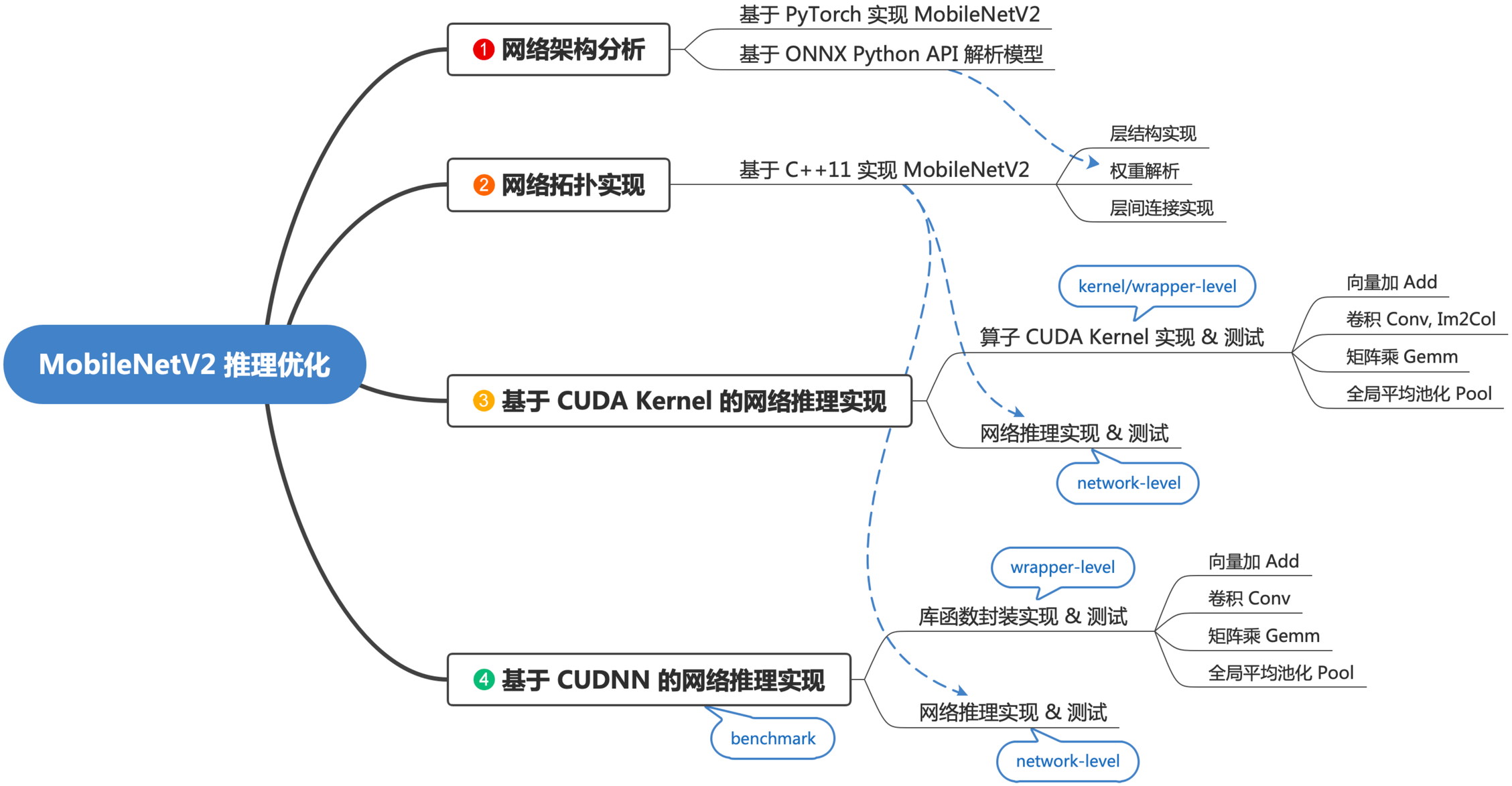
基于 CUDA 的 MobileNetV2 推理优化

实现流程


MobileNetV2 推理优化
1 网络架构分析
基于 PyTorch 实现 MobileNetV2
基于 ONNX Python API 解析模型
2 网络拓扑实现
基于 C++11 实现 MobileNetV2
层结构实现
权重解析
层间连接实现
3 基于 CUDA Kernel 的网络推理实现
kernel/wrapper-level
算子 CUDA Kernel 实现 & 测试
向量加 Add
卷积 Conv, Im2Col
矩阵乘 Gemm
全局平均池化 Pool
网络推理实现 & 测试
network-level
4 基于 CUDNN 的网络推理实现
wrapper-level
库函数封装实现 & 测试
向量加 Add
卷积 Conv
矩阵乘 Gemm
全局平均池化 Pool
网络推理实现 & 测试
network-level
benchmark

Step1　网络架构分析

- 基于 PyTorch 实现 MobileNetV2，对参考 ONNX 文件的网络架构进行还原
- 还原后的 ONNX 模型共有 5 种层：
  - ① 卷积层 Conv2d（#52）
  - ② 激活层 ReLU6（#35）
  - ③ 残差层 ResidualAdd（#10）
  - ④ 全局平均池化层 GlobalAveragePool（#1）
  - ⑤ 线性层 Linear（#1）
- 分析：
  - ① 卷积层数量最多，推理性能瓶颈在卷积层
  - ② 激活层均出现在卷积层后，将两层的实现进行融合
  - ③ 包括 3 种类型的卷积+激活层
    - 1x1 Conv2d + ReLU6
    - 3x3 Conv2d (group) + ReLU6
    - 1x1 Conv2d

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

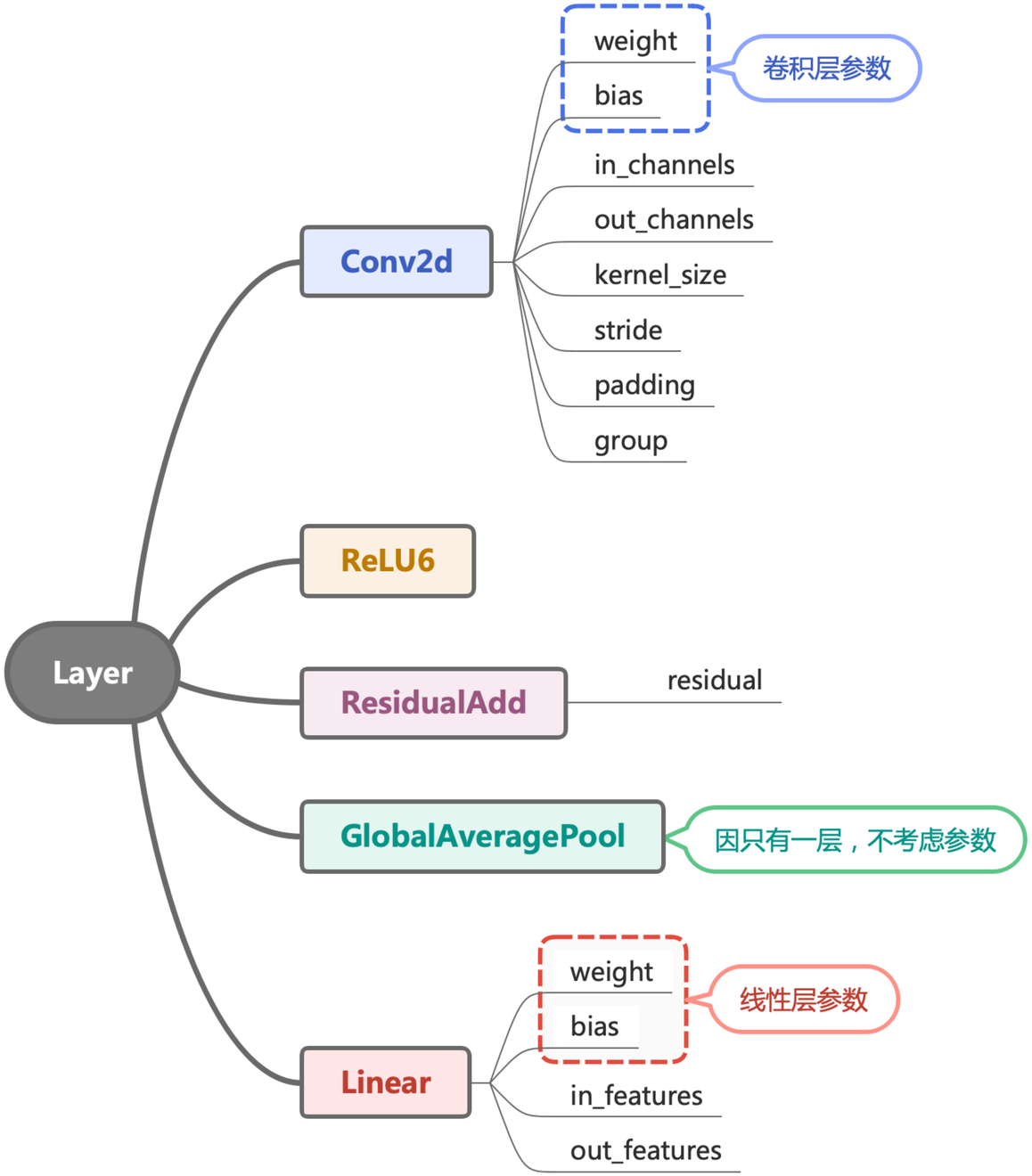
MobileNetV2 架构

Input	Operator	Output
$h \times w \times k$	1x1 conv2d , ReLU6	$h \times w \times (tk)$
$h \times w \times tk$	3x3 dwise s=s, ReLU6	$\frac{h}{s} \times \frac{w}{s} \times (tk)$
$\frac{h}{s} \times \frac{w}{s} \times tk$	linear 1x1 conv2d	$\frac{h}{s} \times \frac{w}{s} \times k'$

Bottleneck 模块（3 种卷积）

Step2　网络拓扑实现

- 基于 C++11 实现 MobileNetV2 拓扑结构，分为 3 步：
  - ① 实现每种层的结构
    - 每种层都继承自基类 Layer，以便层间连接
    - 权重参数包括卷积层参数、线性层参数



Step2　网络拓扑实现

- ② 基于 ONNX Python API 解析层参数
  - 参数存储在 ONNX 模型的 initializer 中，即

    ```
    initializers = onnx.load(input_path).graph.initializer
    ```

  - 包括权重 weight 、偏置 bias
  - 保存为文本文件，以便 C++ 网络读取

- ③ 实现层间连接，进而实现完整网络拓扑
  - 初始化网络各层，并加载上一步得到的参数
  - 在 PyTorch 实现逻辑的基础上，将网络各层按序连接

```
-----------------------------------------
                MobileNetV2
-----------------------------------------
 0: Conv2d(3, 32, 3, 2, 1, 1)
 1: ReLU6
 2: Conv2d(1, 32, 3, 1, 1, 32)
 3: ReLU6
 4: Conv2d(32, 16, 1, 1, 0, 1)
 5: Conv2d(16, 96, 1, 1, 0, 1)
 6: ReLU6
 7: Conv2d(1, 96, 3, 2, 1, 96)
 8: ReLU6
 9: Conv2d(96, 24, 1, 1, 0, 1)
10: Conv2d(24, 144, 1, 1, 0, 1)
11: ReLU6
12: Conv2d(1, 144, 3, 1, 1, 144)
13: ReLU6
14: Conv2d(144, 24, 1, 1, 0, 1)
15: ResidualAdd(Conv2d(96, 24, 1, 1, 0, 1))
```



```
85: ReLU6
86: Conv2d(1, 960, 3, 1, 1, 960)
87: ReLU6
88: Conv2d(960, 160, 1, 1, 0, 1)
89: ResidualAdd(ResidualAdd(Conv2d(576, 160,
90: Conv2d(160, 960, 1, 1, 0, 1)
91: ReLU6
92: Conv2d(1, 960, 3, 1, 1, 960)
93: ReLU6
94: Conv2d(960, 320, 1, 1, 0, 1)
95: Conv2d(320, 1280, 1, 1, 0, 1)
96: ReLU6
97: GlobalAveragePool
98: Linear(1280, 1000)
-----------------------------------------
```

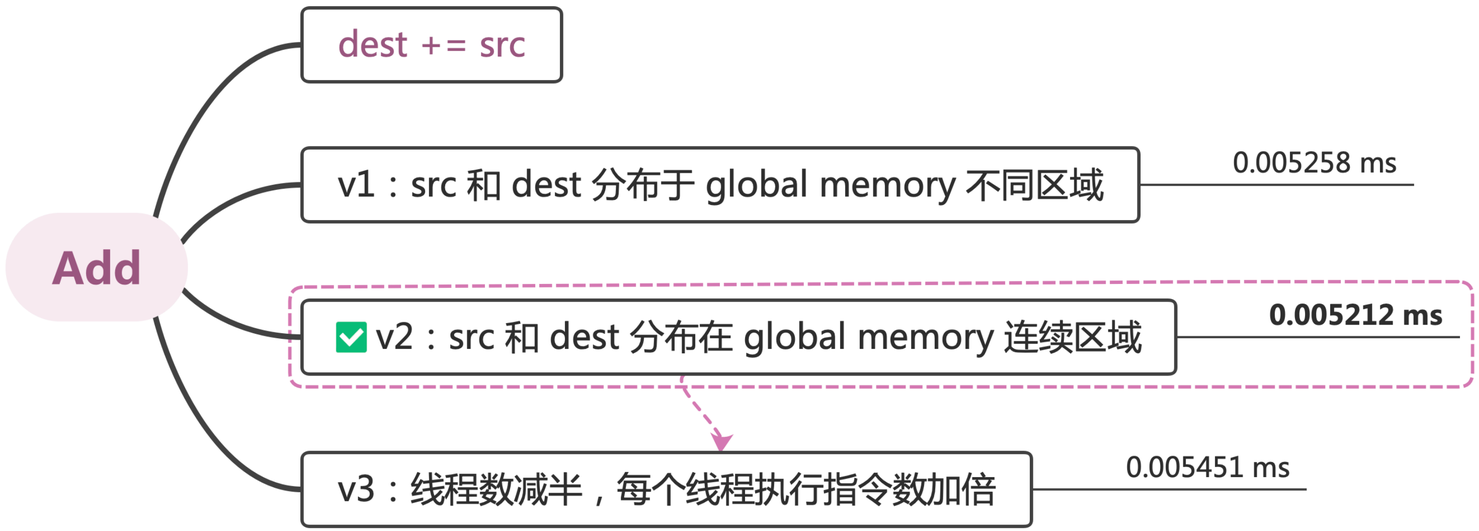
Step3　基于 CUDA Kernel 的网络推理实现

- 首先为每种算子编写 CUDA Kernel，并实现相应的 host 端运行的 wrapper 函数，单独设计测试用例
- 然后，基于 wrapper 函数、C++ 实现的网络拓扑，实现基于 CUDA Kernel 的网络推理

CUDA Kernel	对应的层
向量加 Add	ResidualAdd 层
卷积 Conv (Im2Col)	Conv2d + ReLU6 层
矩阵乘 Gemm	Linear 层
全局平均池化 Pool	GlobalAveragePool 层

Step3　基于 CUDA Kernel 的网络推理实现

向量加 Add

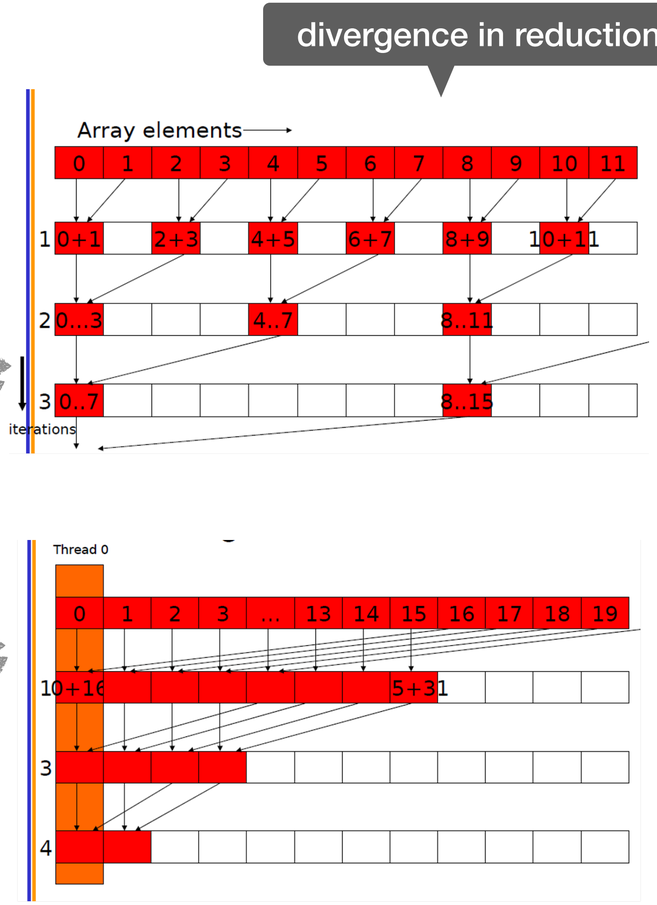


Step3　基于 CUDA Kernel 的网络推理实现

全局平均池化 Pool

GlobalAveragePool

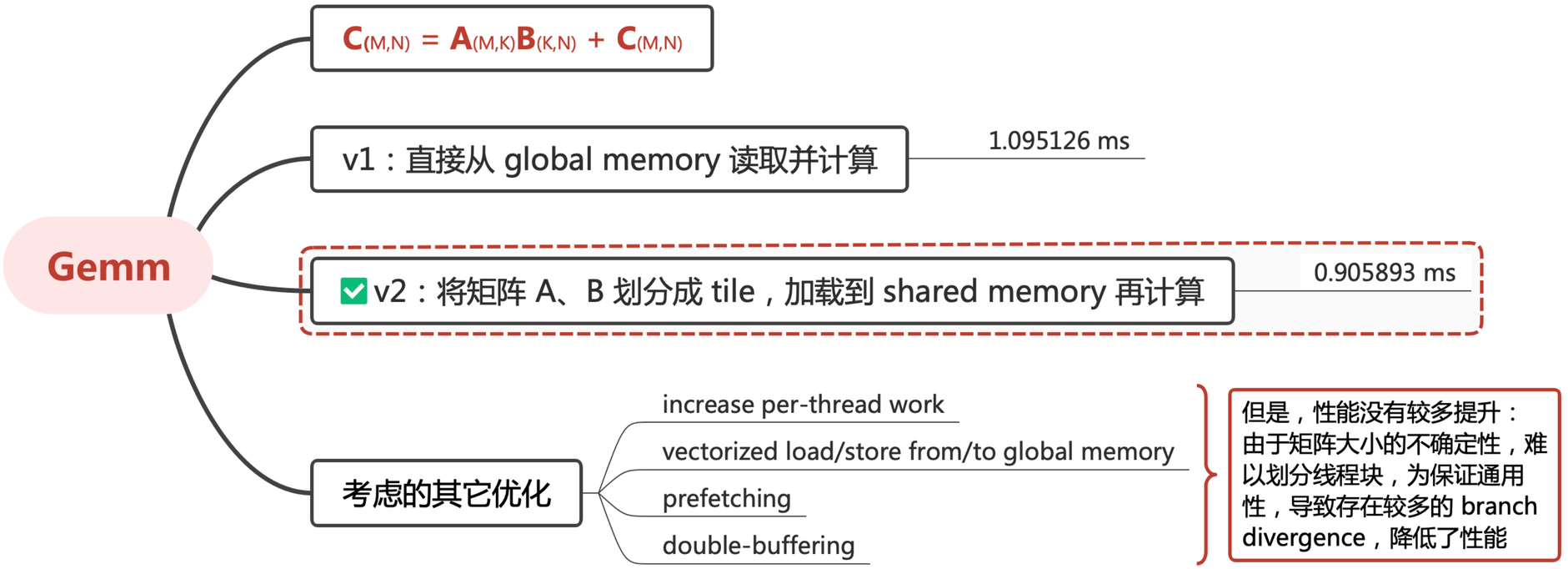
- **Y** = GAP(**X**)，本质是 parallel reduction 问题
- v1：在 window_size = X.h * X.w 大小的线程块内规约 (求和取平均)，在 shared memory 中计算 — 0.004946 ms
- v2：改变了 v1 索引 shared memory 的方式 (strided index) — 0.004317 ms
- v3：顺序寻址，减少 warp 内的 branch divergence — 0.004208 ms
- ✅ v4：当 stride = 32 (warp size) 时展开循环 — 0.003918 ms



Step3　基于 CUDA Kernel 的网络推理实现

矩阵乘 Gemm

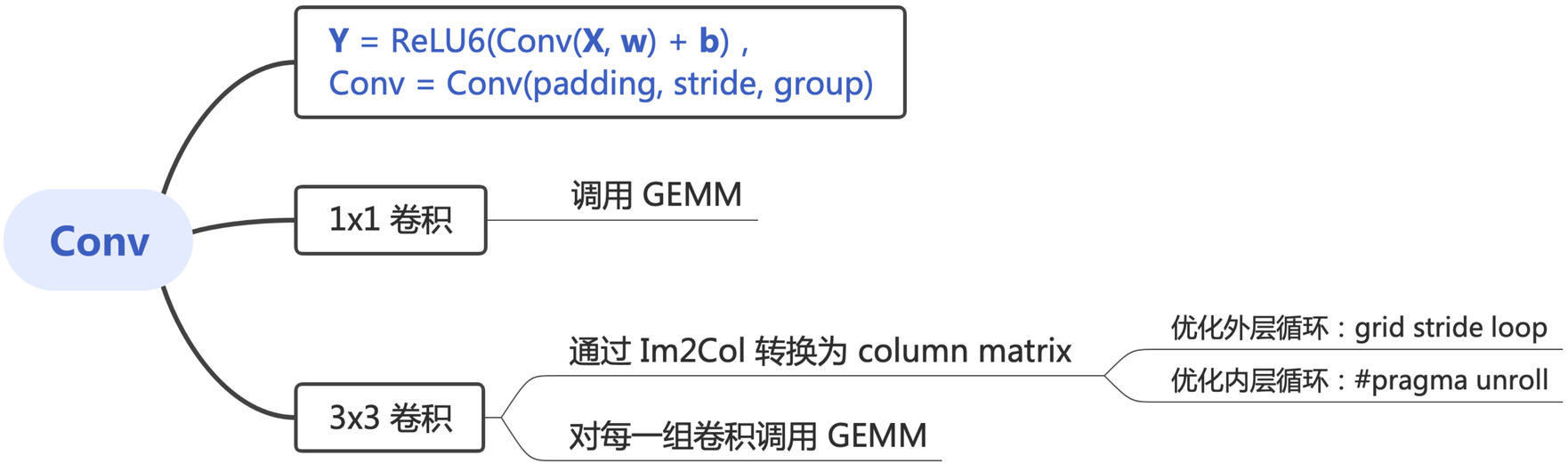
- 经验证，将矩阵组织为 column-major 可以大幅缩减计算时间



Step3　基于 CUDA Kernel 的网络推理实现

卷积 Conv

- 卷积计算的常用方法：① Im2Col + GEMM ② Winograd ③ FFT
  - Winograd 在该网络中存在一定局限性：实现较为复杂，不同参数的卷积实现有一定差别，且主要适用于 stride=1 的情况
  - FFT 主要适用于大卷积核，而网络中主要是 1x1 和 3x3 的卷积核



Step4　基于 CUDNN 的网络推理实现

- 封装 CUDNN 库函数，实现各种算子的 wrapper 函数，并单独设计测试用例

Wrapper 函数	对应的层	对应的 CUDNN 库函数
向量加函数	ResidualAdd 层	`cudnnAddTensor`
卷积函数	Conv2d + ReLU6 层	`cudnnConvolutionBiasActivationForward` 和 `cudnnActivationForward`
矩阵乘函数	Linear 层	`cudnnConvolutionForward` (看成 1x1 卷积) 或 `cublasGemmEx` (使用 CUBLAS)
全局平均池化函数	GlobalAveragePool 层	`cudnnPoolingForward`

- 基于封装好的 wrapper 函数、C++ 实现的网络拓扑，实现基于 CUDNN 的网络推理
- 作为基准，衡量基于我们编写的 CUDA Kernel 的网络推理性能

参考文献

[1] Sandler, Mark, et al. "Mobilenetv2: Inverted residuals and linear bottlenecks." *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*. 2018.

[2] NVIDIA Corporation. "NVIDIA cuDNN Documentation." *available at: https://docs.nvidia.com/deeplearning/cudnn/api/index.html*

[3] NVIDIA Corporation. "NVIDIA cuBLAS Documentation." *available at: https://docs.nvidia.com/cuda/cublas/index.html*

[4] Lavin, Andrew, and Scott Gray. "Fast algorithms for convolutional neural networks." *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*. 2016.

[5] Mark Harris. "CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops." *available at: https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/*

[6] Mark Harris. "Optimizing Parallel Reduction in CUDA." *available at: https://vuduc.org/teaching/cse6230-hpcta-fa12/slides/cse6230-fa12--05b-reduction-notes.pdf*