NeurIPS Tutorial

How to Build Agents to Generate Kernels for Faster LLMs (and Other Models!)

Dec 2025
Mexico City

Presenters:
Sina Rafati, PhD
Hao Li,
Efthimios Gianitsos,
Sharon Zhou, PhD

Team:
Sina Rafati (AMD), Hao Li (AMD), Azalia Mirhoseini (Stanford), David Kanter (ML Commons), Anna Goldie (Google Deepmind), Laurence Moroney (ARM), Vartika Singh (Nvidia), Mark Saroufim (Meta), Simon Guo (Stanford), Kesavan Ramakrishnan (Stanford), Chris Lattner (Modular AI), Sitao Huang (UC Irvine), Efthimios Gianitsos (AMD), Vincent Ouyang (AMD), Nithya Manohar (AMD), Sharon Zhou (AMD)

AMD
together we advance_

Pain of waiting!

Meet the Kernels

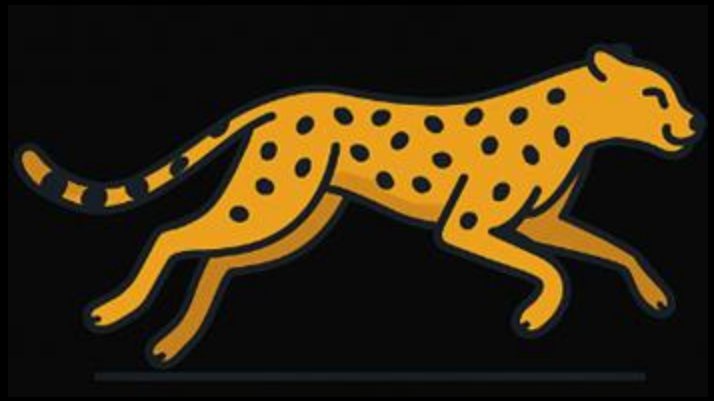
Inefficient Kernel

Optimized Kernel

In GPU land, the chef isn’t Gordon Ramsay yelling at you—
it’s thousands of tiny chefs all screaming ‘Yes, Chef!’ at once.

The speed evolution of kernels

Naïve Kernel



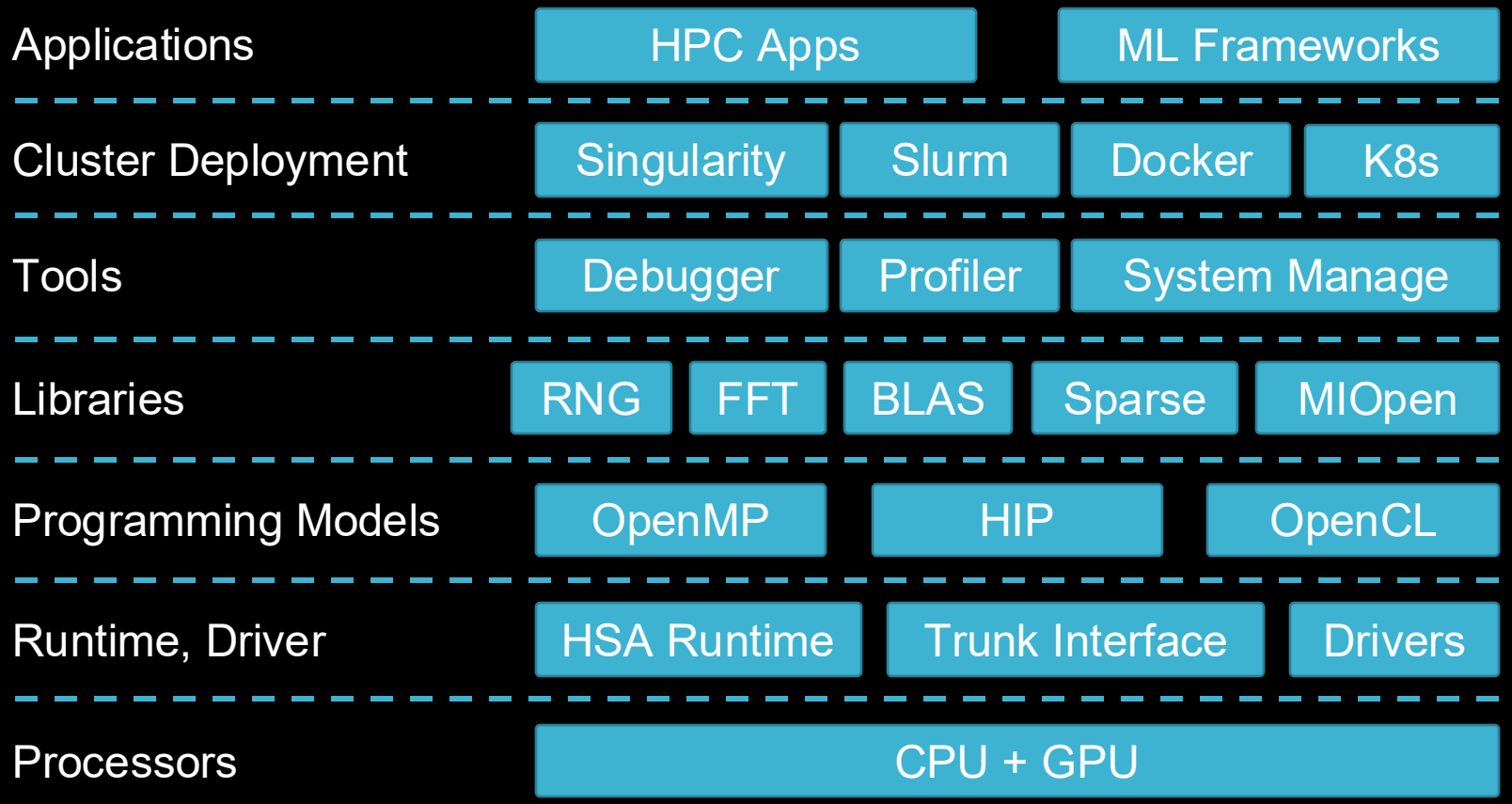
Hand-tuned Kernel

AI-tuned Kernel

Glossary

- HIP: Heterogeneous-compute Interface for Portability
- ACE: Asynchronous Compute Engine
- HWS: Hardware Scheduler
- SPI: Shader Processor Interpolator
- WG: Work Group
- WF: Wave Front
- LDS: Local Data Share
- SGPRs: Scalar General-Purpose Registers
- VGPRs: Vector General-Purpose Registers
- CU: Compute Unit
- SIMD: Single Instruction, Multiple Data

ROCm Ecosystem

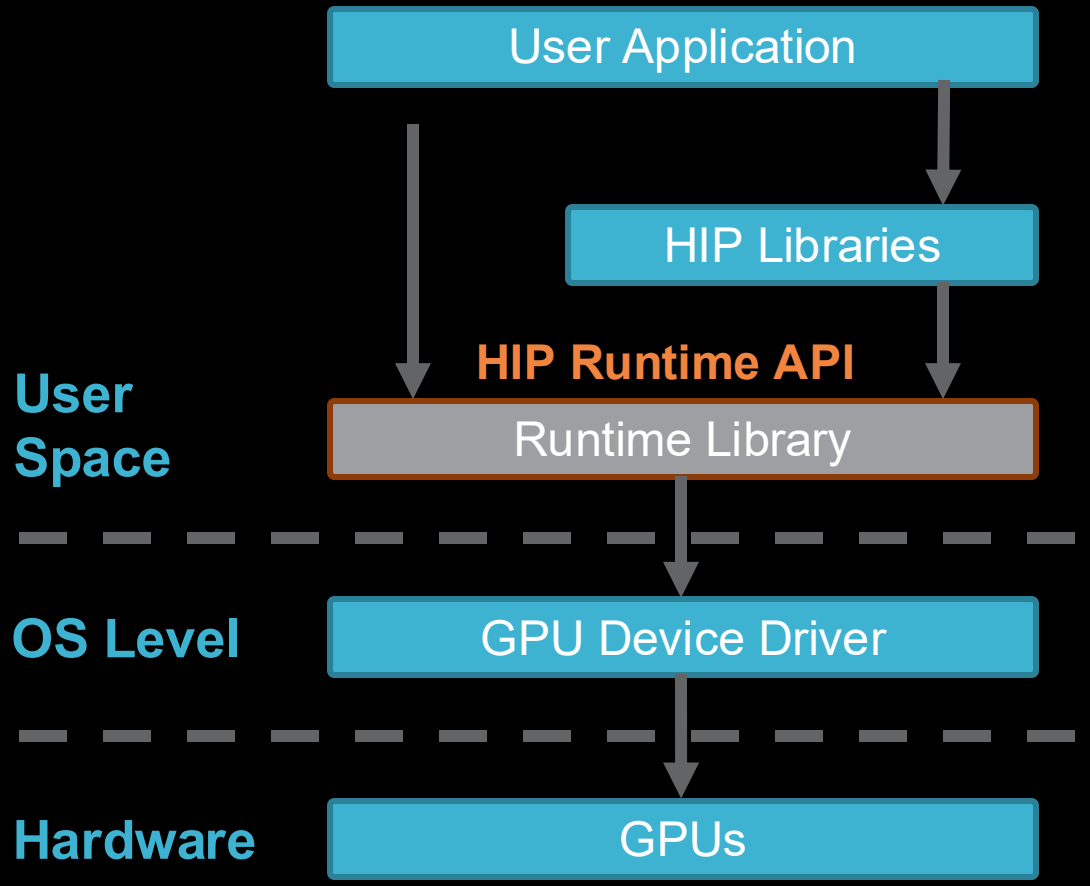


Representation created by Yifan Sun

HIP is a Runtime Library

HIP Runtime APIs

- Device management
- Memory management
- Kernel launching
- Synchronization
- Error Handling



Representation created by Yifan Sun

GPU Arch

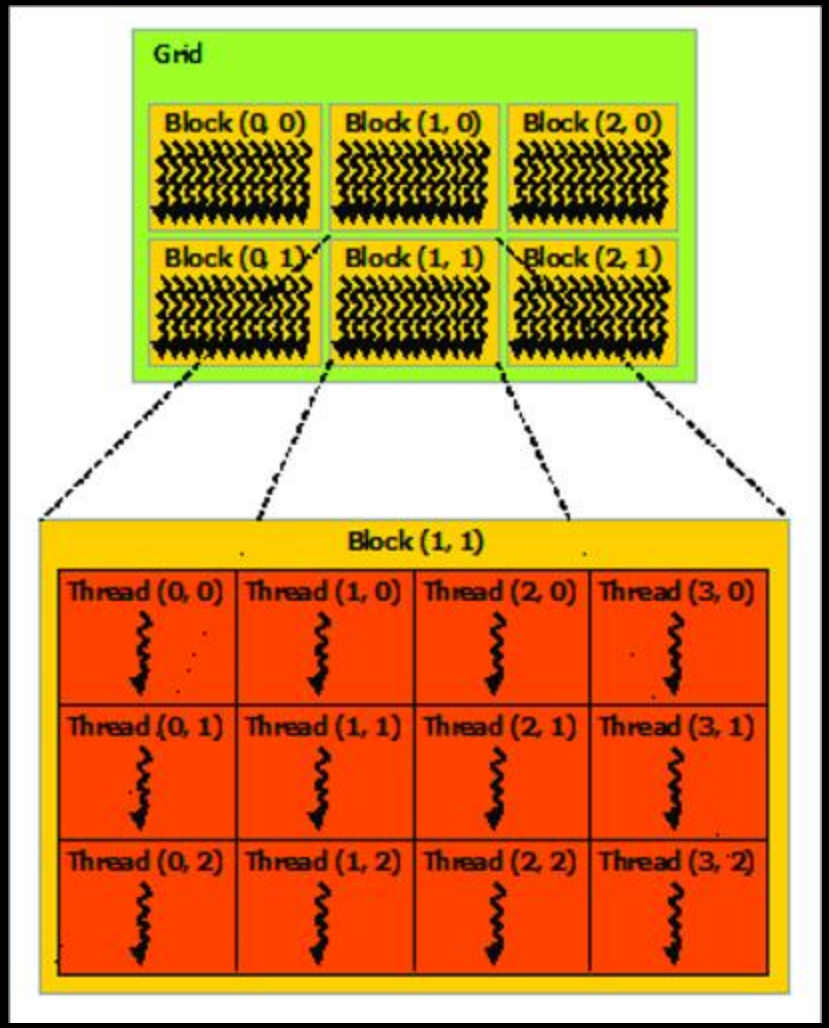
Kernel Launch

Grid

Block/Workgroup

Warp/Wavefront

Thread/Work-item



Thread Indexing

Built-in Variables

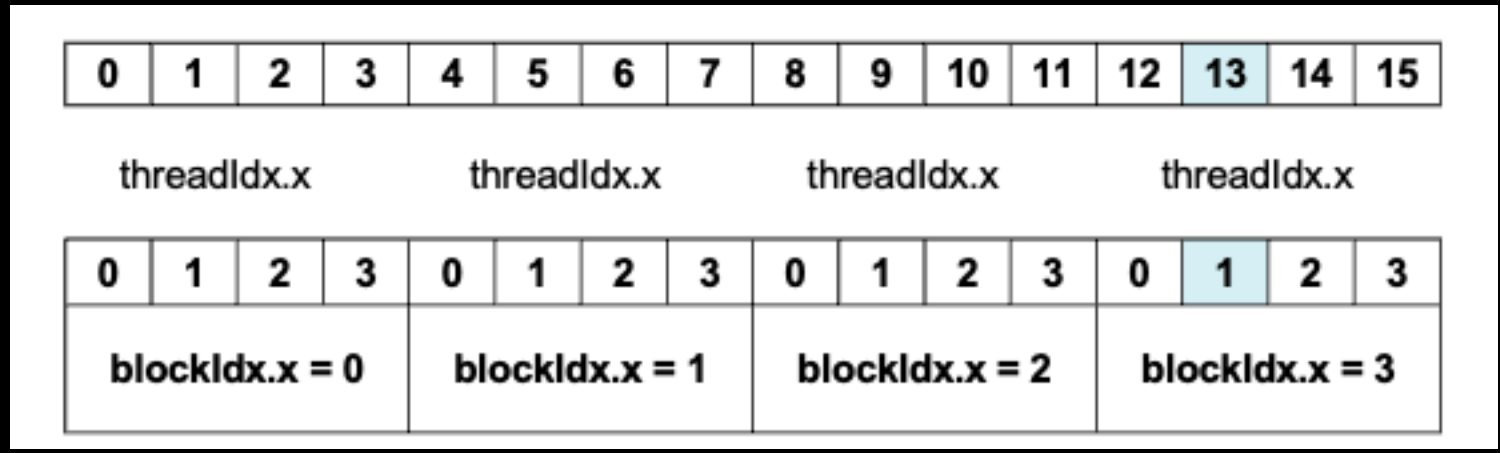
- blockIdx.x, blockIdx.y, blockIdx.z
  Block (0, 1) => blockIdx.x = 0, blockIdx.y = 1
- threadIdx.x, threadIdx.y, threadIdx.z
  Thread (2,1) => threadIdx.x = 2,
  threadIdx.y = 1

```
int i = blockDim.x * blockIdx.x + threadIdx.x
```

Thread Indexing

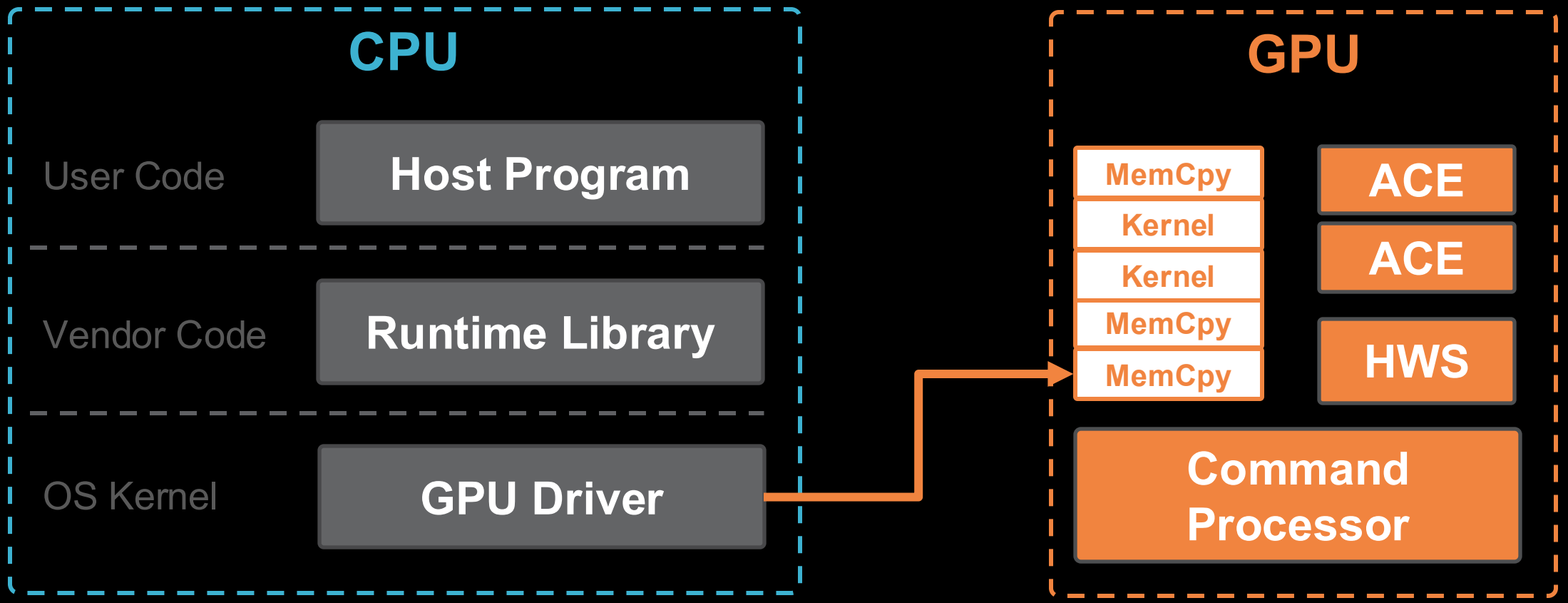
Built-in Variables

```
int i = blockDim.x * blockIdx.x + threadIdx.x
```



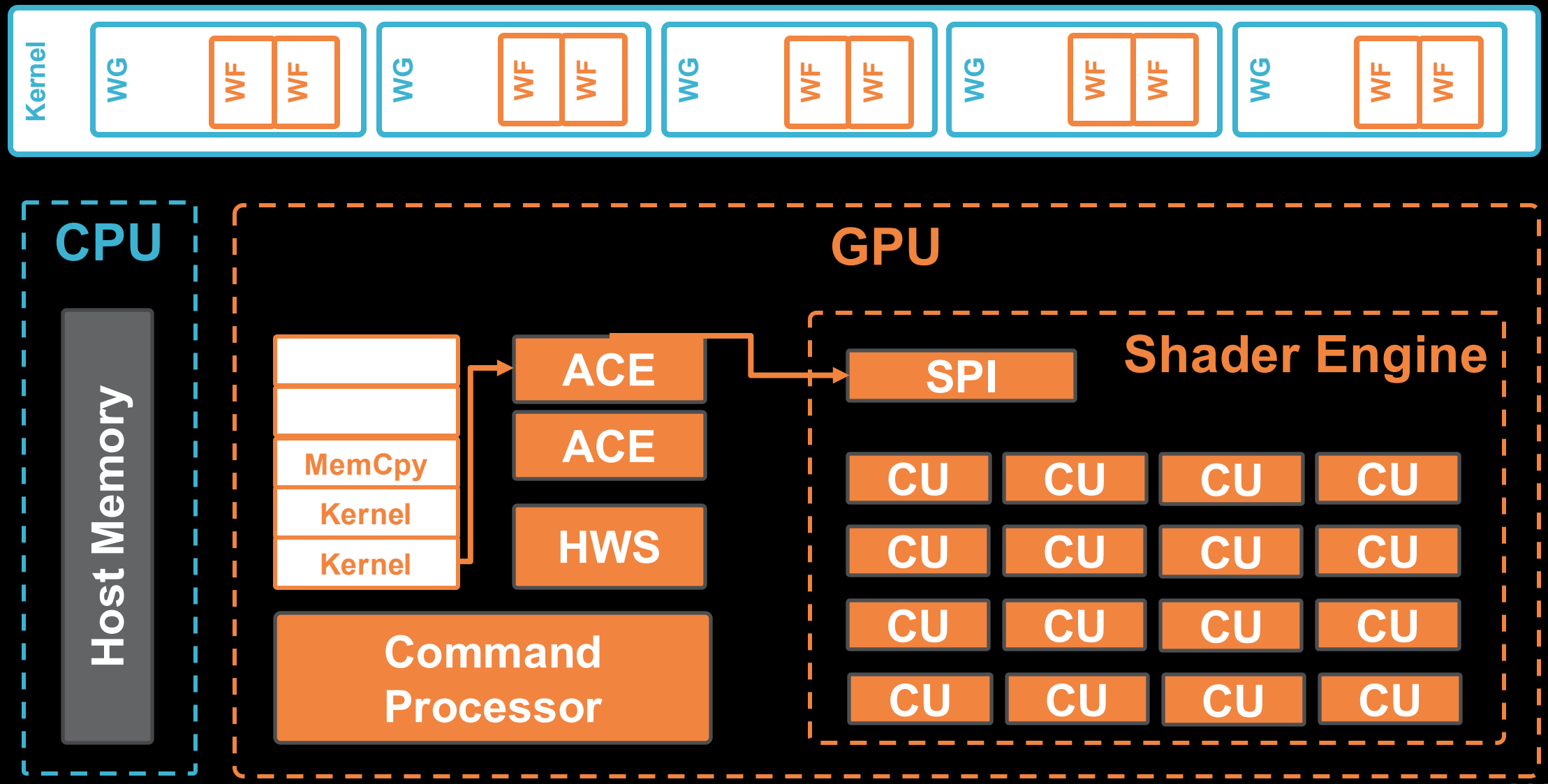
Representation created by Yifan Sun

Asynchronous Compute Engine (ACE)



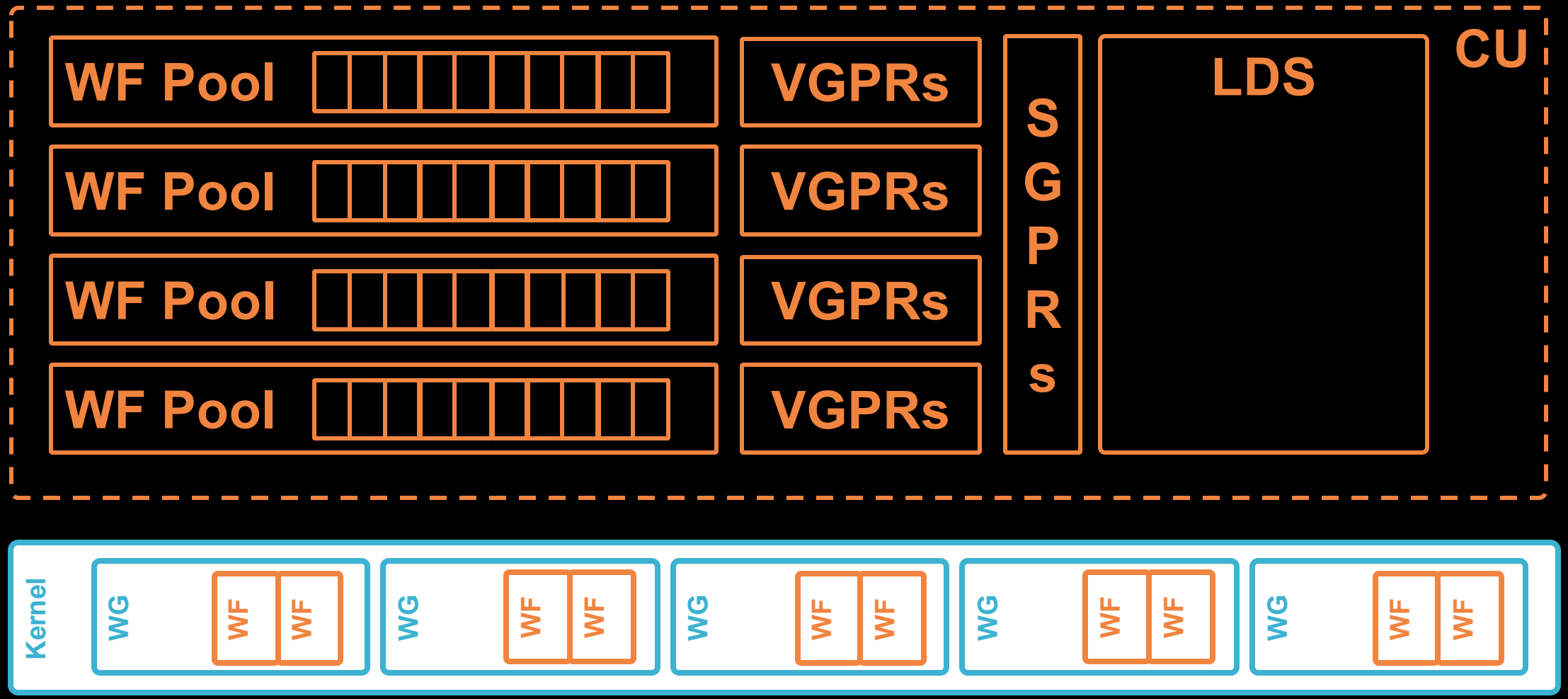
Representation created by Yifan Sun

Kernel Launching



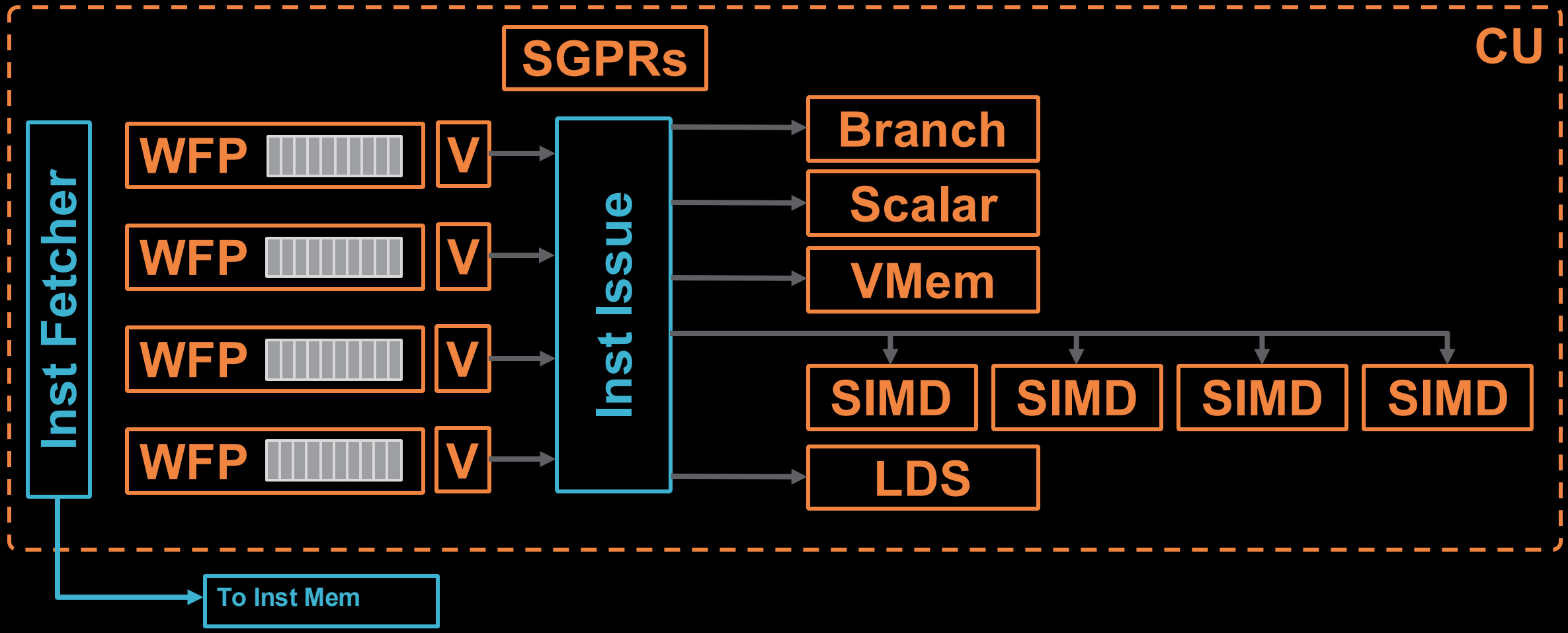
Representation created by Yifan Sun

WF Slot, Registers, LDS



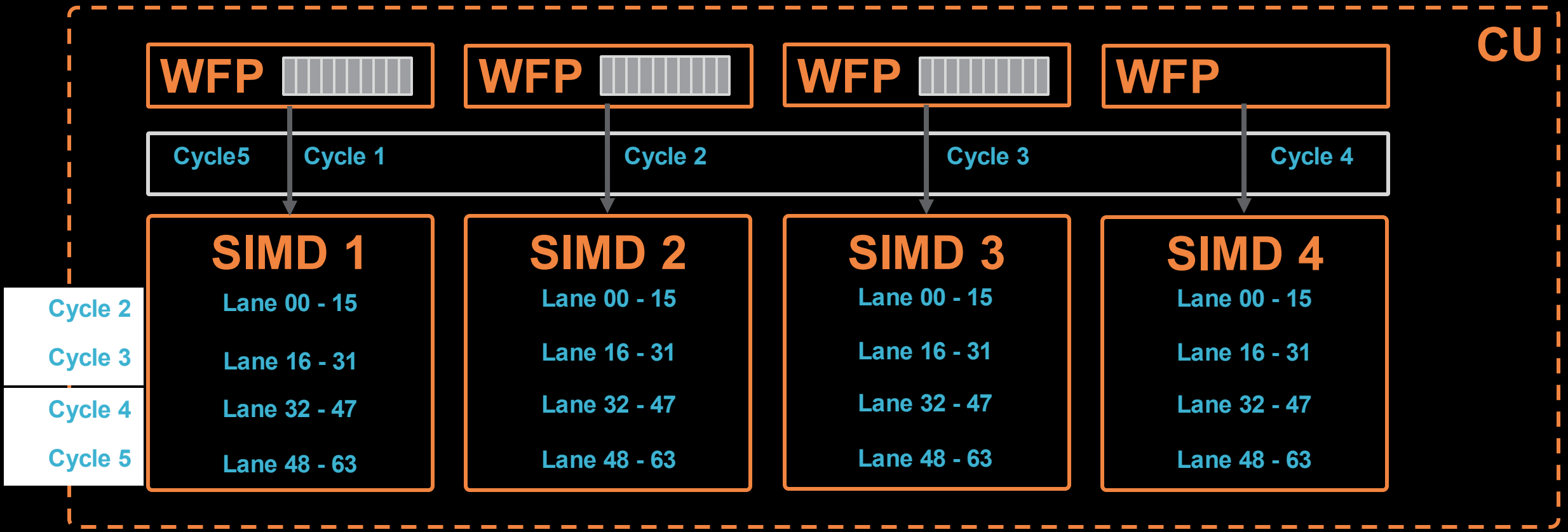
Representation created by Yifan Sun

The Compute Unit



Representation created by Yifan Sun

The SIMD Unit



Representation created by Yifan Sun

Kernel Launch

Header file
- hip/hip_runtime.h

Define GPU code
- Kernel
- Function returns void

Run GPU code
- Launch kernel

```
#include <hip/hip_runtime.h>

__global__ void gpuHello() {
  int id = blockIdx.x * blockDim.x + threadIdx.x;
  printf("Hello World from thread %d\n", id);
}

int main(){
  gpuHello<<<1, 1>>>();
  hipDeviceSynchronize();

  return 0;
}
```

The standard syntax for compiling HIP program is:

```
hipcc hello.cpp -o hello
```

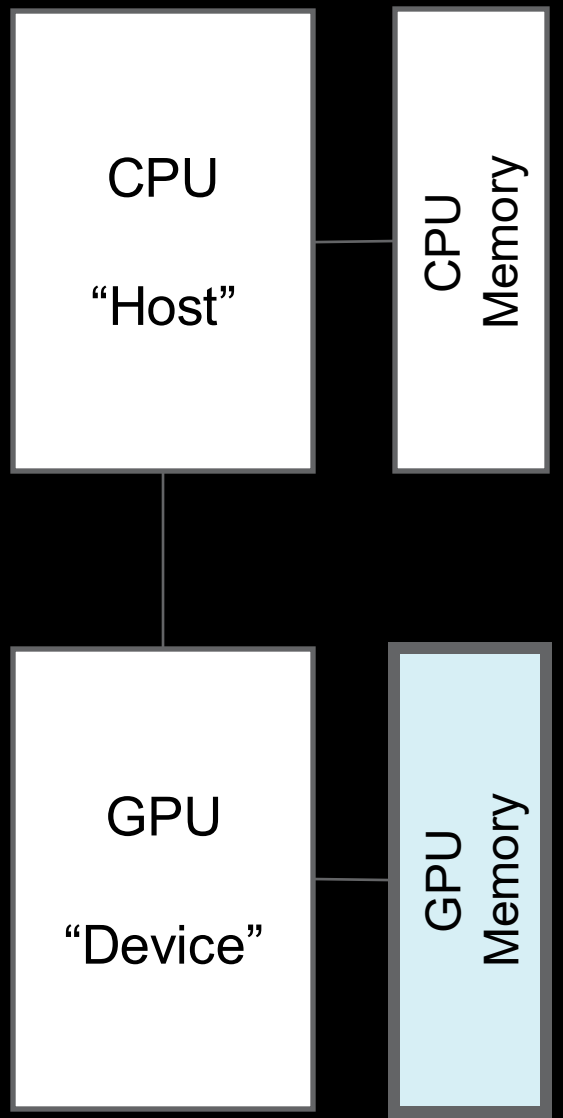
Memory Allocation

```
int *d_A
hipMalloc(&d_A, bytes)
```

HIP provides functions for allocating device memory

```
hipMemcpy(&d_A, &h_A, bytes,
          hipMemcpyHostToDevice)
```

Will copy a total of “bytes” bytes from source “h_A” to the destination “d_A”



Vector Addition

Header file

- hip/hip_runtime.h

```
#include <hip/hip_runtime.h>

#define N 10000
```

Vector Addition

Macros
- #define

GPU code
- __global__ void

```
__global__ void vectorAdd(const float* A, const float* B,
float* C, int N) {

}
```

Vector Addition

CPU code

- CPU memory allocation

```
float *hA = new float[N];
float *hB = new float[N];
float *hC = new float[N];
float *dA, *dB, *dC;
```

Vector Addition

CPU code

- CPU memory allocation
- Array initialization

```
for(int i = 0; i < N; i++){
    hA[i] = 1.0f;
    hB[i] = 2.0f;
}
```

Vector Addition

CPU code

- CPU memory allocation
- Array initialization
- GPU memory allocation

```
hipMalloc((void**)&dA, sizeof(float) * N);
hipMalloc((void**)&dB, sizeof(float) * N);
hipMalloc((void**)&dC, sizeof(float) * N);
```

Vector Addition

CPU code

- CPU memory allocation
- Array initialization
- GPU memory allocation
- Memory Transfer (Host To Device)

```
hipMemcpy(dA, hA, sizeof(float) * N, hipMemcpyHostToDevice);
hipMemcpy(dA, hB, sizeof(float) * N, hipMemcpyHostToDevice);
```

Vector Addition

CPU code

- CPU memory allocation
- Array initialization
- GPU memory allocation
- Memory Transfer (Host To Device)
- Kernel Launch

```
    int blockSize = 256;
int gridSize = (N + blockSize - 1) / blockSize;

vectorAdd<<<gridSize, blockSize>>>(dA, dB, dC, N);
```

Vector Addition

CPU code

- CPU memory allocation
- Array initialization
- GPU memory allocation
- Memory Transfer (Host To Device)
- Kernel Launch
- Memory transfer (Device to Host)

```
hipMemcpy(hC, dC, bytes, hipMemcpyDeviceToHost)
```

Vector Addition

CPU code

- CPU memory allocation
- Array initialization
- GPU memory allocation
- Memory Transfer (Host To Device)
- Kernel Launch
- Memory transfer (Device to Host)
- Memory deallocation

```
hipFree(dA); hipFree(dB); hipFree(dC);
free(A); free(B); free(C);
```

Vector Addition

Header file
- hip/hip_runtime.h

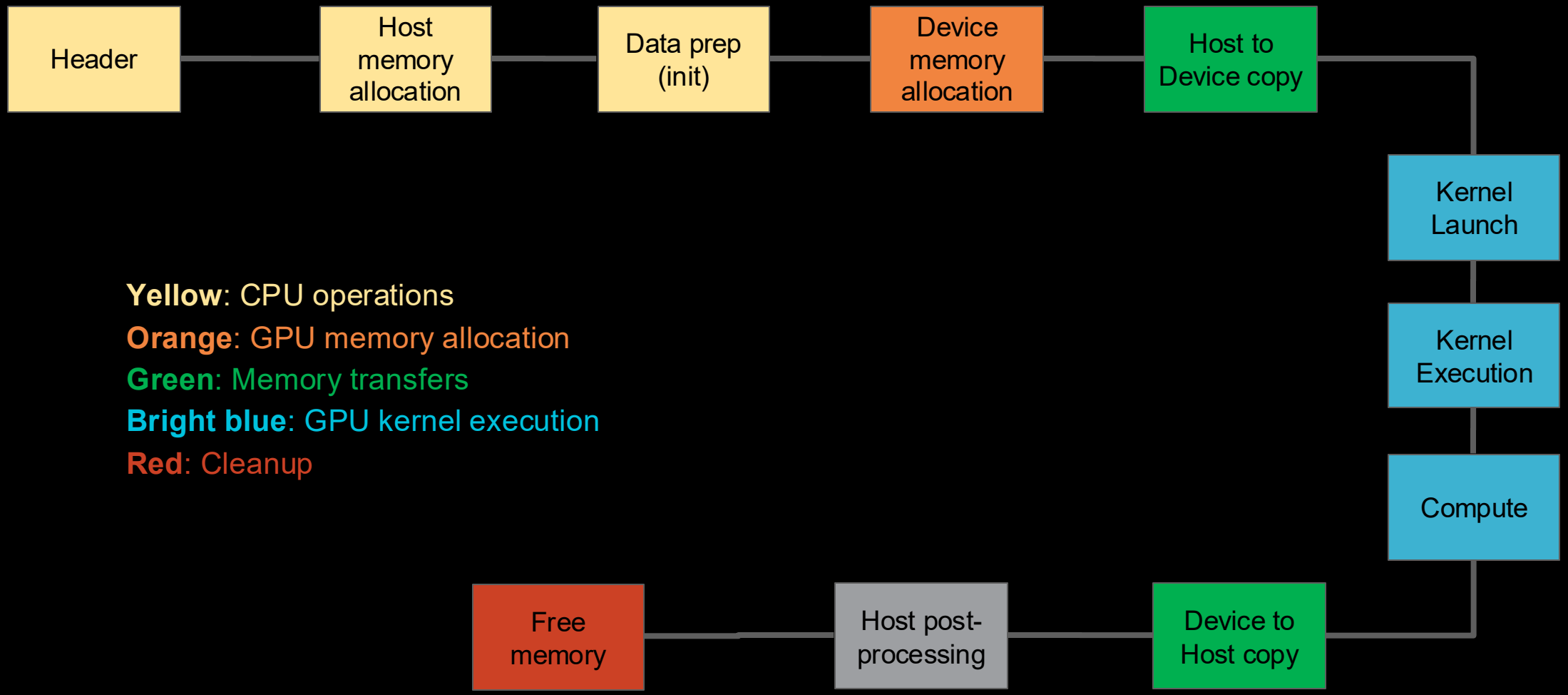
Macros
- #define

GPU code
- __global__ void

CPU code
- CPU memory allocation
- Array initialization
- GPU memory allocation
- Memory Transfer (Host To Device)
- Kernel Launch
- Memory transfer (Device to Host)
- Memory deallocation

```
#include <hip/hip_runtime.h>
#include <iostream>
#include <cmath>

#define HIP_CHECK(cmd) do { hipError_t e = cmd; if(e != hipSuccess){ \
    std::cerr << "HIP error: " << hipGetErrorString(e) << std::endl; exit(EXIT_FAILURE);} } while(0)

__global__ void vectorAdd(const float* A, const float* B, float* C, int N) {
    int gid = blockIdx.x * blockDim.x + threadIdx.x;
    if (gid < N)
        C[gid] = A[gid] + B[gid];
}

int main() {
    const int N = 1 << 20; // 1M elements
    const size_t bytes = N * sizeof(float);

    float *hA = new float[N], *hB = new float[N], *hC = new float[N];
    float *dA, *dB, *dC;
    for(int i = 0; i < N; i++){
        hA[i] = 1.0f;
        hB[i] = 2.0f;
    }

    HIP_CHECK(hipMalloc(&dA, bytes));
    HIP_CHECK(hipMalloc(&dB, bytes));
    HIP_CHECK(hipMalloc(&dC, bytes));

    HIP_CHECK(hipMemcpy(dA, hA, bytes, hipMemcpyHostToDevice));
    HIP_CHECK(hipMemcpy(dB, hB, bytes, hipMemcpyHostToDevice));

    int blockSize = 256;
    int gridSize = (N + blockSize - 1) / blockSize;
    vectorAdd<<<gridSize, blockSize>>>(dA, dB, dC, N);
    HIP_CHECK(hipDeviceSynchronize());

    HIP_CHECK(hipMemcpy(hC, dC, bytes, hipMemcpyDeviceToHost));

    bool correct = true;
    for (int i = 0; i < N; ++i) {
        float expected = hA[i] + hB[i];
        if (fabs(hC[i] - expected) > 1e-5) { correct = false; break; }
    }
    std::cout << (correct ? "Results verified - CORRECT!\n"
                          : "Verification FAILED!\n");

    HIP_CHECK(hipFree(dA)); HIP_CHECK(hipFree(dB)); HIP_CHECK(hipFree(dC));
    free(hA); free(hB); free(hC);
    return 0;
}
```

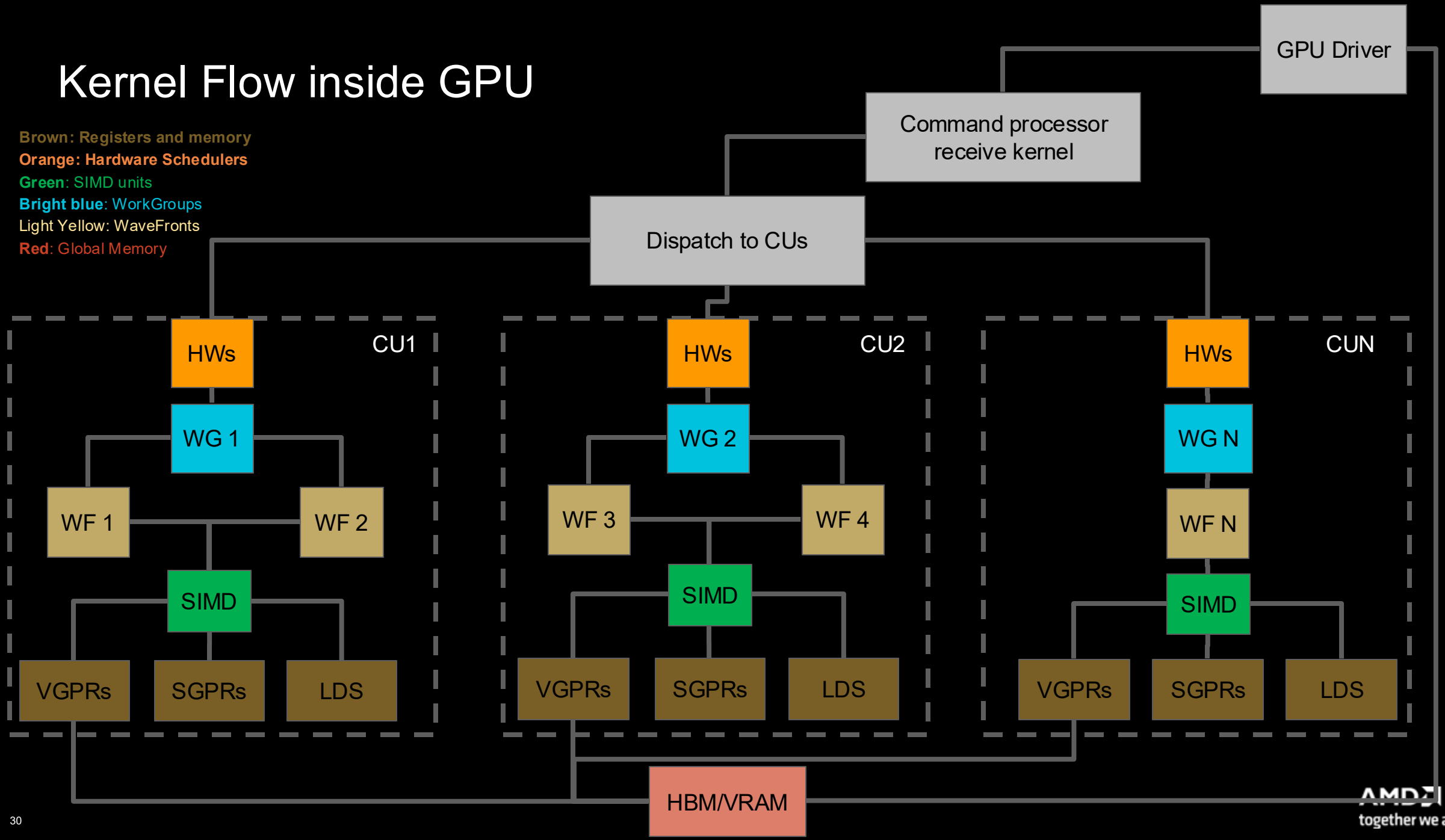
Kernel Flow



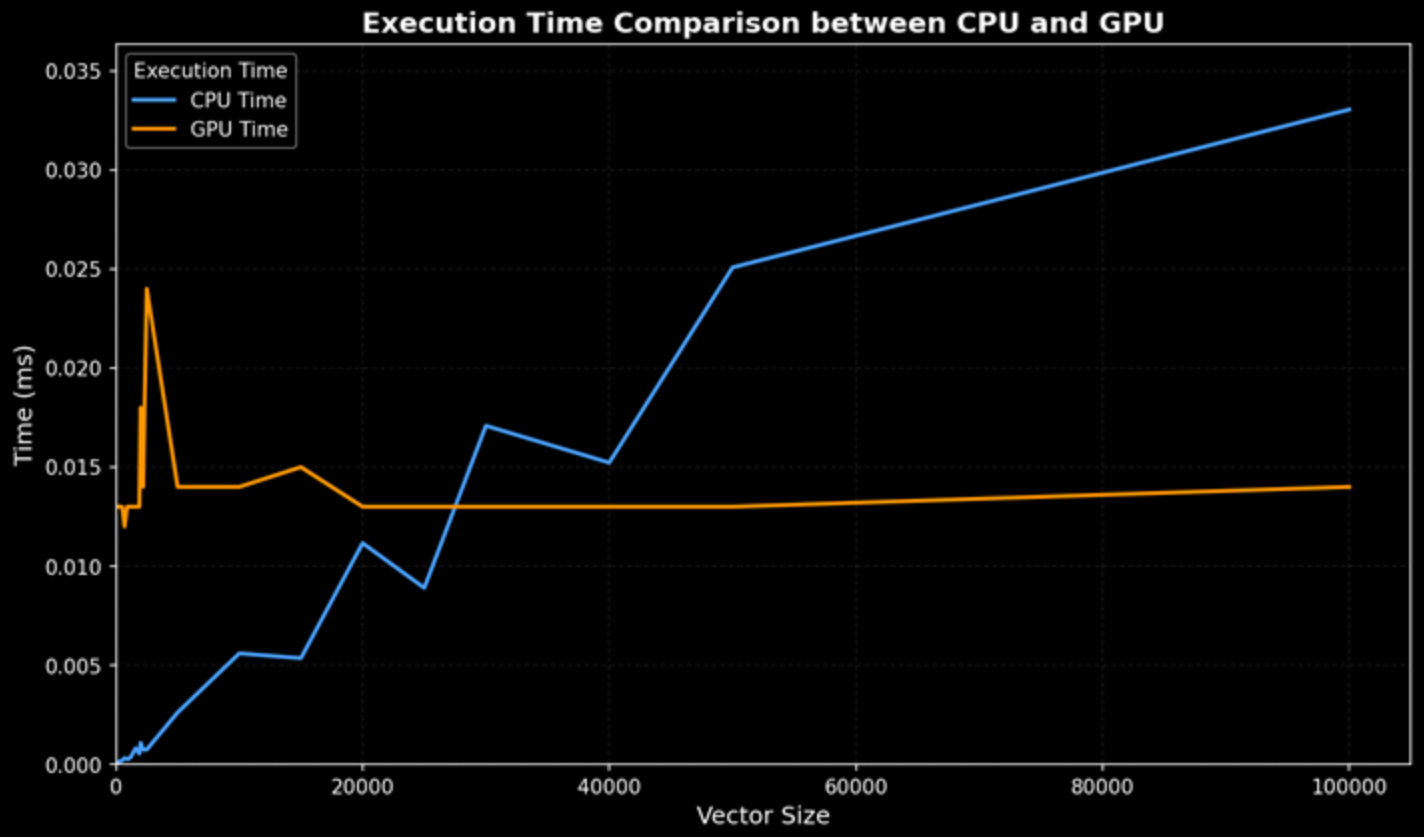
Yellow: CPU operations
Orange: GPU memory allocation
Green: Memory transfers
Bright blue: GPU kernel execution
Red: Cleanup

Kernel Flow inside GPU

Brown: Registers and memory
Orange: Hardware Schedulers
Green: SIMD units
Bright blue: WorkGroups
Light Yellow: WaveFronts
Red: Global Memory

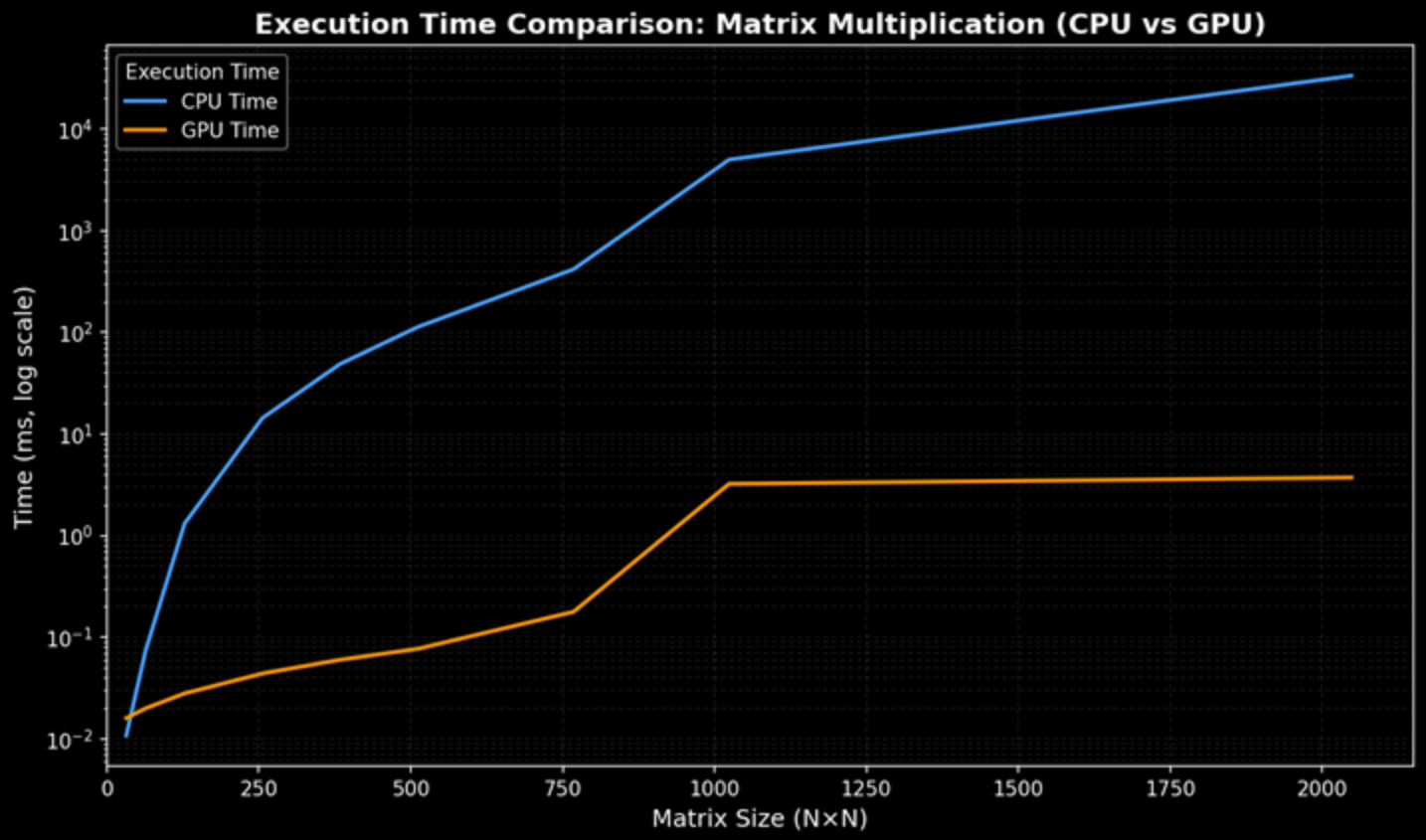


Vector Addition



Code is previous slides

Matrix Multiplication




```
#include <hip/hip_runtime.h>

#define BLOCK_SIZE 16
#define N 256

__global__ void gpu_matrix_multiplication(int
*a,int *b, int *c, int n){
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int sum = 0;
  if( col < n && row < n) {
    for(int i = 0; i < n; i++) {
      sum += a[row * n + i] * b[i * n + col];
    }
    c[row * n + col] = sum;
  }
}

int main(){



  dim3 threadsPerBlock (BLOCK_SIZE, BLOCK_SIZE);
  int n_blocks = ceil(N/BLOCK_SIZE);

  dim3 blocksPerGrid (n_blocks, n_blocks);

  gpu_matrix_multiplication<<<blocksPerGrid,
threadsPerBlock>>>(d_a, d_b, d_c, N);
  hipDeviceSynchronize();

  hipMemcpy(h_c, d_c, sizeof(int)*N*N,
hipMemcpyDeviceToHost);
```

HIP

HIP Kernel Optimization and Hands on Demo

- Explain Two Optimization Techniques
  - Memory Coalescing
  - Loop Unrolling
- Walk through two demos

Function-Type Qualifiers

__global__

Declares a function as a kernel

Called from the host CPU

Executed on the GPU

Always return void

```
__global__ void vector_add(float *out, float *a,
float *b, int n) {

    int id = blockDim.x * blockIdx.x + threadIdx.x;
    if (id < N){
        out[id] = a[id] + b[id];
    }
}
```

__device__

Executes on device (GPU)

Callable from the device only

```
__device__ int get_global_id(void){
    return blockDim.x * blockIdx.x + threadIdx.x;
}

__global__ void myKernel(int *a){
    int id = get_global_id();
}
```

__host__

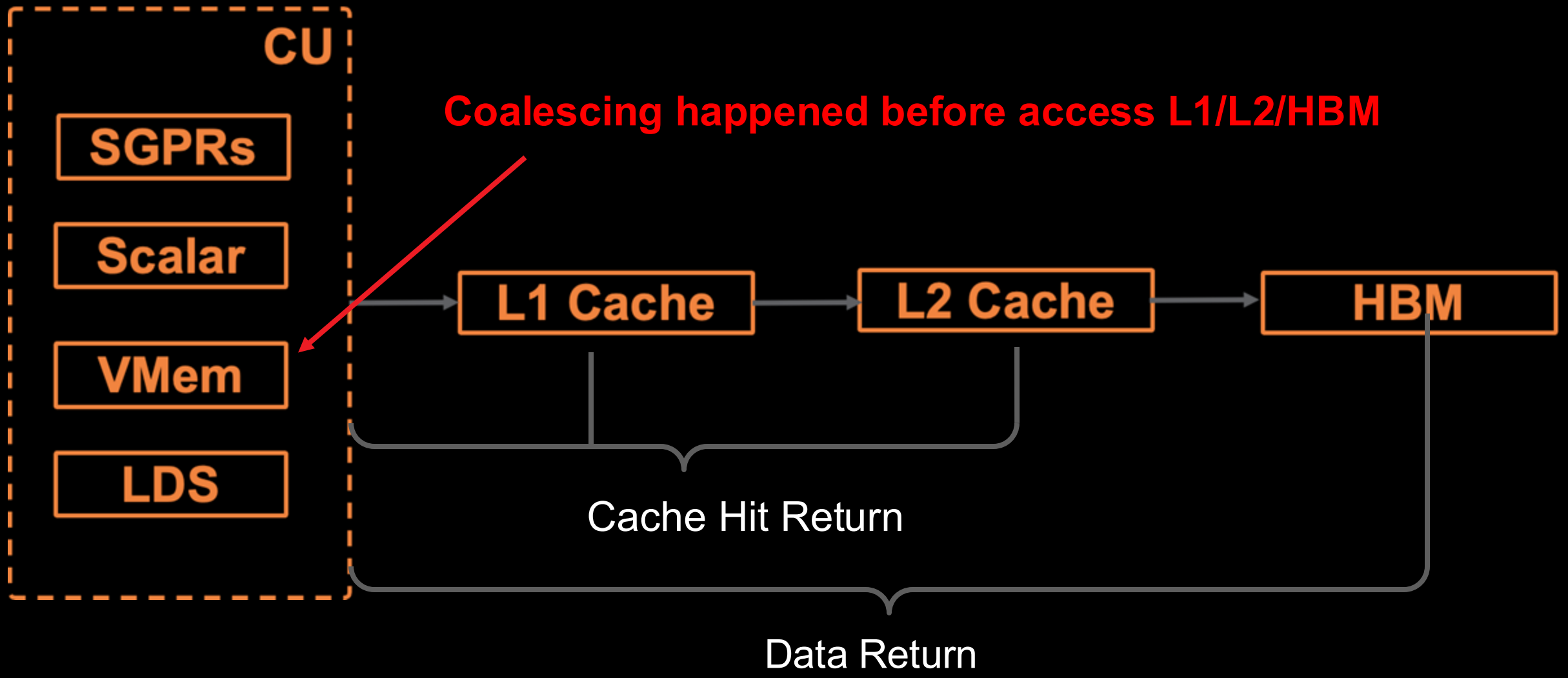
Executed on the host

Callable from the host

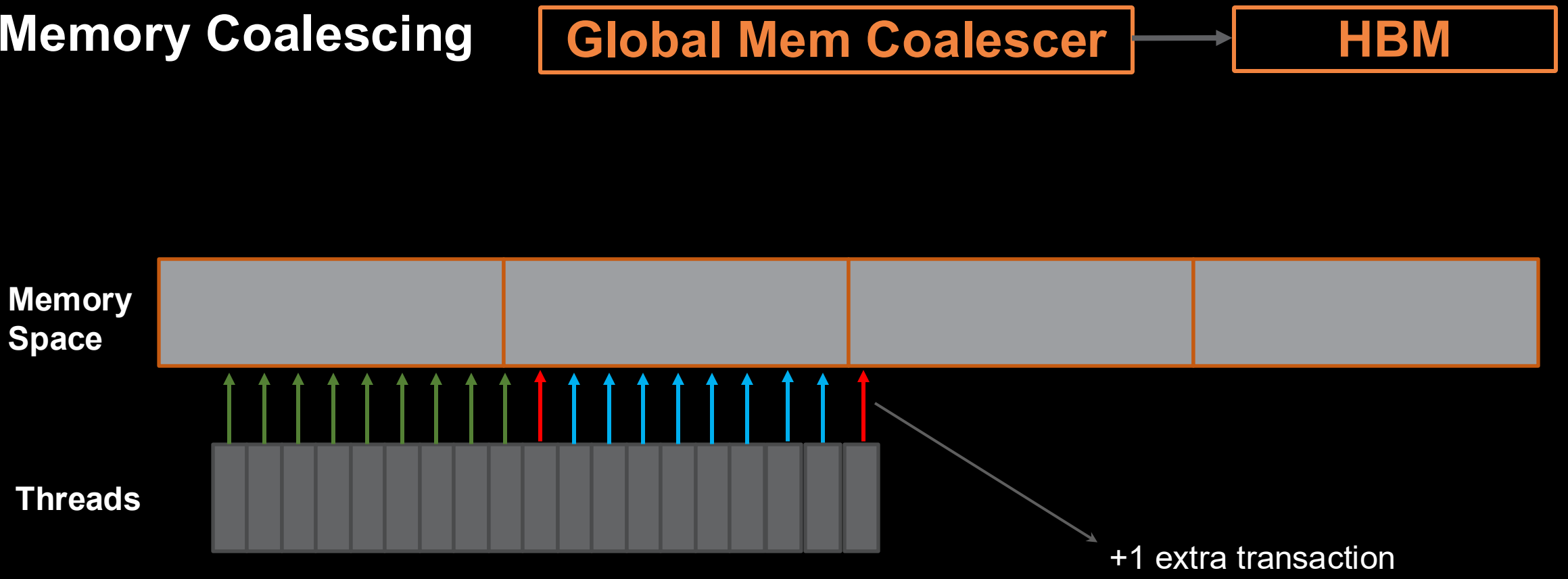
Similar to not having an identifier

```
__host__ int add_numbers(int a, int b){
  return a+b
}

int add_numbers(int a, int b){
  return a+b
}
```

Memory Coalescing (memory transactions)



Memory Coalescing

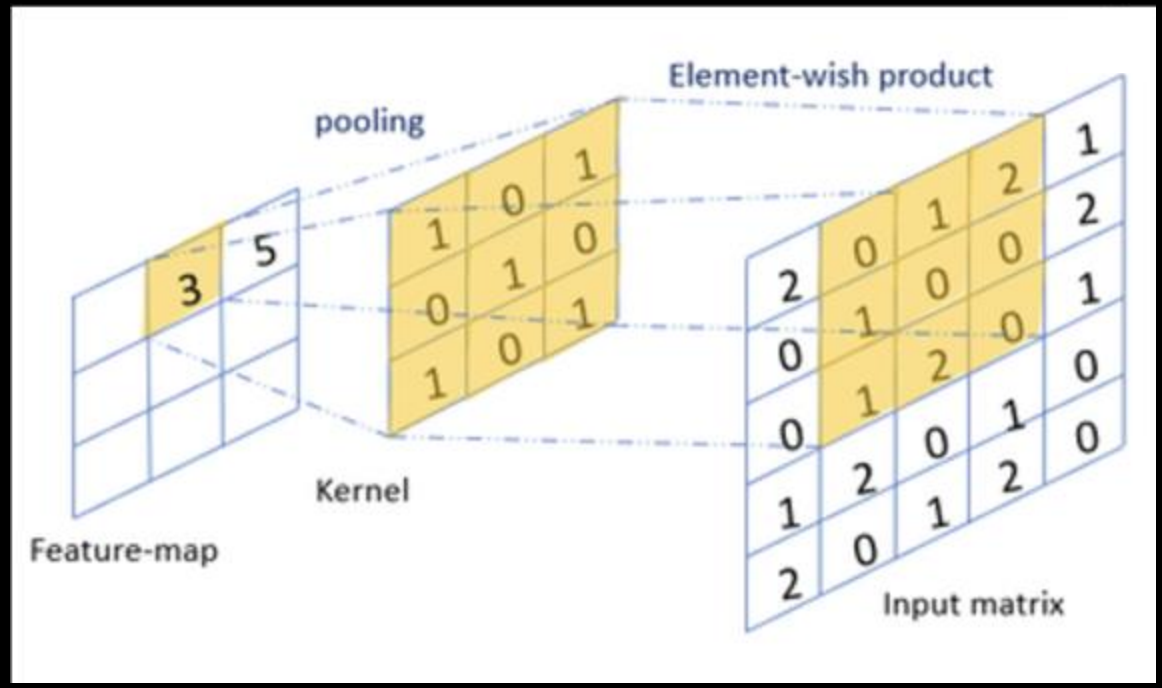


Demo of Memory Coalescing

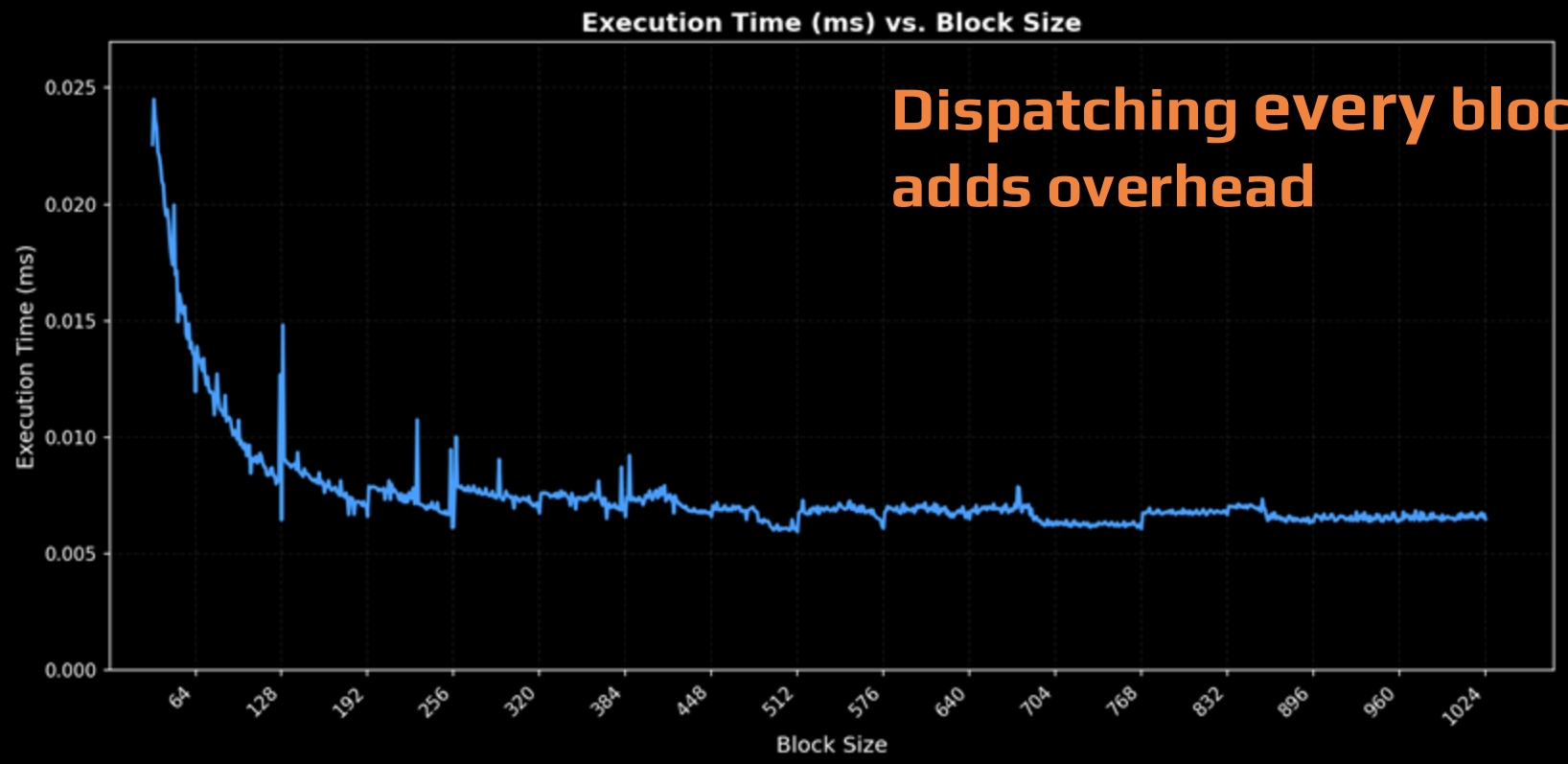
https://staging.amddevcloud.com/ui

Unrolling (for Conv3×3 Example)

```
__global__ void conv3x3_naive_no_unroll(
      const float* __restrict__ input,
      const float* __restrict__ kernel,
      float* __restrict__ output,
      int width,
      int height
) {
     for (int ky = -1; ky <= 1; ++ky)
        for (int kx = -1; kx <= 1; ++kx)
            acc += input[...] * kernel[...];
     ...

}
```



Unrolling (for Conv3×3 Example)

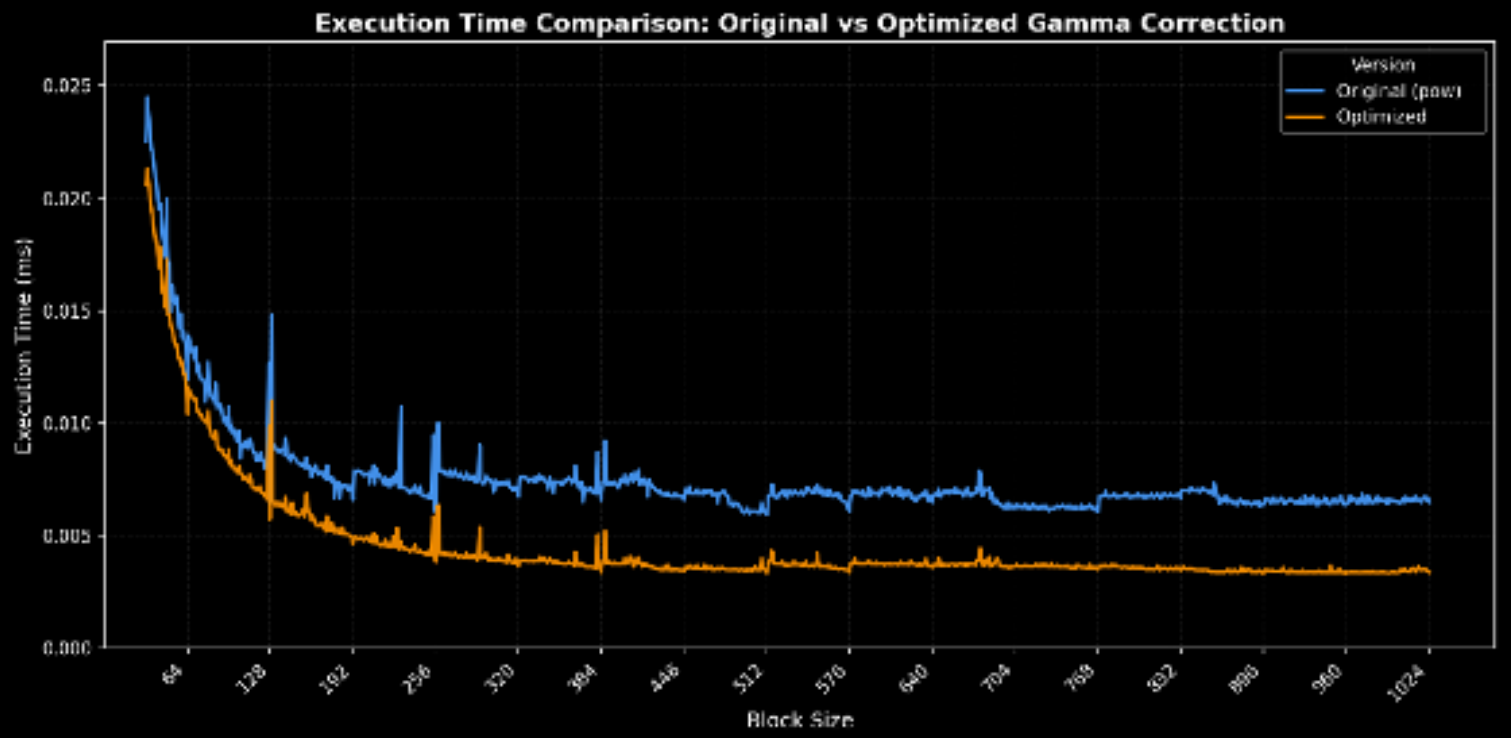


Dispatching every block adds overhead

Reduce the number of threads!
Let each thread to more work!

Unrolling (for Conv3×3 Example)

```
__global__ void conv3x3_unrolled(
      const float* __restrict__ input,
      const float* __restrict__ kernel,
      float* __restrict__ output,
      int width,
      int height
{
     ...
     acc = fmaf(in[row_above - 1], k[0], acc);
     acc = fmaf(in[row_above + 0], k[1], acc);
     acc = fmaf(in[row_above + 1], k[2], acc);

     acc = fmaf(in[center - 1], k[3], acc);
     acc = fmaf(in[center + 0], k[4], acc);
     acc = fmaf(in[center + 1], k[5], acc);

     acc = fmaf(in[row_below - 1], k[6], acc);
     acc = fmaf(in[row_below + 0], k[7], acc);
     acc = fmaf(in[row_below + 1], k[8], acc);
     ...
}
```



Demo of Loop Unrolling

https://staging.amddevcloud.com/ui

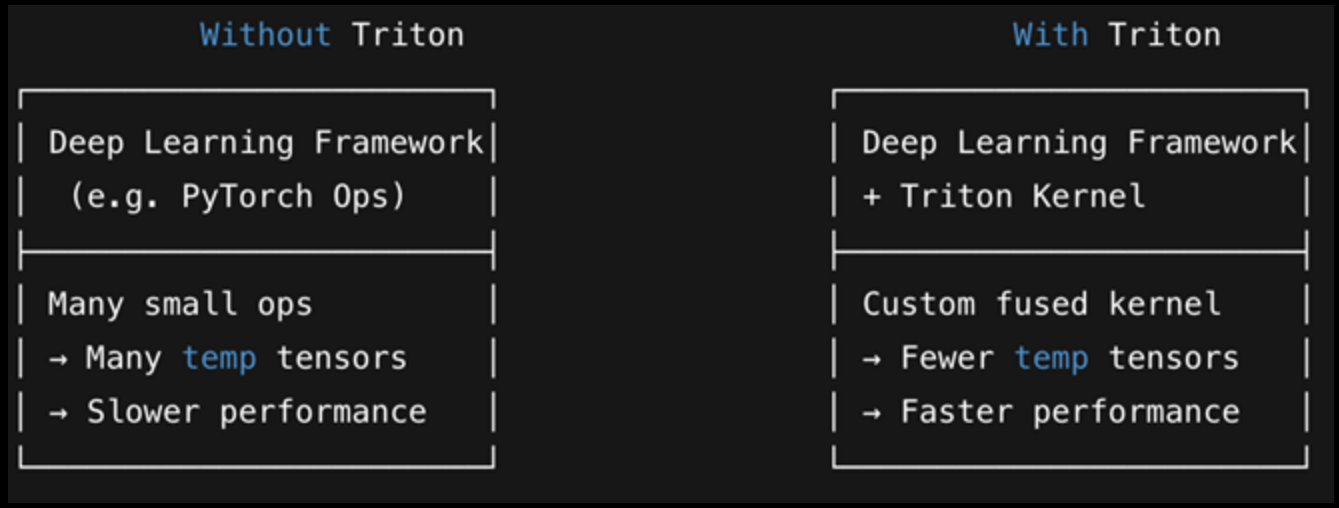
Triton

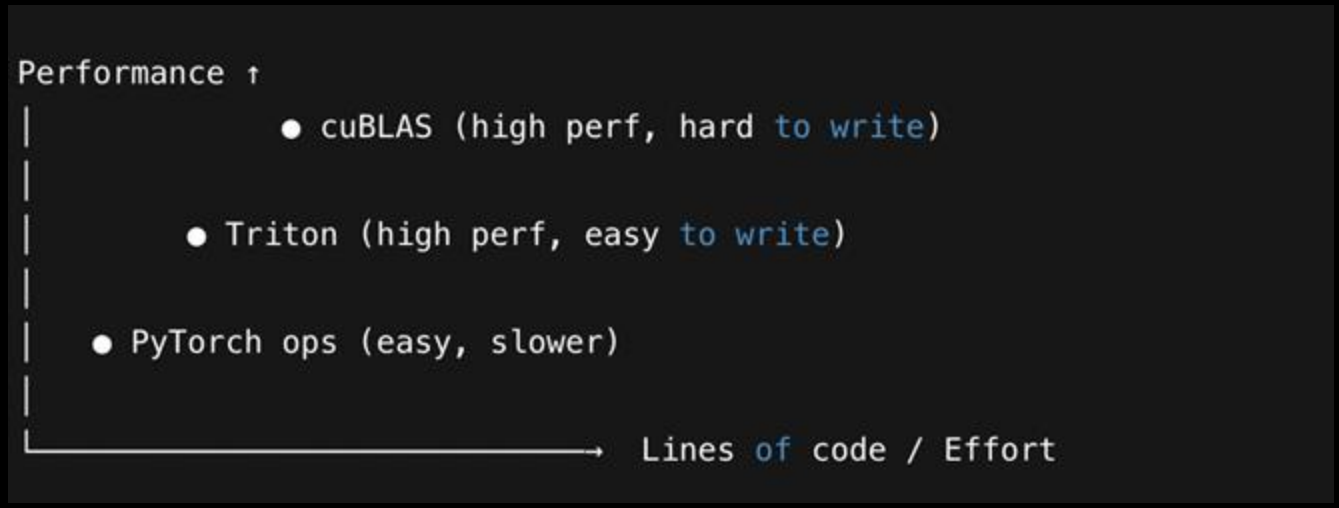
Introducing Triton

Problem: Deep learning frameworks (like PyTorch, TensorFlow) combine lots of operators → create many **temporary tensors** → wastes time & memory.

Old solutions: Writing **custom GPU kernels** (specialized code) solves this, but it's **hard and tedious**.

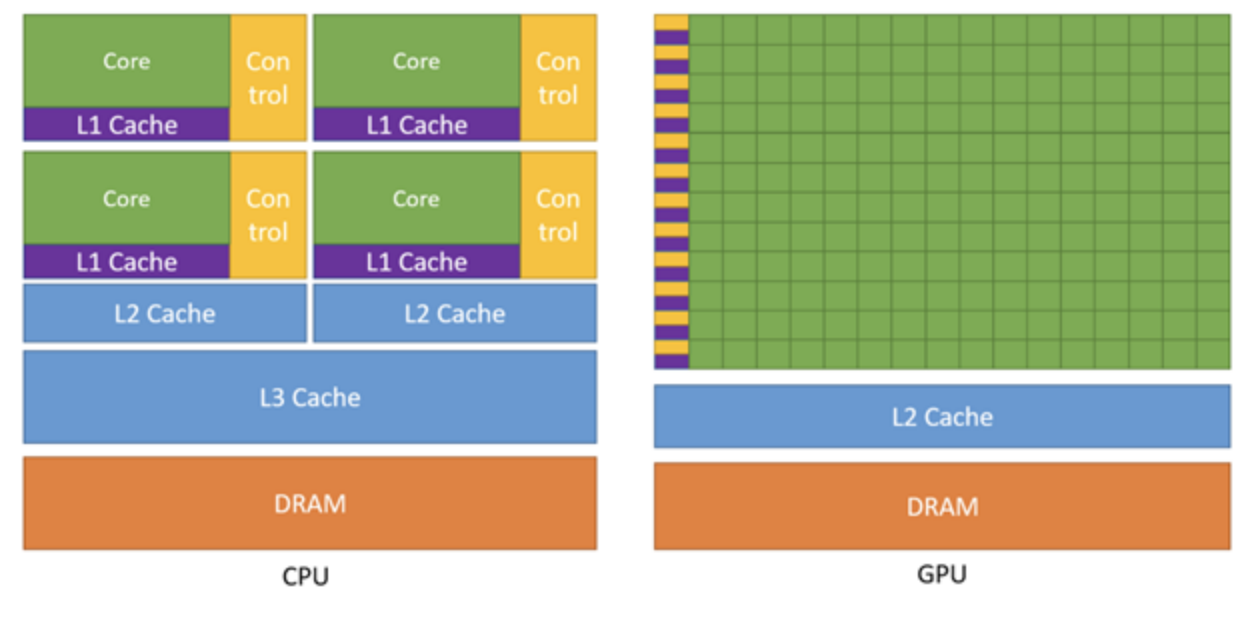
Triton: A language/compiler that makes it easy to write high-performance GPU code (as good as NVIDIA's cuBLAS library) in a few lines.

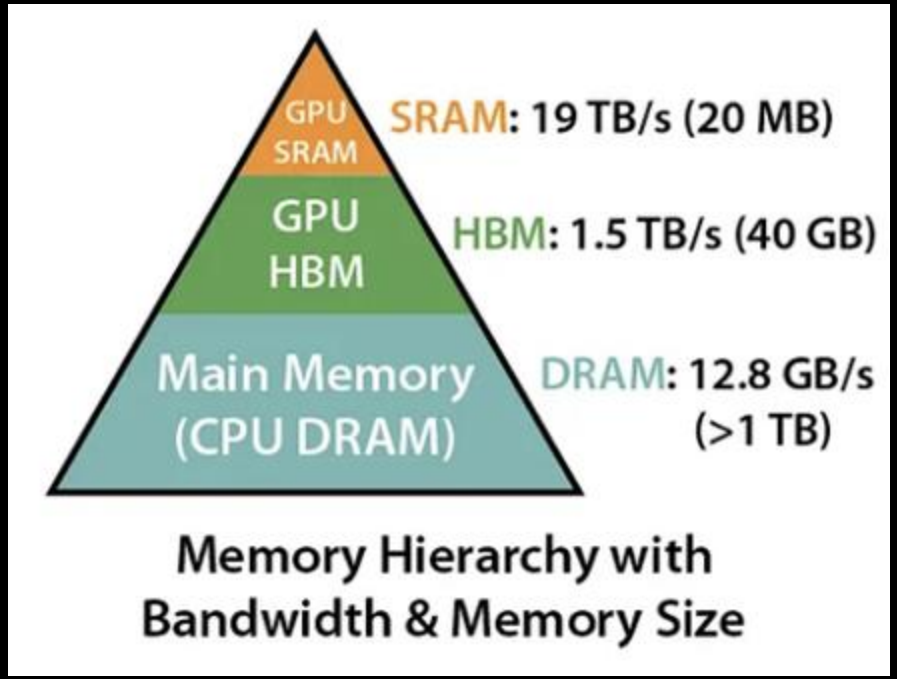




Why is memory read/write so important?

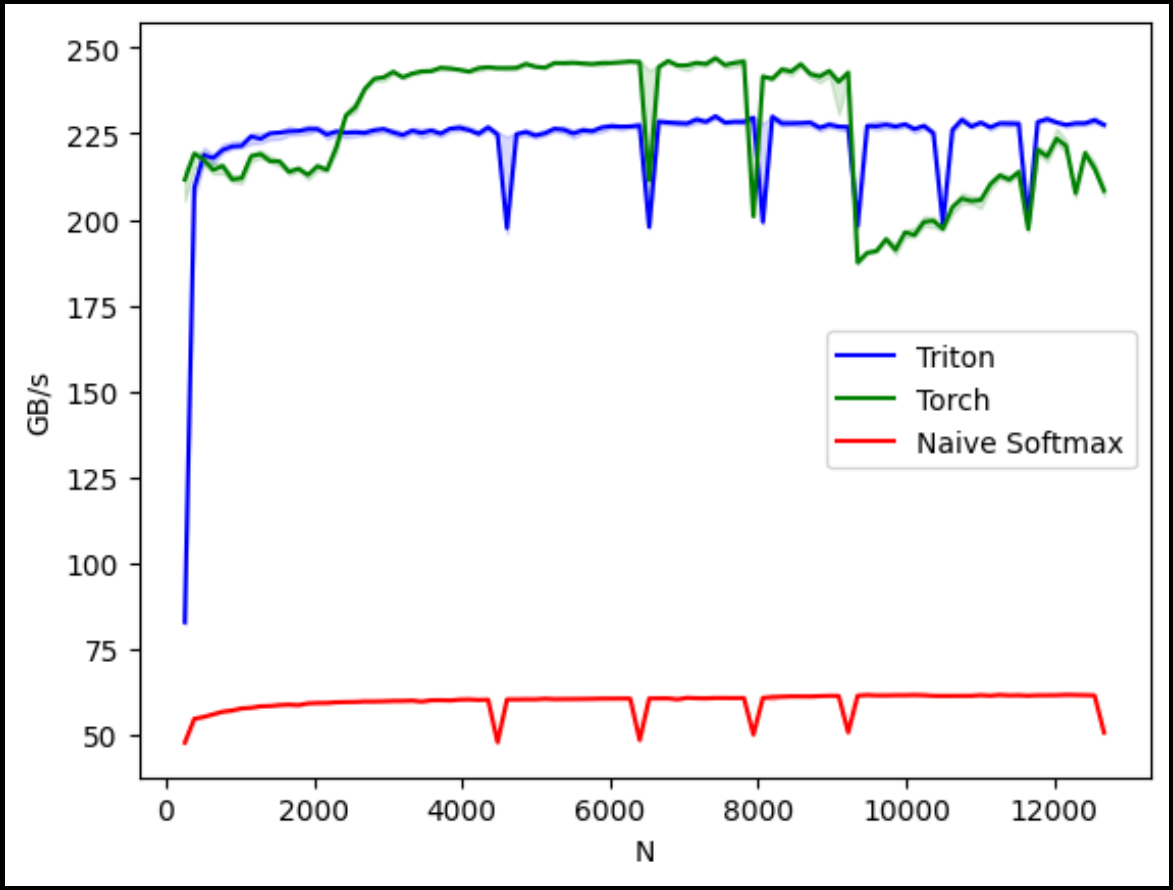
- Open AI's GPT-3/4 around half a billion dollars so far in GPU costs
- Facebook's llama costs are approximated to be a couple of million dollars for one training run





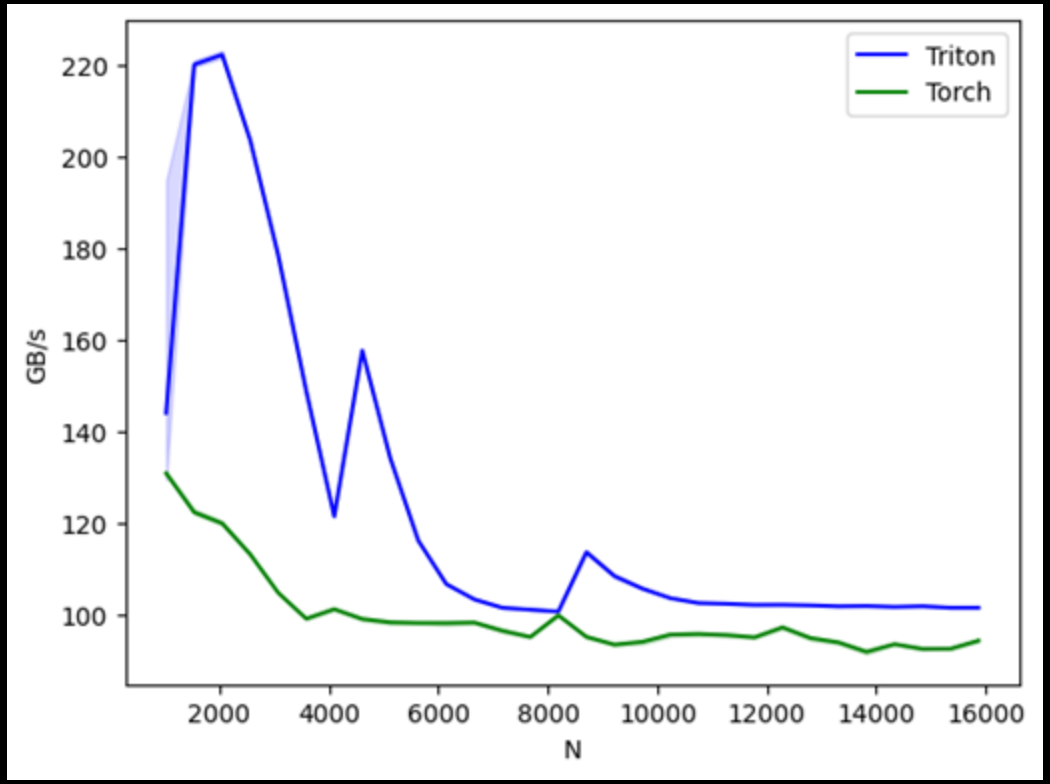
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Demo of softmax



https://drive.google.com/drive/folders/1w70-mXQNv8tULuMMXbryH7Ps_ZQAcPkL?usp=share_link

Demo of layernorm



https://drive.google.com/drive/folders/1w70-mXQNv8tULuMMXbryH7Ps_ZQAcPkL?usp=share_link

Roofline Model

Arithmetic Intensity $I = W/Q$

GPU Hardware

- β Memory BW
  - 256 GB/s
- π Compute TP
  - 10 TFLOPS

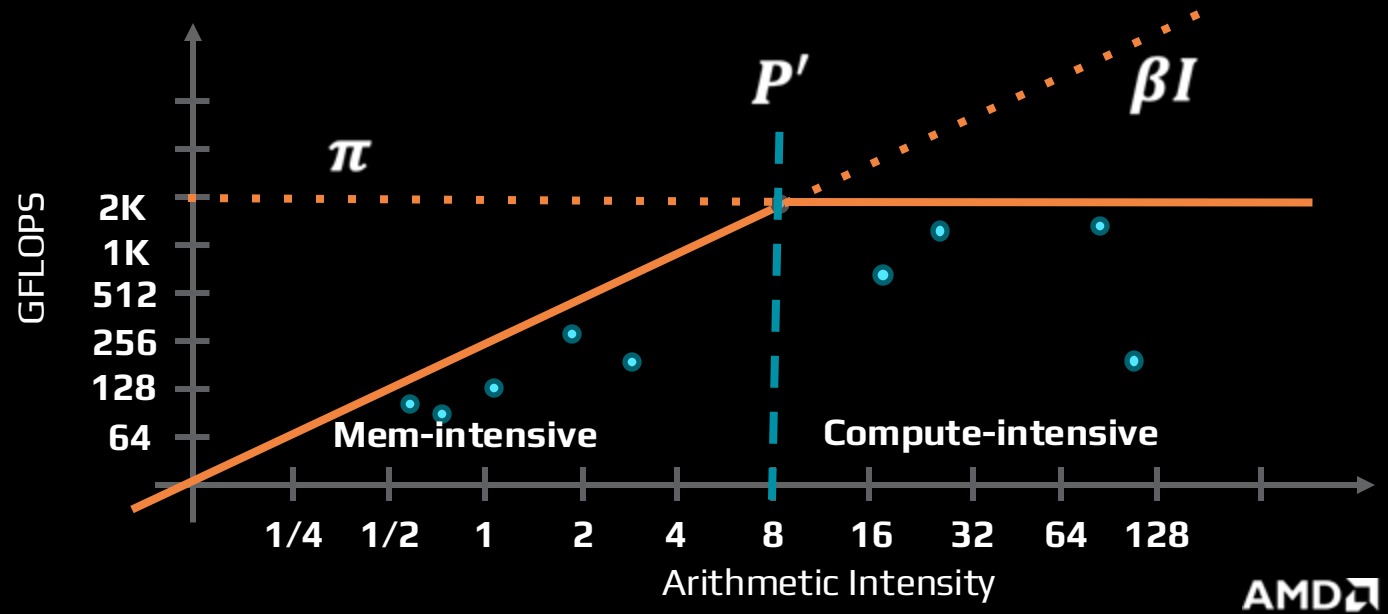
Roofline

$$P = min\begin{cases}\pi \\ \beta I\end{cases}$$

GPU AI $P' = \pi/\beta$

Memory Itensive $I_{wl} < I_{GPU}$

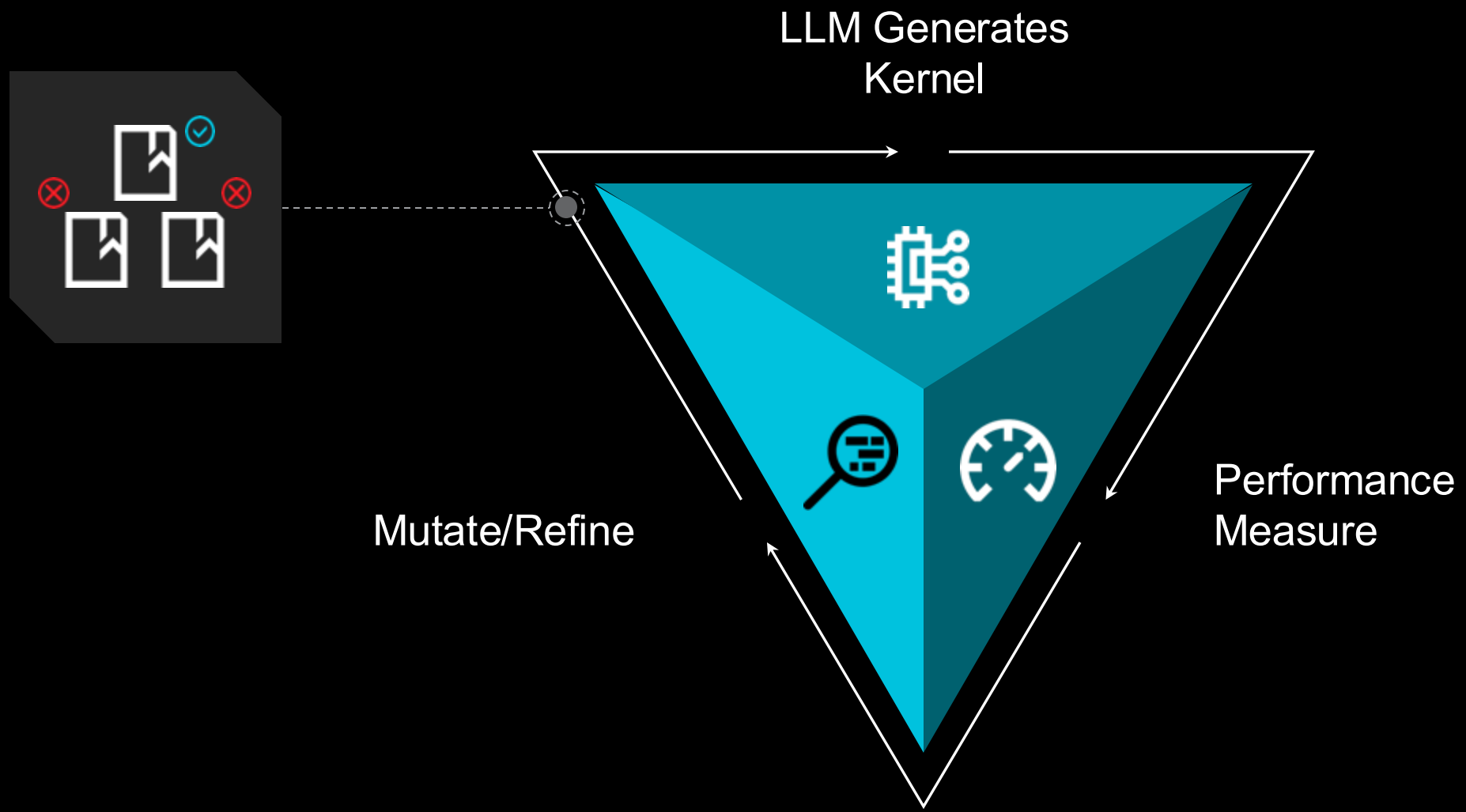
Compute Intensive $I_{wl} > I_{GPU}$



Agentic Framework

The Kernel Hacker’s Grind

Tinder for Kernels



“In the next 3 to 6 months, AI is writing 90% of the code, and in 12 months, nearly all code may be generated by AI.”

Dario Amodei

82% of developers use AI coding tools daily or weekly

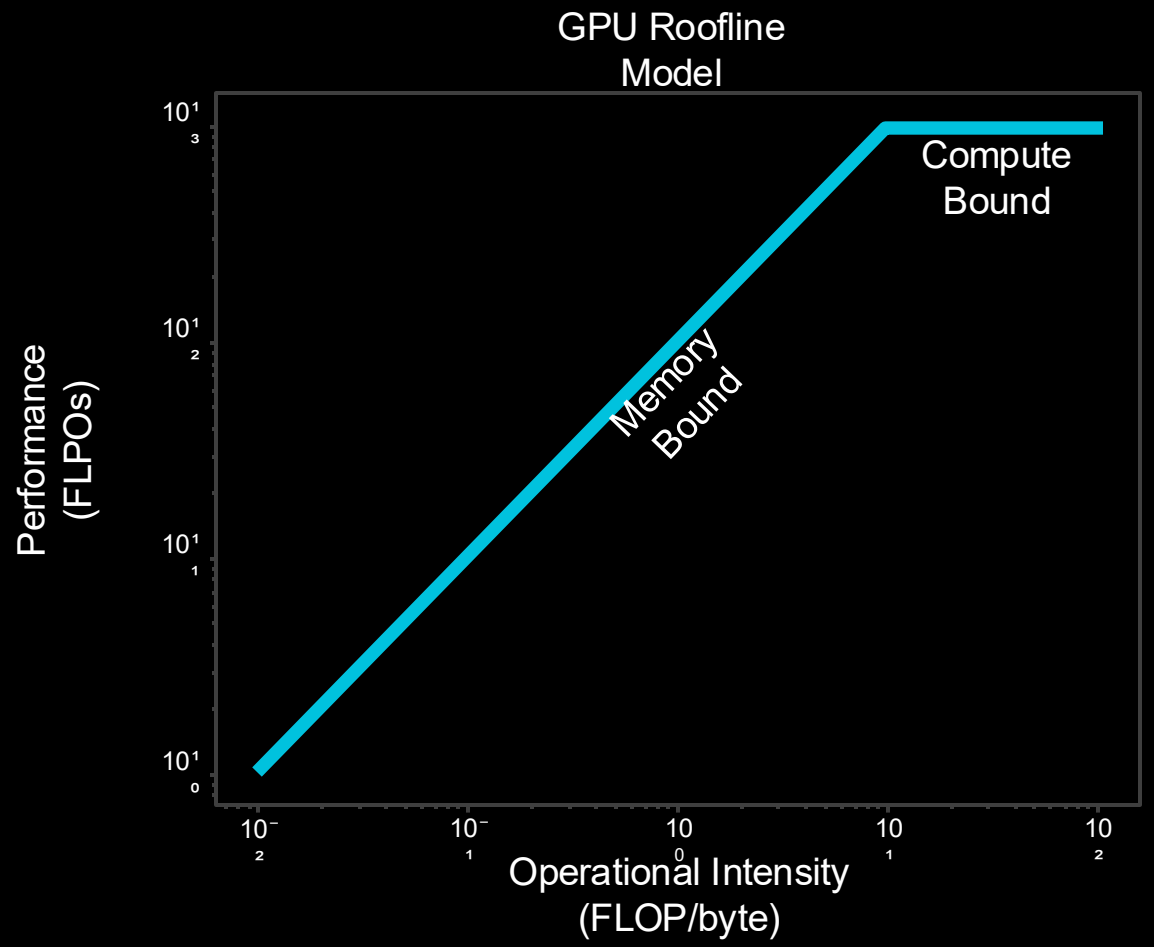
https://www.qodo.ai

Objective

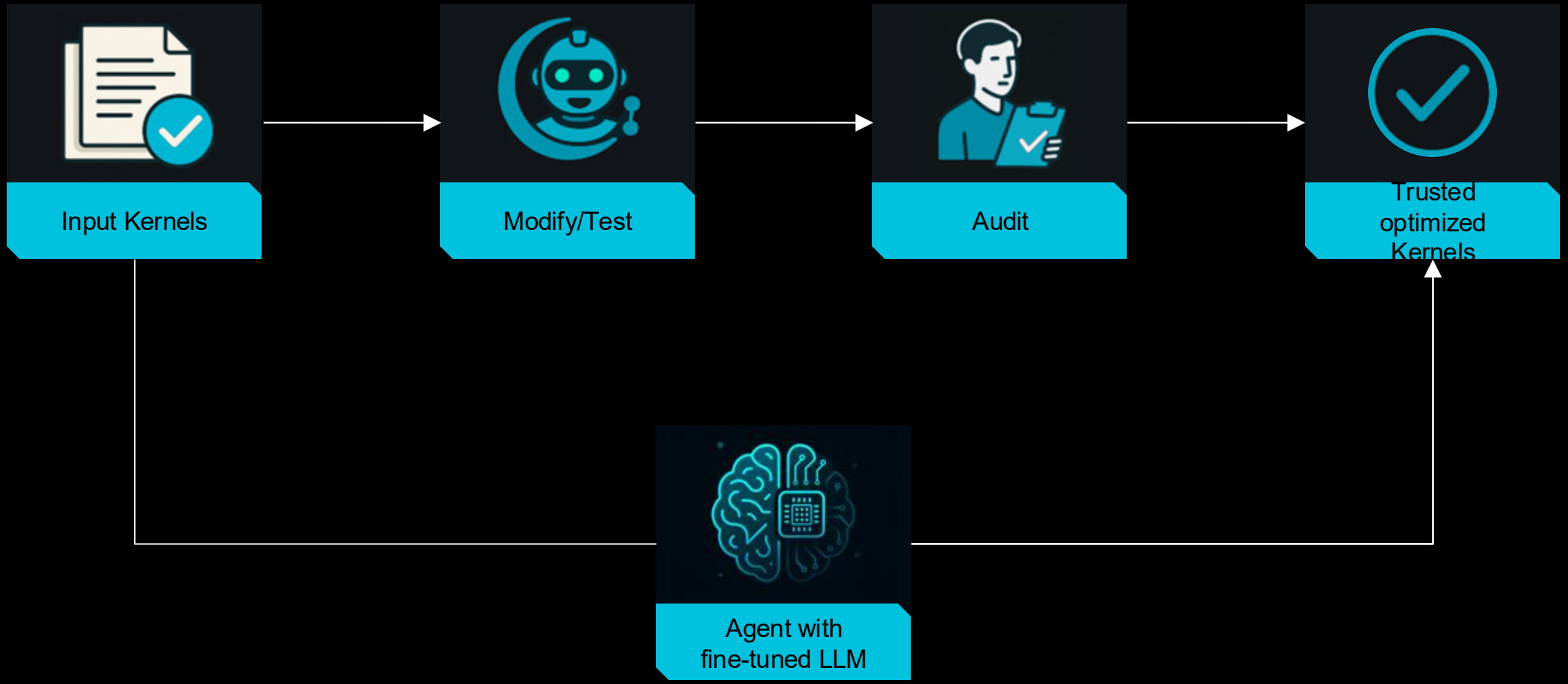
AGIKIT

AMD Generative Intelligence for Kernel Improvement & Tuning

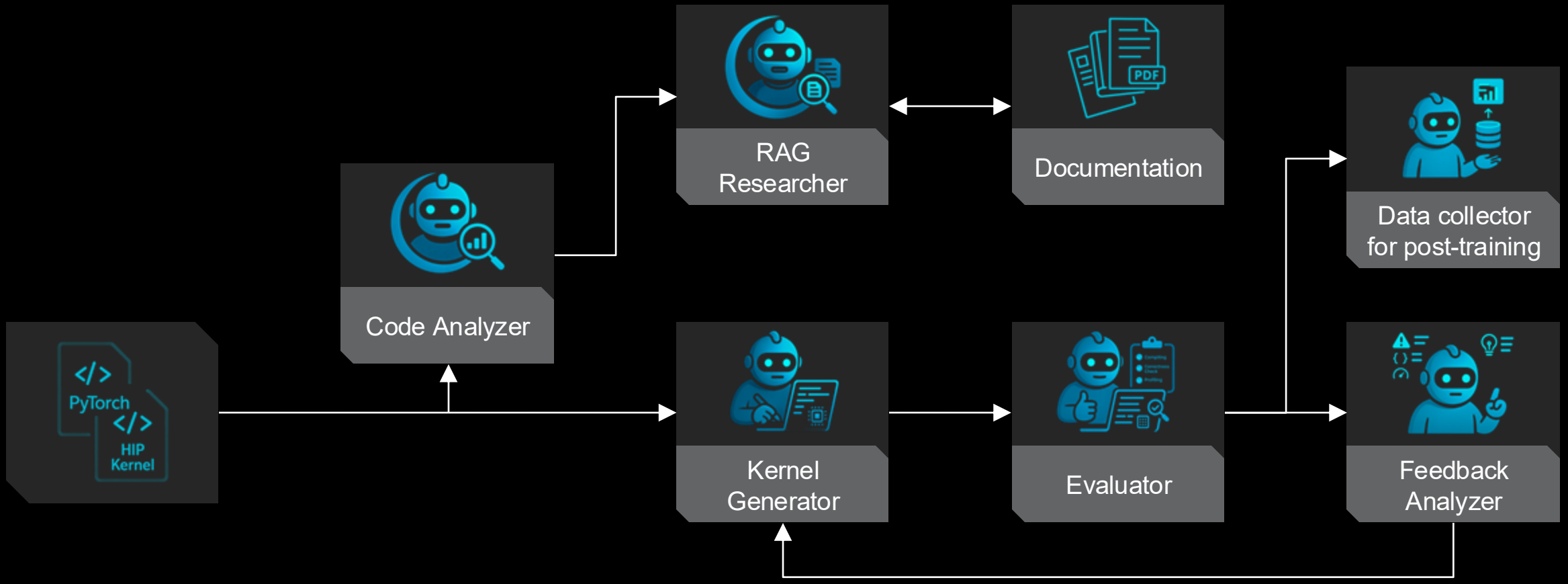
HIP KERNEL



Ultimate Agent

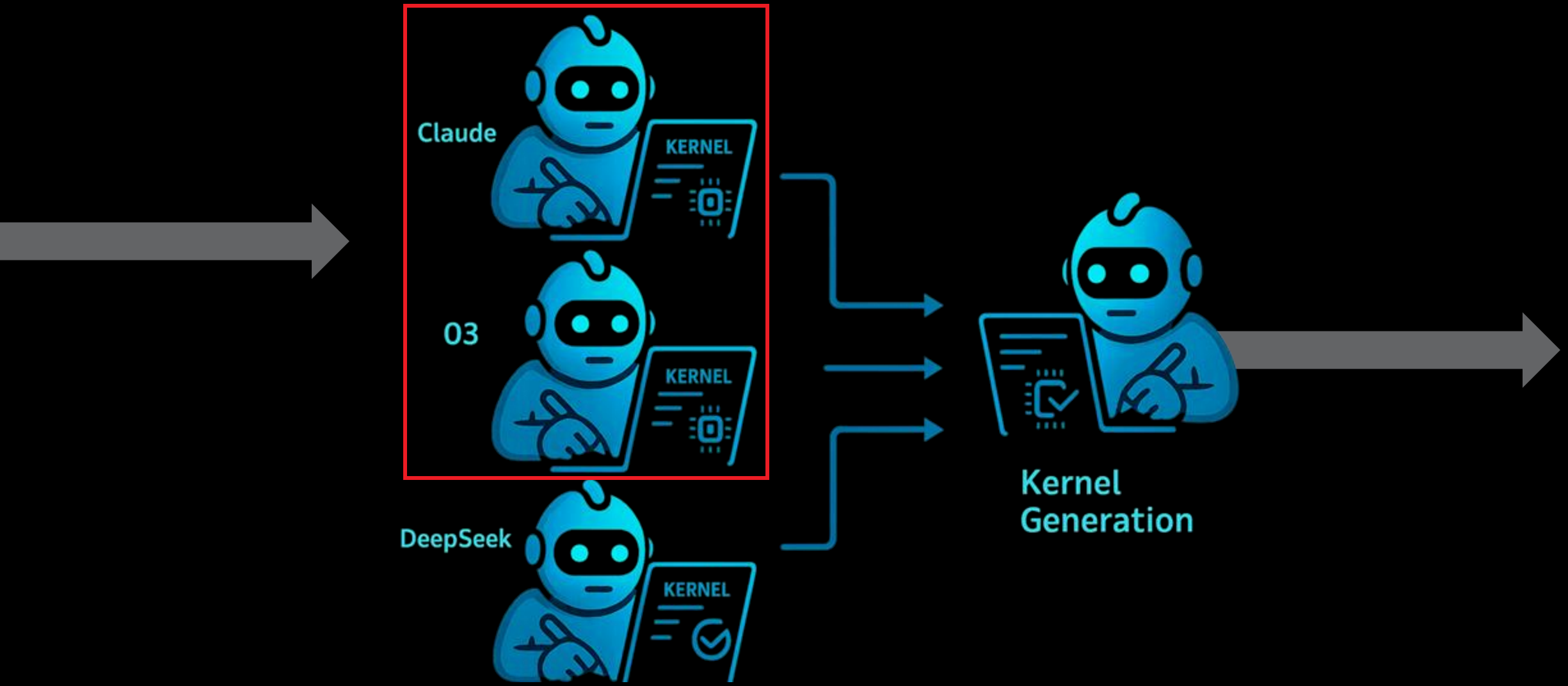


AGIKIT-Agentic Flow



An intelligent loop where code, knowledge, and feedback power continuous kernel innovation.

AGIKIT-Agentic Flow Can Evolve!

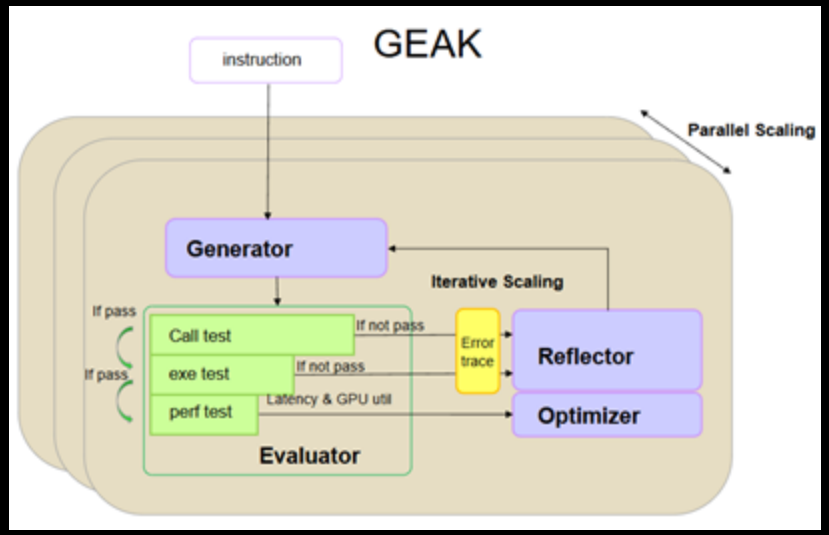


AGIKIT-Agentic KernelBench

```python
class Model(nn.Module):
    """
    Performs 3D tensor-matrix multiplication.
    """
    def __init__(self):
        super(Model, self).__init__()

    def forward(self, A, B):
        """
        Performs 3D tensor-matrix multiplication.

        Args:
            A (torch.Tensor): Input 3D tensor of shape (N, M, K).
            B (torch.Tensor): Input matrix of shape (K, L).

        Returns:
            torch.Tensor: Output tensor of shape (N, M, L),
            resulting from the multiplication of A and B along the last dimension of A.
        """
        return torch.matmul(A, B)

N = 16
M = 1024
K = 2048
L = 768

def get_inputs():
    A = torch.randn(N, M, K)
    B = torch.randn(K, L)
    return [A, B]
```

```cpp
// Allocate device memory
float *d_A, *d_B, *d_C;
size_t size_A = N * M * K * sizeof(float);
size_t size_B = K * L * sizeof(float);
size_t size_C = N * M * L * sizeof(float);

hipError_t err;
err = hipMalloc(&d_A, size_A);
if (err != hipSuccess) {…
}

err = hipMalloc(&d_B, size_B);
if (err != hipSuccess) {…
}

err = hipMalloc(&d_C, size_C);
if (err != hipSuccess) {…
}

// Initialize with test data
float* h_A = new float[N * M * K];
float* h_B = new float[K * L];

// Simple initialization
for (int i = 0; i < N * M * K; i++) {
    h_A[i] = (i % 100) * 0.01f;
}
for (int i = 0; i < K * L; i++) {
    h_B[i] = (i % 100) * 0.01f;
}
```

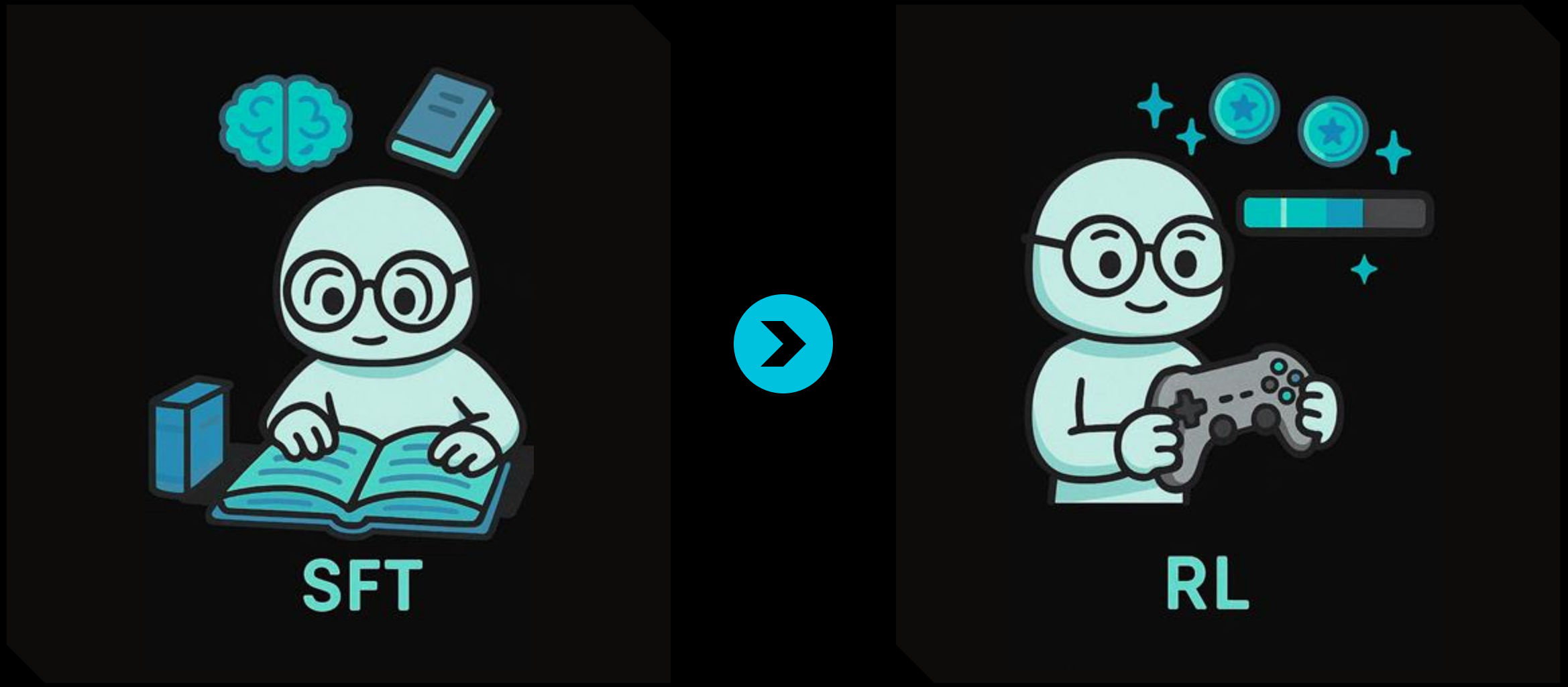
Demo of Hip2Hip using Agentic AI



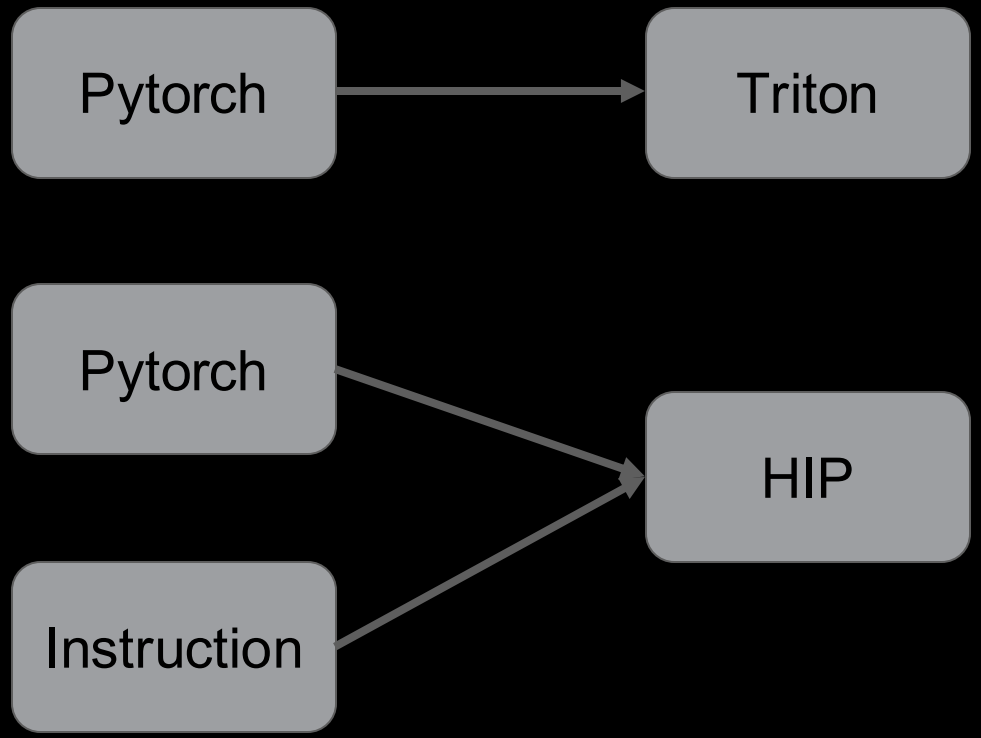
https://staging.amddevcloud.com/ui

https://github.com/AMD-AGI/GEAK-agent

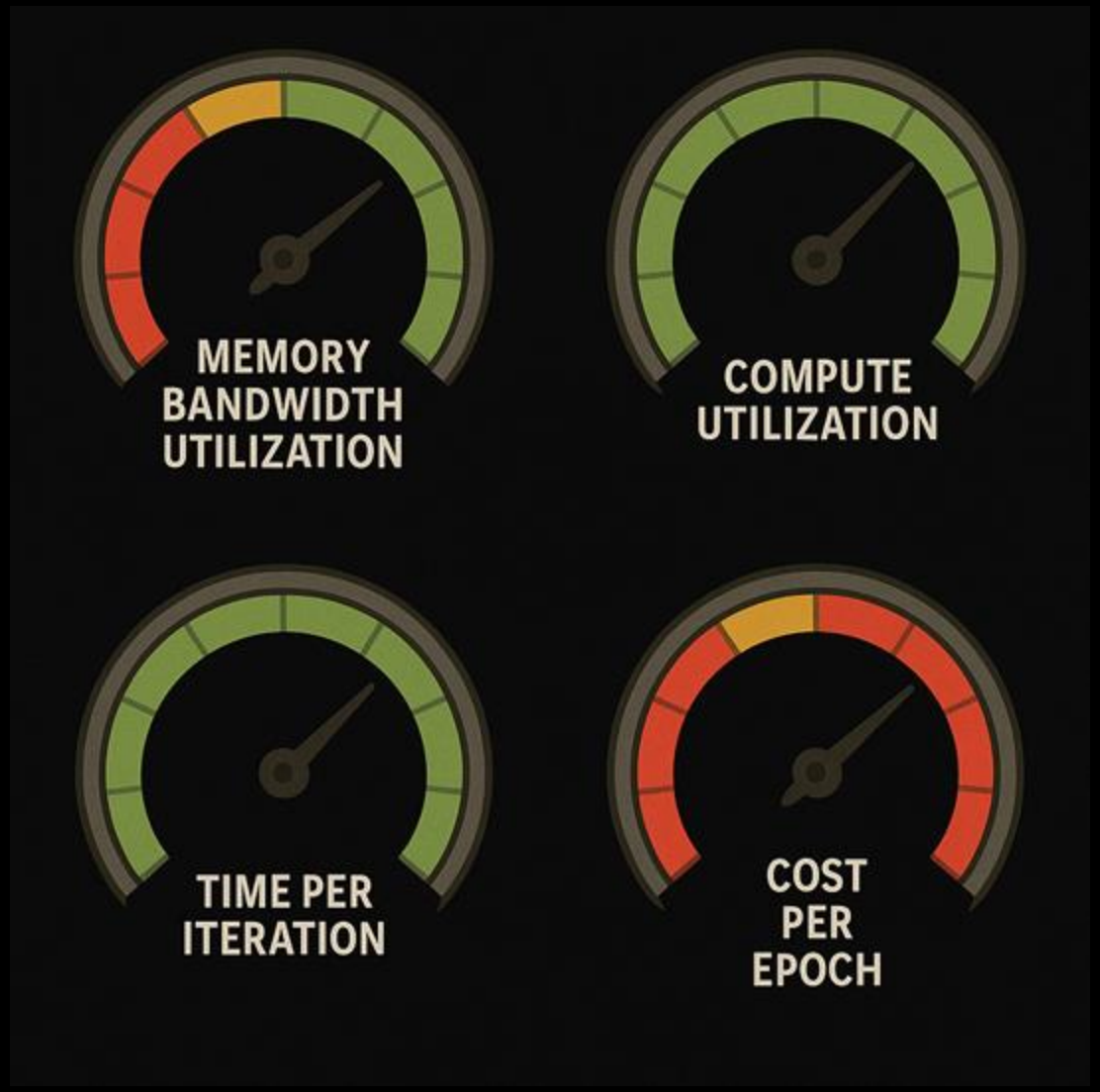
AI Student → Gamer (SFT → RL)



Dataset Coming Soon!

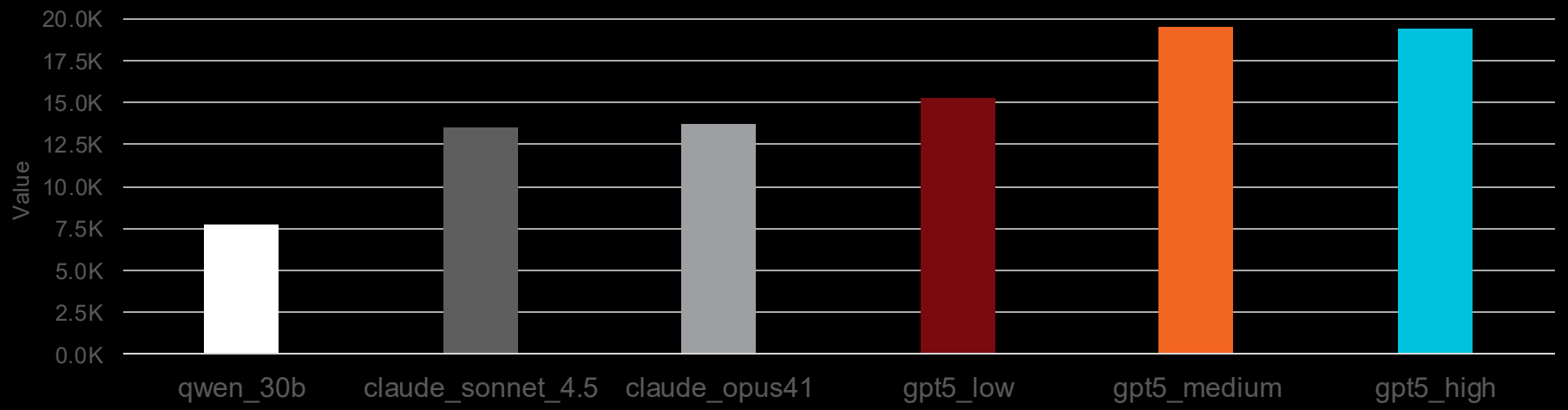


Benchmarking



Agentic KernelBench Results Summary

Compiling → Correctness → Performance

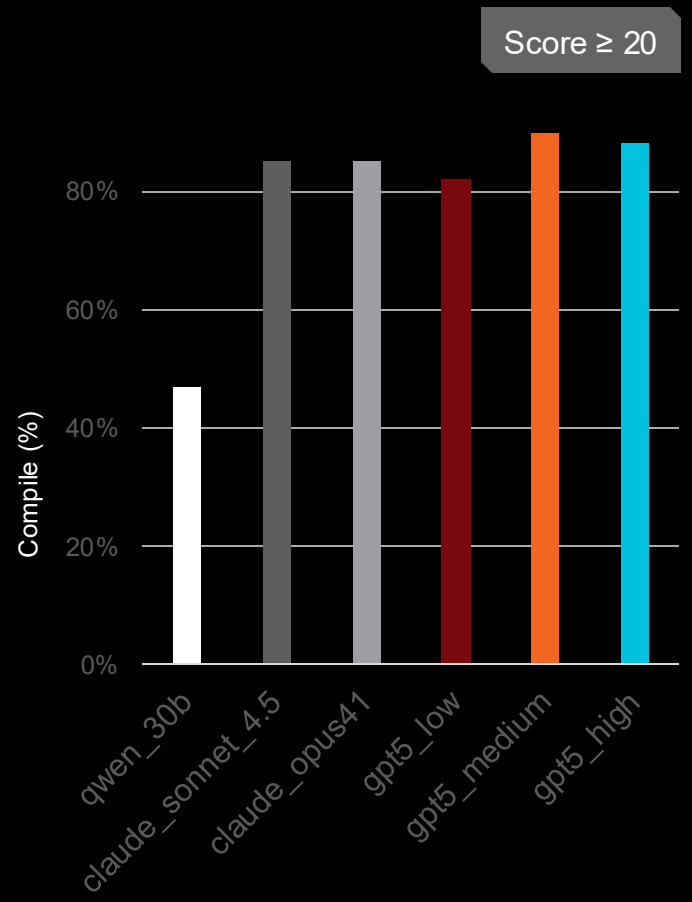


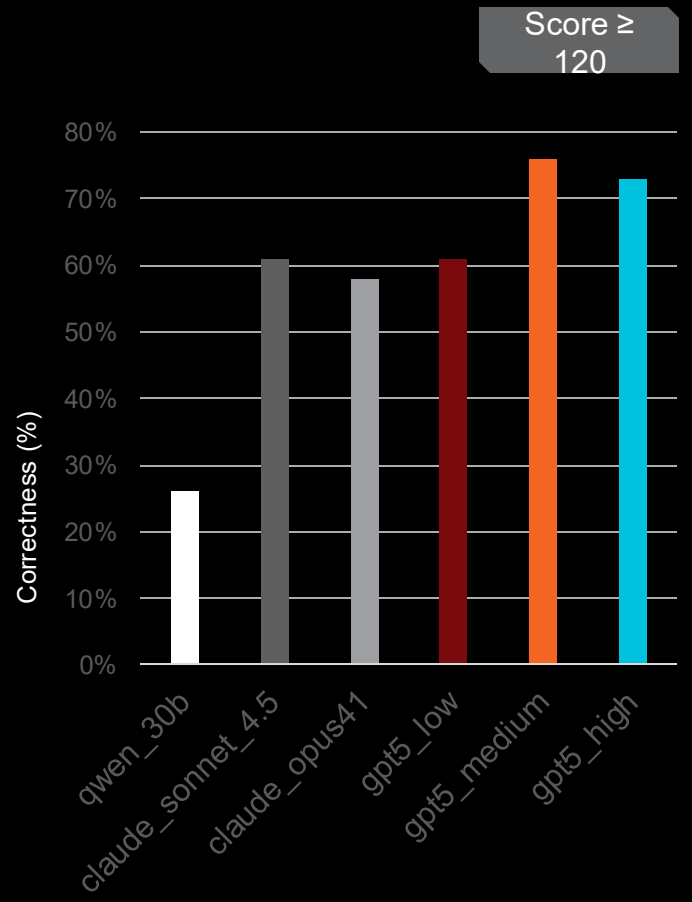
Level 1- Kernel Bench

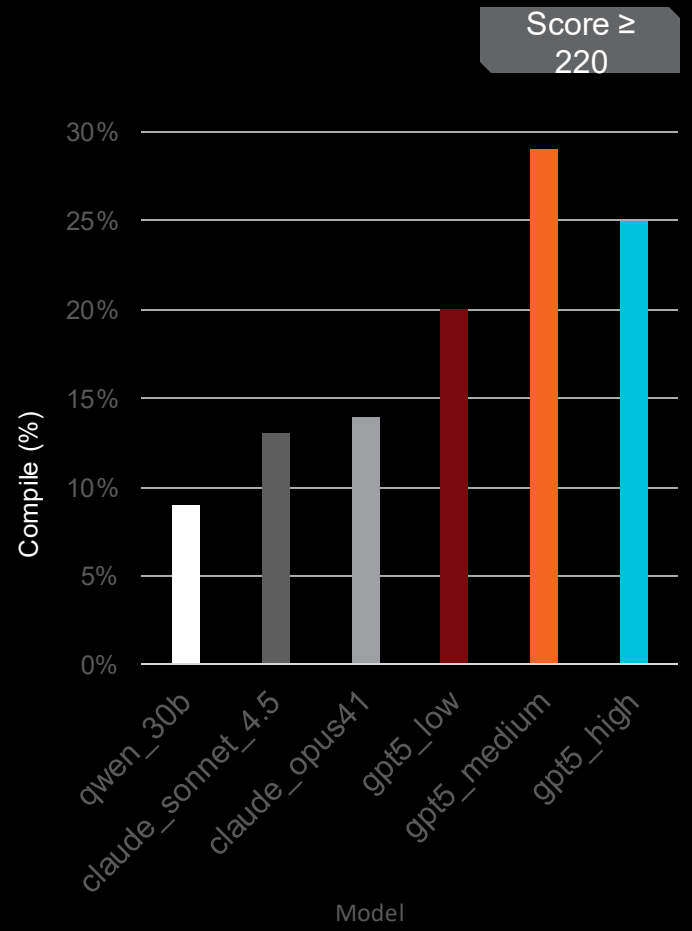
KernelBench: Stanford
Vincent Ouyang: AMD

Engineering projections as of October 2025, subject to change.

Agentic KernelBench Results Summary







KernelBench: Stanford
Vincent Ouyang: AMD

Engineering projections as of October 2025, subject to change.

Demo of benchmarking using KernelBench

KernelBench: https://github.com/ScalingIntelligence/KernelBench

https://drive.google.com/drive/folders/1w70-mXQNv8tULuMMXbryH7Ps_ZQAcPkL?usp=share_link

Future?

- Workloads that instantly adapt to any hardware
- Self-optimizing systems — your server writes its own kernels
- Performance that evolves in real time, not at compile time
- AI-driven code generation: machines teaching themselves to go faster
- The future: where your server grabs coffee while waiting… and maybe drinks it too

Acknowledgements

AGIKIT


Vincent Ouyang


Nithya Manohar


Hao Li

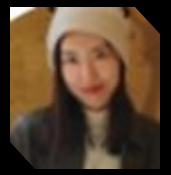

Efthimios Gianitsos


Sina Rafati

GEAK


Zicheng Liu


Dong Li


Ziqiong Liu


Pratik Prabhanjan Brahma

ROCm KernelGen


Zhaoyi Li

Omni


Muhammad Awad


Keith Lowery


Cole Ramos

Kernel Eng


John Tyler


Muhammad Osama

Leadership


Sharon Zhou


Emad Barsoum

THANK YOU