PyramidDNN 模型分析（training)

现状

Paddle 比竞品跑整个模型要慢 25%左右

跑 3 个 EPOC，根据 vtune log, Paddle 花了 747s, 竞品花了 610s, 即 Paddle 多花了 747-610=137s, 慢了 137/610=22.4%

分析

多花了 137s, 时间都去哪儿了？

经过比较 vtune log 里的时间绝对值，我们认为多出来的时间(137s)来自 3 个方面：

1. 35s(5.7%): Paddle 使用 Python Reader（单独线程）读取数据，主线程也是在 Python 层接收数据，然后调用 C .so 进行 executor run。在主线程中 Python 和 C 之间会比较多的切换；而竞品的 Python 程序（线程）只是启动，主线程（executte_native_thread_routine)完全运行 C 代码（包括 data reader, optimizer, 等等), 没有 C/Python 间的切换

2. 55s (9.0%): Paddle 的 CPUSearchPrymidHashOPKernel（前向）实现，比竞品对应的 BatchPyramidHashLayer(前向）实现，要慢 60s （两者反向的性能基本一致）。而在 60s 里，有 36s 是因为 Paddle 里的 memcpy 比竞品里的__memcpy_sse2_unligned 多花的

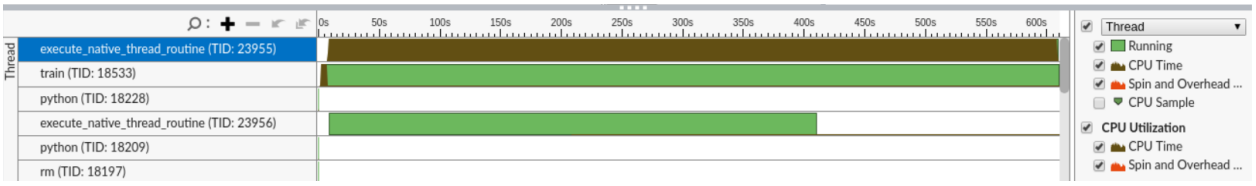
3. 剩余 47s(7.7%): 这个主要是图的差异造成的。竞品把 Embedding 和 SeqPool(sum)合在一起，并且用稀疏矩阵乘做了实现，比 Paddle 里单独的 LookupTable OP + SeqPool OP 优化了很多。这个 Intel team 正在做优化。

下面是具体分析：

线程及 Python/C 边界

竞品和 Paddle 进程，都包含了数十个线程（竞品 35 vs Paddle 48), 但有效线程（实际干活的）只有少数 3 到 4 个。

竞品:

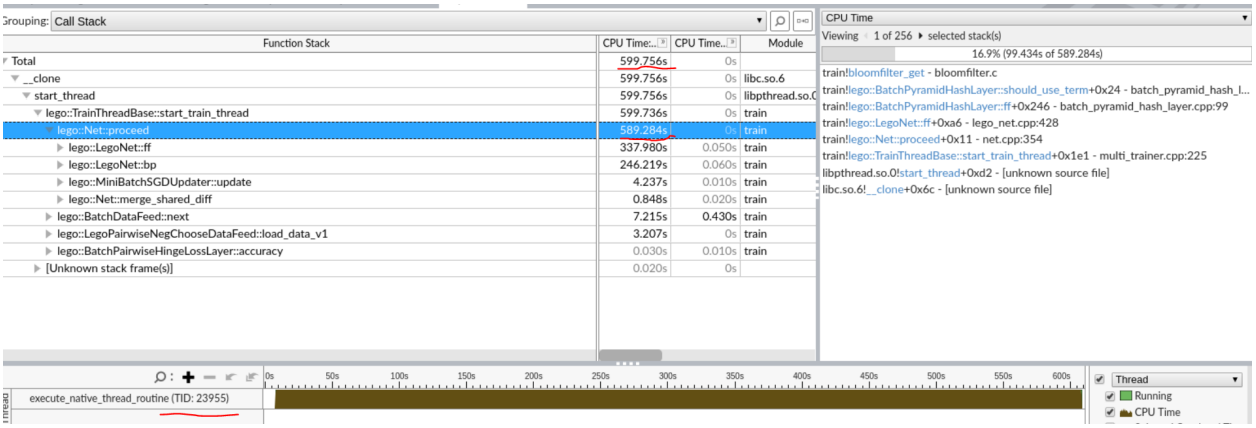


可以看到，竞品的 Python 线程（TID:18228)会启动 C 的线程（TID:18533)并进而启动主线程（TID:23955)和另一个辅助线程（TID:23956, 貌似用于 write something, 几乎不占 CPU).

ElapsedTime (610s)的绝大多数时间里，主线程在一个 CPU core 上全速运行。其它 3 个线程要么根本不被调度（Python 线程），要么几乎不占 CPU 时间（TID 18533 和辅助线程 23956)

注：图中绿色表示 Running(即线程被调度了），棕色表示实际占用 CPU core 的比例。

让我们来近距离观察一下竞品的主线程（Top-down view with Filter)

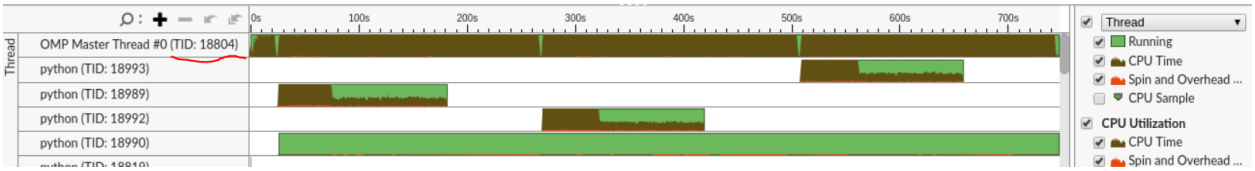


用于算子运算（竞品::Net::Proceed)的时间为 589s, 和总的 Elapsed Time 相比，差距为 610s - 589s = 21s. 在这 21s 里：

- 10s 为从 Python 线程启动到该主线程启动的时间（即 610s - 599.756s = 10s)

- 另 10s 为该主线程中用于 Data loading 的时间（竞品::BatchDataFeed::next, 7.215s; …）

Paddle:

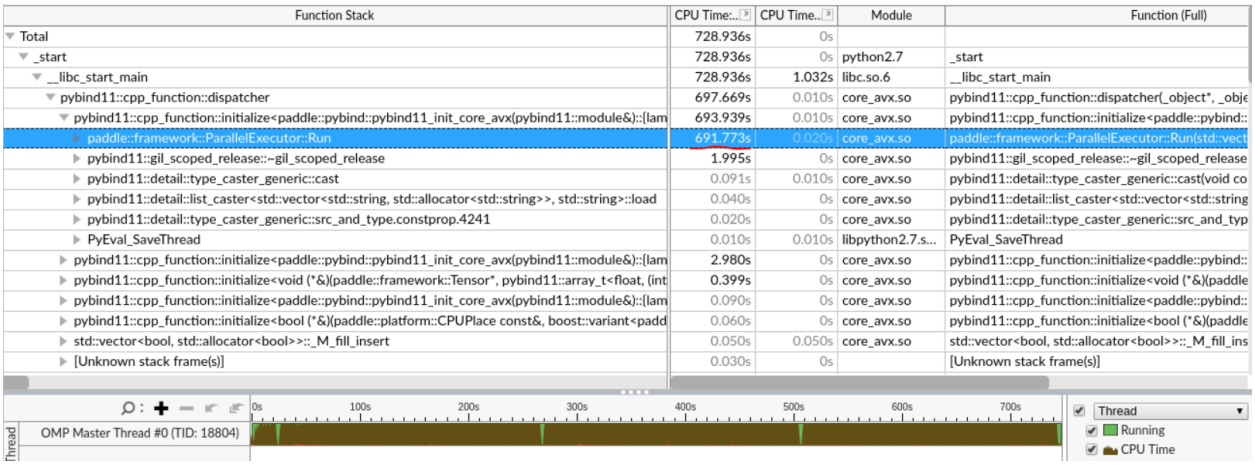


Paddle 这里有三类线程：

- TID 18990: 这是由主线程中 data_feed.py 创建出来用于管理 data feed 线程的辅助线程，大部分时间都是在运行但 wait(都是绿的），基本不占 CPU（没有棕色）
- TID 18989/18992/18993: 这三个线程是按需被创建（并销毁）的，实际用于 data load 并 feed 给主线程的。它们每个平均存活了 150s 左右，在存活过程中是一直活跃运行的（前 50s 基本占了一个 CPU 核 100%，后 100s 也占了 20%~30%左右）
- TID 18804: 主线程

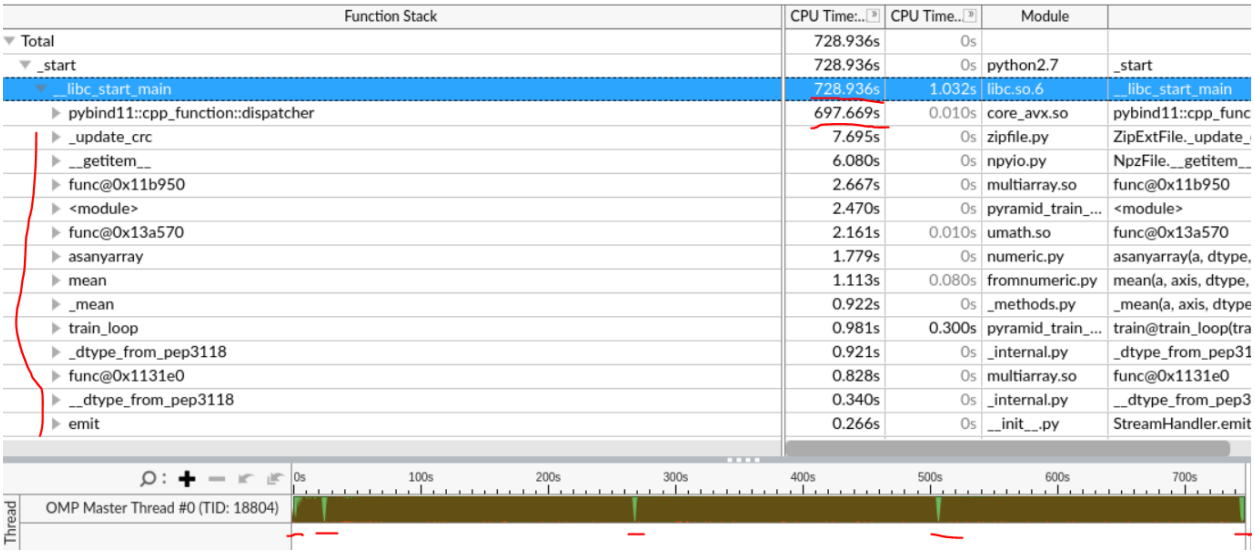
可以看到，每当要启动一个 data feed 线程加载数据时，主线程会有一个明显的等待（注意上图中主线程 bar 上面 3 个绿色的豁口），虽然这个等待并不花多长时间（每个豁口在 2~3s 左右）。data feed 线程会持续加载更多的数据(buffering)用于之后不断 feed 给主线程。

让我们来近距离观察一下 Paddle 的主线程（Top-down view with Filter)



用于算子运算的时间为 691.773s (paddle::framework::ParallelExecutor::Run), 和总的 Elapsed Time 747s 相比，差距为 56s. 这个和竞品 （21s）相比，是比较大的，即 **Paddle 里的“非算子运算”时间较长。**

我们再近距离看看这 56s:



Function Stack	CPU Time:...	CPU Time...	Module	
Total	728.936s	0s		
_start	728.936s	0s	python2.7	_start
__libc_start_main	728.936s	1.032s	libc.so.6	__libc_start_main
pybind11::cpp_function::dispatcher	697.669s	0.010s	core_avx.so	pybind11::cpp_func
_update_crc	7.695s	0s	zipfile.py	ZipExtFile._update_
__getitem__	6.080s	0s	npyio.py	NpzFile.__getitem__
func@0x11b950	2.667s	0s	multiarray.so	func@0x11b950
<module>	2.470s	0s	pyramid_train_...	<module>
func@0x13a570	2.161s	0.010s	umath.so	func@0x13a570
asanyarray	1.779s	0s	numeric.py	asanyarray(a, dtype,
mean	1.113s	0.080s	fromnumeric.py	mean(a, axis, dtype,
_mean	0.922s	0s	_methods.py	_mean(a, axis, dtype
train_loop	0.981s	0.300s	pyramid_train_...	train@train_loop(tra
_dtype_from_pep3118	0.921s	0s	_internal.py	_dtype_from_pep31
func@0x1131e0	0.828s	0s	multiarray.so	func@0x1131e0
__dtype_from_pep3118	0.340s	0s	_internal.py	__dtype_from_pep3
emit	0.266s	0s	__init__.py	StreamHandler.emit

- 18s (747s - 728.9s) 为主线程等待（例如 wait for data feed 线程)的时间，见上图中那些绿色的豁口
- 31s（728.9s - 697.669s)为主线程中 Python 级别运行的代价，见上图中的红色竖线。例如 zipfile.py/_update_crc 7.695s; npio.py/_getitem_ 6.080s。
- 7s (697.669s - 691.733s)为 python 到 C 间 pybind 的代价及其它

所以，“**Paddle 里的“非算子运算”时间较长**”，原因是：

- 竞品所有的“非算子”逻辑（例如 data reader)也都是以 C 方式实现在主线程里，Python 代码只在起始阶段运行一下（python 线程 TID:18228）。主线程开始运行后无需等待任何其它线程，也不运行任何 Python 代码
- Paddle 的“非算子”逻辑一部分实现在其它线程里（如 data feed 线程），一部分以 Python 代码形式实现在主线程里。这部分在主线程里运行的 Python 代码，效率差（31s)，且增加了 Pybind 的代价（7s)

核心算子（CPUSearchPrymidHashOPKernel）

从 vtune log 很容易看出，PyramidHash OP 的前向 kernel 是最耗时，也是竞品/Paddle 差异最大的 kernel.

竞品:

Function Stack	CPU Time:...	CPU Time...	Module	
start_thread	599.756s	0s	libpthread.so.0	start_thread
lego::TrainThreadBase::start_train_thread	599.736s	0s	train	lego::TrainThreadBase::start
lego::Net::proceed	589.284s	0s	train	lego::Net::proceed(void)
lego::LegoNet::ff	337.980s	0.050s	train	lego::LegoNet::ff(void)
lego::BatchPyramidHashLayer::ff	237.632s	2.308s	train	lego::BatchPyramidHashLay
lego::BatchPyramidHashLayer::should_use_term	119.222s	0.610s	train	lego::BatchPyramidHashLay
bloomfilter_get	118.612s	99.434s	train	bloomfilter_get
murmurhash3_x64_128	19.178s	13.551s	train	murmurhash3_x64_128
lego::BatchPyramidHashLayer::hash_embedding_ff	114.524s	10.991s	train	lego::BatchPyramidHashLay
__memcpy_sse2_unaligned	78.517s	78.517s	libc.so.6	__memcpy_sse2_unaligned
XXH32_endian_align.constprop.3	15.693s	15.693s	train	XXH32_endian_align.constp
XXH32	9.153s	9.153s	train	XXH32
[Import thunk memcpy]	0.170s	0.170s	train	[Import thunk memcpy]
std::vector<int, std::allocator<int>>::push_back	0.539s	0s	train	std::vector<int, std::allocato
rand_r	0.360s	0.360s	libc.so.6	rand_r
count<__gnu_cxx::__normal_iterator<int*, std::vector<int> >, int>	0.259s	0.259s	train	count<__gnu_cxx::__normal
vector	0.080s	0s	train	vector

Paddle:

Function Stack	CPU Time:...	CPU Time...	Module
paddle::framework::OperatorWithKernel::RunImpl	685.503s	0.070s	core_avx.so
paddle::framework::OperatorWithKernel::RunImpl	679.726s	0.210s	core_avx.so
std::_Function_handler<void (paddle::framework::ExecutionContext co	292.891s	0.030s	core_avx.so
paddle::operators::CPUSearchPyramidHashOPKernel<paddle::platfo	289.336s	16.140s	core_avx.so
bloomfilter_get	141.179s	113.838s	core_avx.so
murmurhash3_x64_128	26.790s	16.865s	core_avx.so
[Import thunk murmurhash3_x64_128]	0.550s	0.550s	core_avx.so
memcpy	114.592s	114.592s	libc.so.6
XXH32_finalize.constprop.9	8.847s	8.847s	core_avx.so
[Import thunk memcpy]	5.011s	5.011s	core_avx.so
murmurhash3_x64_128	0.910s	0.910s	core_avx.so
[Import thunk XXH32]	0.869s	0.869s	core_avx.so
rand_r	0.250s	0.250s	libc.so.6
std::_Sp_counted_base<(__gnu_cxx::_Lock_policy)2>::_M_release	0.179s	0.030s	core_avx.so
paddle::framework::Tensor::dims	0.170s	0.170s	core_avx.so
paddle::framework::Tensor::mutable_data<float>	0.150s	0s	core_avx.so
paddle::framework::Tensor::mutable_data<int>	0.148s	0.010s	core_avx.so

可以看出，Paddle 比竞品要慢 55s（292.891s - 237.632s). 具体看：

- Paddle 的 memcpy 耗时 114.592s, 比竞品的__memcpy_sse2_unaligned(耗时 78.517s)，要多花了 35s
- Paddle 的 bloomfilter_get (141.179s)比竞品的 bloomfilter_get (118.612s)，要多花了 23s

建议：

- _memcpy_sse2_unaligned 只是 memcpy 的一个具体实现。竞品/Paddle 性能差异这么大，要么是两者 copy 的数据量不一样，要么是 Paddle 的 memcpy 执行时走到的非_memcpy_sse2_unaligned(而是一个性能更差的分支）