AI编译器-后端优化

后端优化

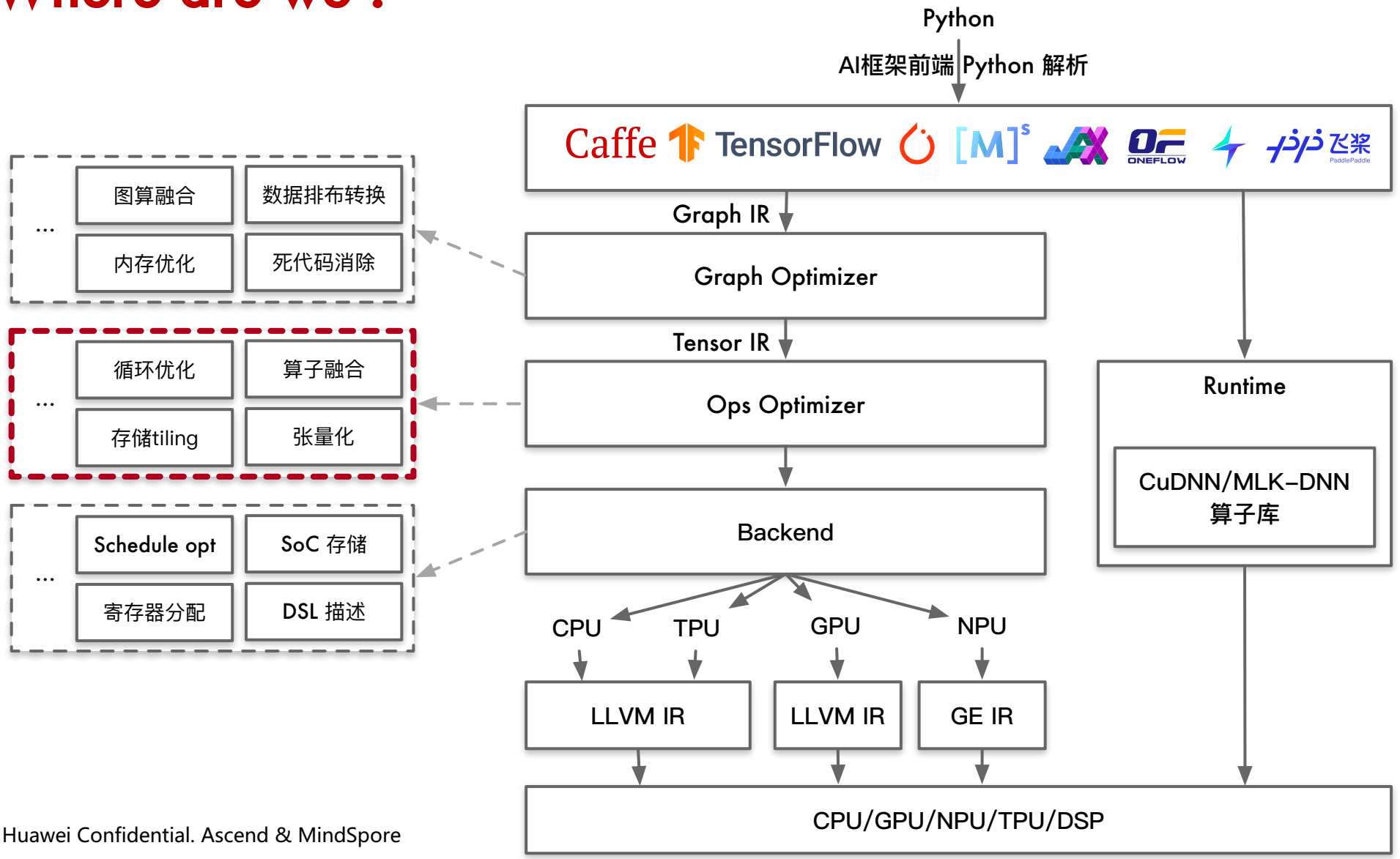
ZOMI

Talk Overview

I. AI 编译器后端优化

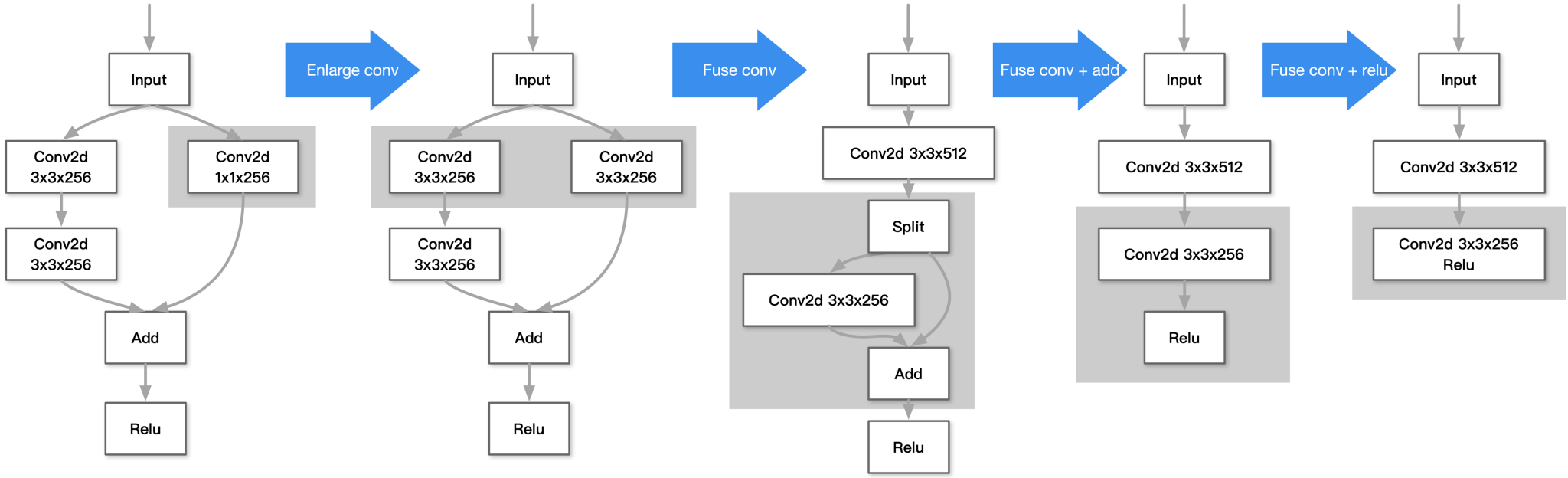
- 后端优化概念
- 算子执行与调度
- 算子循环优化
- Auto-Tuning
- Polyhedral

Where are we ?



后端优化与前端优化的区别

- **前端优化**：输入计算图，关注计算图整体拓扑结构，而不关心算子的具体实现。在 AI 编译器的前端优化，对算子节点进行融合、消除、化简等操作，使计算图的计算和存储开销最小。
- **后端优化**：关注算子节点的内部具体实现，针对具体实现使得性能达到最优。重点关心节点的输入，输出，内存循环方式和计算的逻辑。



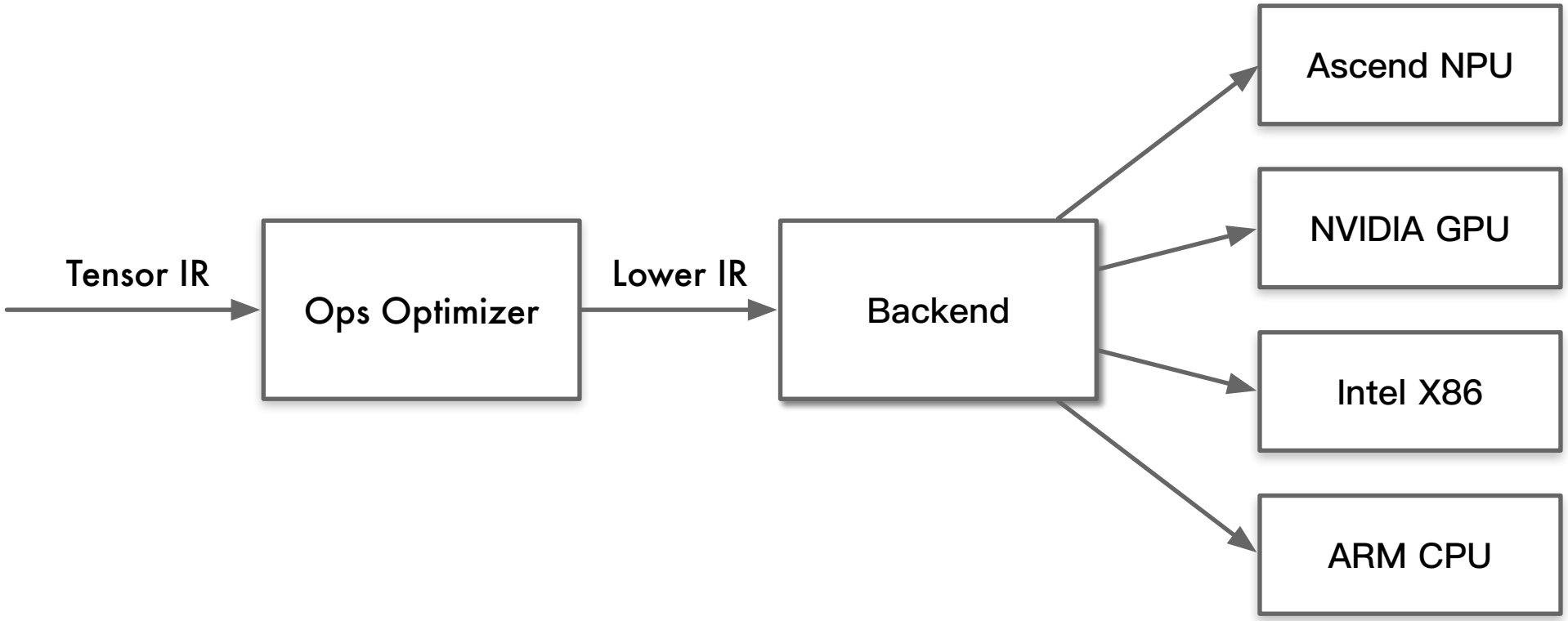
后端优化与前端优化的区别

- **后端优化**：关注算子节点的内部具体实现，针对具体实现使得性能达到最优。重点关心节点的输入，输出，内存循环方式和计算的逻辑。


```
for (int n = 0; n < o_n; ++n) {
        for (int c = 0; c < o_c; ++c) {
            for (int j = 0; j < o_h; ++j) {
                for (int i = 0; i < o_w; ++i) {
                    int d_start = n * i_c * i_h * i_w + j * i_w + i;
                    int temp = 0;
                    for (int kk = 0; kk < k_c; ++kk) {
                        for (int kj = 0; kj < k_h; ++kj) {
                            for (int ki = 0; ki < k_w; ++ki) {
                                int k_idx = kk * k_h * k_w + kj * k_w + ki;
                                int d_idx = d_start + kk * i_h * i_w + kj * i_w + ki;
                                temp += inputs->data[d_idx] * kernel->data[k_idx];
                            }
                        }
                    }
                    res[n * o_c * o_h * o_w + j * o_w + i] = temp;
                }
            }
        }
```

AI 编译器后端部分

1）生成低级IR；2）后端优化；3）代码生成



AI 编译器后端：生成低级IR

- I）生成低级IR：不同 AI 编译器内部低级 IR 形式和定义不同，但是对于同一算子，算法的原理实质相同。对于每个具体的算子，需要用AI编译器底层的接口来定义算法，再由编译器来生成内部的低级IR。

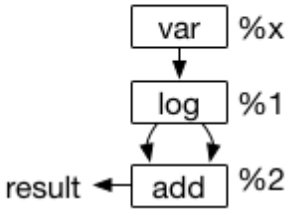
Python Code

```
x = relay.var("x")
v1 = relay.log(x)
v2 = relay.add(v1, v1)
f = relay.Function([x], v2)
```

Text Form

```
fn (%x) {
  %1 = log(%x)
  %2 = add(%1, %1)
  %2
}
```

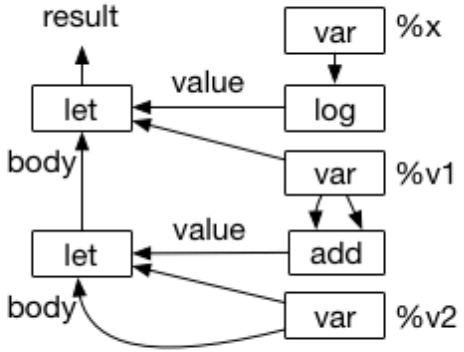
AST Structure



Dataflow

```
x = relay.var("x")
sb = relay.ScopeBuilder()
v1 = sb.let("v1", relay.log(x))
v2 = sb.let("v2", relay.add(v1, v1))
sb.ret(v2)
f = relay.Function([x], sb.get())
```

```
fn (%x) {
  let %v1 = log(%x)
  let %v2 = add(%v1, %v1)
  %v2
}
```



A-normal Form

AI 编译器后端：后端优化

- 2）后端优化：针对不同的硬件架构/微架构，不同的算法实现的方式有不同的性能，目的是找到算子的最优实现方式，达到最优性能。同一算子不同形态如Conv1x1、 Conv3x3、 Conv7x7都会有不同的循环优化方法。

AI 编译器后端：代码生成

- 3）代码生成：对优化后的低级 IR 转化为机器指令执行，现阶段最广泛的借助成熟的编译工具来实现，非AI 编译器的核心内容。如把低级IR 转化成为 LLVM、NVCC 等成功编译工具的输入形式，然后调用其生成机器指令。

Question?

- 为什么后端优化不直接用传统的通用编译器，如 GCC、LLVM 呢？

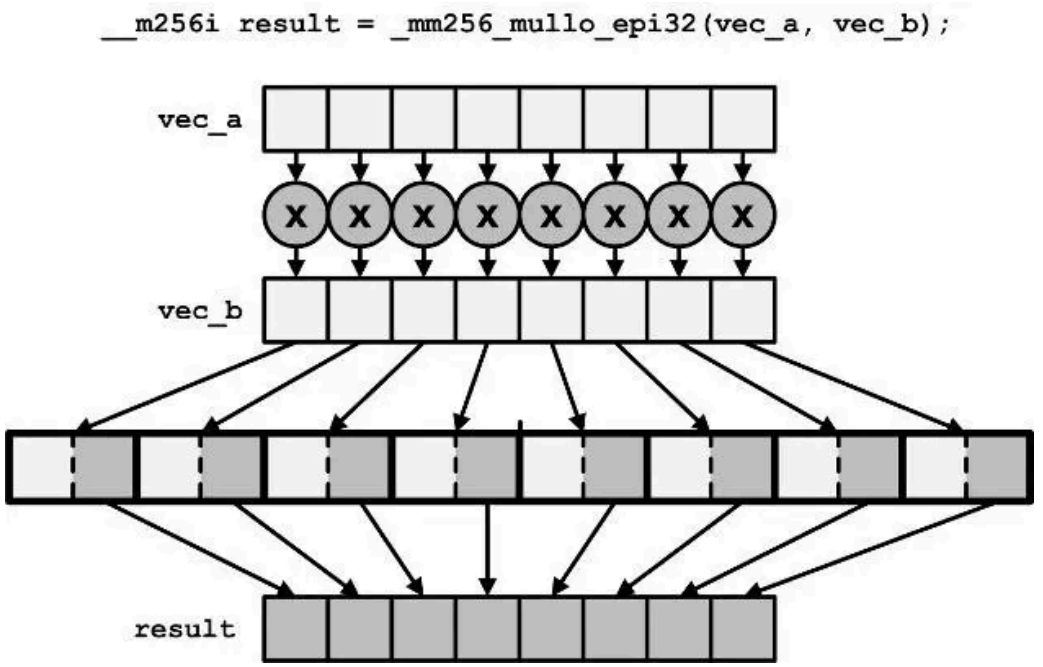
Question?

- 为什么后端优化不直接用传统的通用编译器，如 GCC、LLVM 呢？

1. 深度学习中主要数据为张量（Tensor），传统编译器不擅长对张量计算优化。
2. 通用编译器主要针对通用编程语言，缺少领域特定语言 DSL 支持，以及相关的特殊优化。

算子分类：访存密集型

- **访存密集型**：如 RNN 训练任务，由于 RNN 网络模型结构的计算密度更低，瓶颈转移到host端的 Op Launch 上，因此kernel之间甚至出现了大量空白。对于访存密集型算子部分的工作来说，新硬件带来了更大的性能优化空间。



算子分类：计算密集型

- **计算密集型**：计算密度是指一个程序在单位访存量下所需的计算量，单位是 FLOPs/Byte，计算密度较大，程序性能受硬件最大计算峰值（下文简称为算力）限制，称为计算密集型程序。

序号	batch	ic	ih	iw	kh	kw	stride	pad	dilation	group	oc	oh	ow	计算量(MFLOPs)	访存量(MB)	计算密度
1	1	3	224	224	7	7	2	3	1	1	64	112	112	236.028	3.673	64.267
2	1	128	28	28	3	3	1	1	1	1	128	28	28	231.211	1.328	174.088
3	1	512	7	7	1	1	1	0	1	1	1024	7	7	51.380	2.287	22.465
4	1	128	28	28	3	3	1	1	1	128	128	28	28	1.806	0.770	2.346

序号	batch	ic	ih	iw	kh	kw	stride	pad	dilation	group	oc	oh	ow	计算量(MFLOPs)	访存量(MB)	计算密度	理论计算速度(GFLOPs/s)	理论推理时间
3	1	512	7	7	1	1	1	0	1	1	1024	7	7	51.380	2.287	22.465	2156.653	23.824
5	1	16	448	448	3	3	1	1	1	16	16	448	448	57.803	24.501	2.359	226.487	255.214

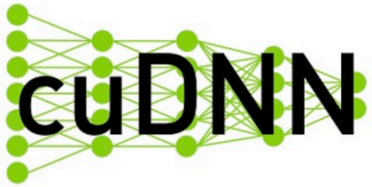
算子优化的挑战

- **优化手段多样**：要在不同情况下权衡优化及其对应参数，对于优化专家来说也是相当耗费精力
- **通用性与移植性**：不同类型的硬件架构差异，使得优化方法要考虑的因素也有很大不同
- **优化间相互影响**：各种优化之间可能会相互制约，相互影响。这意味着找到最优的优化方法组合与序列就是一个困难的组合优化问题，甚至是 NP 问题。

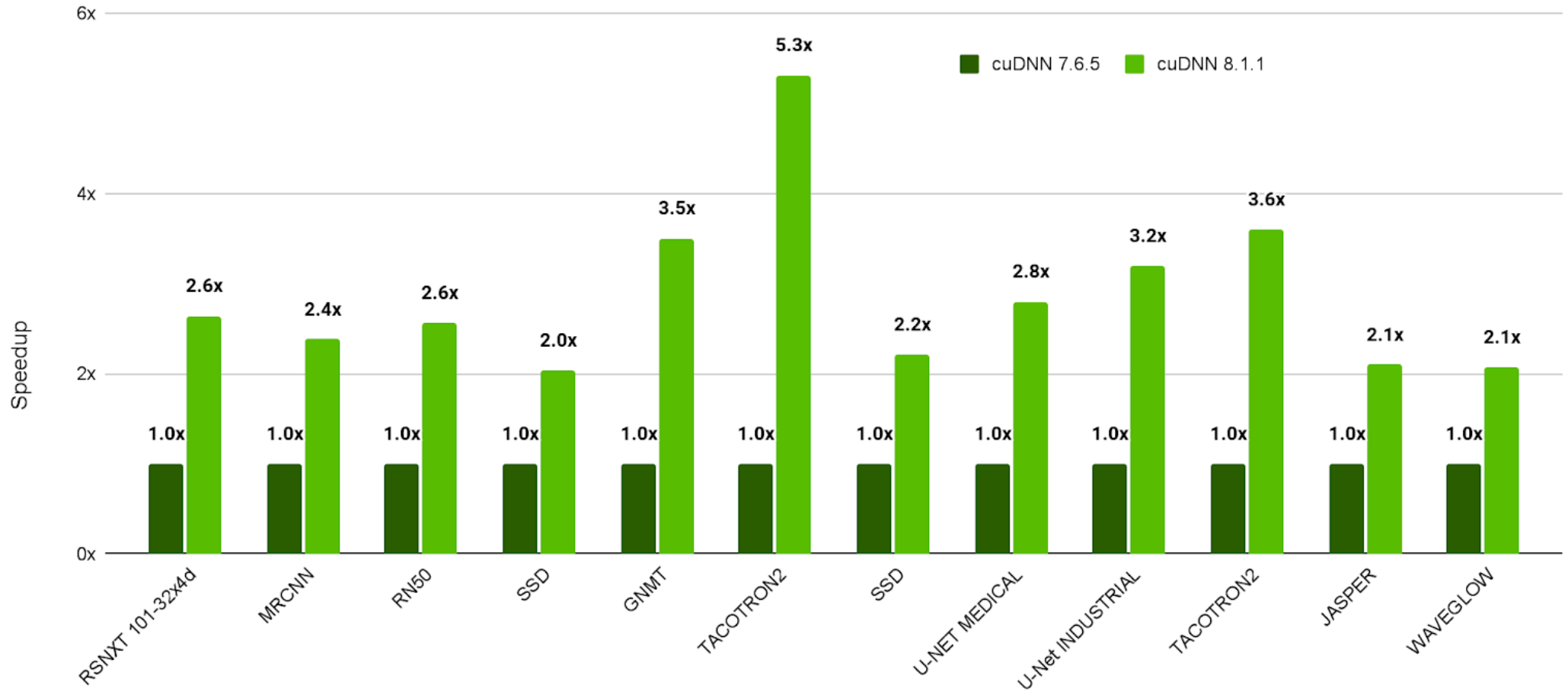
算子库

- 目的：针对访存密集型和计算密集型的算子进行优化。对于一个算子，要实现正确的逻辑计算或许不难，但要结合硬件能力达到高性能就比较难了，要达到极致的性能更是难上加难。

- 业界一个最为常见的方式是将预置的算子实现封装成计算库。如 CuDNN、CuBLAS、OpenBLAS、 Eigen 这样优秀的计算库。



A100 OVER 5X FASTER THAN V100 WITH CUDNN 8.1



Question?

1. 如何应对AI领域算子**迭代更新**快？如BN-> Layer BN->Group BN
2. 如何解决同一算子在多**平台移植**后一致性问题？
3. 如何面对算子**组合爆炸**问题？如参数多样，融合大算子等

自动化生成

1. 如何应对AI领域算子**迭代更新**快？如BN-> Layer BN->Group BN
2. 如何解决同一算子在多**平台移植**后一致性问题？
3. 如何面对算子**组合爆炸**问题？如参数多样，融合大算子等

自动 kernel 生成的方式，它的目标是对于给定算法自动生成目标平台上的高性能的实现。

自动化生成

1. A Survey on Compiler Autotuning using Machine Learning
2. Machine Learning in Compiler Optimization
3. Machine Learning in Compilers: Past, Present and Future

1）AI 编译器中正在尝试融合机器学习方法

Auto Tuning

2）算子计算加速正在用编译技术来解决

Polyhedral