Open Source @ Huawei
OPEN SOURCE
源生无限 同行致远

2025华为星空大会
开源开放，共建AI时代新生态

PyPTO：Tile 编程与白盒优化

冯思远　上海创智学院助理教授

算力与访存的鸿沟：难以逾越的“内存墙”

GPU 型号	V100	A100	H100	H200	B200
显存容量	32 GB	80 GB	80 GB	144 GB	192 GB
bf16 算力(TFLOPs)	125	312	989	989	2250
显存带宽 (TB/s)	0.9	2.0	3.25	4.8	8.0
算力提升	1x	2.5x	**7.9x**	7.9x	**18x**
访存提升	1x	2.2x	**3.6x**	5.3x	**8.9x**

原始算力的提升速度远超过**内存带宽**的提升速度

算力与访存的鸿沟：难以逾越的“内存墙”

GPU 型号	V100	A100	H100	H200	B200
显存容量	32 GB	80 GB	80 GB	144 GB	192 GB
bf16 算力(TFLOPs)	125	312	989	989	2250
显存带宽 (TB/s)	0.9	2.0	3.25	4.8	8.0
算力提升	1x	2.5x	**7.9x**	7.9x	**18x**
访存提升	1x	2.2x	**3.6x**	5.3x	**8.9x**
访存特性	N/A	cp.async	TMA, Warp Specialize		TMEM

原始算力的提升速度远超过**内存带宽**的提升速度

增加硬件和软件**复杂度**，大幅**提高编程门槛**

Tile编程：以块为单位，屏蔽细粒度细节

Tensor

完整输入输出，计算方式与**硬件无关**
难以结合硬件特性进行优化

Tile

切分后的**块状**输入输出，计算方式与硬件**部分相关**
通常映射到硬件的**单一核心**，需要**考虑 SRAM 优化**

Element

单个输入输出**元素**，计算方式与硬件**高度绑定**
需要**完整考虑**硬件内存层级、异步单元等**细节**

Tile编程：以块为单位，屏蔽细粒度细节

Tensor

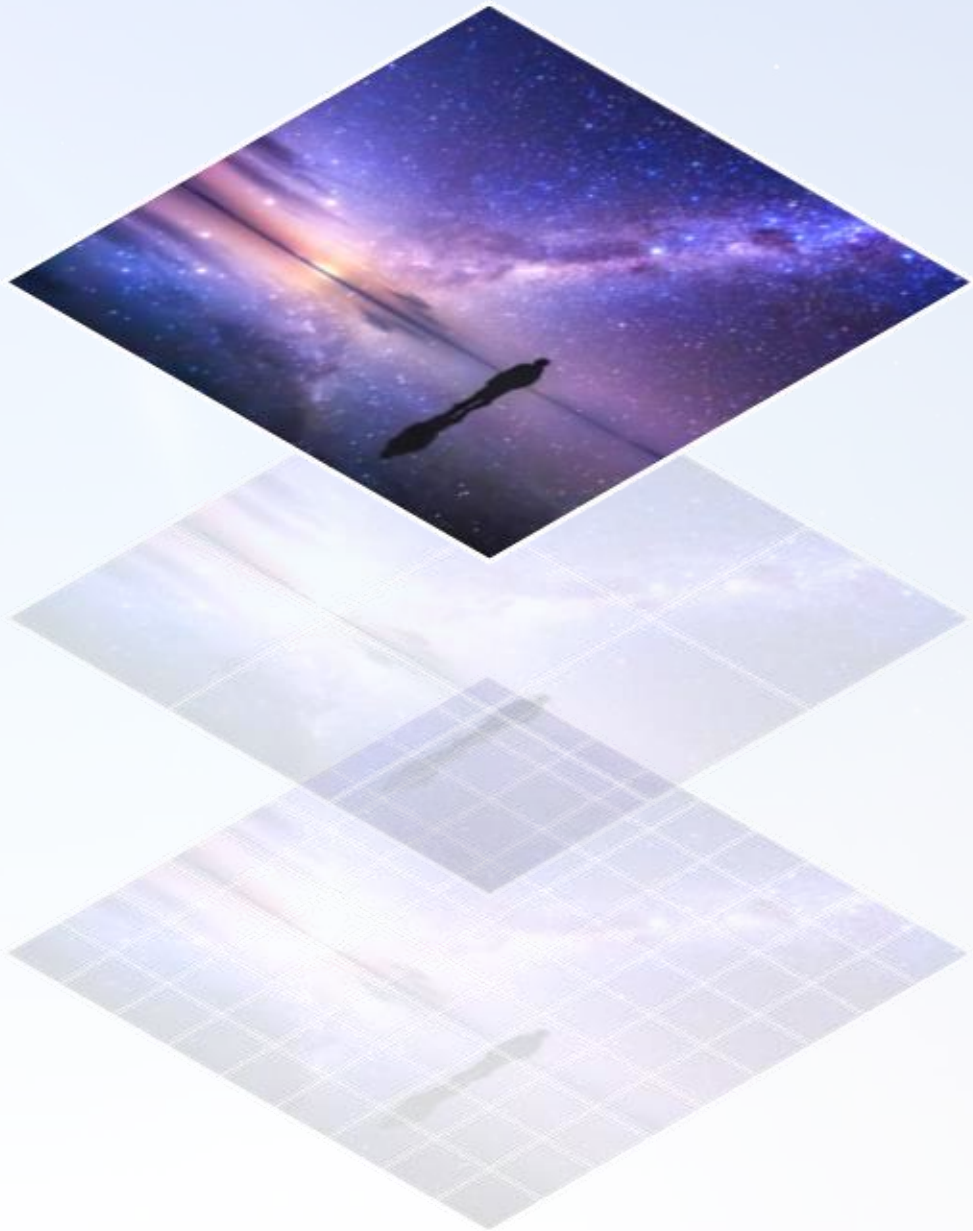
完整输入输出，计算方式与**硬件无关**
难以结合硬件特性进行优化

Tile

切分后的**块状**输入输出，计算方式与硬件**部分相关**
通常映射到硬件的**单一核心**，需要**考虑 SRAM 优化**

Element

单个输入输出**元素**，计算方式与硬件**高度绑定**
需要**完整考虑**硬件内存层级、异步单元等**细节**



Tile编程：以块为单位，屏蔽细粒度细节

Tensor

完整输入输出，计算方式与**硬件无关**
难以结合硬件特性进行优化

Tile

切分后的**块状**输入输出，计算方式与硬件**部分相关**
通常映射到硬件的**单一核心**，需要**考虑 SRAM 优化**

Element

单个输入输出**元素**，计算方式与硬件**高度绑定**
需要**完整考虑**硬件内存层级、异步单元等**细节**

Tile编程：以块为单位，屏蔽细粒度细节

Tensor

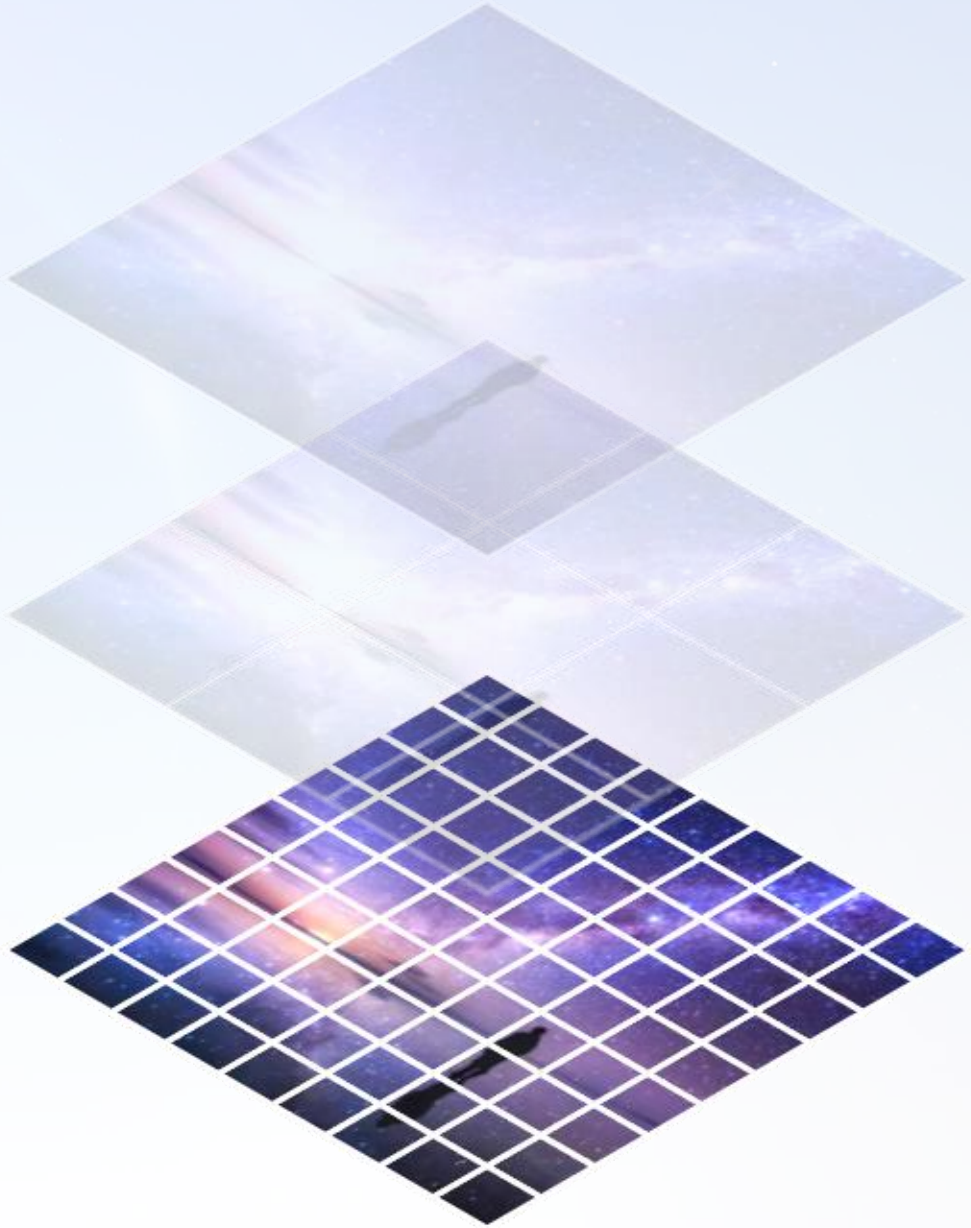
完整输入输出，计算方式与**硬件无关**
难以结合硬件特性进行优化

Tile

切分后的**块状**输入输出，计算方式与硬件**部分相关**
通常映射到硬件的**单一核心**，需要**考虑 SRAM 优化**

Element

单个输入输出**元素**，计算方式与硬件**高度绑定**
需要**完整考虑**硬件内存层级、异步单元等**细节**



Tile编程：以块为单位，屏蔽细粒度细节

Tensor

完整输入输出，计算方式与**硬件无关**
难以结合硬件特性进行优化

Tile

切分后的**块状**输入输出，计算方式与硬件**部分相关**
通常映射到硬件的**单一核心**，需要**考虑 SRAM 优化**

Element

单个输入输出**元素**，计算方式与硬件**高度绑定**
需要**完整考虑**硬件内存层级、异步单元等**细节**

生态破局 – Triton：在 Python 中释放 GPU 算力

以 **Tile 为核心**的编程范式

以 **Python 为前端**的编程语言

屏蔽了**内存层级**和**指令细节**

开发者专注于快速的**算法迭代**

```python
@triton.jit
def add_kernel(
    x_ptr,  # *Pointer* to first input vector.
    y_ptr,  # *Pointer* to second input vector.
    output_ptr,  # *Pointer* to output vector.
    n_elements,  # Size of the vector.
    BLOCK_SIZE: tl.constexpr,
):
    pid = tl.program_id()
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)

    mask = offsets < n_elements
    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)
    output = x + y

    tl.store(output_ptr + offsets, output, mask=mask)
```

多层优化 – TileLang：融合专家经验的 Tile 编程

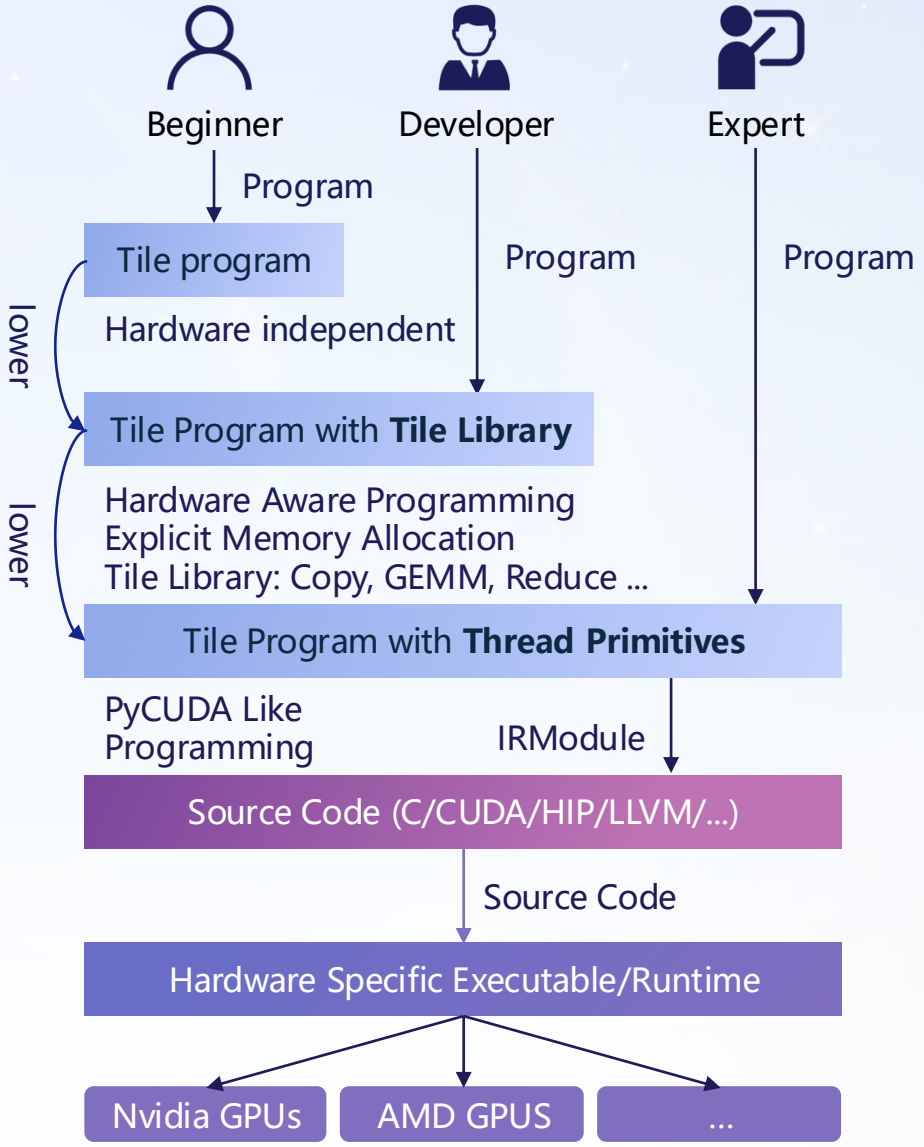
多层级编程接口

混合了 **Tile** 编程和 **SIMT** 编程

给**专家**提供了**细粒度**的优化接口

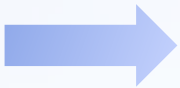
极致性能和**更灵活**的编程范式

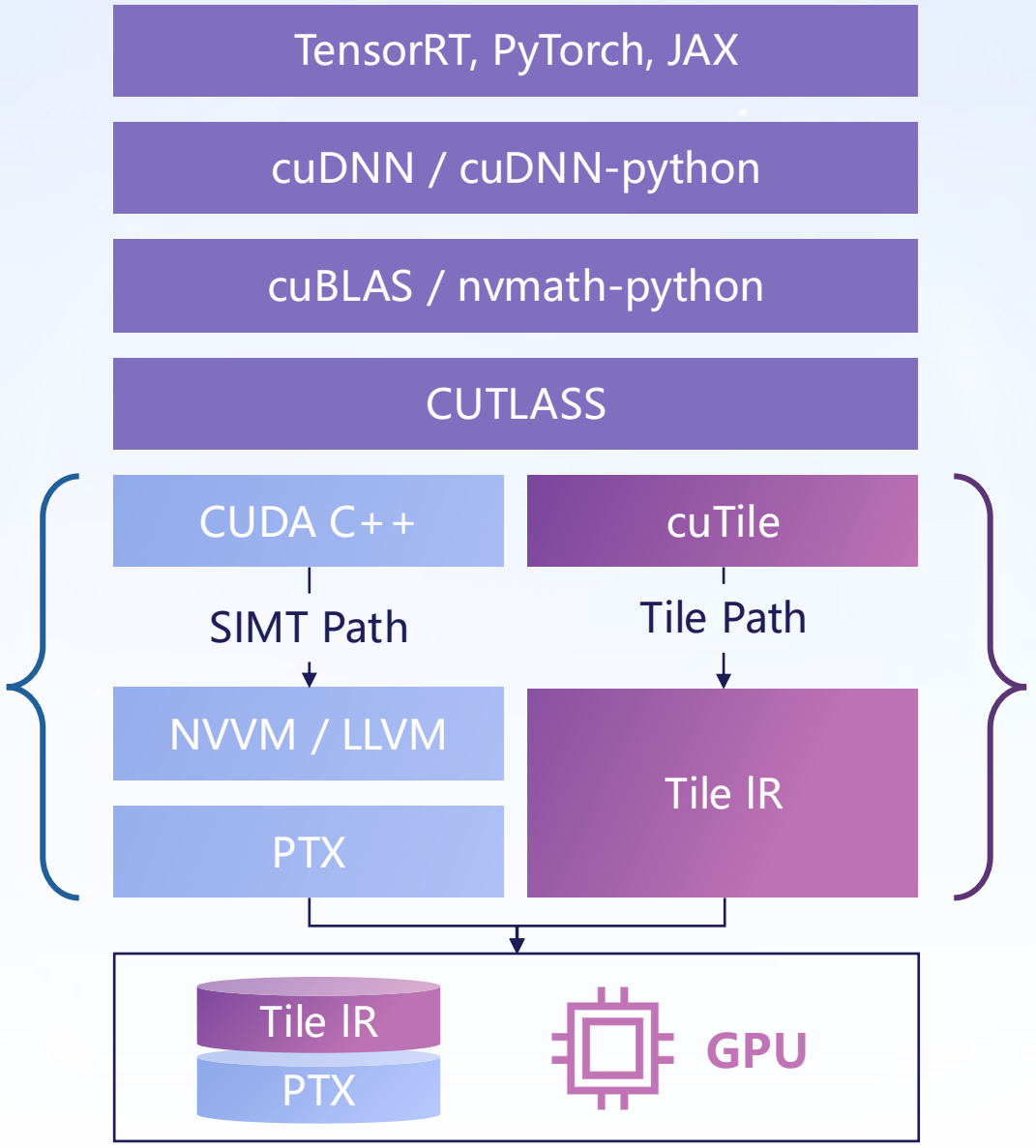
对普通开发者**不友好**



硬件原生 – CuTile：跨代兼容的 Tile 原生指令集

由于硬件迭代
难以跨代兼容

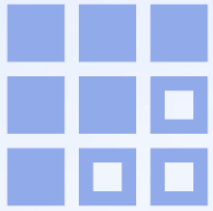




屏蔽部分特性
实现跨代兼容

PyPTO 融合算子开发框架

Tile 编程



以 Tile 为粒度编程，
屏蔽不必要的优化细节。

Python 原生

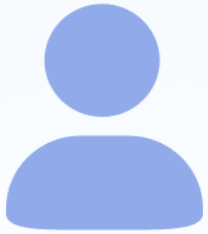
提供 Python 原生接口，
降低用户开发门槛。

PyPTO

跨代兼容

通过底层 Tile 指令集，
实现指令集跨代兼容。

人机协作



充分发挥专家经验，
编译流程白盒化。

PyPTO：Python 原生的 Tile 编程语言


```python
@pypto.frontend.jit
def test_softmax(
  input: pypto.Tensor((M, N), pypto.DT_FP32),
) -> pypto.Tensor((M, N), pypto.DT_FP32):
  pypto.set_vec_tile_shapes(32, 32)

  rowmax = pypto.amax(input, -1, True)
  sub_res = pypto.sub(input, rowmax)
  exp_res = pypto.exp(sub_res)
  esum = pypto.sum(exp_res, -1, True)
  output = pypto.div(exp_res, esum)

  return output

def main():
  x = torch.randn((M, N), dtype=torch.float32)
  output = test_softmax(x)
```

通过 frontend.jit 装饰器构建PyPTO 函数

原生 **Tensor 输入输出**

通过编译**配置参数**控制编译优化

类似 PyTorch 的语法 **构造 Tensor Graph**

原生函数调用，且**原生支持 torch Tensor**

PyPTO：Python 原生的 Tile 编程语言


```python
@pypto.frontend.jit
def test_softmax(
  input: pypto.Tensor((M, N), pypto.DT_FP32),
) -> pypto.Tensor((M, N), pypto.DT_FP32):
  pypto.set_vec_tile_shapes(32, 32)

  rowmax = pypto.amax(input, -1, True)
  sub_res = pypto.sub(input, rowmax)
  exp_res = pypto.exp(sub_res)
  esum = pypto.sum(exp_res, -1, True)
  output = pypto.div(exp_res, esum)

  return output

def main():
  x = torch.randn((M, N), dtype=torch.float32)
  output = test_softmax(x)
```

通过 **frontend.jit** 装饰器构建PyPTO 函数

原生 **Tensor 输入输出**

通过编译**配置参数**控制编译优化

类似 PyTorch 的语法 **构造 Tensor Graph**

原生函数调用，且**原生支持 torch Tensor**

PyPTO：Python 原生的 Tile 编程语言


```python
@pypto.frontend.jit
def test_softmax(
  input: pypto.Tensor((M, N), pypto.DT_FP32),
) -> pypto.Tensor((M, N), pypto.DT_FP32):
  pypto.set_vec_tile_shapes(32, 32)

  rowmax = pypto.amax(input, -1, True)
  sub_res = pypto.sub(input, rowmax)
  exp_res = pypto.exp(sub_res)
  esum = pypto.sum(exp_res, -1, True)
  output = pypto.div(exp_res, esum)

  return output

def main():
  x = torch.randn((M, N), dtype=torch.float32)
  output = test_softmax(x)
```

通过 frontend.jit 装饰器构建PyPTO 函数

原生 **Tensor 输入输出**

通过编译**配置参数**控制编译优化

类似 PyTorch 的语法 **构造 Tensor Graph**

原生函数调用，且**原生支持 torch Tensor**

PyPTO：Python 原生的 Tile 编程语言


```python
@pypto.frontend.jit
def test_softmax(
  input: pypto.Tensor((M, N), pypto.DT_FP32),
) -> pypto.Tensor((M, N), pypto.DT_FP32):
  pypto.set_vec_tile_shapes(32, 32)

  rowmax = pypto.amax(input, -1, True)
  sub_res = pypto.sub(input, rowmax)
  exp_res = pypto.exp(sub_res)
  esum = pypto.sum(exp_res, -1, True)
  output = pypto.div(exp_res, esum)

  return output

def main():
  x = torch.randn((M, N), dtype=torch.float32)
  output = test_softmax(x)
```

通过 **frontend.jit** 装饰器构建PyPTO 函数

原生 **Tensor 输入输出**

通过编译**配置参数**控制编译优化

类似 PyTorch 的语法 **构造 Tensor Graph**

原生函数调用，且**原生支持 torch Tensor**

PyPTO：Python 原生的 Tile 编程语言


```python
@pypto.frontend.jit
def test_softmax(
  input: pypto.Tensor((M, N), pypto.DT_FP32),
) -> pypto.Tensor((M, N), pypto.DT_FP32):
  pypto.set_vec_tile_shapes(32, 32)

  rowmax = pypto.amax(input, -1, True)
  sub_res = pypto.sub(input, rowmax)
  exp_res = pypto.exp(sub_res)
  esum = pypto.sum(exp_res, -1, True)
  output = pypto.div(exp_res, esum)

  return output

def main():
  x = torch.randn((M, N), dtype=torch.float32)
  output = test_softmax(x)
```

通过 **frontend.jit** 装饰器构建PyPTO 函数

原生 **Tensor 输入输出**

通过编译**配置参数**控制编译优化

类似 PyTorch 的语法 **构造 Tensor Graph**

原生函数调用，且**原生支持 torch Tensor**

PyPTO：Python 原生的 Tile 编程语言


```python
@pypto.frontend.jit
def test_softmax(
  input: pypto.Tensor((M, N), pypto.DT_FP32),
) -> pypto.Tensor((M, N), pypto.DT_FP32):
  pypto.set_vec_tile_shapes(32, 32)

  rowmax = pypto.amax(input, -1, True)
  sub_res = pypto.sub(input, rowmax)
  exp_res = pypto.exp(sub_res)
  esum = pypto.sum(exp_res, -1, True)
  output = pypto.div(exp_res, esum)

  return output

def main():
  x = torch.randn((M, N), dtype=torch.float32)
  output = test_softmax(x)
```

通过 **frontend.jit** 装饰器构建PyPTO 函数

原生 **Tensor 输入输出**

通过编译**配置参数**控制编译优化

类似 PyTorch 的语法 **构造 Tensor Graph**

原生函数调用，且**原生支持 torch Tensor**

PyPTO：Python 原生的 Tile 编程语言


```python
@pypto.frontend.jit
def test_softmax(
  input: pypto.Tensor((M, N), pypto.DT_FP32),
) -> pypto.Tensor((M, N), pypto.DT_FP32):
  pypto.set_vec_tile_shapes(32, 32)

  rowmax = pypto.amax(input, -1, True)
  sub_res = pypto.sub(input, rowmax)
  exp_res = pypto.exp(sub_res)
  esum = pypto.sum(exp_res, -1, True)
  output = pypto.div(exp_res, esum)

  return output

def main():
  x = torch.randn((M, N), dtype=torch.float32)
  output = test_softmax(x)
```

通过 **frontend.jit** 装饰器构建PyPTO 函数

原生 **Tensor 输入输出**

通过编译**配置参数**控制编译优化

类似 PyTorch 的语法 **构造 Tensor Graph**

原生函数调用，且**原生支持 torch Tensor**

跨代兼容的 Tile 虚拟指令集

PTO 虚拟指令集通过 Tile 的抽象，使其指令与硬件实现解耦，实现跨代兼容

```python
@pypto.frontend.jit
def test_softmax(
  input: pypto.Tensor((M, N), pypto.DT_FP32),
) -> pypto.Tensor((M, N), pypto.DT_FP32):
  pypto.set_vec_tile_shapes(32, 32)

  rowmax = pypto.amax(input, -1, True)
  sub_res = pypto.sub(input, rowmax)
  exp_res = pypto.exp(sub_res)
  esum = pypto.sum(exp_res, -1, True)
  output = pypto.div(exp_res, esum)

  return output
```

用户 Python 代码

编译

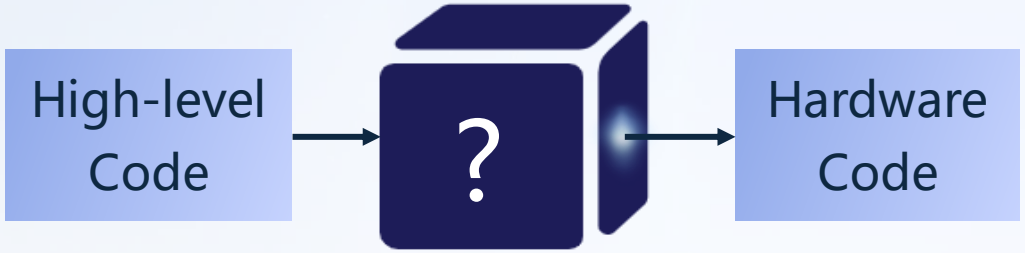
```
// Load
TLoad(ubTensor_0, gmTensor_1, …);
set_flag(PIPE_MTE2, PIPE_V, EVENT_ID0);

// Compute
wait_flag(PIPE_MTE2, PIPE_V, EVENT_ID0);
TPairMax(ubTensor_2, ubTensor_0, ubTensor_1);
TSub(ubTensor_3, ubTensor_0, ubTensor_2);
TExp(ubTensor_3, ubTensor_3);
…
set_flag(PIPE_V, PIPE_MTE3, EVENT_ID0);

// Write back
wait_flag(PIPE_V, PIPE_MTE3, EVENT_ID0);
TStore(gmTensor, ubTensor, …);
```

底层 PTO 虚拟指令集

注：生成 PTO 指令代码仅作示例

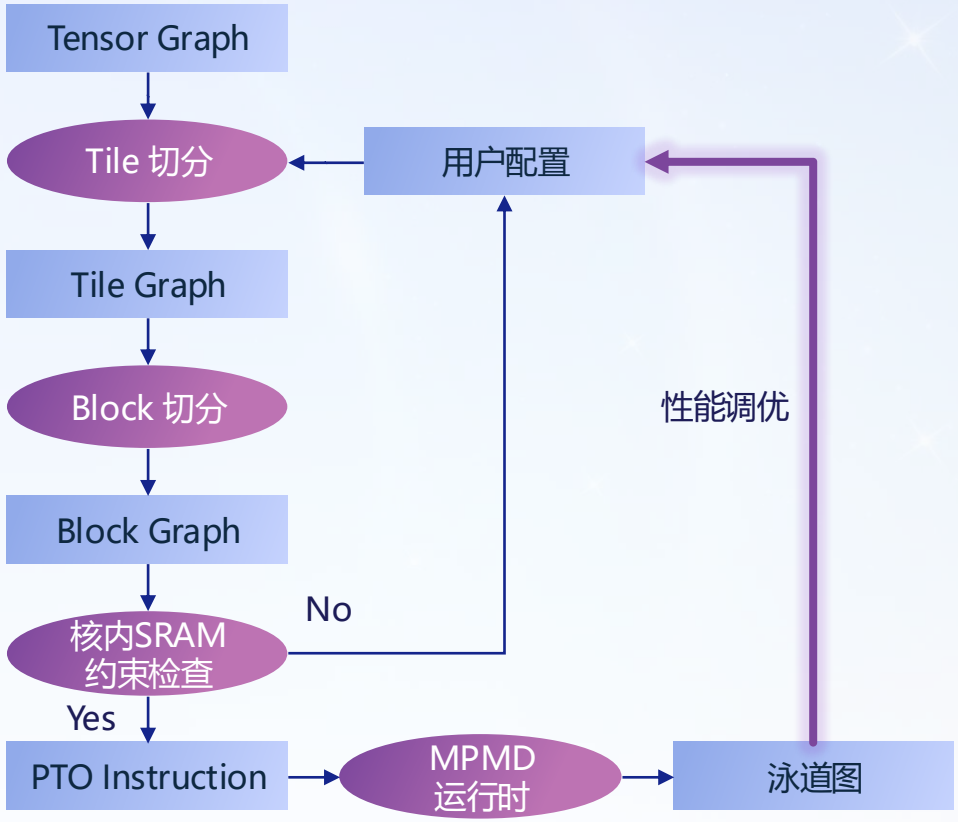
PyPTO：白盒编译与人机协同

黑盒编译



传统编译器内部过程**不可见**，专家**难以介入优化**

PyPTO 白盒编译

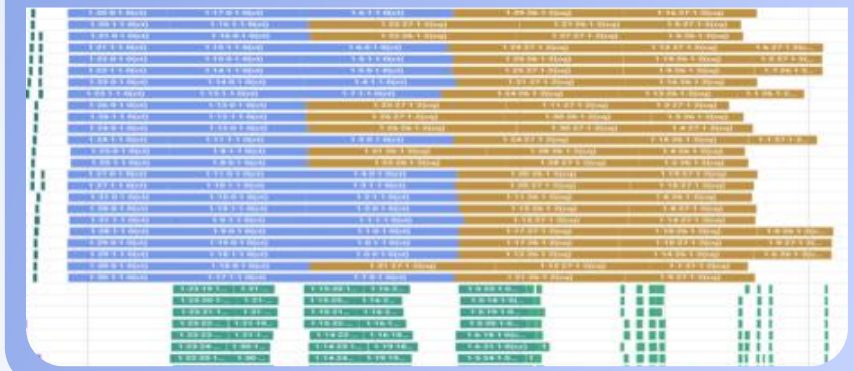


PyPTO 将编译过程**透明化**，允许用户在关键环节**直接参与优化**

Human-in-the-Loop 闭环调优

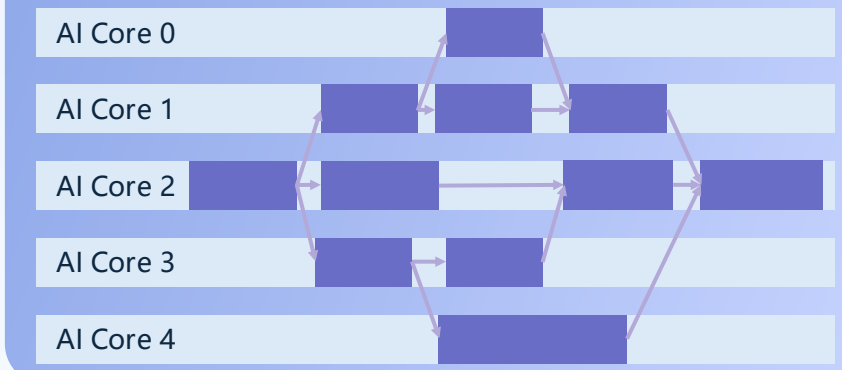
通过**可视化工具**和**可配置参数**，用户可以直接**观察**、**分析**并**控制**编译优化过程，从而实现极致性能

调优结果：泳道图



性能分析

编译优化：MPMD 任务图



定位问题

用户配置：Tile Config

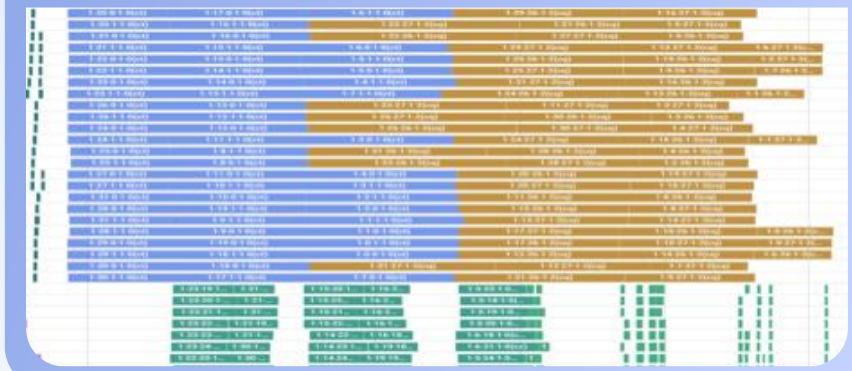
```
pypto.set_vec_tile_shapes(64, 64)
pypto.set_cube_tile_shapes(
    [16, 16], [64, 64], [64, 64]
)
```

修改配置

Future: Agent-in-the-Loop 闭环调优

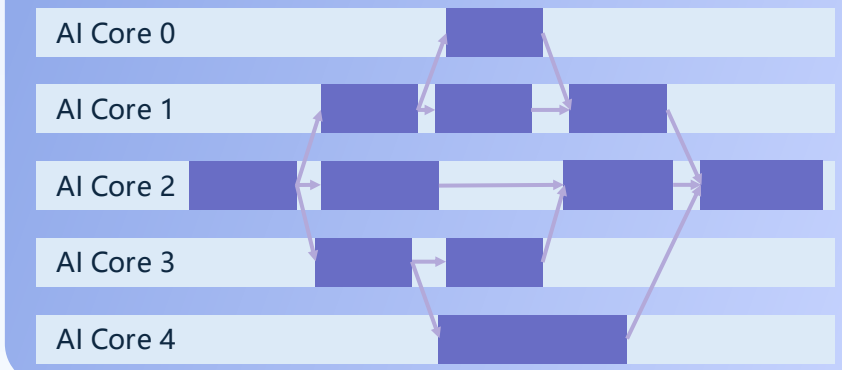
通过**结构化性能分析工具**和**可配置参数**， AI Agent 可以直接**闭环分析、优化**程序

调优结果：泳道图



性能分析

编译优化：MPMD 任务图



定位问题

用户配置：Tile Config

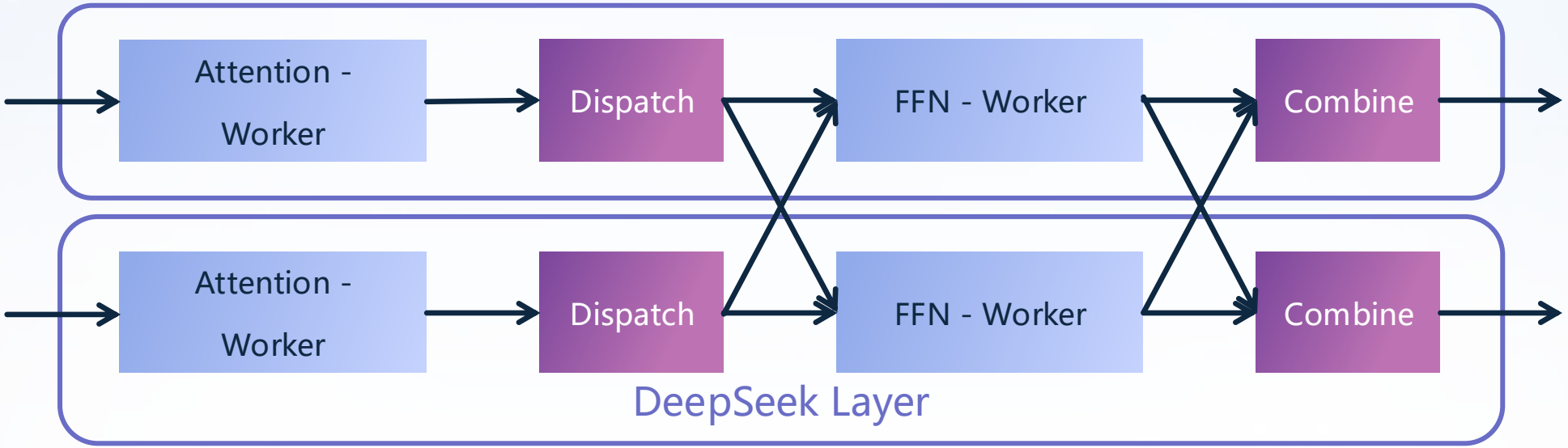
```
pypto.set_vec_tile_shapes(64, 64)
pypto.set_cube_tile_shapes(
    [16, 16], [64, 64], [64, 64]
)
```

修改配置

基于白盒编译的 Mega-Kernel

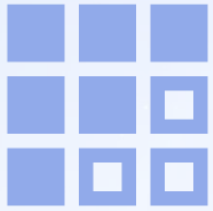
DeepSeek R1 推理实例，达成主线 **0.9x-1.0x** 性能

Ascend C	PyPTO	Ascend C	PyPTO
26	4	57.8k	1.03k
算子数量		总代码行数 **56x** 缩减	



PyPTO 融合算子开发框架

Tile 编程



以 Tile 为粒度编程，
屏蔽不必要的优化细节。

Python 原生

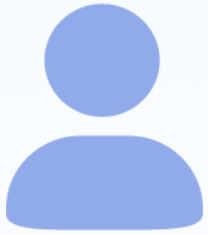
提供 Python 原生接口，
降低用户开发门槛。

PyPTO

跨代兼容

通过底层 Tile 指令集，
实现指令集跨代兼容。

人机协作



充分发挥专家经验，
编译流程白盒化。

Roadmap

底座标准化 Standardization

完善基础功能，建立工业级稳定性

- Python 前端与 IR
- 编译系统基础
- 运行时与兼容性
- 开发工具链

开发体验深度化 Empowerment

开放深层编程接口，支持复杂场景

- 分层编程接口
- 增量编译、协同优化
- 通信-计算融合
- 智能化工具链

价值驱动共建 Ecosystem

拓展领域边界，构筑繁荣开源生态

- 全场景执行能力
- 泛 AI 计算场景
- 社区共建生态体系
- AI 辅助算子开发调优

PyPTO 将在近期全量开源，敬请期待

THANKS