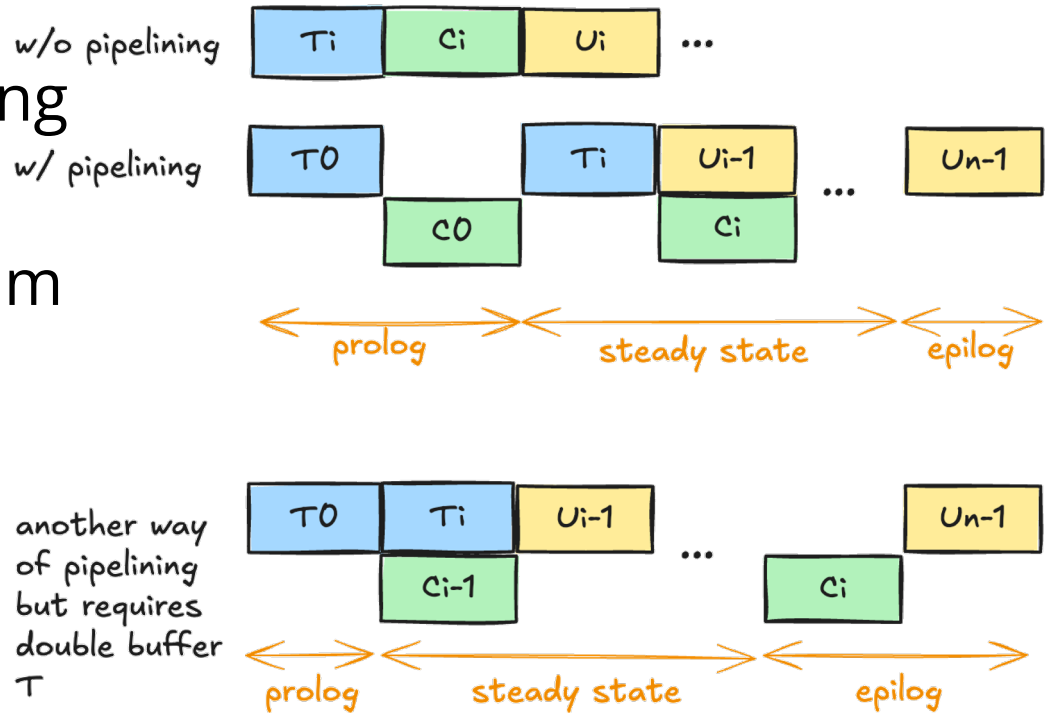
```
arrive_barrier %empty_mbar[k % D]
```

CUDA and Tensor Core Pipelining

- Focus on compute WG
  - Leverage asynchronous MMA
  - GEMM: overlap address computation and mma
  - FMHA: overlap softmax and mma

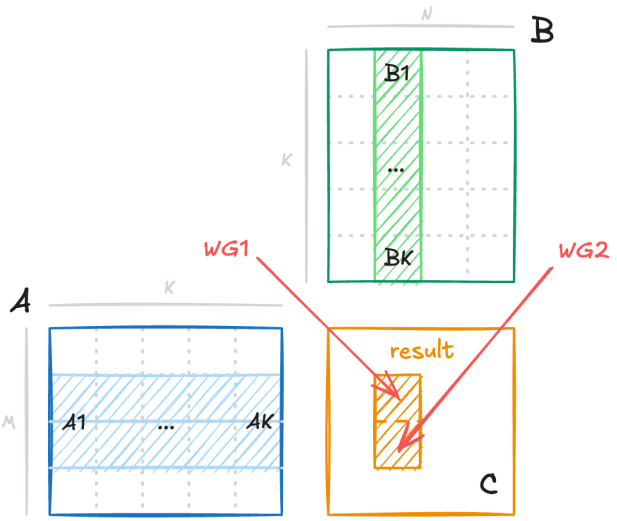
- After aref generation before aref lowering

- Extract different stages from the program
  - Tensor Core Stage: T_i
  - CUDA Core Stage: C_i
  - Downstream Tensor Core Stage: U_j

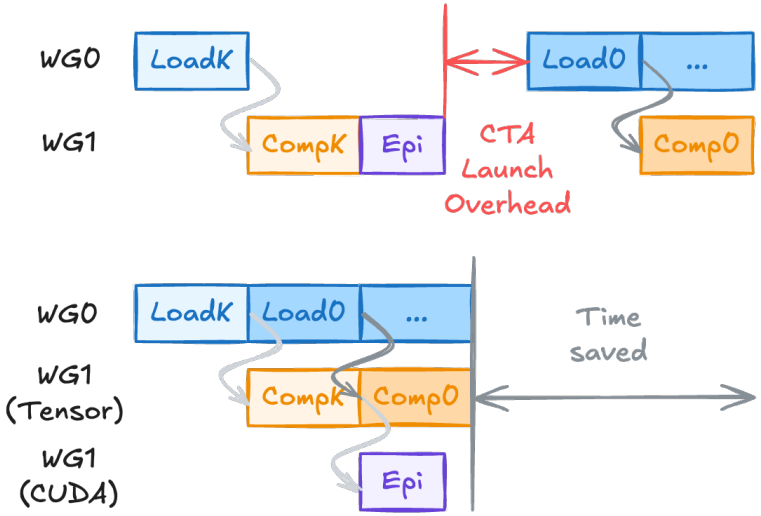


Source level optimizations

- Support the following source level optimizations
- Cooperative Warp Groups
  - Two WGs cooperate on a larger tile
  - Higher register pressure, mitigated via setmaxreg
  - Warp IDs are marked in the IR, codegen handles the rest



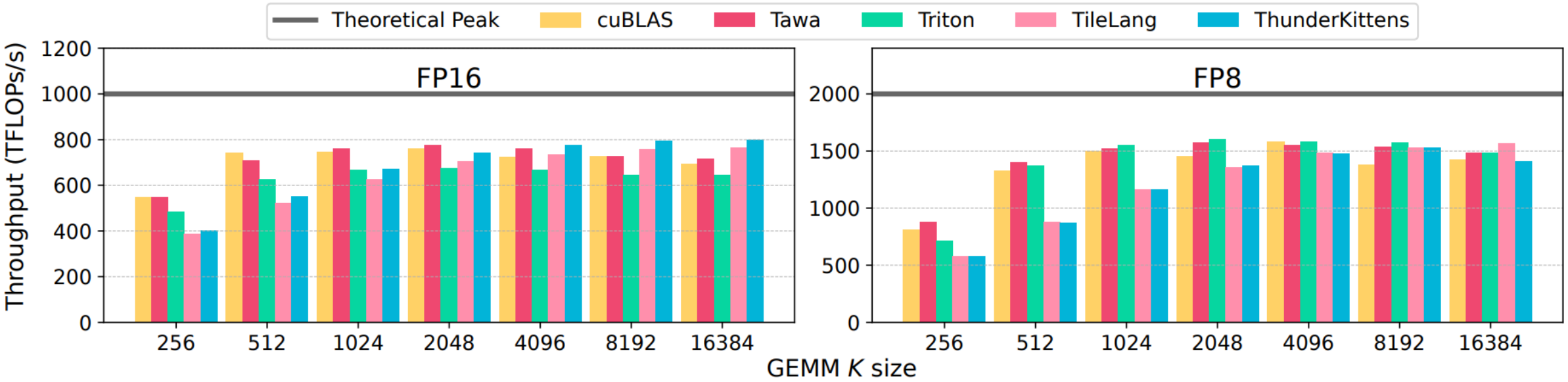
- Persistent Kernel
  - Launch num_sm CTAs
  - Looping over ALL tiles with the stride of num_sm
  - Writes output when processing the final k tile (complex control flow)
  - Reduce CTA scheduling overhead
  - Overlap the epilog with the next tile



Experimental Settings

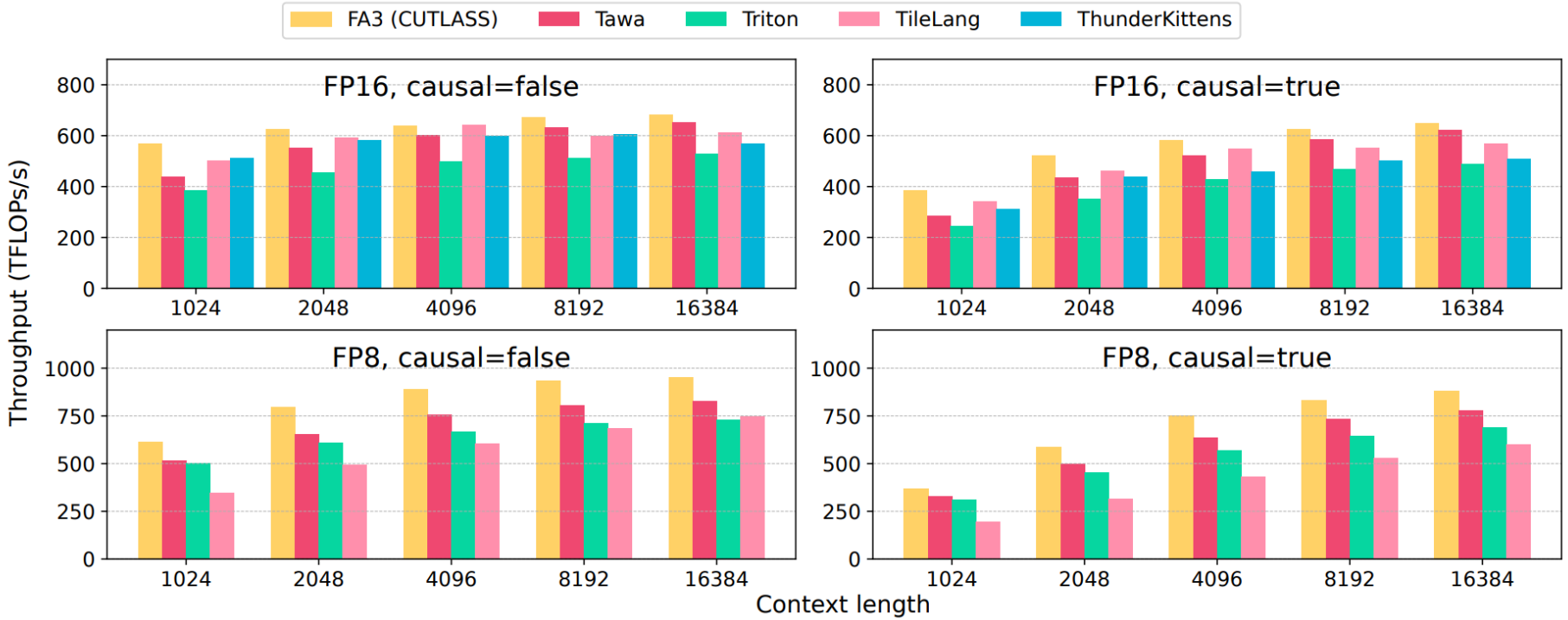
- Baselines
  - cuBLAS v12.7
  - Triton (0c7edf, one-year-old baseline, not yet had WS in late 2024)
    - **Default software pipelining (swp) is on**. Pipeline depth is predetermined.
  - Triton-Aref (Tawa)
    - Aref size and MMA pipeline depths are manually chosen to maximize performance
  - TileLang
  - ThunderKittens
- Hardware
  - GPU: NVIDIA H100 SXM5 HBM3e
  - CUDA 12.7
- Benchmark
  - GEMM 8192x8192, varied K from 128 to 2048
    - 8 warp groups for non-warp-specialized kernels, and 4+8 warp groups for warp-specialized kernels
  - MHA N=4, D=128

GEMM FP16 & FP8 Results



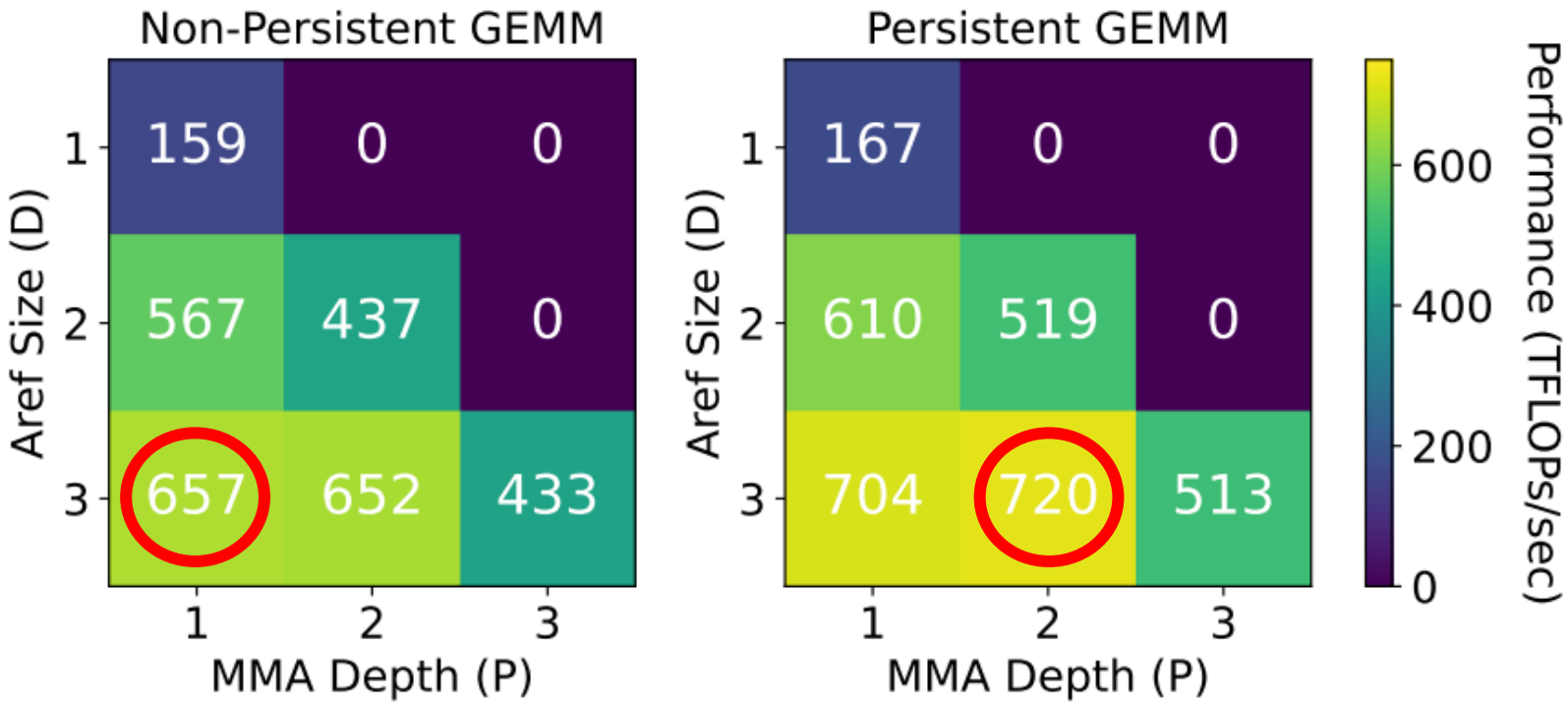
- 8192x8192xK, best tuned tile size
- cuBLAS: **1.01x** on FP16, 1.06x on FP8
- Triton (swp): **1.13x** on FP16, 1.02x on FP8 (overheads dominate)
- Both TileLang and ThunderKittens achieve good performance on FP16, but worse on FP8

Flash Attention FP16 & FP8 Results



- N=4, D=128
- FP16: **96% of FA3**, 1.21x over Triton
- FP8: 89% of FA3, 1.11x over Triton
- Perform better with long sequences
- ThunderKittens failed in FP8

Hyperparameter Section



- Increasing aref size generally improves performance
- Setting MMA pipeline depth to 1 or 2 is enough
- In all the cases, persistent kernels are better than non-persistent kernels

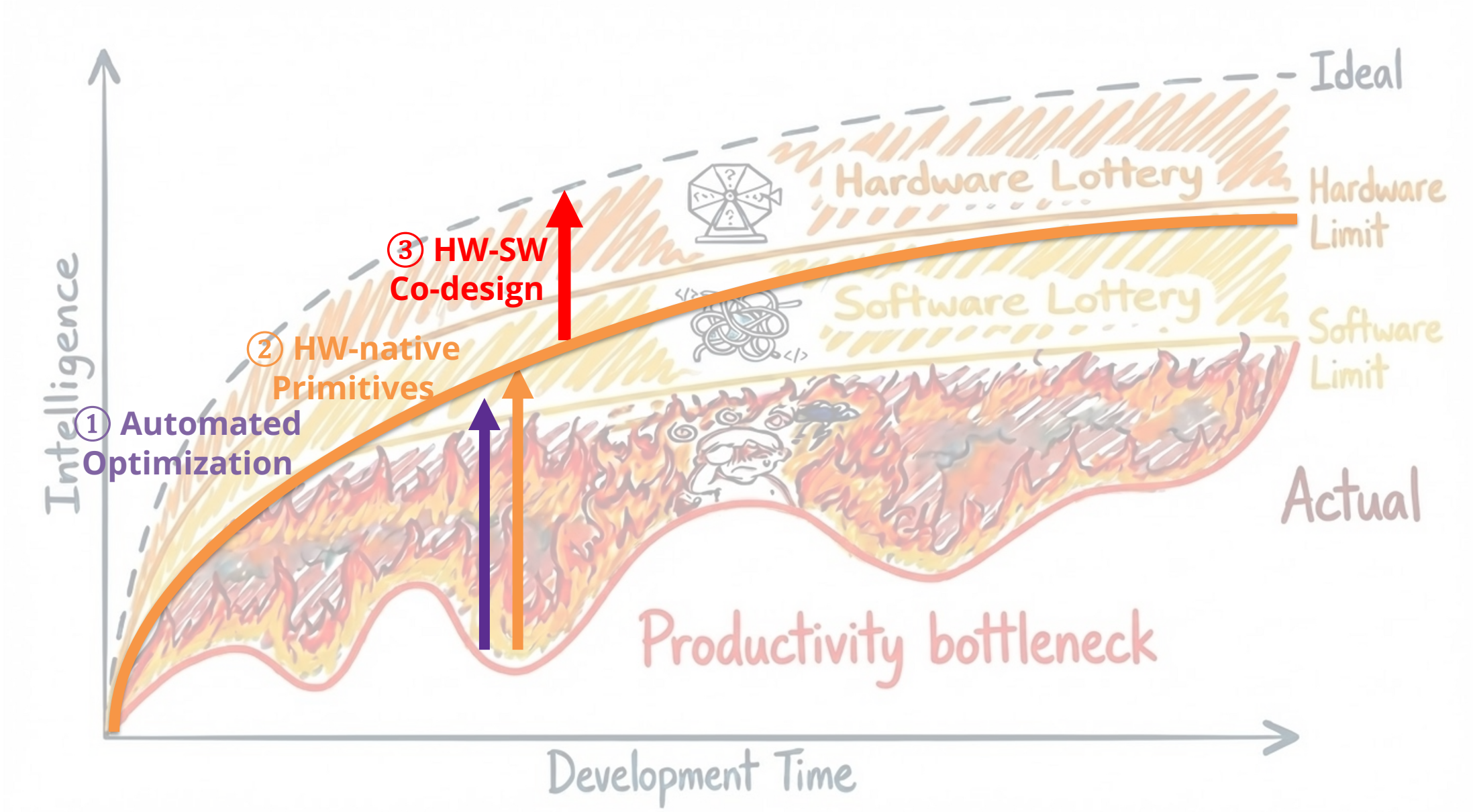
Conclusions

- Proposed a general asynchronous dataflow communication structure **aref** for warp specialization
  and implemented end-to-end in the Triton compiler

- Development branch: https://github.com/triton-lang/triton/tree/aref_auto_ws
  - Most features have been merged into main (enable_warp_specialization=True)
  - NVWS dialect: https://github.com/triton-lang/triton/pull/6288
  - End-to-end aref: https://github.com/triton-lang/triton/pull/8262

- Blackwell optimization
  - Actively developing, search aref in upstream Triton
  - Check out Chris Sullivan's talk "**A Performance Engineer's Guide to NVIDIA Blackwell GPUs in Triton**" @ Triton Conference'25

From Software Programming to Hardware Generation



Allo: A Programming Model for Composable Accelerator Design

Hongzheng Chen*, Niansong Zhang*, Shaojie Xiang,
Zhichen Zeng, Mengjia Dai, Zhiru Zhang

PLDI'24

* Equal contribution

Challenge: Balancing Manual Control & Compiler Optimization

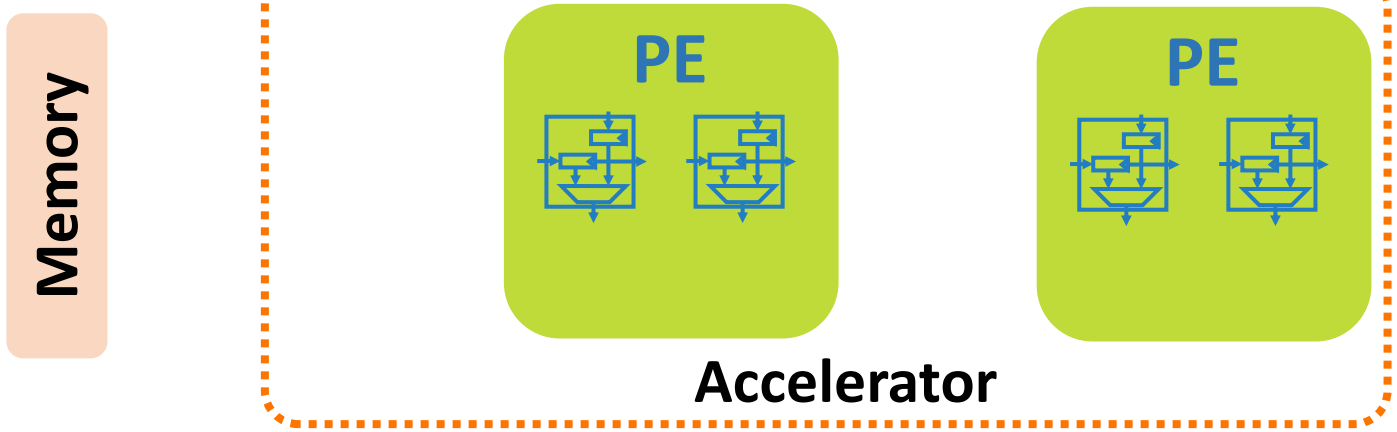
```
void systolic_tile(int8_t A_tile[2][768],
 int8_t B_tile[768][2],
 int8_t C_tile[2][2]) {
 #pragma dataflow
 hls::stream<int8_t> A_fifo[2][3], B_fifo[2][3];
 #pragma stream variable=A/B_fifo depth=3
 #pragma partition variable=A/B/C_tile complete dim=1
 for (int k4 = 0; k4 < 768; k4++) {
  for (int m = 0; m < 2; m++) {
   int8_t v105 = A_tile[m][k4];
   A_fifo[m][0].write(v105);}
   // ... write B_fifo
  }}
  for (int Ti = 0; Ti < 2; ++Ti) {
  #pragma HLS unroll
   for (int Tj = 0; Tj < 2; ++Tj) {
   #pragma HLS unroll
    // ... load A/B_fifo
    PE_kernel(A_in, A_out, B_in, B_out, C, Ti, Tj);
}}}

void matmul(int8_t A[512][768], int8_t B[768][768],
 int8_t C[512][768]
) {
 int8_t local_A[2][768], local_B[768][2], local_C[2][2];
 for (int mi = 0; mi < 256; mi++) {
  for (int ni = 0; ni < 384; ni++) {
   // ... load A, B
   systolic_tile(local_A, local_B, local_C);
}}}
```

Compute Customization

Vanilla Matmul (1% theoretical peak perf.)

+ Compute customization (~30% peak)
  + Loop tiling
  + Loop unrolling
  + Loop & function pipelining

Increasing programming effort

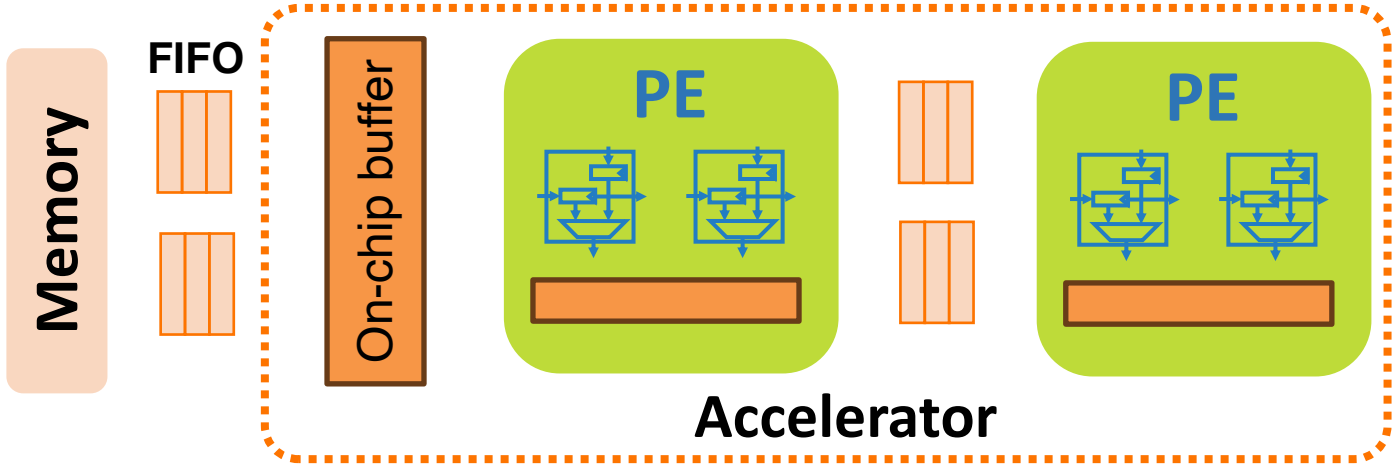


Challenge: Balancing Manual Control & Compiler Optimization

```
void systolic_tile(int8_t A_tile[2][768],
 int8_t B_tile[768][2],
 int8_t C_tile[2][2]) {
 #pragma dataflow
 hls::stream<int8_t> A_fifo[2][3], B_fifo[2][3];
 #pragma stream variable=A/B_fifo depth=3
 #pragma partition variable=A/B/C_tile complete dim=1
 for (int k4 = 0; k4 < 768; k4++) {
  for (int m = 0; m < 2; m++) {
   int8_t v105 = A_tile[m][k4];
   A_fifo[m][0].write(v105);}
   // ... write B_fifo
  }}
 for (int Ti = 0; Ti < 2; ++Ti) {
 #pragma HLS unroll
  for (int Tj = 0; Tj < 2; ++Tj) {
  #pragma HLS unroll
   // ... load A/B_fifo
   PE_kernel(A_in, A_out, B_in, B_out, C, Ti, Tj);
}}}

void matmul(int8_t A[512][768], int8_t B[768][768],
 int8_t C[512][768]
) {
 int8_t local_A[2][768], local_B[768][2], local_C[2][2];
 for (int mi = 0; mi < 256; mi++) {
  for (int ni = 0; ni < 384; ni++) {
   // ... load A, B
   systolic_tile(local_A, local_B, local_C);
}}}
```

Memory Customization

Compute Customization



Vanilla Matmul (1% theoretical peak perf.)

+ Compute customization (~30% peak)
  + Loop tiling
  + Loop unrolling
  + Loop & function pipelining

+ Custom memory hierarchy (~50% peak)
  + Tiling & data reuse buffers
  + Memory banking/partitioning

Increasing programming effort

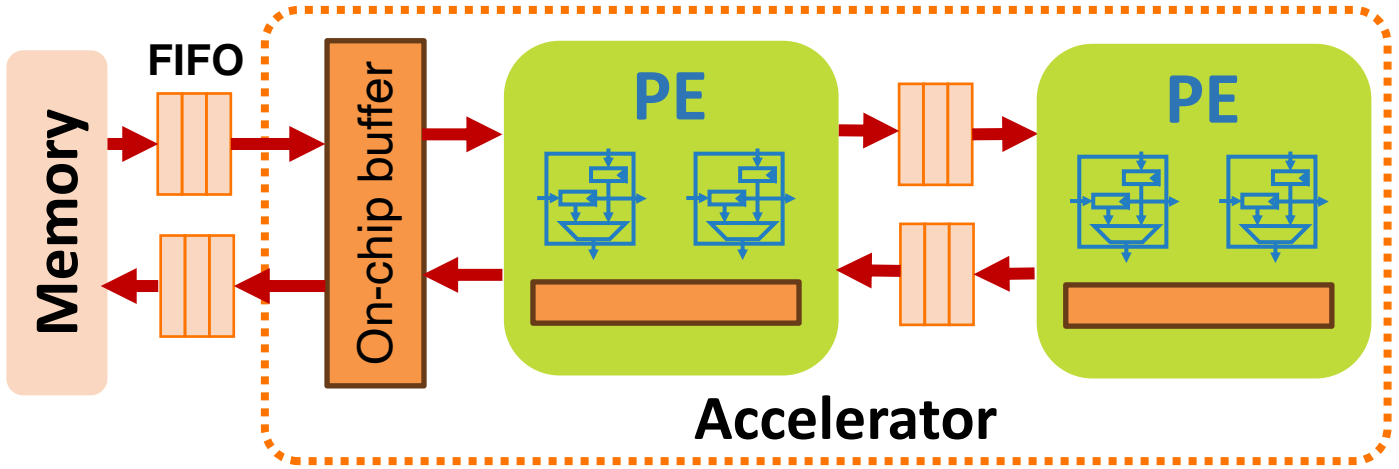
Challenge: Balancing Manual Control & Compiler Optimization

```
void systolic_tile(int8_t A_tile[2][768],
 int8_t B_tile[768][2],
 int8_t C_tile[2][2]) {
 #pragma dataflow
 hls::stream<int8_t> A_fifo[2][3], B_fifo[2][3];
 #pragma stream variable=A/B_fifo depth=3
 #pragma partition variable=A/B/C_tile complete dim=1
 for (int k4 = 0; k4 < 768; k4++) {
  for (int m = 0; m < 2; m++) {
   int8_t v105 = A_tile[m][k4];
   A_fifo[m][0].write(v105);}
   // ... write B_fifo
  }}
 for (int Ti = 0; Ti < 2; ++Ti) {
 #pragma HLS unroll
  for (int Tj = 0; Tj < 2; ++Tj) {
  #pragma HLS unroll
   // ... load A/B_fifo
   PE_kernel(A_in, A_out, B_in, B_out, C, Ti, Tj);
}}}

void matmul(int8_t A[512][768], int8_t B[768][768],
 int8_t C[512][768]
) {
 int8_t local_A[2][768], local_B[768][2], local_C[2][2];
 for (int mi = 0; mi < 256; mi++) {
  for (int ni = 0; ni < 384; ni++) {
   // ... load A, B
   systolic_tile(local_A, local_B, local_C);
}}}
```

Communication Customization

Memory Customization

Compute Customization



Vanilla Matmul (1% theoretical peak perf.)

+ Compute customization (~30% peak)
  + Loop tiling
  + Loop unrolling
  + Loop & function pipelining

+ Custom memory hierarchy (~50% peak)
  + Tiling & data reuse buffers
  + Memory banking/partitioning

+ Data movement optimization (~95% peak)
  + Data streaming
  + Data packing (vectorization)
  + Memory coalescing
  + Systolic communication

Increasing programming effort

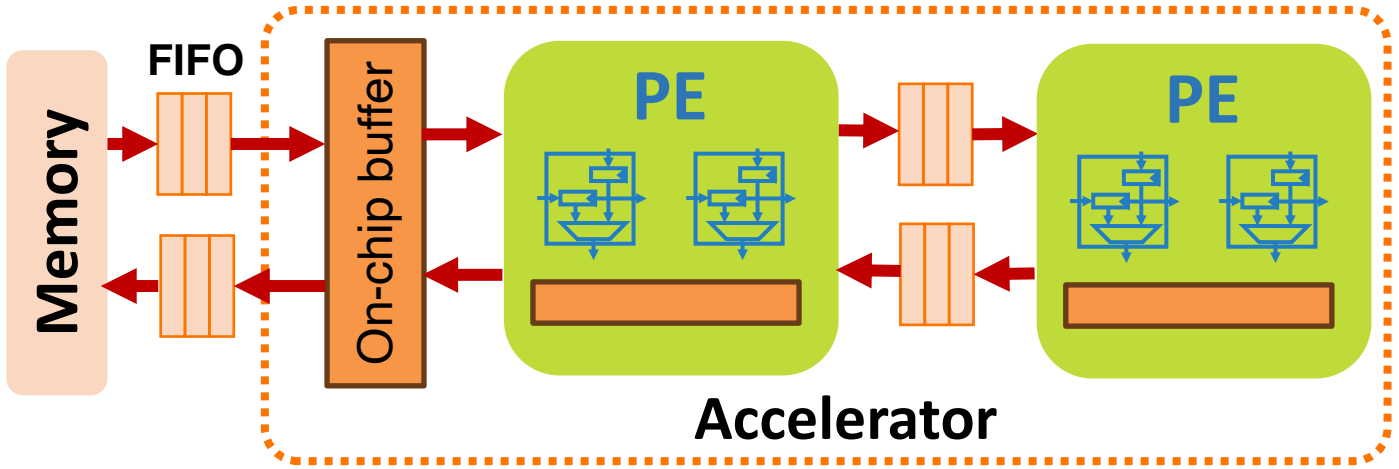
Challenge: Balancing Manual Control & Compiler Optimization

```
void systolic_tile(int8_t A_tile[2][768],
 int8_t B_tile[768][2],
 int8_t C_tile[2][2]) {
 #pragma dataflow
 hls::stream<int8_t> A_fifo[2][3], B_fifo[2][3];
 #pragma stream variable=A/B_fifo depth=3
 #pragma partition variable=A/B/C_tile complete dim=1
 for (int k4 = 0; k4 < 768; k4++) {
  for (int m = 0; m < 2; m++) {
   int8_t v105 = A_tile[m][k4];
   A_fifo[m][0].write(v105);}
   // ... write B_fifo
  }}
 for (int Ti = 0; Ti < 2; ++Ti) {
 #pragma HLS unroll
  for (int Tj = 0; Tj < 2; ++Tj) {
  #pragma HLS unroll
   // ... load A/B_fifo
   PE_kernel(A_in, A_out, B_in, B_out, C, Ti, Tj);
}}}

void matmul(int8_t A[512][768], int8_t B[768][768],
 int8_t C[512][768]
) {
 int8_t local_A[2][768], local_B[768][2], local_C[2][2];
 for (int mi = 0; mi < 256; mi++) {
  for (int ni = 0; ni < 384; ni++) {
   // ... load A, B
   systolic_tile(local_A, local_B, local_C);
}}}
```

Communication Customization

Memory Customization

Compute Customization



Vanilla Matmul (1% theoretical peak perf.)

+ Compute customization (~30% peak)
  + Loop tiling — Existing HLS compiler
  + Loop unrolling — e.g., ScaleHLS [HPCA'22]
  + Loop & function pipelining

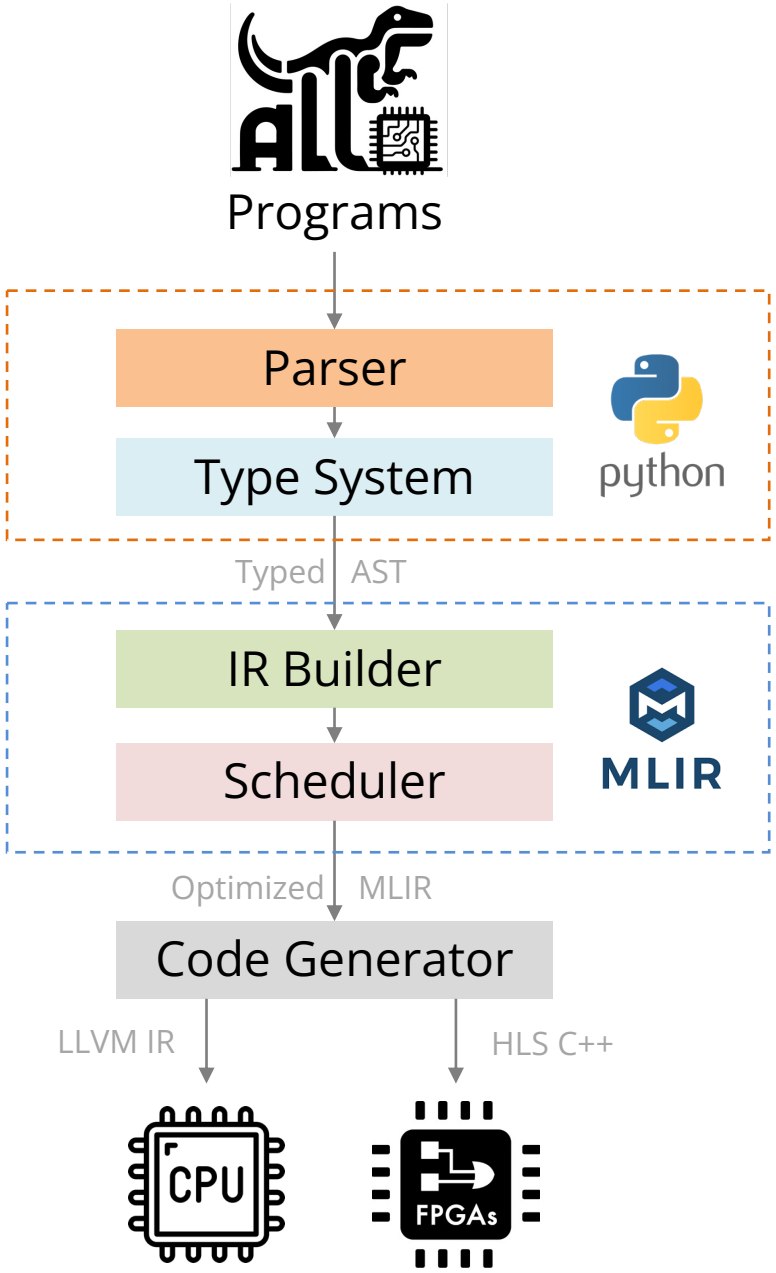
+ Custom memory hierarchy (~50% peak)
  + Tiling & data reuse buffers
  + Memory banking/partitioning

+ Data movement optimization (~95% peak)
  + Data streaming
  + Data packing (vectorization)
  + Memory coalescing
  + Systolic communication

Increasing programming effort

~500 lines of HLS code for a small systolic array
vendor-specific, hard to maintain & reuse

Allo Accelerator Design Language (ADL) and Compiler



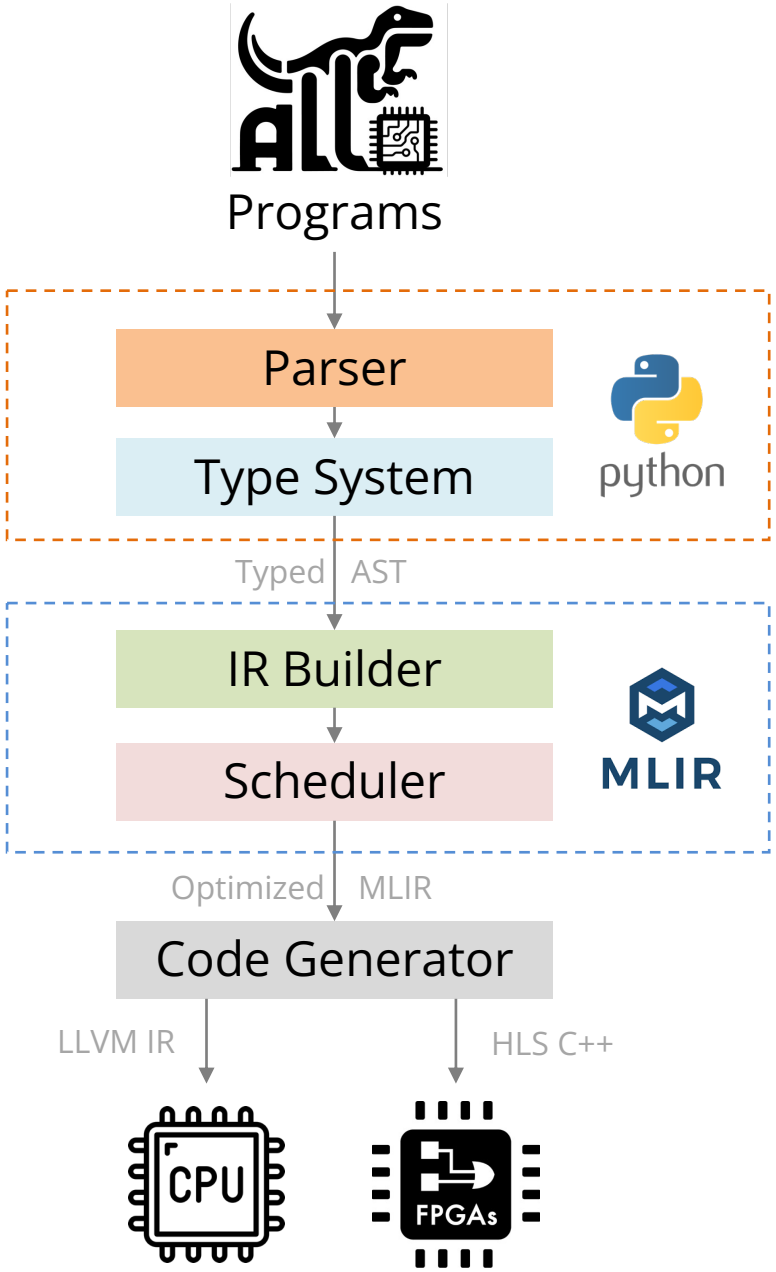
Pythonic: No need to learn a new DSL!

Algorithm Specification

```
def matmul(A: float32[M, K],
           B: float32[K, N])
           -> float32[M, N]:
  C: float32[M, N] = 0.0
  for i, j, k in allo.grid(M, N, K):
    C[i, j] += A[i, k] * B[k, j]
  return C
```

- Free-form imperative programming
- Python native keywords (e.g., for, if, else)
- Explicit type annotation

Allo Accelerator Design Language (ADL) and Compiler



Pythonic: No need to learn a new DSL!

Algorithm Specification

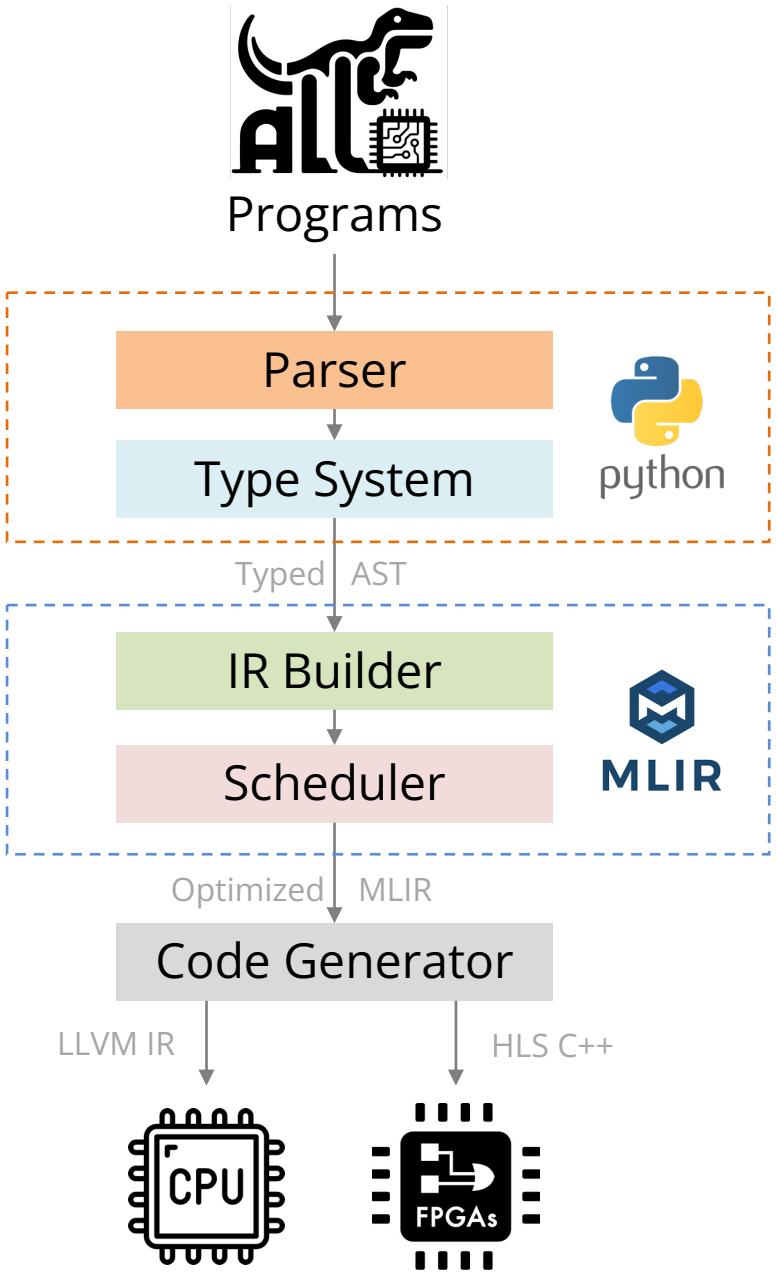
```
def matmul(A: float32[M, K],
           B: float32[K, N])
           -> float32[M, N]:
  C: float32[M, N] = 0.0
  for i, j, k in allo.grid(M, N, K):
    C[i, j] += A[i, k] * B[k, j]
  return C
```

Decoupled customization

Compute Cust.
Memory Cust.
Comm. Cust.

```
s = allo.customize(matmul)
s.reorder("k", "j")
s.buffer_at(s.C, axis="i")
s.pipeline("j")
```

Allo Accelerator Design Language (ADL) and Compiler



Stepwise verifiable rewrites

```
s = allo.customize(gemm)
s.reorder("k", "j")
print(s.module)

s.buffer_at(s.C, axis="i")
print(s.module)

s.pipeline("j")
print(s.module)
```

MLIR

```
module {
func.func @gemm(%arg0, %arg1) -> memref<1024x1024xf32> {
 %alloc = memref.alloc() {name = "C"} : memref<1024x1024xf32>
 affine.for %arg2 = 0 to 1024 {
  affine.for %arg3 = 0 to 1024 {
   affine.for %arg4 = 0 to 1024 {
    ...
   } {loop_name = "j", pipeline}
  } {loop_name = "k", op_name = "S_k_0", reduction}
  ...
 } {loop_name = "i", op_name = "S_i_j_0"}
 return %alloc : memref<1024x1024xf32>
}}
```

```
module {
func.func @gemm(%arg0: memref<1024x1024xf32>, %arg1:
memref<1024x1024xf32>) -> memref<1024x1024xf32> {
 %alloc = memref.alloc() {name = "C"} : memref<1024x1024xf32>
 affine.for %arg2 = 0 to 1024 {
  %alloc_0 = memref.alloc() : memref<1024xf32>
  affine.for %arg3 = 0 to 1024 {
   ...
  } {buffer, loop_name = "j_init", pipeline_ii = 1 : i32}
  affine.for %arg3 = 0 to 1024 {
   affine.for %arg4 = 0 to 1024 {
    ...
   } {loop_name = "j"}
  } {loop_name = "k", op_name = "S_k_0", reduction}
  affine.for %arg3 = 0 to 1024 {
   ...
  } {buffer, loop_name = "j_back", pipeline_ii = 1 : i32}
 } {loop_name = "i", op_name = "S_i_j_0"}
 return %alloc : memref<1024x1024xf32>
}}
```

```
module {
func.func @gemm(%arg0: memref<1024x1024xf32>, %arg1:
memref<1024x1024xf32>) -> memref<1024x1024xf32> {
 %alloc = memref.alloc() {name = "C"} : memref<1024x1024xf32>
 affine.for %arg2 = 0 to 1024 {
  affine.for %arg3 = 0 to 1024 {
   affine.for %arg4 = 0 to 1024 {
    ...
   } {loop_name = "k"}
  } {loop_name = "j"}
 } {loop_name = "i", op_name = "S_i_j_0"}
 return %alloc : memref<1024x1024xf32>
}}
```

Transforming GEMM to Systolic Array

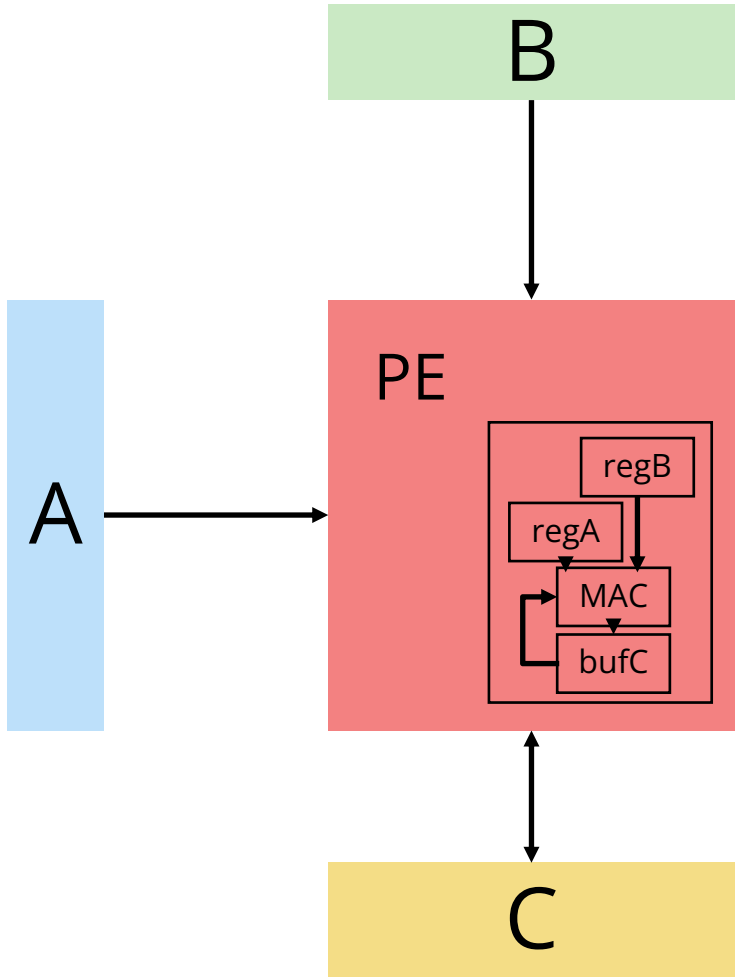
➔ Algorithm specification

```
def matmul(A: int8[M, K],
           B: int8[K, N],
           C: int16[M, N]):
  for i, j in allo.grid(M, N, "PE"):
    for k in range(K):
      C[i, j] += A[i, k] * B[k, j]
```

➔ Schedule construction

```
s = allo.customize(matmul)
```

➔ Architectural Diagram



* Schedule: A sequence of customization primitives

Transforming GEMM to Systolic Array

➔ Algorithm specification

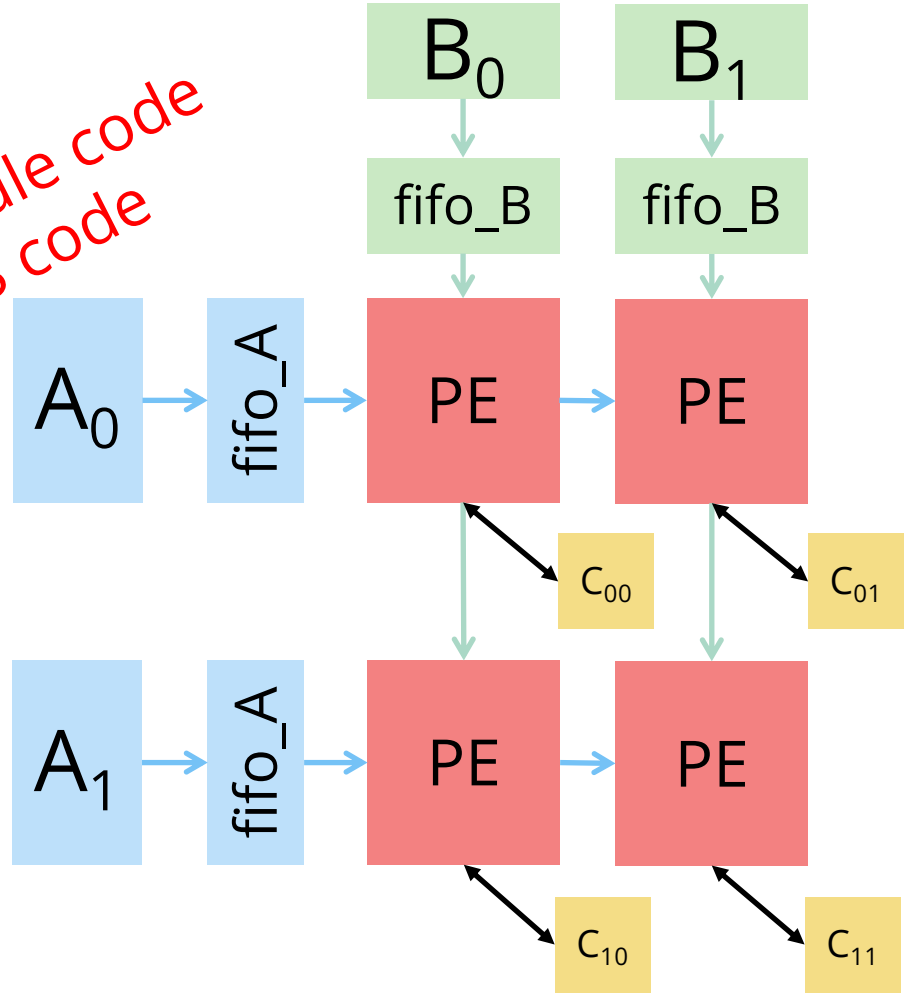
```
def matmul(A: int8[M, K],
           B: int8[K, N],
           C: int16[M, N]):
  for i, j in allo.grid(M, N, "PE"):
    for k in range(K):
      C[i, j] += A[i, k] * B[k, j]
```

➔ Schedule construction

```
s = allo.customize(matmul)
buf_A = s.buffer_at(s.A, "j")
buf_B = s.buffer_at(s.B, "j")
pe = s.unfold("PE", axis=[0, 1],
                    factor=[M, N])
s.partition(s.A, dim=0)
s.partition(s.B, dim=1)
s.partition(s.C, dim=[0, 1])
s.to(buf_A, pe, axis=0, depth=M + 1)
s.to(buf_B, pe, axis=1, depth=N + 1)
```

Only 8 lines of schedule code vs ~500 lines HLS code

➔ Architectural Diagram



* Schedule: A sequence of customization primitives

Transforming GEMM to Systolic Array

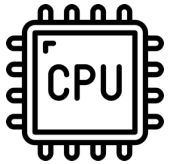
➔ Algorithm specification


```
def matmul(A: int8[M, K],
           B: int8[K, N],
           C: int16[M, N]):
  for i, j in allo.grid(M, N, "PE"):
    for k in range(K):
      C[i, j] += A[i, k] * B[k, j]
```

➔ Schedule construction

```
s = allo.customize(matmul)
buf_A = s.buffer_at(s.A, "j")
buf_B = s.buffer_at(s.B, "j")
pe = s.unfold("PE", axis=[0, 1],
                    factor=[M, N])
s.partition(s.A, dim=0)
s.partition(s.B, dim=1)
s.partition(s.C, dim=[0, 1])
s.to(buf_A, pe, axis=0, depth=M + 1)
s.to(buf_B, pe, axis=1, depth=N + 1)
```

➔ CPU Simulation




```
mod = s.build(target="llvm")

A = np.random.randint(-8, 8, size=(M, K)) \
      .astype(np.int8)
B = np.random.randint(-8, 8, size=(K, N)) \
      .astype(np.int8)
C = np.zeros((M, N), dtype=np.int16)
mod(A, B, C)
np.testing.assert_allclose(C, A @ B, atol=1e-3)
```

Transforming GEMM to Systolic Array

➔ Algorithm specification

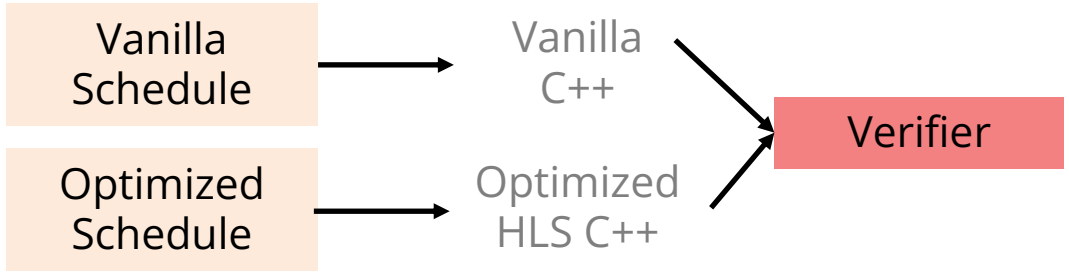
```
def matmul(A: int8[M, K],
           B: int8[K, N],
           C: int16[M, N]):
  for i, j in allo.grid(M, N, "PE"):
    for k in range(K):
      C[i, j] += A[i, k] * B[k, j]
```

➔ Schedule construction

```
s = allo.customize(matmul)
buf_A = s.buffer_at(s.A, "j")
buf_B = s.buffer_at(s.B, "j")
pe = s.unfold("PE", axis=[0, 1],
                    factor=[M, N])
s.partition(s.A, dim=0)
s.partition(s.B, dim=1)
s.partition(s.C, dim=[0, 1])
s.to(buf_A, pe, axis=0, depth=M + 1)
s.to(buf_B, pe, axis=1, depth=N + 1)
```

➔ Formal Verification [FPGA'24 Best Paper*]

```
mod = s.verify()
```



- A formal equivalence checker of source-to-source HLS transformations via symbolic execution
  - Support statically interpretable control-flow (SICF)
- Verification in time/space linear w.r.t. OPs executed
  - ~500K Ops/sec in verification throughput
  - A complex 64x64 systolic array verified in 16 minutes

* L.-N. Pouchet et al., "Formal Verification of Source-to-Source Transformation for HLS", FPGA 2024 (Best Paper Award)

Transforming GEMM to Systolic Array

➔ Algorithm specification

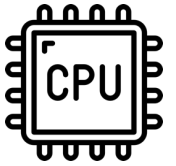

```
def matmul(A: int8[M, K],
           B: int8[K, N],
           C: int16[M, N]):
  for i, j in allo.grid(M, N, "PE"):
    for k in range(K):
      C[i, j] += A[i, k] * B[k, j]
```

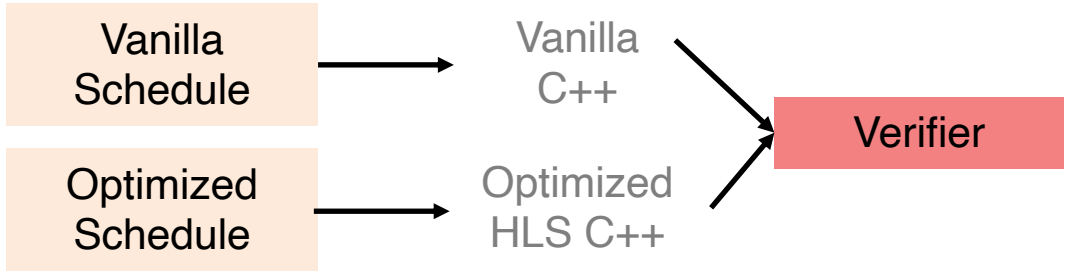
➔ Schedule construction

```
s = allo.customize(matmul)
buf_A = s.buffer_at(s.A, "j")
buf_B = s.buffer_at(s.B, "j")
pe = s.unfold("PE", axis=[0, 1],
                    factor=[M, N])
s.partition(s.A, dim=0)
s.partition(s.B, dim=1)
s.partition(s.C, dim=[0, 1])
s.to(buf_A, pe, axis=0, depth=M + 1)
s.to(buf_B, pe, axis=1, depth=N + 1)
```

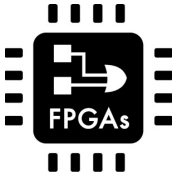
➔ Formal Verification [FPGA'24 Best Paper*]

```
mod = s.verify()
```





➔ Bitstream generation

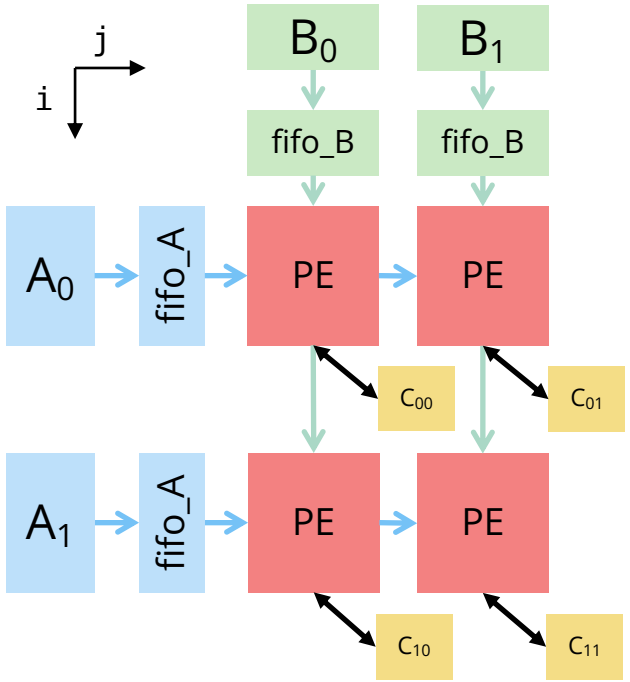



```
mod = s.build(target="vitis_hls",
              mode="hw",
              project="systolic.prj")

# Automatically invoke the HLS toolchain
mod(A, B, C)
```

Transforming GEMM to Systolic Array

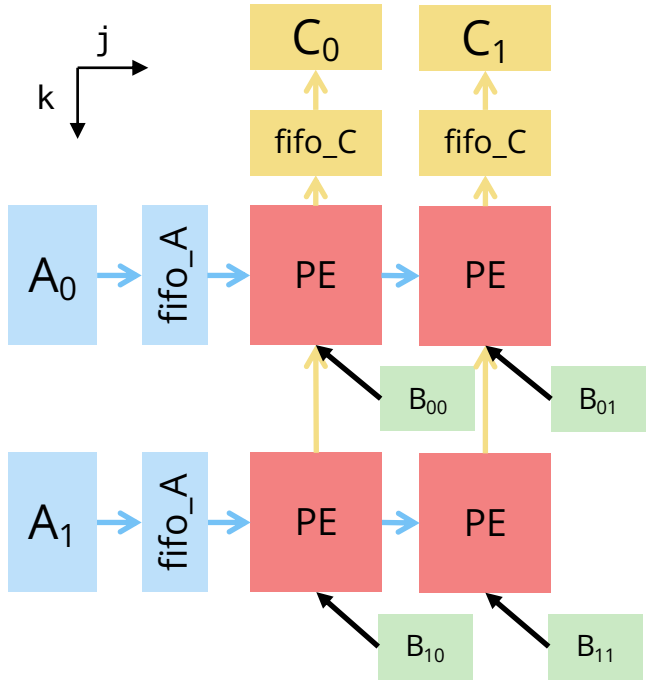
➔ Spatial loop: i, j

```
s = allo.customize(matmul)
buf_A = s.buffer_at(s.A, "j")
buf_B = s.buffer_at(s.B, "j")
pe = s.unfold("PE", axis=[0, 1],
                    factor=[M, N])
s.partition(s.A, dim=0)
s.partition(s.B, dim=1)
s.partition(s.C, dim=[0, 1])
s.relay(buf_A, pe, axis=0, depth=M + 1)
s.relay(buf_B, pe, axis=1, depth=N + 1)
```



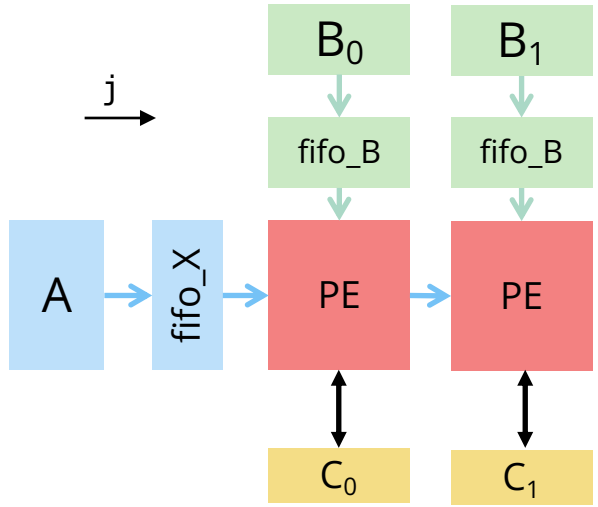
➔ Spatial loop: k, j

```
s = allo.customize(matmul)
s.reorder("k", "j", "i")
buf_A = s.buffer_at(s.A, "j")
buf_C = s.buffer_at(s.C, "j")
pe = s.unfold("PE", axis=[0, 1],
                    factor=[K, N])
s.partition(s.A, dim=0)
s.partition(s.B, dim=[0, 1])
s.partition(s.C, dim=1)
s.relay(buf_A, pe, axis=0)
s.relay(buf_B, pe, axis=1)
```



➔ Spatial loop: j

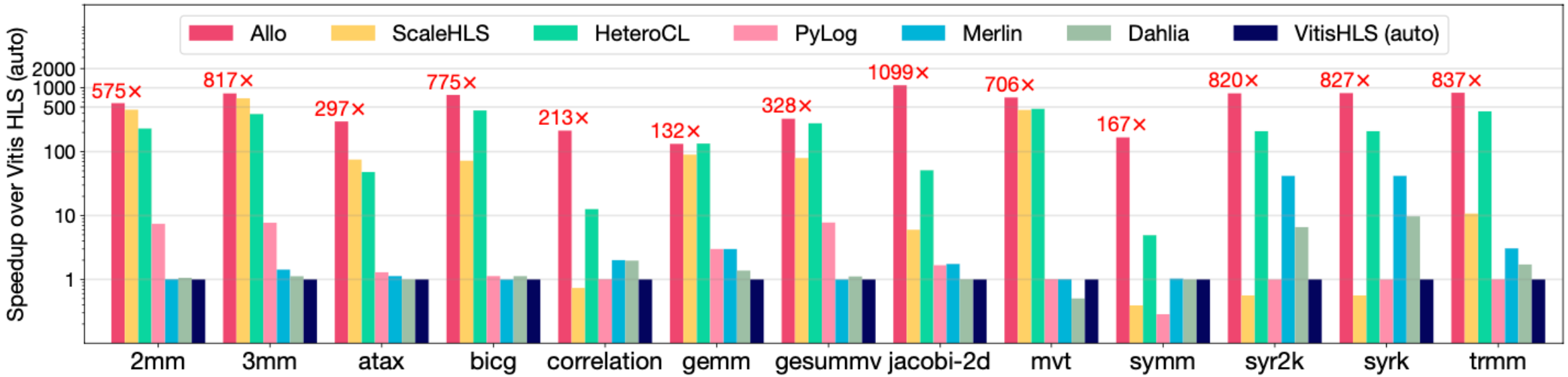
```
s = allo.customize(matmul)
buf_A = s.buffer_at(s.A, "j")
buf_B = s.buffer_at(s.B, "j")
pe = s.unfold("PE", axis=1,
                    factor=M)
s.partition(s.B, dim=1)
s.partition(s.C, dim=1)
s.relay(buf_A, pe, axis=0)
s.relay(buf_B, pe, axis=1)
```



Realize different dataflows with minimal schedule code

Single-Kernel Evaluation

- Benchmarks from PolyBench; Target hardware: AMD U280 FPGA
- Normalized against AMD VitisHLS-auto (pragmas automatically inserted)
- Other baselines
  - ADLs: HeteroCL [FPGA'19], Dahlia [PLDI'20], PyLog [TC'21]
  - Automated DSE for HLS: ScaleHLS [HPCA'22], Merlin [TRETS'22]



Allo achieves (much) higher performance by optimizing data placement with a custom memory hierarchy

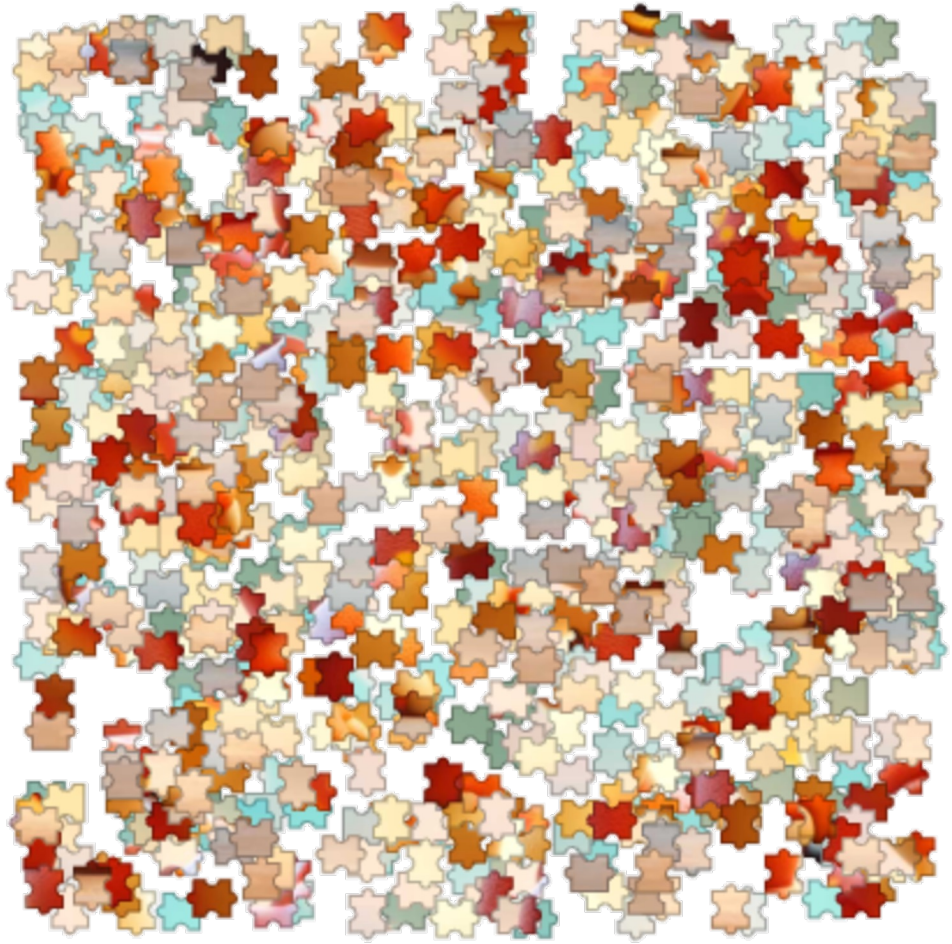
Single-Kernel Evaluation

- Compared to ScaleHLS [HPCA'22], Allo achieves
  - Lower latency with much more effective use of compute resources
  - Higher post place-and-route frequency due to better pipelining

Benchmark	Allo						ScaleHLS				
	Latency (cycles)	II	DSP Usage	PnR Freq. (MHz)	Lines of Allo Custm.	Compile Time (s)	Latency (cycles)	II	DSP Usage	PnR Freq. (MHz)	Compile Time (s)
atax	4.9K (↓ 3.9×)	1	403 (↑ 2.9×)	411	9	1.0	19.4K	4	141	329	36.1
correlation	498.7K (↓ 290.5×)	1	4168 (↑ 38.2×)	362	19	0.8	144.9M	667	109	305	638.8
jacobi-2d	58.8K (↓ 183.1×)	1	3968 (↑ 72.1×)	411	17	0.9	10.8M	28	55	308	47.9
symm	405.7K (↓ 427.4×)	1	1208 (↑ 201.3×)	402	15	1.0	182.4M	13	6	397	3.5
trmm	492.6K (↓ 78.0×)	1	101 (↑ 14.4×)	414	12	0.8	38.4M	4	7	382	1.4

Allo achieves (much) higher performance by optimizing data placement with a custom memory hierarchy

How to Compose Optimized Kernels into Complete Accelerator?



Optimized Kernels

High-performance Accelerator

Behavioral Composition

➔ Given optimized kernel implementations

```
def matmul(A: int8[M, K],
           B: int8[K, N],
           C: int16[M, N])
```

Mismatched interface

```
s2.partition(s.A, dim=1)
```

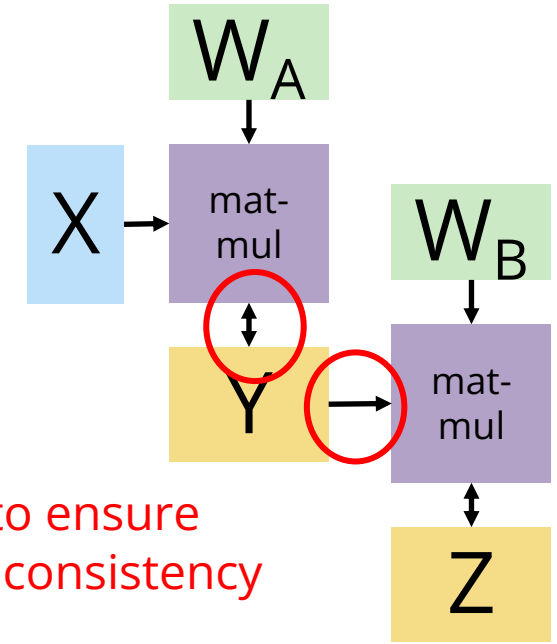

```
s1.partition(s.C, dim=0)
```

Matched interface!


```
s2.partition(s.A, dim=0)
```

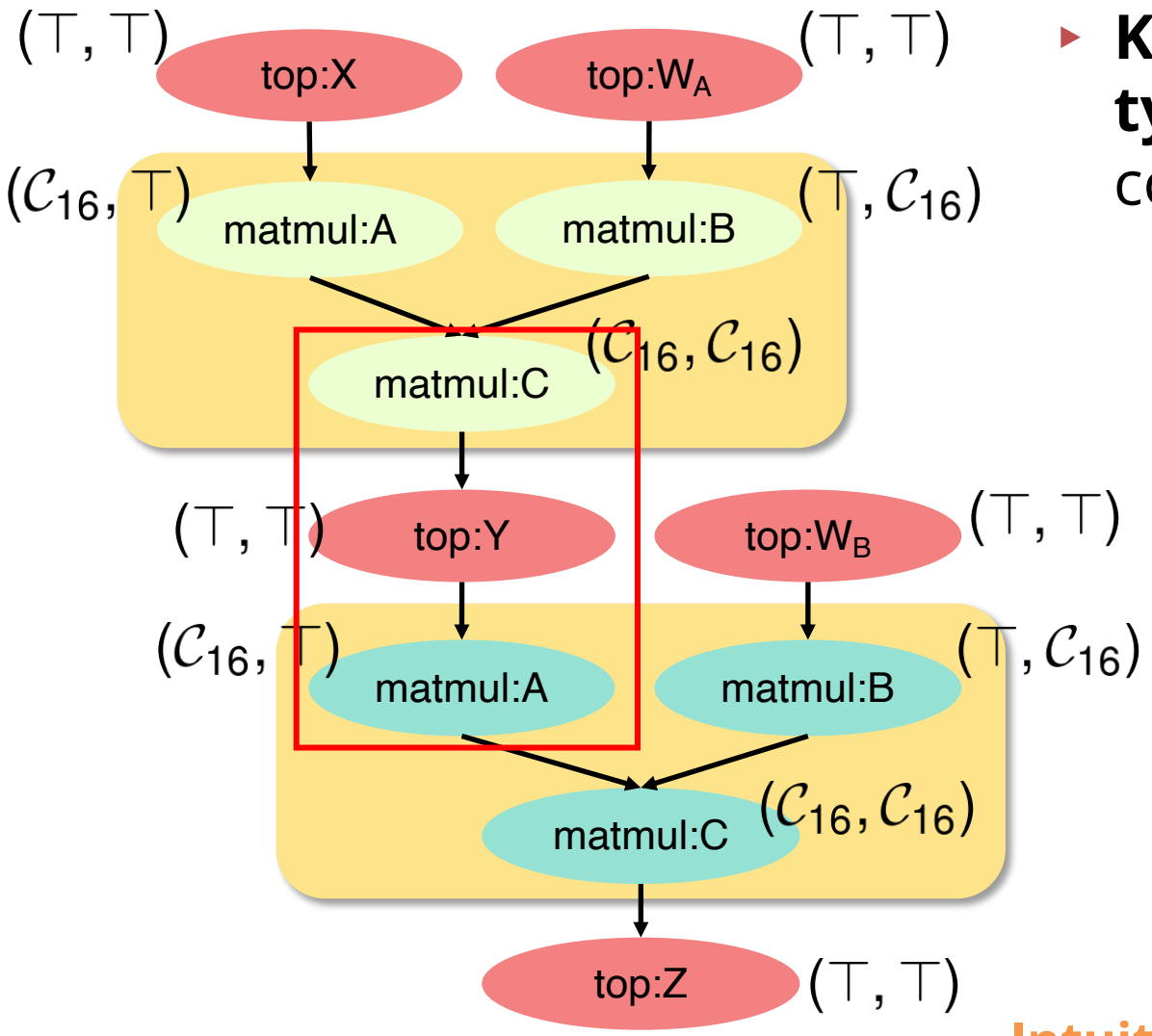
➔ Algorithm specification (Hierarchical)

```
def top(X: int8[M, K], W_A: int8[K, N],
        W_B: int8[N, K], Y: int8[M, K]):
  Y: int8[M, N] = 0
  Z: int8[M, K] = 0
  matmul(X, W_A, Y)
  matmul(Y, W_B, Z)
  return Z
```



Need to ensure interface consistency

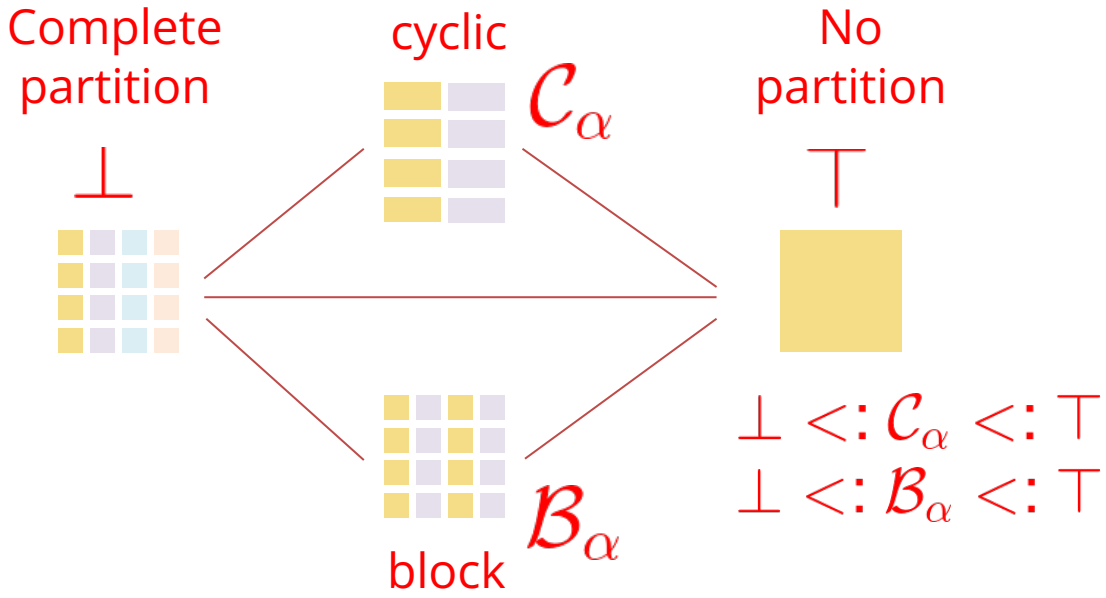
Behavioral Composition



Hierarchical use-def graph

- **Key idea: Model memory banking as a type** to ensure kernel interfaces are consistent

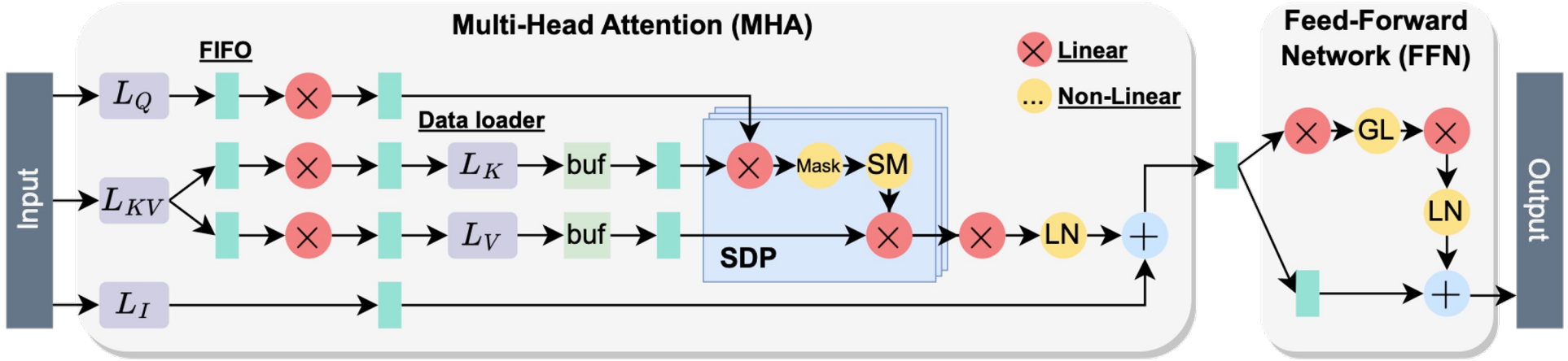
Subtyping relation forms a **lattice**!



Intuition:
We can supply more read/write parallelism, but not less!

Multi-Kernel Evaluation: A Complete LLM Accelerator

- GPT2 model (the only open-source LLM in the GPT family)
  - 355M parameters, 24 hidden layers, 16 heads
  - W4A8 quantization




```
def GPT_layer(inp: float32[L, D], ...)
                -> float32[L, D]:
  # 1. Multi-Head Attention (MHA)
  Q = Linear_layer_qkv(inp, Wq, Bq)
  K = Linear_layer_qkv(inp, Wk, Bk)
  V = Linear_layer_qkv(inp, Wv, Bv)
  attn_sf_outp = Self_attention(Q, K, V)
  # ...
  # 2. Feed Forward Network (FFN)
  # ...
  return ffn_res_outp
```

Compose all the schedules together

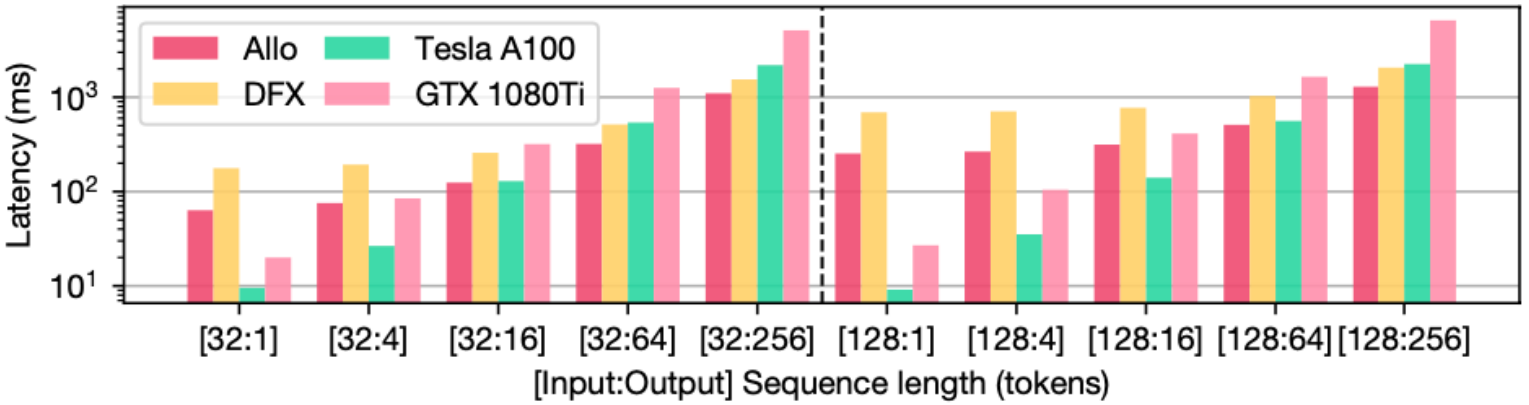
```
s = allo.customize(GPT_layer)
s.compose([s_qkv, ..., s_gelu])
```

LLM Accelerator Evaluation

- GPT2: single-batch, low-latency, generative inference settings
  - Full design running on an AMD Alveo U280 FPGA (16nm) at 250MHz
  - 2.2x speedup in prefill stage compared to DFX [MICRO'22] (an FPGA-based overlay)
  - **In decode stage, 1.9x speedup for long output sequences and 5.7x more energy-efficient vs. NVIDIA A100 GPU**
  - Fewer than 50 lines of schedule code in Allo

A100 GPU *7nm*
Max M: 16415 (temporal)
Bandwidth: **1935 GB/s**

Alveo U280 (ours) *16nm*
Max M: 1289
Bandwidth: **498 GB/s**

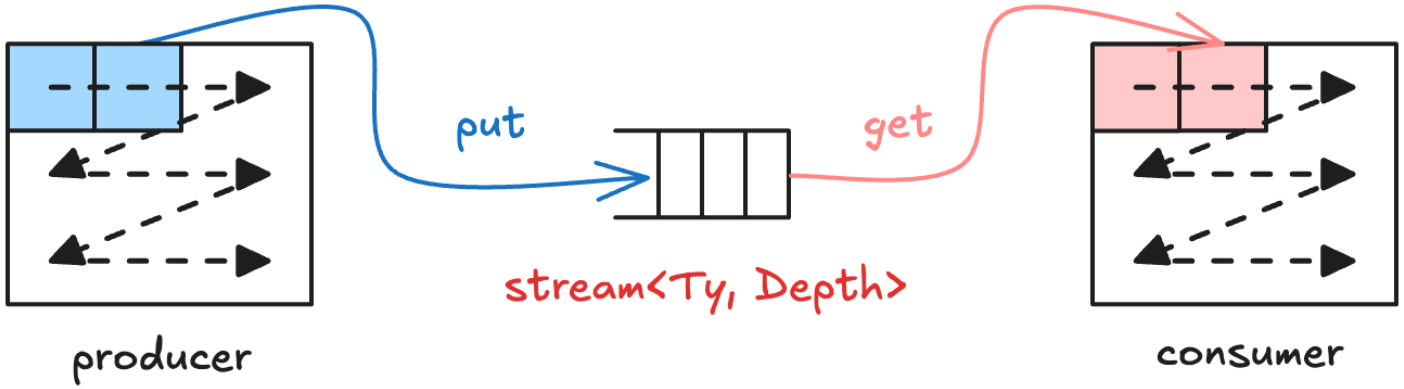


	Allo	DFX
Device	U280	U280
Freq.	250MHz	200MHz
Quant.	W4A8	fp16
BRAM	384 (19.0%)	1192 (59.1%)
DSP	1780 (19.73%)	3533 (39.2%)
FF	652K (25.0%)	1107K (42.5%)
LUT	508K (39.0%)	520K (39.9%)

H. Chen et al., "Understanding the Potential of FPGA-Based Spatial Acceleration for Large Language Model Inference", ACM TRETS (FCCM Journal Track), 2024.

Structural Composition: Producer-Consumer

- Data communication (Stream) as a **first-class citizen**
  - Enable type checking on read/write consistency

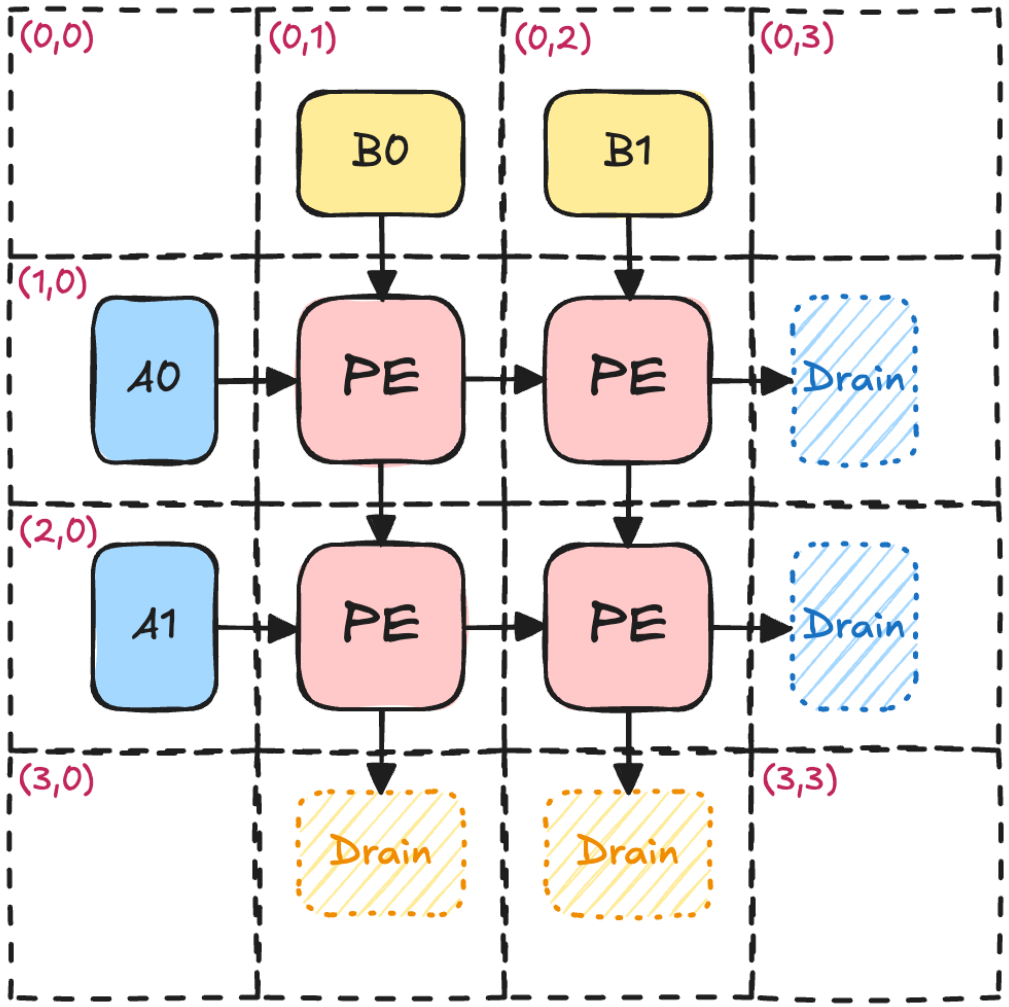



```
Ty = float32
M, N, K = 16, 16, 16

@df.unit()
def top():
 pipe: Stream[Ty, 2]

 @df.work(mapping=[1])
 def producer(A: Ty[M, N]):
  for i, j in allo.grid(M, N):
   # send data
   pipe.put(A[i, j])

 @df.work(mapping=[1])
 def consumer(B: Ty[M, N]):
  for i, j in allo.grid(M, N):
   # receive data
   B[i, j] = pipe.get() + 1
```

S. Fang*, **H. Chen*** et al., "Dato: A Task-Based Programming Model for Dataflow Accelerators", arXiv, 2025.

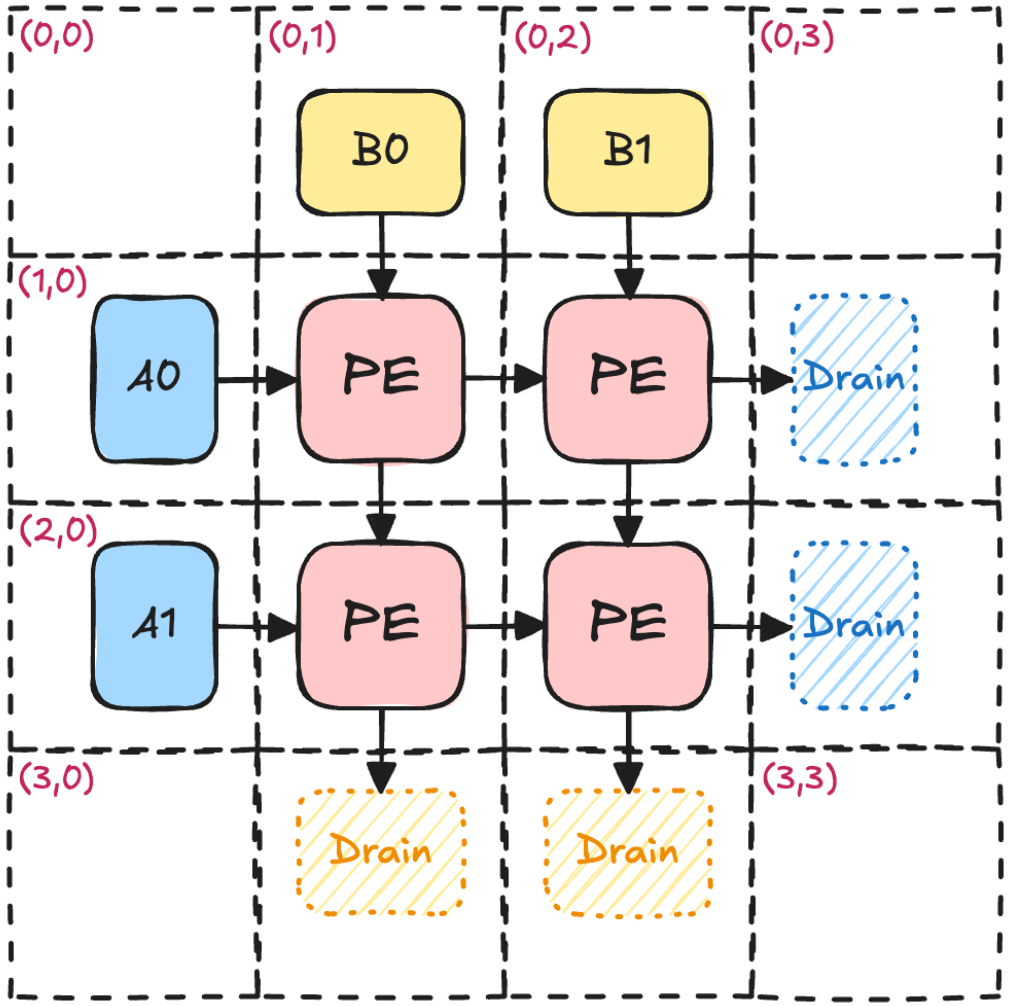
Structural Composition: Systolic Array

```
@df.region()
def top():
  fifo_A: Stream[int8, 4][P0, P1]
  fifo_B: Stream[int8, 4][P0, P1]

  @df.kernel(mapping=[P0, P1])
  def gemm(A: int8[M, K],
           B: int8[K, N],
           C: int16[M, N]):
    i, j = df.get_pid()
    # peripheral kernels
    with allo.meta_if(i in {0, M + 1}
                      and j in {0, N + 1}):
      pass
    with allo.meta_elif(j == 0):
      # i > 0
      for k in range(K):
        fifo_A[i, j + 1].put(A[i - 1, k])
    with allo.meta_elif(i == 0):
      # j > 0
      for k in range(K):
        fifo_B[i + 1, j].put(B[k, j - 1])
```



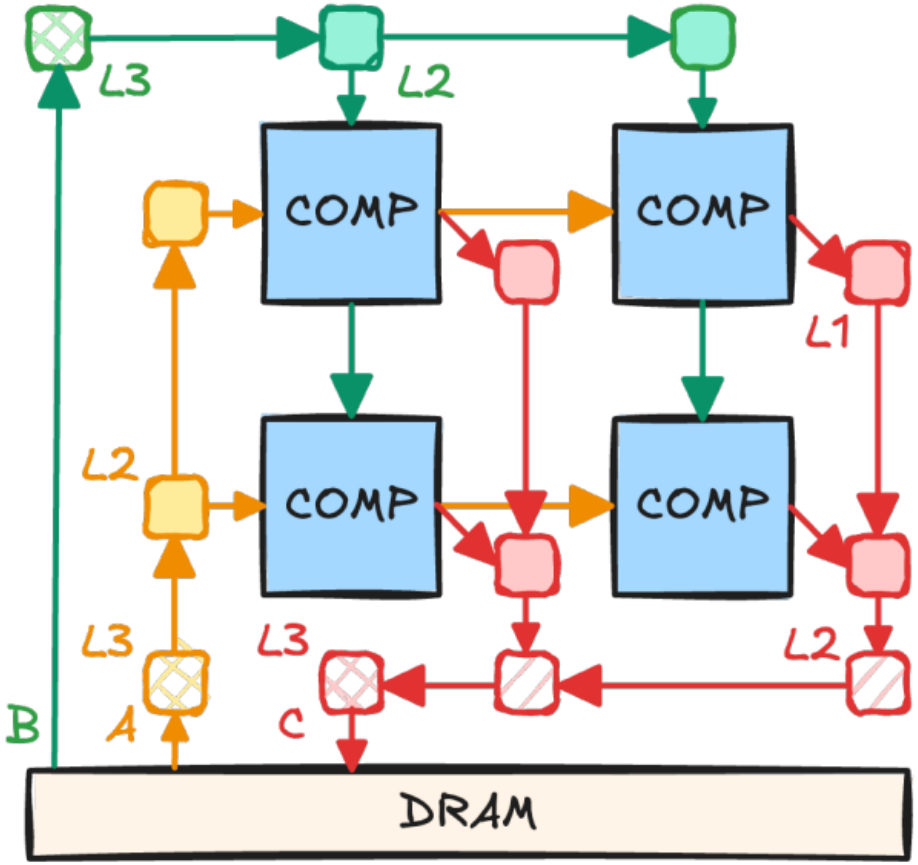
Structural Composition: Systolic Array

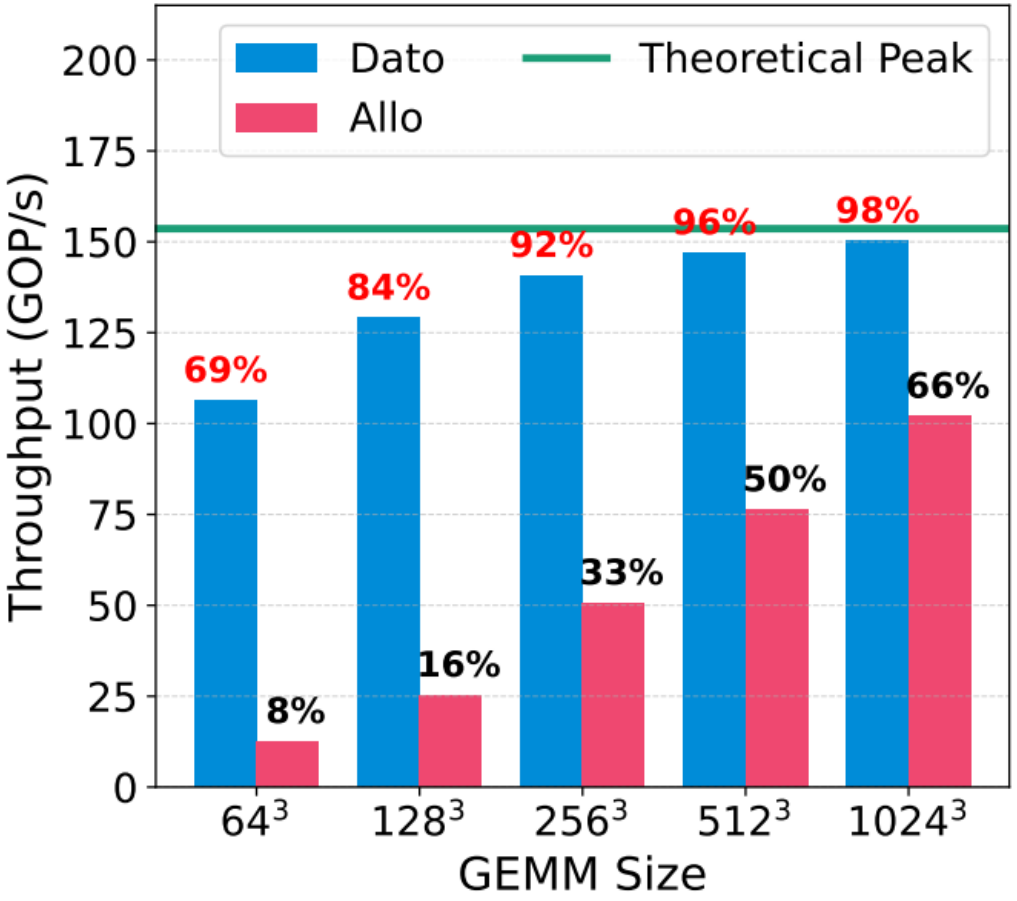
```
@df.region()
def top():
  fifo_A: Stream[int8, 4][P0, P1]
  fifo_B: Stream[int8, 4][P0, P1]


  # drain
  with allo.meta_elif(i == M + 1 and j > 0):
    for k in range(K):
      b: int8 = fifo_B[i, j].get()
  with allo.meta_elif(j == N + 1 and i > 0):
    for k in range(K):
      a: int8 = fifo_A[i, j].get()
  # main body
  with allo.meta_else():
    c: int16 = 0
    for k in range(K):
      a: int8 = fifo_A[i, j].get()
      b: int8 = fifo_B[i, j].get()
      c += a * b
      fifo_A[i, j + 1].put(a)
      fifo_B[i + 1, j].put(b)
      C[i - 1, j - 1] = c
```



Experiments: FPGA

- Same interface for designing high-performance systolic array
- Target Alveo U280 FPGA (300MHz)

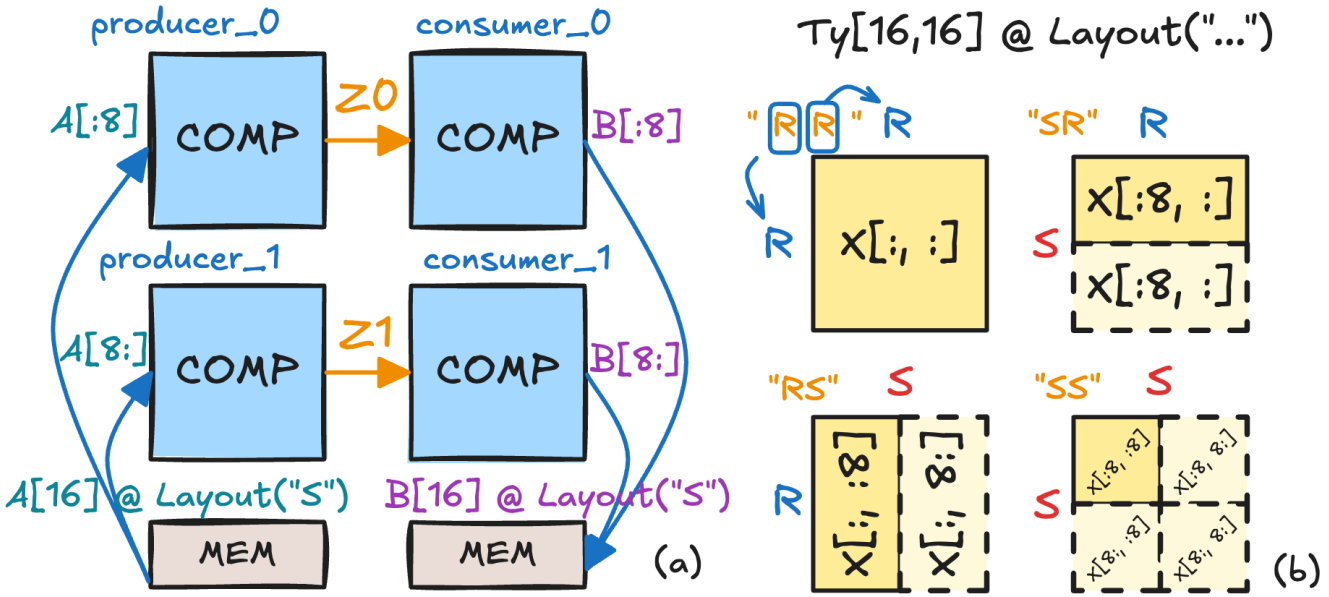




Allo Dataflow Programming Interface

- Layout as a refinement type of the tensor

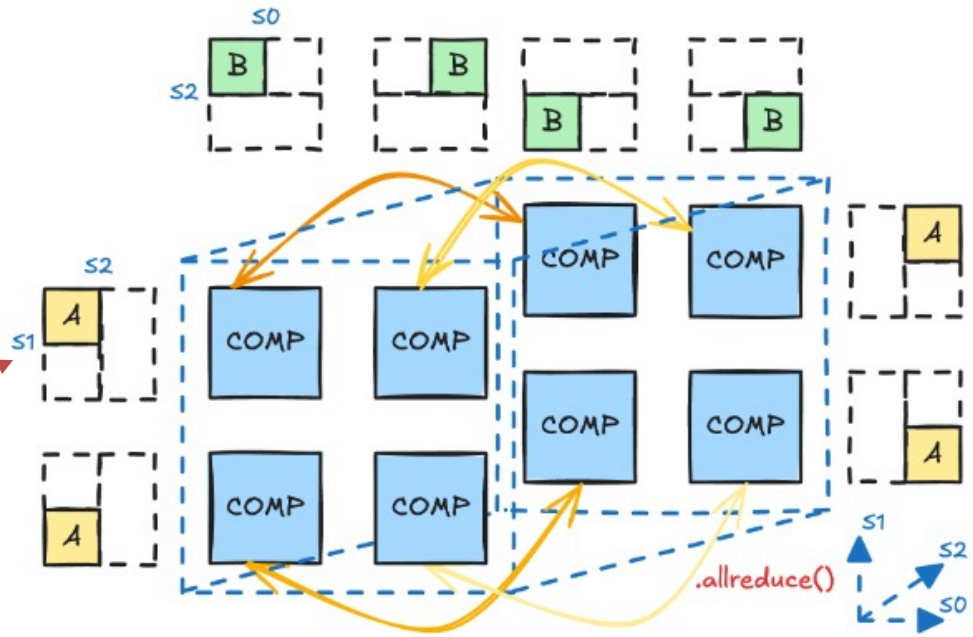
```
import allo
import allo.dataflow as df
from allo.ir.types import int8, Stream
from allo.memory import Layout
Ty, M, P0 = int8, 16, 2

S = Layout.Shard
R = Layout.Replicate

@df.unit()
def top():
 Z: Stream[Ty[M // P0]][P0]

 @df.work(mapping=[P0])
 def producer(A: Ty[M] @ [S(0)]):
  wid = df.get_wid()
  Z[wid].put(A[:])

 @df.work(mapping=[P0])
 def consumer(B: Ty[M] @ [S(0)]):
  wid = df.get_wid()
  B[:] = Z[wid].get() + 1
```

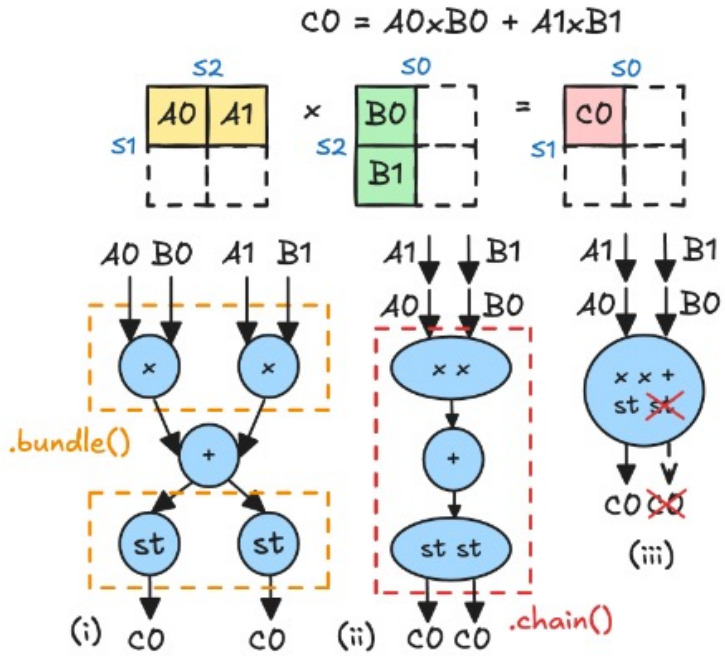


Allo Dataflow Programming Interface

- Virtual mapping
  - .bundle() and .chain() at the IR

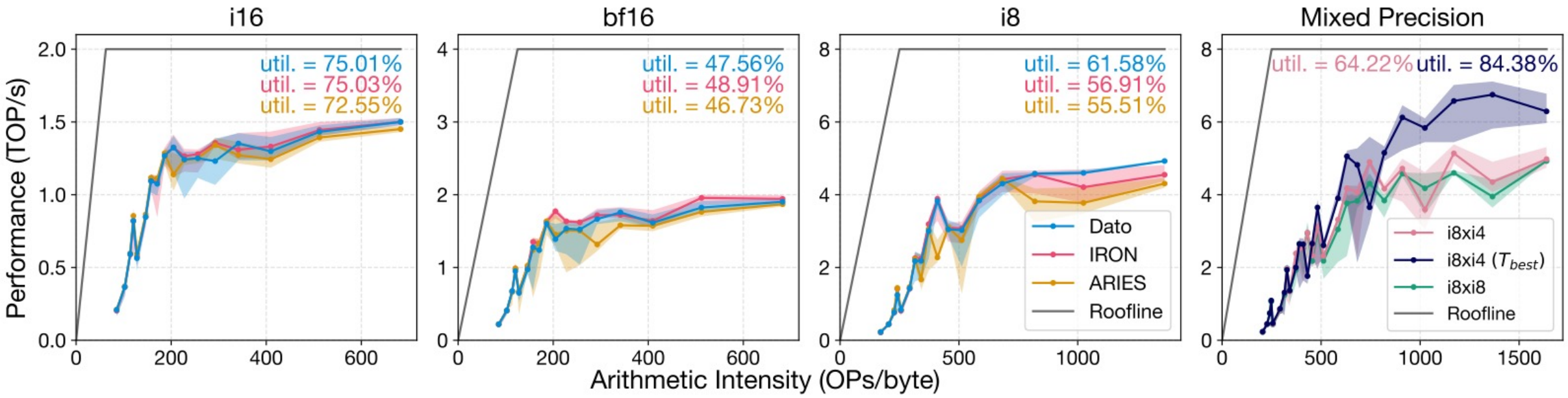
```
import allo
from allo.memory import Layout

S = Layout.Shard
Pk, Pm, Pn = 2, 2, 2

@df.unit()
def top():

 @df.work(mapping=[Pn, Pm, Pk])
 def gemm(
   A: Ty[M, K] @ [S(1),S(2)],
   B: Ty[K, N] @ [S(2),S(0)],
   C: Ty[M, N] @ [S(1),S(0)]):
  part_C = allo.matmul(A, B)
  allo.allreduce(part_C, op="+")
  C[:, :] = part_C
```





Experiments on NPU

- AMD XDNA1 NPU (4x5 compute tiles)
- GEMM (M, N, K in {256, 512, 1024, 2048}
- MLIR-AIR results similar to IRON (MLIR-AIE) thereby not shown in the figures

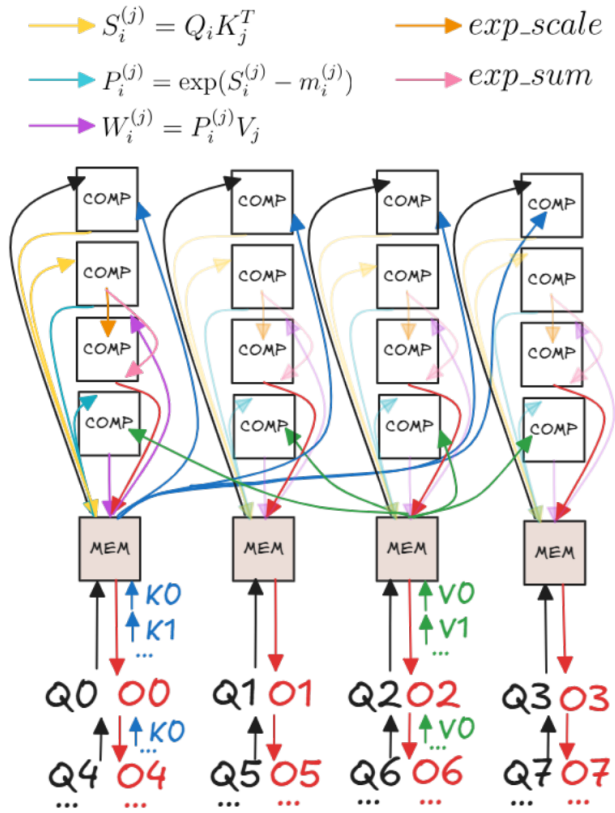


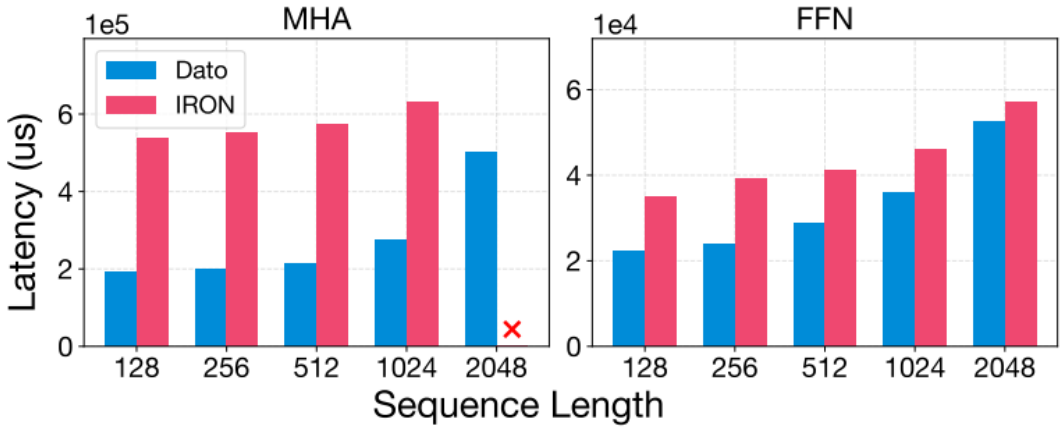
Experiments: MHA

- Flash Attention on NPU, mapping to the whole array
- MHA: H=12, D=64, L=[128,2048]
- FFN: D=768, 3072

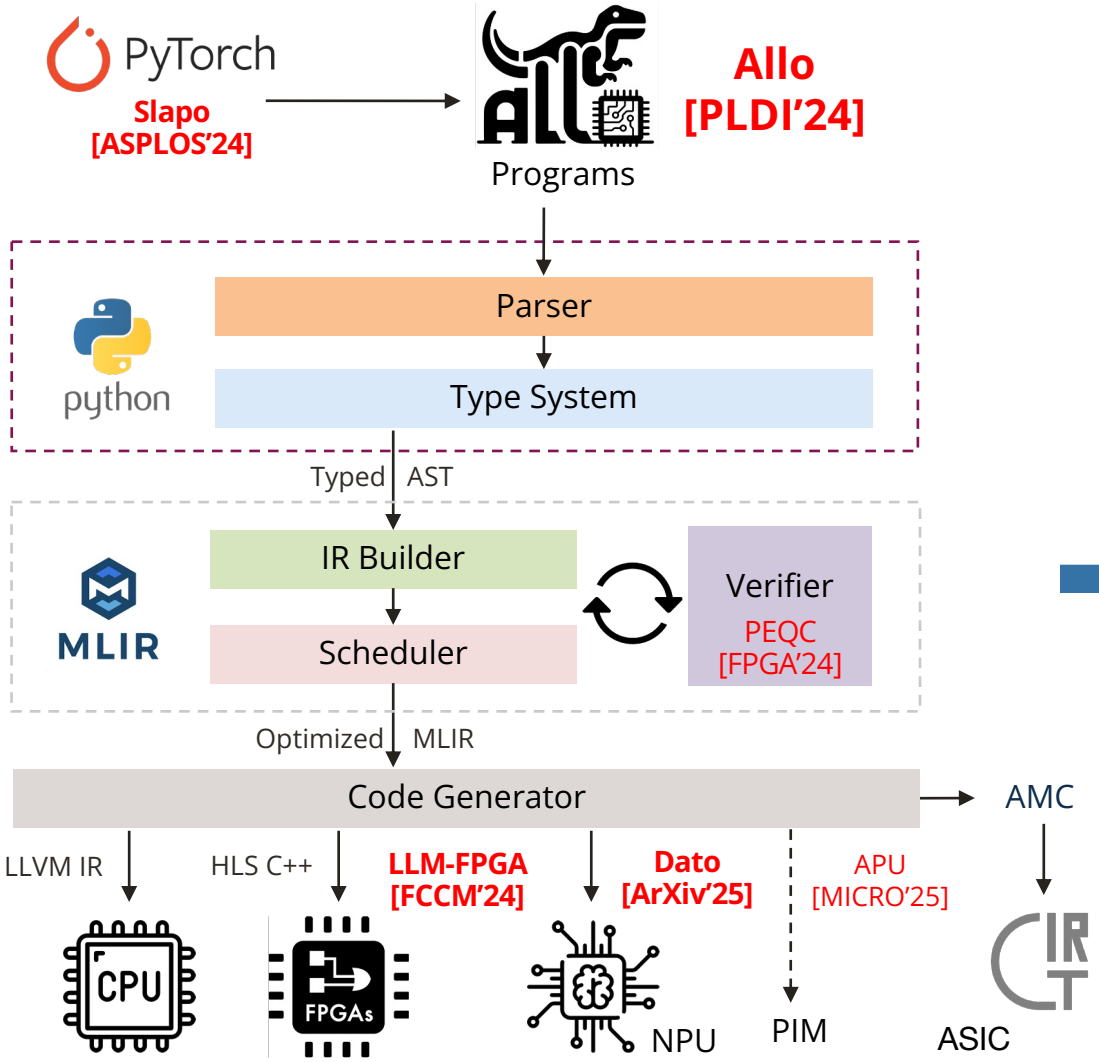
```
import allo
from allo.ir.types import bfloat16,\
   Stream, Layout

S = Layout.Shard
R = Layout.Replicate
Ty = bfloat16
P0, P1 = N // Tq, N // Tkv

@df.unit()
def top():
 q: Stream[Ty[Tq,d]][P0,P1]
 s: Stream[Ty[Tq,Tkv]][P0,P1]
 w: Stream[Ty[Tq,Tkv]][P0,P1]
 exp_sum: Stream[Ty[Tq]][P0]
 exp_scale: Stream[Ty[Tq]][P0]
 o: Stream[Ty[Tq,d]][P0,P1]

@df.work(mapping=[P0,1])
def send_q(Q: Ty[N,d] @ [S(0), R]):
 tid, _ = df.get_wid()
 for i in range(P1):
  q[tid,i].put(Q)

@df.work(mapping=[P0,P1])
def gemm0(K: Ty[d,N] @ [R, S(1)]):
 tid, tj = df.get_wid()
 s[tid,tj].put(allo.matmul(
   q[tid,tj].get(), K))
```

```
@df.work(mapping=[P0,1])
def softmax():
 ti, _ = df.get_wid()
 max_: Ty[Tkv]
 sum_: Ty[Tkv]
 init_softmax_kernel(max_,sum_)
 for i in range(P1):
  weight: Ty[Tq,Tkv]
  scale: Ty[Tkv]
  online_softmax_kernel(
   s[ti,i].get(), max_, sum_, weight, scale)
  exp_scale[ti].put(scale)
  w[ti,i].put(weight)
 exp_sum[ti].put(sum_)

@df.work(mapping=[P0,P1])
def gemm1(V: Ty[N,d] @ [S(1), R]):
 ti, tj = df.get_wid()
 o[ti,tj].put(allo.matmul(
      w[ti,tj].get(),V))

@df.work(mapping=[P0,1])
def acc(O: Ty[N,d] @ [S(0), R]):
 output: Ty[Tq,d] = 0
 ti, _ = allo.get_wid()
 for i in range(P1):
  rescale_kernel(output, exp_scale[ti].get())
  output[:,:] = allo.add(output, o[ti,i].get())
 scale_kernel(output,exp_sum[ti].get(),0)
```





Allo ADL Summary



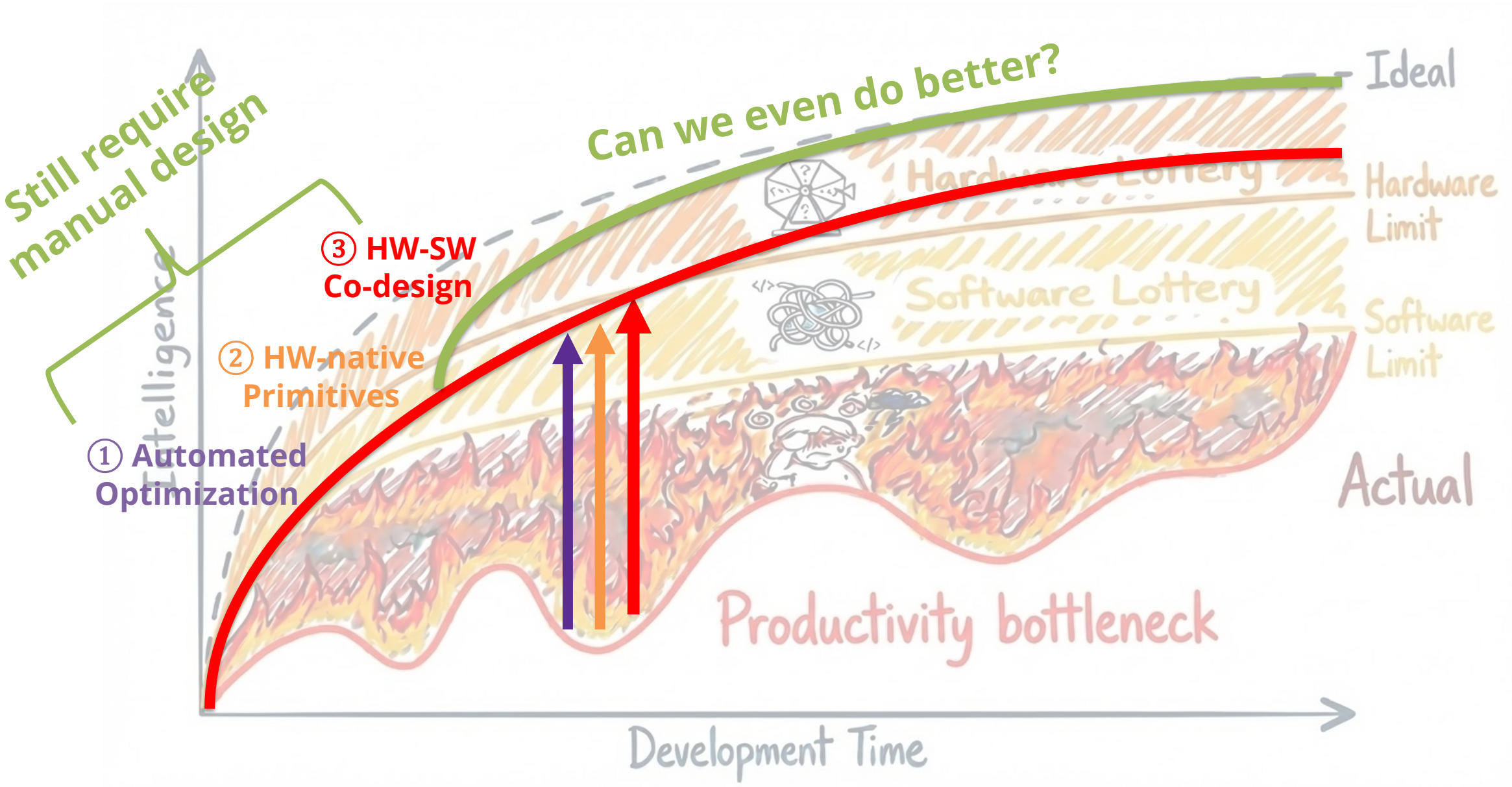
cornell-zhang/**allo**

Allo Accelerator Design and Programming Framework (PLDI'24)

https://github.com/cornell-zhang/allo

38 Contributors · 51 Issues · 29 Discussions · 351 Stars · 64 Forks

From Hardware Generation to …



What's Next?

- **Learning the bitter lesson –** Most significant progress in AI has come from scaling**,** where more compute, data, and general meta-methods beat specialized, human-designed strategies in the long run

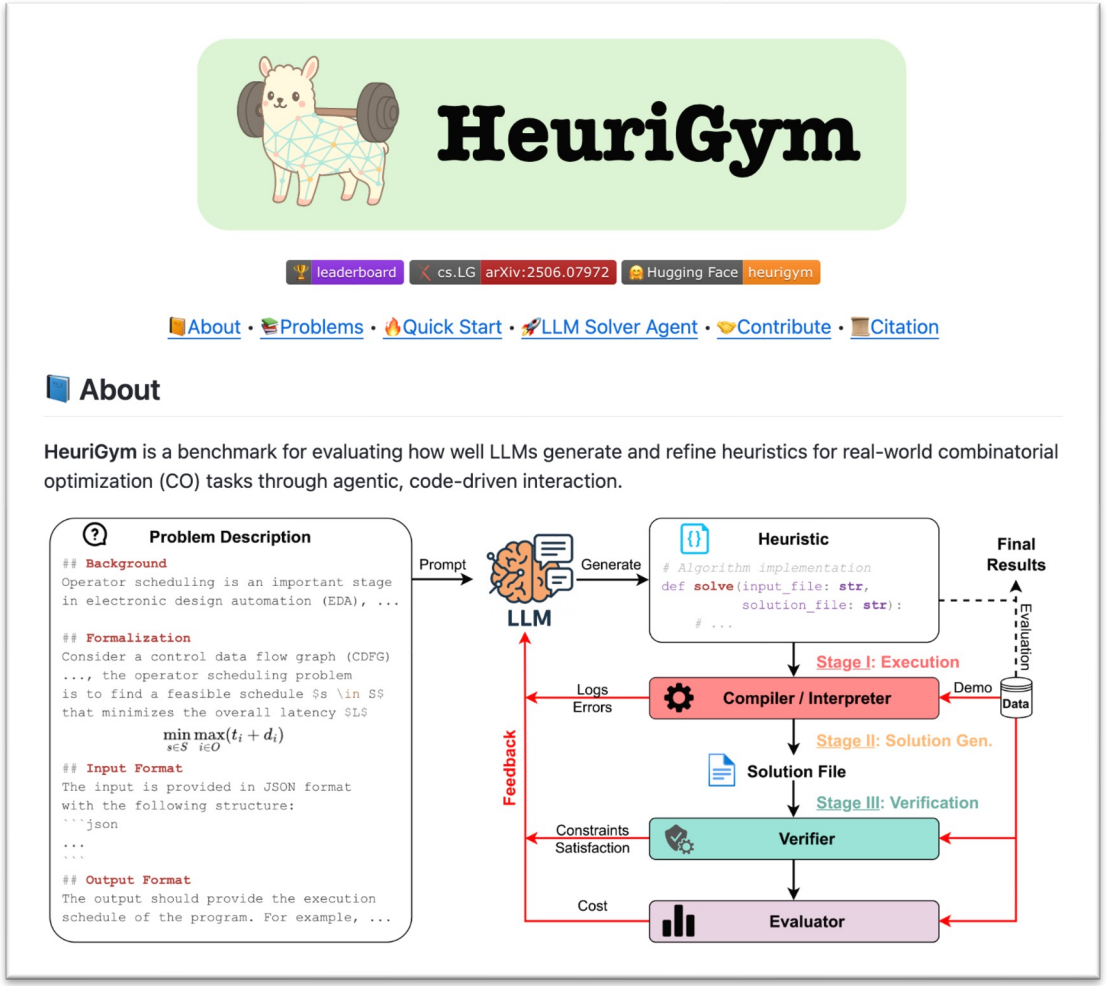
- LLM agents are eating the world.

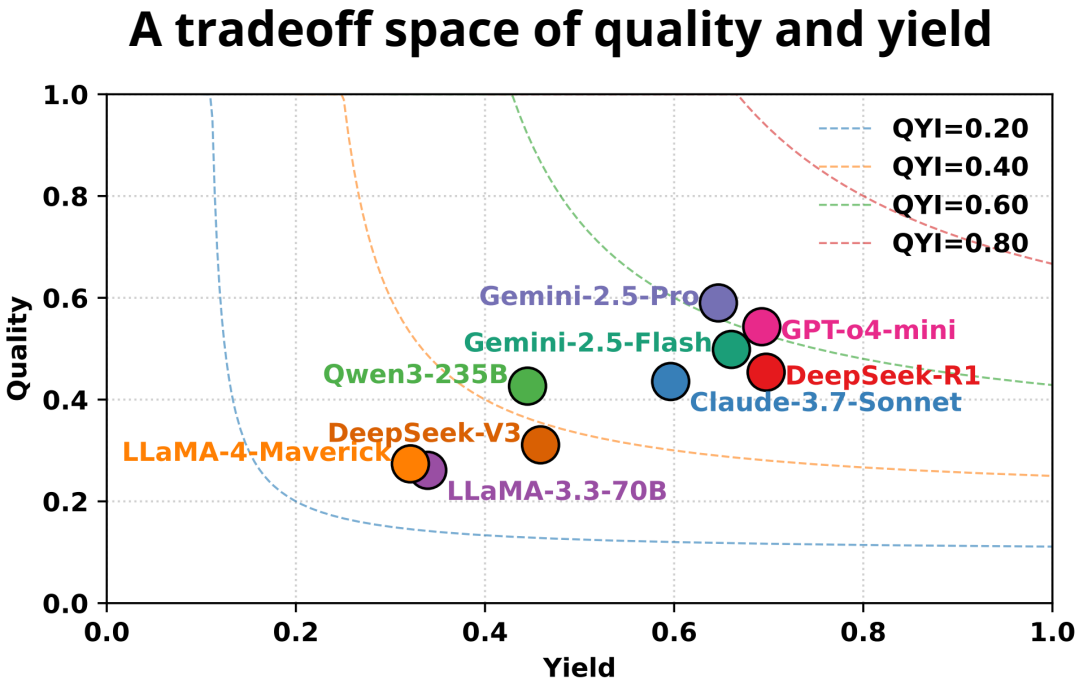
- Can they help tackle system problems?

- Both Tawa and Allo have manually designed optimization algorithms, can we use LLM to generate those algorithms?

* Rich Sutton, "The Bitter Lesson", 2019

HeuriGym: Agentic Benchmark for LLM-Crafted Heuristics





- SoTA LLMs in early 2025 achieve a quality-yield index (QYI) score of **0.6**, where 1.0 = human expert
- More results available on HeuriGym leaderboard

https://github.com/cornell-zhang/heurigym

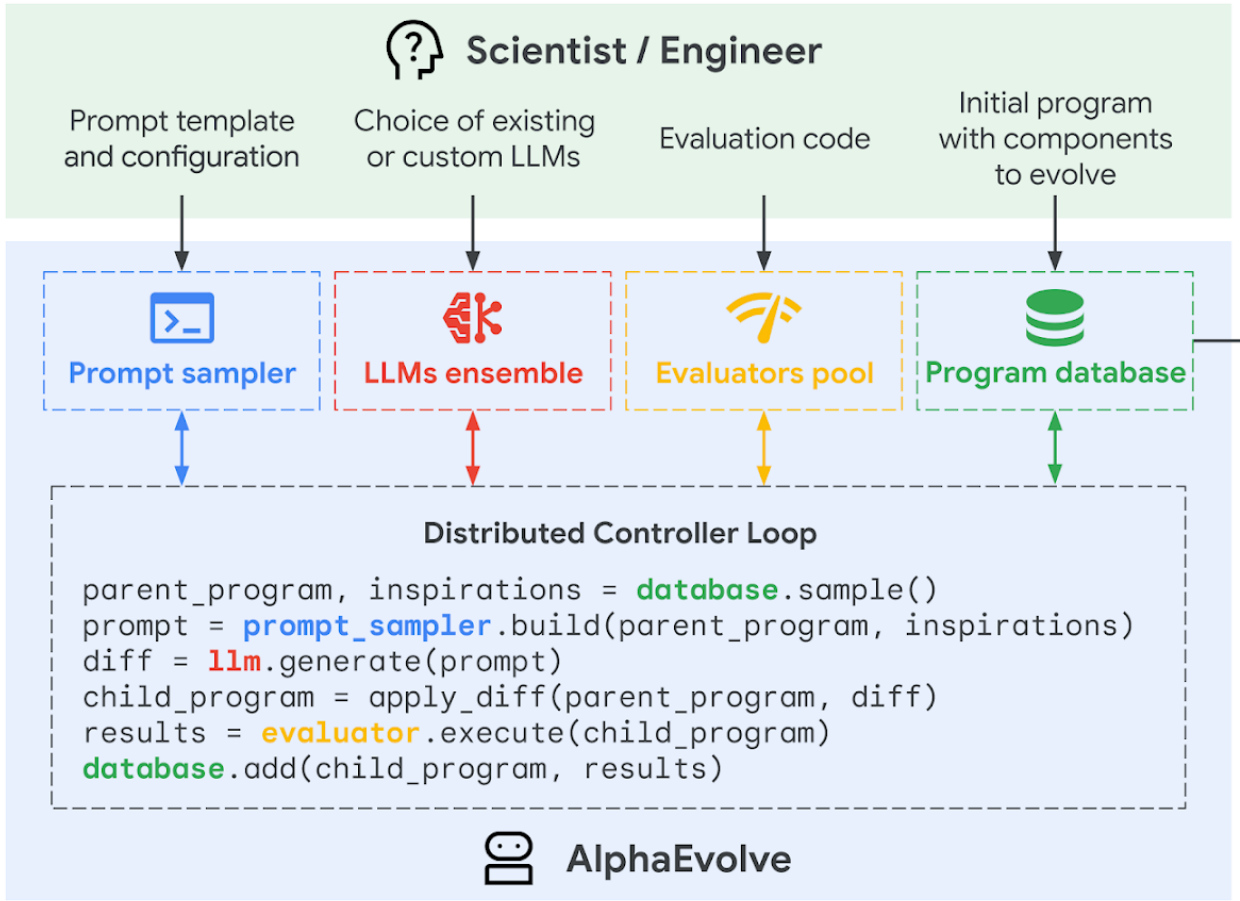
H. Chen et al., "HeuriGym: An Agentic Benchmark for LLM-Crafted Heuristics in Combinatorial Optimization", ICLR, 2026.

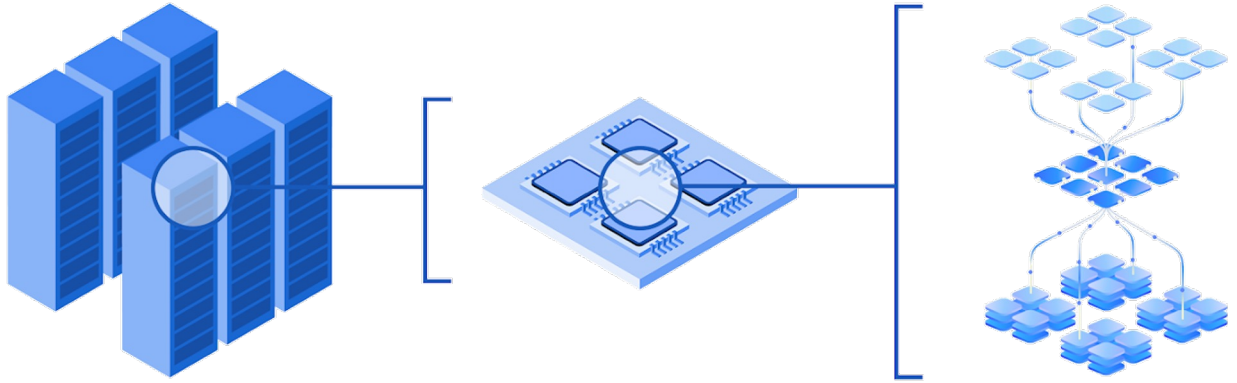
Magellan: Autonomous Discovery of Novel Compiler Optimization Heuristics with AlphaEvolve

Hongzheng Chen, Alexander Novikov, Ngân (NV) Vũ,
Hanna Alam, Zhiru Zhang, Aiden Grossman,
Mircea Trofin, Amir Yazdanbakhsh

CGO'26 C4ML

AlphaEvolve in a Nutshell





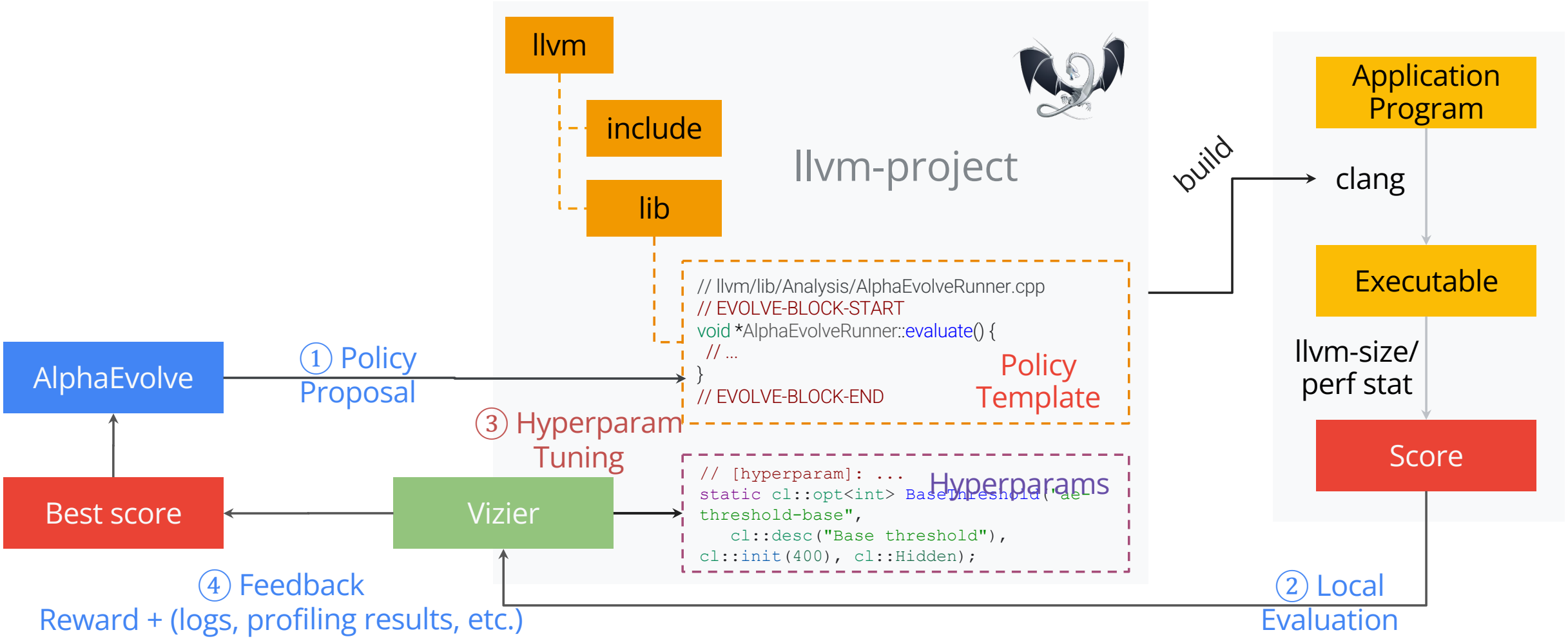
Data Center Optimization
Borg Scheduling

Hardware Optimization
TPU Circuit Design

Software Optimization
Gemini Training

* https://deepmind.google/blog/alphaevolve-a-gemini-powered-coding-agent-for-designing-advanced-algorithms/

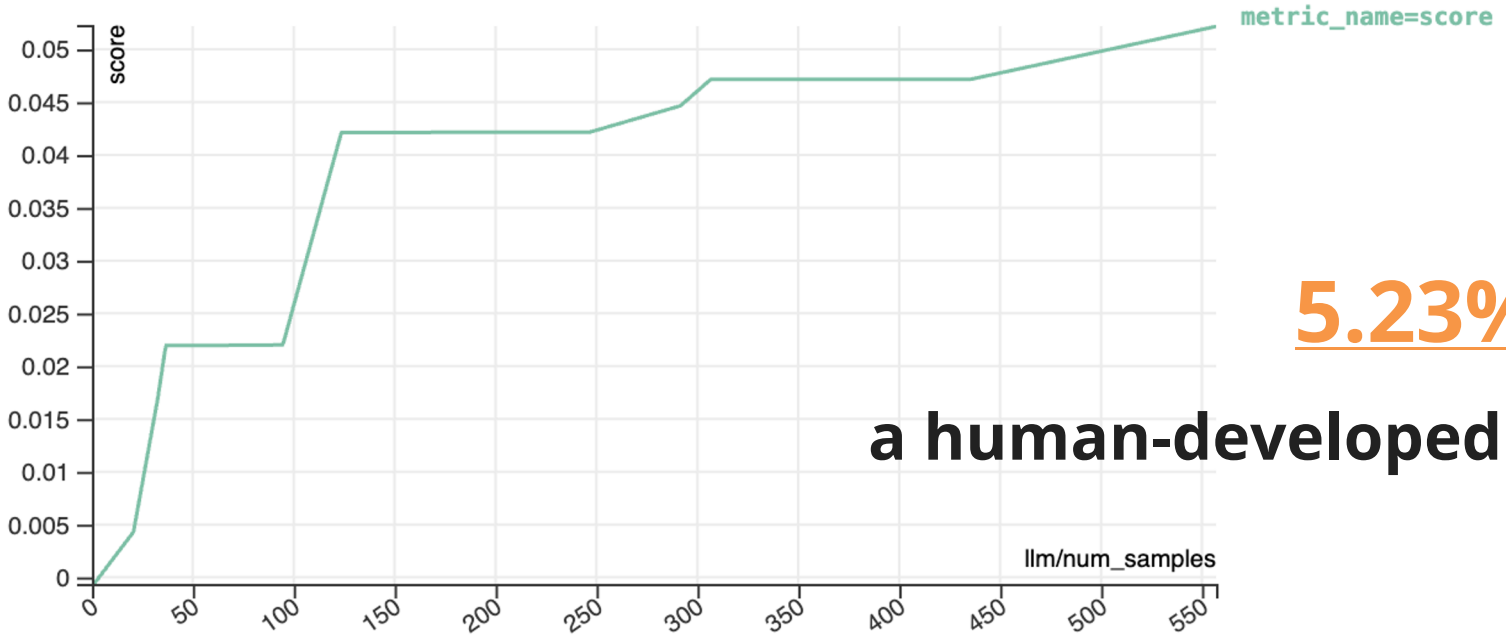
Evolving Compiler (passes) using LLM Agents



The system works in **an autonomous loop**: A two-stage pipeline that decouples high-level policy design (LLM-driven) from low-level parameter optimization (executed via an autotuner).

Case Study I: Function Inlining for Size (w/o autotuner)

- Automatically discovered the entire heuristic based on arbitrary LLVM API calls
  - Evaluated on internal search application and started from a naïve heuristic
  - Require more trials and errors compared to giving pre-defined features
  - Production Ready → The generated heuristic is clean C++ that is ready to be directly deployed upstream with minimal manual intervention



5.23% better than

a human-developed heuristic after 1.5 days!

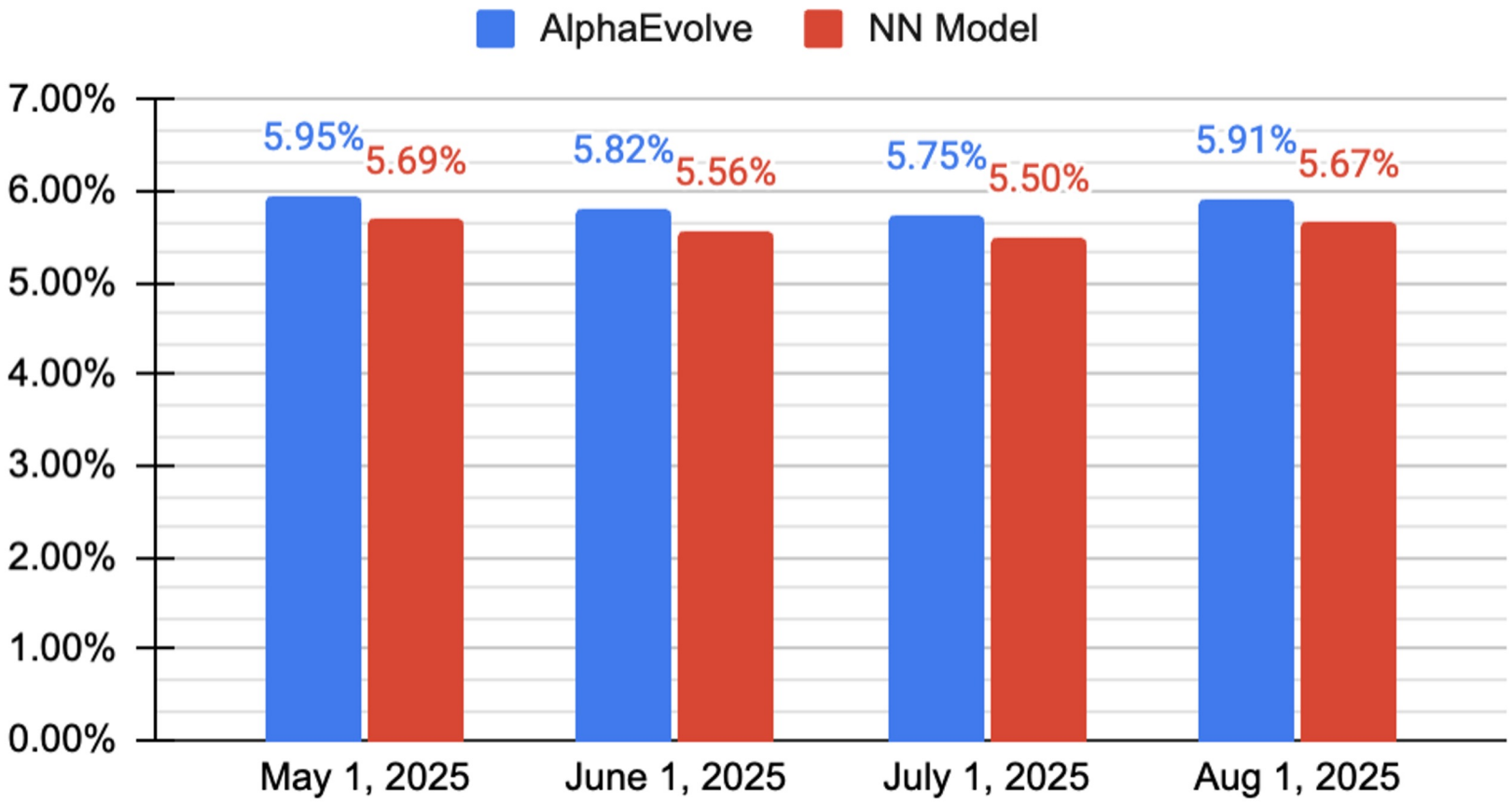
Case Study I: Function Inlining for Size (w/ Autotuner)

- Use the workflow combining AlphaEvolve+Vizier
  - 15 iterations * 10 Vizier samples / iter
  - Significant boost on program sampling efficiency!
  - Only 2/15 ≈ 13% of programs are invalid! 5x reduction compared w/ previous one
  - Only 5 hours to achieve the same level of size reduction! 7x time reduction

More than +5% better than

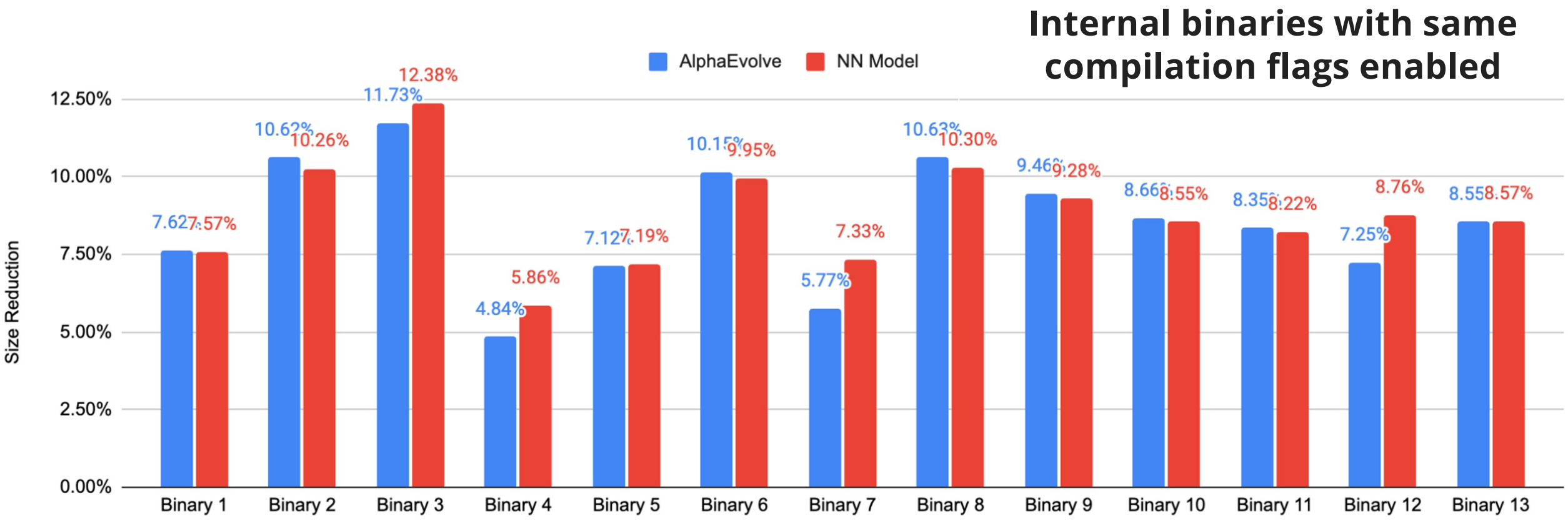
a human-developed heuristic with only 5 hours!

Case Study I: Function Inlining for Size
Temporal Generalization



- Evaluated on four different timestamps with one-month interval
- Compared w/ a two-year old TensorFlow NN model (we didn't retrain it!)

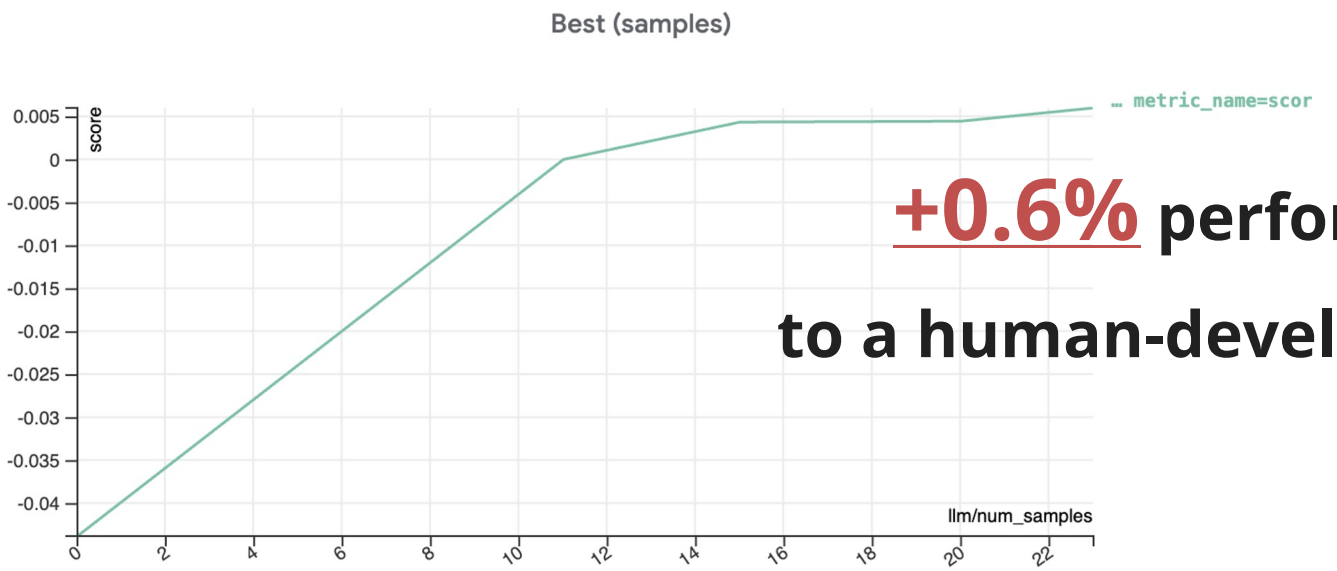
Case Study I: Function Inlining for Size
Domain Generalization



On average (10+ production applications), AlphaEvolve achieves a **8.79%** size reduction on average, on par with the NN model's 8.52%.

Case Study II: Function Inlining for Performance

- Finally **broke the performance ceiling**!
  - **First time** achieving performance improvement using ML for this problem *at this problem complexity and from a "best practices" baseline*
  - Used the clang compiler as the benchmark
  - ① Started from a policy with -4.7% regression obtained by Gemini-2.5-Pro
  - 23 iterations * ② **40** Vizier samples / iter (3 days)
  - 7 hours per iteration, 4 parallel samples, avg 30 min per sample



+0.6% performance improvement compared ③ to a human-developed heuristic w/ Gemini-3-Pro!

Takeaways & Future Work

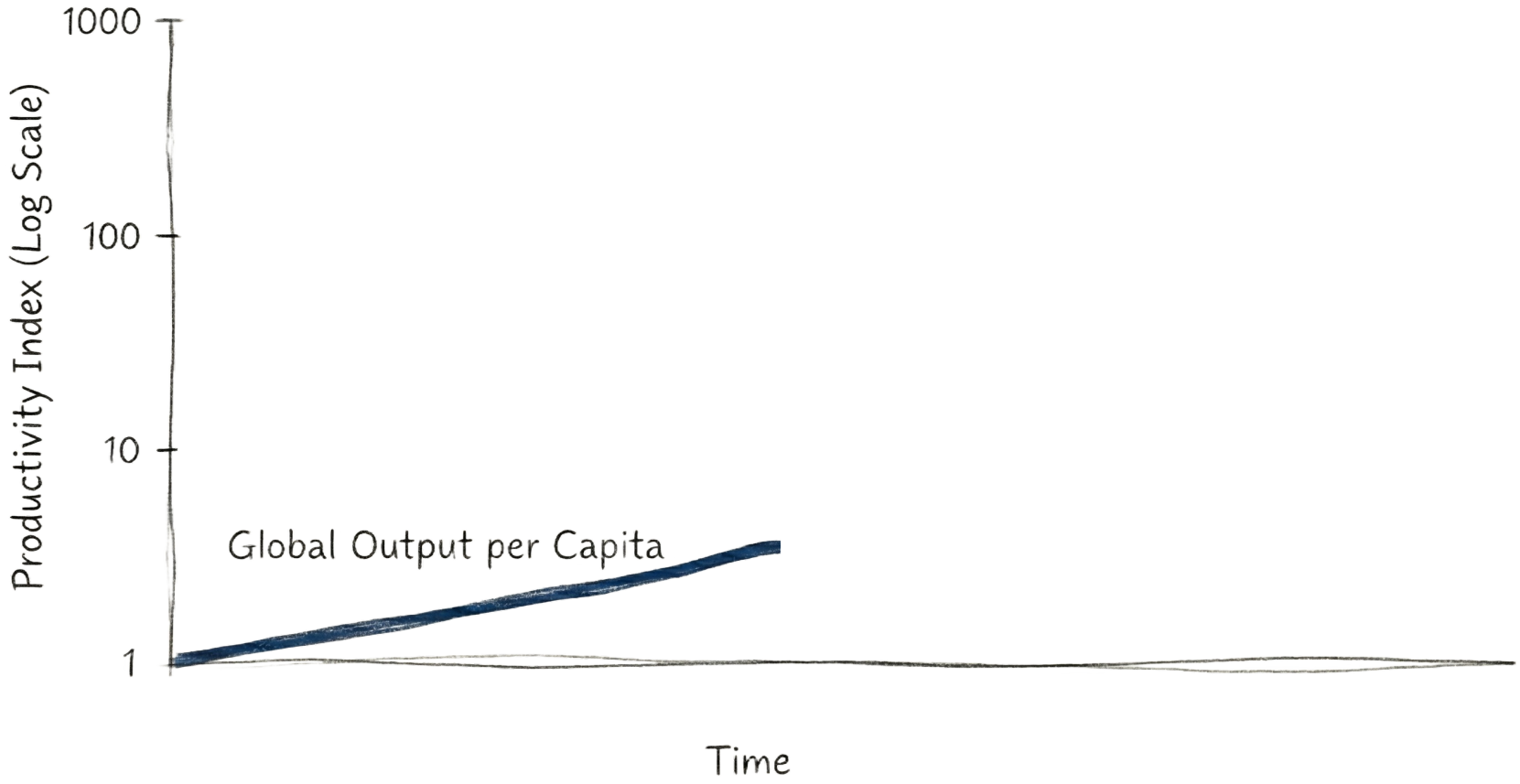
- **Automated Discovery & Productivity Gain** → AlphaEvolve boosts productivity by automating the labor-intensive process of heuristic design.
- **Human-Competitive Results** → The system consistently generates heuristics that perform on par with, and can sometimes surpass, those created by human experts.
- **Faster iteration pipeline** → The incoporation of autotuner boosts sampling efficiency.

Next steps:

- **Scale up evaluations:** Explore the boundaries of AlphaEvolve and see if it has signals on convergence.
- **Tackle "Green-Field" Problems:** Apply AlphaEvolve (or its variants) to novel compiler domains (e.g., GPU and NPU optimization problems) to evaluate its ability to innovate where little or no prior human expertise exists.

"We're moving from the age of scaling to the age of **research**."

-- Ilya Sutskever



Thank You! Questions?

Cornell University

Contact: Hongzheng Chen
hzchen@cs.cornell.edu