Deeplearning4j的本机CPU优化

本页指南将介绍在DL4J和ND4J上调试或提升CPU系统性能的几种方法。让我们先来定义一些术语：

OpenMP

OpenMP是一个开源的并行编程API，支持C/C++/Fortran语言。ND4j使用以C++编写的后端，因此我们用OpenMP来改善CPU的并行计算性能。

CPU、内核、超线程

一个CPU通常是由多个内核组成的单个物理单元。每个内核都能独立处理指令，彼此互不依赖。每个内核也都采用超线程技术，在此类系统中显示为额外的一组内核。

举例而言，假设您安装的是英特尔i7-4790 CPU，这就是一个物理CPU、四个物理内核，共八条线程。

如果您用的是英特尔®至强®双处理器E5-2683 v4系统，那么这就是两个物理CPU，各有16个物理内核（共32个物理内核）以及32个虚拟内核（共64个内核）。AMD也有类似的架构，只不过命名方法略有不同。他们的层级是CPU -> 模块 -> 内核，但总体思路是一致的。

SIMD

MD是***单指令流多数据流***的缩写，这是一种以并行方式对不同数组/向量元素应用一些特定指令的并行计算模式。我们主要将SIMD用于内循环，或者因为太小而不必动用OpenMP的循环。

JVM vs 本机并行

按其设计，ND4J一般用一条JVM线程来执行操作，但如果任何特定的操作涉及到基本代码的

本机部分，那么该操作可以用一条以上的本机线程来加速执行。所以，除非您有非常特殊的硬件设置，否则不宜将JVM和本机并行模式结合起来同时使用。 理由很简单：

- CPU/GPU的计算能力受到各种限制：CPU的内核数量以及浮点运算单元数量，或者CUDA的多处理器数量、常驻线程块数量、共享内存和设备内存带宽。
- CPU缓存一致性：取决于您的工作流的模式，CPU缓存带来的性能提升可能要大得多，而解决问题所需的额外线程也会大大减少。
- PCIe带宽：限制CUDA性能的因素之一有可能是主机 -> GPU RAM的传输，因为PCIe带宽并不是无限的资源。

综上所述，对于不同的任务，取得最佳性能表现的方法也不同，具体的最佳方法始终取决于具体的任务。

并行计算与性能

很有可能出现的一种情况是，由于系统十分强大，如果不限制并行计算，反而会影响性能发挥。试想您有一个20核的CPU，但您的任务是对一个含有256个元素的数组进行线性运算。在这种情况下，一条并行线程也不必启动。这是因为，同样的运算用单个线程 + CPU的SIMD通道来进行速度更快，避免了启动和操作一条新线程的开销。

正因为如此，在启动新的线程之前需要评估每项运算的有效并行上限。

术语和基本概念都已介绍完毕，接下来让我们来探讨性能调试。

性能调试

OMP_NUM_THREADS

`OMP_NUM_THREADS`环境变量决定有多少条OpenMP线程将被用于BLAS调用和其他本机调用。ND4J会尝试估计该项参数的最佳值，但在某些情况下，自定义该项参数的值可能会带来更好的性能表现。一般而言，此处的“最佳值”是一个CPU的物理内核数量。但是请注意，该项参数设定的是启动的线程的最大数量和实际数量。任何一项单独的运算都有可能会（也多半确实会）低于该设定值。

这也就是说，如果您有一套使用超线程技术的双8核CPU系统，那么系统的内核总数将是32，而`OMP_NUM_THREADS`的最佳设定值则应是8。 如果您有一套使用超线程技术的四10核CPU系统，那么系统的内核总数将会是80，而`OMP_NUM_THREADS`的最佳设定值则应是10。 如果您有一套使用超线程技术的单16核CPU系统，那么系统的内核总数将会是32，而`OMP_NUM_THREADS`的最佳设定值则应在8到16之间，具体取决于实际工作量大小。

并行计算的阈值

如果您认为特定的CPU可以取得更好的性能，比如您的CPU支持AVX-152指令集，那么您可以尝试为不同的运算类型更改并行计算的阈值。为此可以采用我们发现的特殊方法：

```
1.  NativeOpsHolder.getInstance().getDeviceNativeOps().setElementThreshold
    (16384)
2.  NativeOpsHolder.getInstance().getDeviceNativeOps().setTADThreshold(64)
```

调用`.setElementThreshold()`可以指定一条OpenMP线程处理的数组元素数量。这也就是说，如果您的CPU支持AVX-512，那么您就可以将该项值设定得足够高，以避免生成过多线程，同时改用SIMD。

`.setTADThreshold()`也有类似的功能。它可以指定一条OpenMP线程处理的张量（TAD）数量。您可以按照CPU型号（以及CPU缓存容量）来提高或降低该项设定值。

英特尔MKL

ND4J默认使用OpenBLAS，但是ND4J/DL4J也可以选用性能顶尖的英特尔MKL计算库。这一选项是自动启用的。如果您的$PATH上有MKL库，它们就会被用于BLAS运算。英特尔MKL支持Linux、OSX和Windows。若您安装了英特尔MKL，也可以在编译ND4J时预先建立与MKL的链接。英特尔目前免费提供MKL的社区许可。

Spark环境

在分布式环境下，某些自定义设置可以起到帮助作用。

首先，您应当考虑将每个节点的执行器数量设为不包括超线程内核的值。如此节约的空间还能

用于本机运算的内部多线程安排。

此外在分布式环境下还可以考虑`OMP_NUM_THREADS`的值。由于Spark提供的并发Java线程数量会比较高，本机并行的线程数量应当减少至2～4条。

因此，开始性能调试时不妨先设定`OMP_NUM_THREADS=4`。

用源码构建

请注意：手动编译需要具备C/C++领域的知识技能。但是，如果您在用源码构建时对环境有充分的了解，那么还可以通过一些额外的选项来进一步提高性能：

- 向量化数学计算库 - 这类库可以使CPU SIMD能力更好地用于数学计算功能，机器学习领域普遍使用。
- 英特尔MKL可用libsvml
- glibc 2.22版以上的Linux可用libmvec
- -march=native

这项通用的编译优化可以让您按当前的硬件架构进行代码编译。在现代化处理器上，这通常可以起到改善向量化的作用。

CPU后端故障排除

ND4J_SKIP_BLAS_THREADS

如果您的BLAS环境不同寻常，或者在调用`Nd4jBlas.setMaxThreads()`前后出现问题 - 请将环境变量`ND4J_SKIP_BLAS_THREADS`设为任意值。如此一来，该方法就不会被触发，但您还必须手动设定`OMP_NUM_THREADS`变量。

回退模式

我们最近发现，几种主流的BLAS库在某些平台上可能会变得不稳定，在不同状况下导致系统崩溃。为了解决这一问题（以及未来可能出现的问题），我们提供了可选的“回退模式”（fallback mode），让ND4J采用内部解决方案，避开潜在问题。其作用相当于现代化

操作系统用户所熟知的“安全模式”。

若要激活回退模式，只需设定这一特殊的环境变量：`ND4J_FALLBACK`。请在启动应用程序之前将其设置为`true`或者`1`。这一变量可以在Apache Spark环境中使用，也可在一个独立的应用程序中使用。

具体运作方式

ND4J的本机后端以C++语言构建，内部采用OpenMP。其基本思路是隐式并行：单个JVM线程转变为运算调用过程中使用的可变数量的线程。

如此就简化了Java的流程及内存管理（亦即您始终确信是单个线程在访问特定的INDArray实例），同时具体运算采用OpenMP线程 + SIMD优化循环来改善性能。

我们采用两种类型的内部并行：

- 元素层级的并行：INDArray中的每个元素由独立的OpenMP线程或SIMD通道来处理。
- TAD层级的并行：每个OpenMP线程在原始操作数之内处理自己的张量。

作者：Vyacheslav Kokorin

原文地址

https://deeplearning4j.org/native

更多文档可以查看 https://github.com/sjsdfg/deeplearning4j-issues。
欢迎star