LiveVideoStackCon 音视频技术大会

2020 北京

多媒体开启 新视界

MULTIMEDIA BRIDGE TO A WORLD OF VISION

入门TensorRT

加速基于深度学习模型的视频处理

季光

NVIDIA Devtech China

重新认识GPU

- GPU->GPGPU
  - 最早用于图形加速(Graphics Processing)
  - 后来用于通用计算(General-Purpose)加速
- GPU的并行计算能力强
  - 乘加器多
  - 访存带宽高
- GPU内包括多个流处理器(Streaming Processor)，简称SM

Streaming Processor

- SM是相对完整的计算单元
  - 带指令调度器
  - 带访存单元
  - 带L1 cache
- 每个SM含有众多Fused Multiply-Add单元
  - 红框部分2时钟周期可完成32个fp32 FMA
  - 折合单时钟周期16个fp32 FMA
- 每个SM含有多个Tensor Core
  - 8时钟周期内完成16x8与8x8矩阵乘
  - 折合单时钟周期256个fp32 OP

GA102 (GTX 3090/Quadro A6000/NVIDIA A40)

GPU的算力

- FP32 (CUDA Core)

  128 (fma/clk) x 2 (op/fma)

  x 1.4 GHz x 84

  = 358 GFLOPS x 84

  ~= 30 TFLOPS

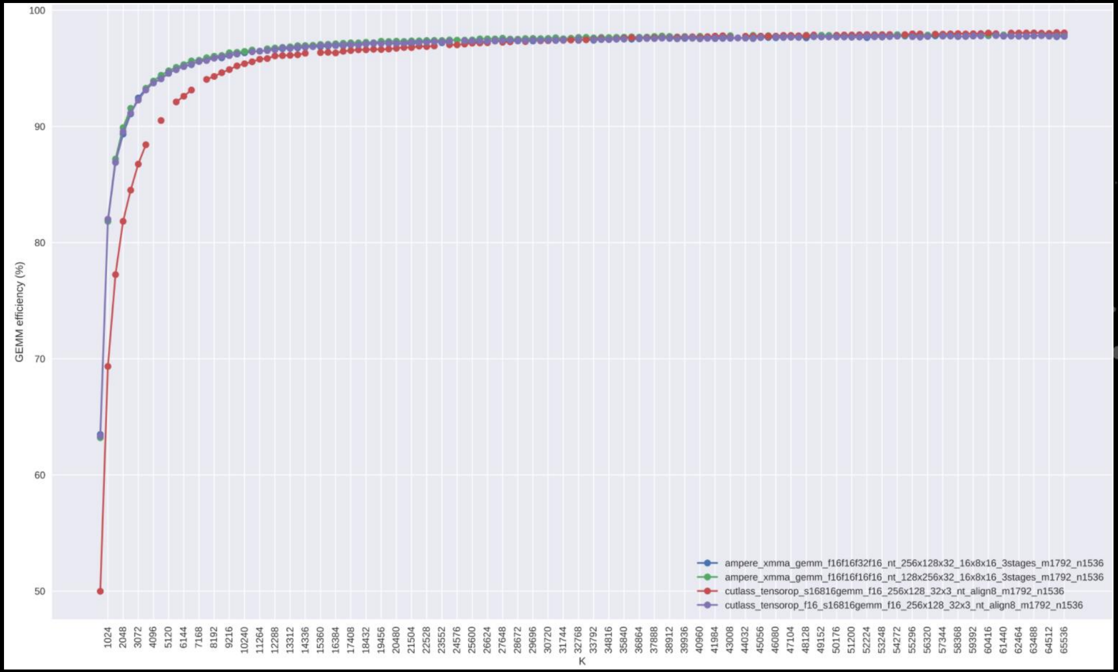
- FP16 (Tensor Core)

  256 (op/TC/clk) x 4 (TC/sm)

  x 1.4 GHz x 84

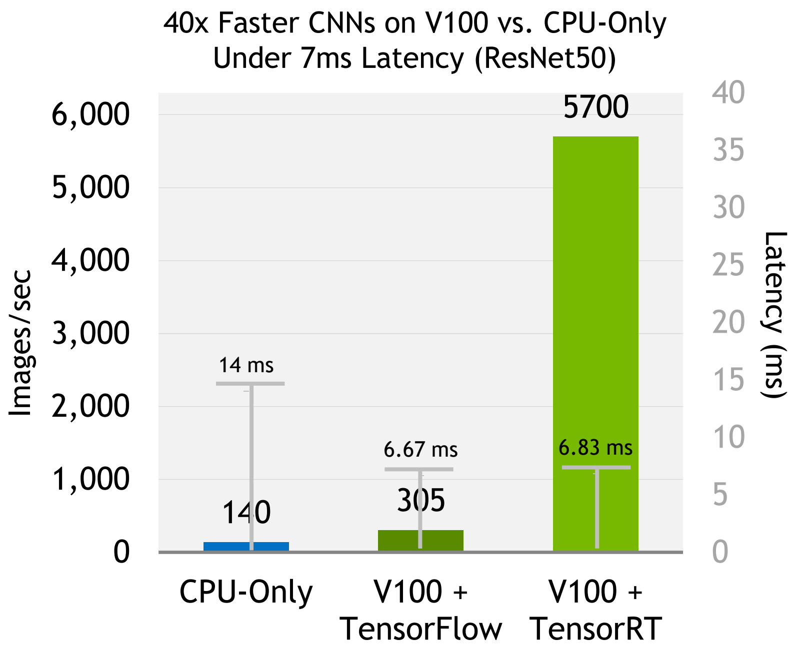
  ~= 120 TFLOPS

- SOL: Speed of Light

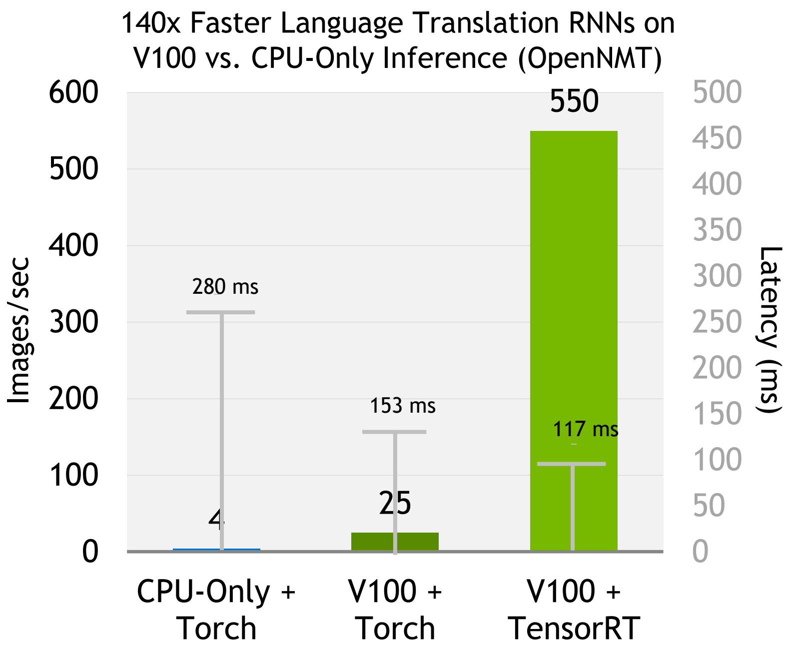


TensorRT: 助力实现模型推理的SOL

- TensorRT专门用于模型推理
  - 模型训练好之后，需要从TF/PyTorch/MXNet等DL框架迁移到TensorRT
- TensorRT所做的性能优化
  - 支持fp16/int8
    - 对数值进行精度转换与缩放，充分利用硬件的低精度高通量计算能力
  - 自动选取最优kernel
    - 矩阵乘法、卷积有多种CUDA实现方式，根据数据大小和形状自动选取最优实现
  - 计算图优化
    - 通过kernel融合、减少数据拷贝等手段，生成DNN的优化计算图



Inference throughput (images/sec) on ResNet50. **V100 + TensorRT**: NVIDIA TensorRT (FP16), batch size 39, Tesla V100-SXM2-16GB, E5-2690 v4@2.60GHz 3.5GHz Turbo (Broadwell) HT On **V100 + TensorFlow**: Preview of volta optimized TensorFlow (FP16), batch size 2, Tesla V100-PCIE-16GB, E5-2690 v4@2.60GHz 3.5GHz Turbo (Broadwell) HT On. **CPU-Only:** Intel Xeon-D 1587 Broadwell-E CPU and Intel DL SDK. Score doubled to comprehend Intel's stated claim of 2x performance improvement on Skylake with AVX512.



Inference throughput (sentences/sec) on OpenNMT 692M. **V100 + TensorRT**: NVIDIA TensorRT (FP32), batch size 64, Tesla V100-PCIE-16GB, E5-2690 v4@2.60GHz 3.5GHz Turbo (Broadwell) HT On. **V100 + Torch**: Torch (FP32), batch size 4, Tesla V100-PCIE-16GB, E5-2690 v4@2.60GHz 3.5GHz Turbo (Broadwell) HT On. **CPU-Only:** Torch (FP32), batch size 1, Intel E5-2690 v4@2.60GHz 3.5GHz Turbo (Broadwell) HT On

使用TensorRT的三种方法

- 调用框架内部集成的TRT: TF-TRT/TRTorch/MXNet-TensorRT
  - 易于使用
  - 未达到最佳效率
- ONNX Parser：现有框架->ONNX->TRT
  - 效率较好
  - 可能导出或导入环节遭遇失败
- API搭建：使用C++/Python TRT API逐层搭建网络
  - 适应性最强，效率最高
  - 难度最高

ONNX Parser: 以ESRGAN为例

```
#在此之前加载PyTorch模型
dummy_input = torch.randn(1, 3, 270, 480, device='cuda')
input_names = ['input']
output_names = ['output']
torch.onnx.export(model, dummy_input, "esrgan.onnx",
     verbose=True, opset_version=11,
     dynamic_axes={"input": [0, 2, 3]},
     input_names=input_names, output_names=output_names)
```

```
#TensorRT/bin/trtexec --onnx=esrgan.onnx --saveEngine=esrgan.trt \
  --optShapes=input:1x3x128x128 \
  --minShapes=input:1x3x32x32 \
  --maxShapes=input:2x3x270x480 --fp16 --verbose
```

请关注GTC China 2020 Talk: Best Practices of TensorRT ONNX Parser

API搭建: 基本框架

```python
import tensorrt as trt

logger = trt.Logger(trt.Logger.VERBOSE)
builder = trt.Builder(logger)
builder.max_workspace_size = 1 << 30

network = builder.create_network()
inputLayer = network.add_input("input", trt.DataType.FLOAT, (c, h, w))
# ...
# Add network layers
# ...
network.mark_output(outputLayer.get_output(0))

engine = builder.build_engine(network)
context = engine.create_execution_context()
context.execute_async(batch_size=n, bindings=[d_input, d_output])
```

API搭建: 以EDVR为例

- 导出weights

```
#net为网络模型，并且已调用torch.load()加载了weights
weights = {name : param.cpu().detach().numpy()
                  for name, param in net.named_parameters()}
import numpy as np
np.savez('edvr.npz', **weights)
```

- 搭建简单层

```
num_feat = 128
conv_first = network.add_convolution(x, num_feat, (3, 3),
         weights['conv_first.weight'], weights['conv_first.bias'])
conv_first.stride = (1, 1)
conv_first.padding = (1, 1)
print('conv_first', conv_first.get_output(0).shape)
```

API搭建: 利用函数与循环结构

```python
def make_layer(network, weights, last_layer, num_feat, n_block, prefix):
    for i in range(n_block):
        conv = network.add_convolution(last_layer.get_output(0), num_feat, (3, 3),
            weights[prefix + f'.{i}.conv1.weight’],
            weights[prefix + f'.{i}.conv1.bias'])
        conv.stride = (1, 1)
        conv.padding = (1, 1)
        last_layer = network.add_activation(conv.get_output(0),
            tensorrt.ActivationType.RELU)
    return last_layer

feat_l1 = make_layer(network, weights, feat_l1, num_feat, 5, 'feature_extraction')
```

API搭建: 自己编写TensorRT Plugin

- 如果TensorRT的层或层的组合得不到我想要的计算：编写Plugin
  - 必须用C++
  - cuBLAS/CUDNN/NPP/CUDA C实现计算过程
  - 填写Plugin接口
  - 填写Plugin Creator接口，用宏(全局对象)注册Creator
- 与Python的互操作
  - 把Plugin编译成动态链接库
  - Python脚本加载动态链接库，Plugin Creator被自动注册到TRT Plugin库
  - Python脚本里调用TRT Plugin的通用构造函数

API搭建: 把PyTorch Extension移植为Plugin

- PyTorch Extension
  - 包装了C++函数与CUDA kernel，可在PyTorch中调用
- 移植目标：原始代码尽量不改，新代码尽量简洁
- 要点
  - Python脚本要执行torch.cuda.init()
  - Plugin的enqueue 函数里，用torch::from_blob把CUDA指针转换成at::Tensor
  - 注意编译与链接选项

API搭建: Shape Tensor

- TensorRT 5及以前：不支持动态形状的输入
  - 只有batch维可变，在构建时数据不指定batch维
  - 运行时指定batch大小
  - 这种模型称作implicit batch
- TensorRT 6及以后：支持动态形状的输入
  - 构建时：input tensor可指定某维大小为-1
  - 运行时：需先为context指定具体input tensor形状
  - 需与explicit batch联合使用，即数据必须指定batch维

API搭建: 若使用Shape Tensor，构建时须指定batch大小

<table>
<tr><td colspan="2" rowspan="2">产品经理请你出来走两步</td><td colspan="2">构建时是否指定数据的batch大小</td></tr>
<tr><td>否
(implicit batch)</td><td>是
(explicit batch)</td></tr>
<tr><td rowspan="2">输入数据有没有-1维</td><td>无
(静态)</td><td>正确用法</td><td>batch大小将无法改变</td></tr>
<tr><td>有
(动态)</td><td>构建会失败</td><td>正确用法</td></tr>
</table>

- Shape Tensor用于获取、操作Tensor的形状
- Shape Tensor也可成为网络的输入与输出

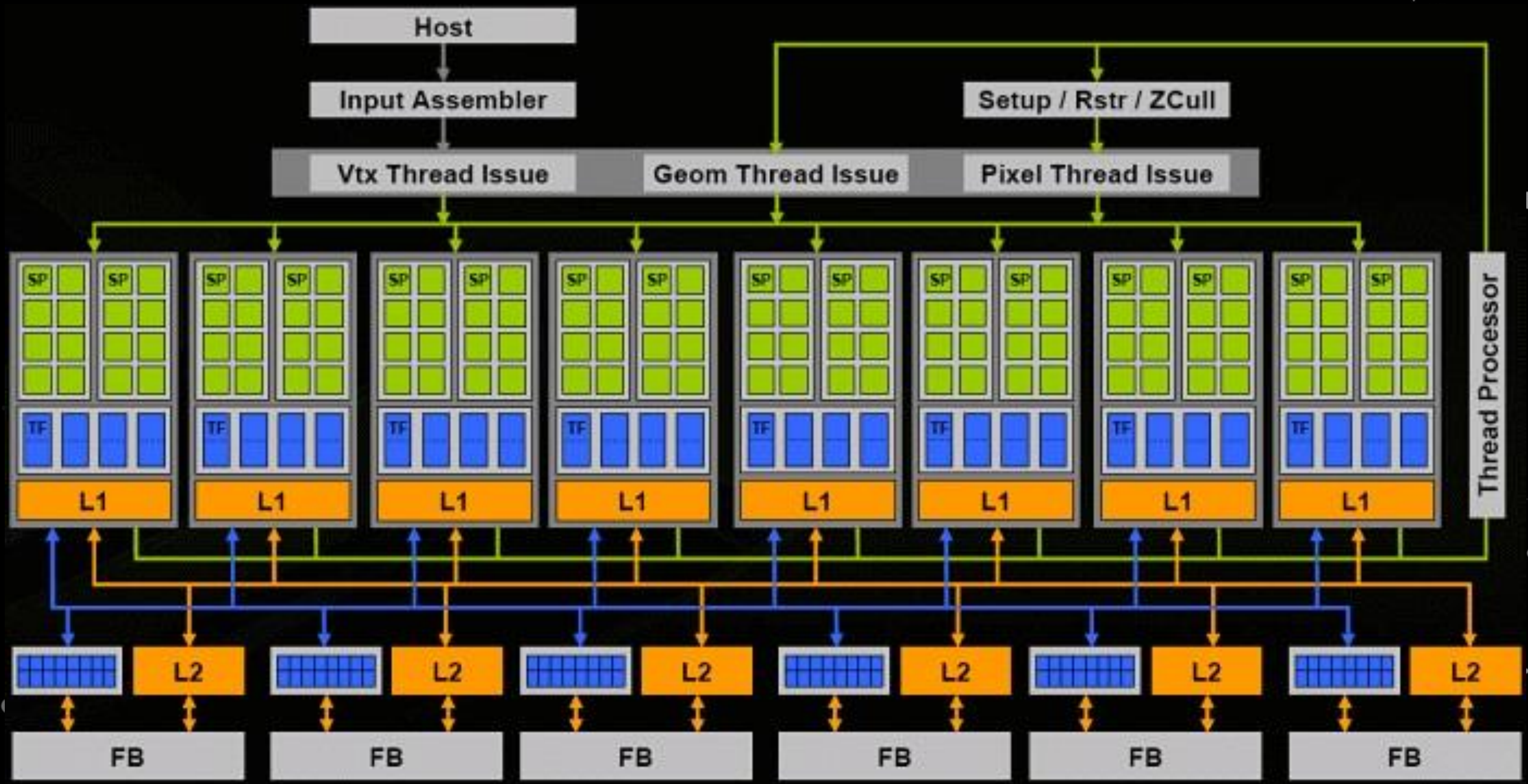
API搭建: 总结要点

- 首先简化模型
  - 如果输入形状可变，先选定一个固定形状
  - 如果需要plugin，先用简单计算替换
- 渐进式搭建：先搭建基本结构，再逐步完善
  1. 搭建基本网络结构，保证中间结果的形状正确（可初步评测加速效果）
  2. 逐层比对中间结果，修正网络各层设置
  3. 实现plugin，实现固定形状的正确计算（可较准确评测加速效果）
  4. 利用shape tensor支持动态形状，实现全功能
- 补充测试代码，方便对比中间结果
  - TRT: 自动读取engine的输出数量及形状，并自动分配缓冲区
  - TF: Eager Execution

GPU探索之路

- 更灵活方便地编程：ONNX Parser + Plugin
    - 混用Parser/API搭建/CUDA C
- 更快地跑程序：深度优化TRT模型
    - 自己写定制kernel：CUDA C
    - 寻找热点：Nsight Sytems
    - 优化kernel：Nsight Compute

1	ExecutionContext::enqueue [70.651 ms]	5.54196s	70.651 ms
379	(Unnamed Layer* 290) [PluginV2IOExt] [1.263 ms]	5.57241s	1.263 ms
391	(Unnamed Layer* 86) [PluginV2IOExt] [1.245 ms]	5.57629s	1.245 ms
371	(Unnamed Layer* 426) [PluginV2IOExt] [1.243 ms]	5.56983s	1.243 ms
367	(Unnamed Layer* 494) [PluginV2IOExt] [1.242 ms]	5.56854s	1.242 ms
383	(Unnamed Layer* 222) [PluginV2IOExt] [1.239 ms]	5.57372s	1.239 ms
470	(Unnamed Layer* 507) [PluginV2IOExt] [1.239 ms]	5.58861s	1.239 ms
478	(Unnamed Layer* 235) [PluginV2IOExt] [1.238 ms]	5.5937[illegible]	1.238 ms
387	(Unnamed Layer* 154) [PluginV2IOExt] [1.235 ms]	5.57501s	1.235 ms
375	(Unnamed Layer* 358) [PluginV2IOExt] [1.235 ms]	5.57113s	1.235 ms
480	(Unnamed Layer* 167) [PluginV2IOExt] [1.234 ms]	5.59498s	1.234 ms
482	(Unnamed Layer* 99) [PluginV2IOExt] [1.234 ms]	5.59625s	1.234 ms
476	(Unnamed Layer* 303) [PluginV2IOExt] [1.233 ms]	5.59243s	1.233 ms
474	(Unnamed Layer* 371) [PluginV2IOExt] [1.230 ms]	5.59116s	1.230 ms
472	(Unnamed Layer* 439) [PluginV2IOExt] [1.229 ms]	5.58989s	1.229 ms
504	(Unnamed Layer* 514) [Reduce] [1.171 ms]	5.59985s	1.171 ms
615	(Unnamed Layer* 718) [Convolution] [611.234 µs]	5.61028s	611.234 µs
10	(Unnamed Layer* 12) [Convolution] + (Unnamed Lay...	5.54518s	574.660 µs
14	(Unnamed Layer* 20) [Convolution] + (Unnamed Lay...	5.54744s	573.572 µs
12	(Unnamed Layer* 16) [Convolution] + (Unnamed Lay...	5.54631s	573.060 µs
6	(Unnamed Layer* 4) [Convolution] + (Unnamed Laye...	5.54291s	572.644 µs
8	(Unnamed Layer* 8) [Convolution] + (Unnamed Laye...	5.54404s	571.396 µs
9	(Unnamed Layer* 10) [Convolution] + (Unnamed Lay...	5.54461s	564.100 µs
7	(Unnamed Layer* 6) [Convolution] + (Unnamed Laye...	5.54348s	559.717 µs
13	(Unnamed Layer* 18) [Convolution] + (Unnamed Lay...	5.54688s	556.037 µs
5	(Unnamed Layer* 2) [Convolution] + (Unnamed Laye...	5.54235s	555.205 µs
11	(Unnamed Layer* 14) [Convolution] + (Unnamed Lay...	5.54575s	554.661 µs
501	(Unnamed Layer* 512) [Convolution] [553.316 µs]	5.59889s	553.316 µs
455	(Unnamed Layer* 95) [Slice] [520.711 µs]	5.58312s	520.711 µs
467	(Unnamed Layer* 503) [Slice] [517.991 µs]	5.58779s	517.991 µs
459	(Unnamed Layer* 231) [Slice] [517.638 µs]	5.58468s	517.638 µs
463	(Unnamed Layer* 367) [Slice] [517.382 µs]	5.58623s	517.382 µs
457	(Unnamed Layer* 163) [Slice] [517.223 µs]	5.5839s	517.223 µs
465	(Unnamed Layer* 435) [Slice] [517.031 µs]	5.58701s	517.031 µs
461	(Unnamed Layer* 299) [Slice] [516.199 µs]	5.58546s	516.199 µs
16	(Unnamed Layer* 30) [Slice] [458.345 µs]	5.54818s	458.345 µs
32	(Unnamed Layer* 373) [Slice] [457.258 µs]	5.5512s	457.258 µs
26	(Unnamed Layer* 237) [Slice] [457.033 µs]	5.55018s	457.033 µs
35	(Unnamed Layer* 441) [Slice] [457.002 µs]	5.55172s	457.002 µs
23	(Unnamed Layer* 169) [Slice] [456.969 µs]	5.54966s	456.969 µs
29	(Unnamed Layer* 305) [Slice] [456.938 µs]	5.55069s	456.938 µs
20	(Unnamed Layer* 101) [Slice] [455.946 µs]	5.54915s	455.946 µs
18	(Unnamed Layer* 33) [Slice] [455.369 µs]	5.54867s	455.369 µs



Host
Input Assembler
Setup / Rstr / ZCull
Vtx Thread Issue
Geom Thread Issue
Pixel Thread Issue
SP
TF
L1
Thread Processor
L2
FB

谢谢

季光
gji@nvidia.com