

OpenBLAS项目与矩阵乘法优化 | AI 研习社

本文作者：叨叨 2017-04-21 18:10

导语：PerfXLab 澎峰科技创始人分享

提起矩阵计算，学过《高等数学》的人可能都听过，但若不是这个领域的研究者，恐怕也只停在“听过”的程度。在矩阵计算领域，开源项目OpenBLAS影响巨大，除IBM、华为等巨头公司在使用外，还吸引了全球的研究院校、开发者们关注。

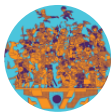
雷锋网 AI 研习社近日有幸邀请到了澎峰科技创始人、OpenBLAS项目创始人和主要维护者张先轶，他将为我们介绍OpenBLAS开源项目以及矩阵乘法的优化。

嘉宾介绍

张先轶，中国科学院博士，MIT博士后，OpenBLAS开源项目创始人和主要维护者，PerfXLab 澎峰科技创始人。曾获2016年中国计算机学会科技进步二等奖。



张先轶师从张云泉教授，在中科院取得高性能计算博士学位。读博期间，基于GotoBLAS的原有基础，他



叨叨
编辑

雷锋网北京编辑。关注人工智能，略杂。微信（yougo5654）可以找我。

发私信

- 当月热门文章
- 一记实锤后，苹果终于承认降低手机的性能
- 前员工们 diss 贾跃亭“三大罪状”，法拉第未来穷途末路了吗
- 语音交互中的“等待体验”研究
- 比特币 2017：暴涨、暴跌、再涨、再暴跌的财富神话
- 360手机发布“充电宝级”产品 N6，李开新说这是涅槃重生

最新文章

- 谷歌发布 TensorFlow 1.5，支持动态图机制和 TensorFlow Lite
- Facebook 发布 wav2letter 包，用于端到端自动语音识别
- 关于应用机器学习作为搜索问题的入门简介
- 迎来 PyTorch，告别 Theano

创建了开源矩阵计算库OpenBLAS，领导团队不断进行修补和维护，目前在矩阵计算的细分领域，成为影响力较大的开源项目。

他在MIT萌生创业想法，归国之后，针对“深度学习”，创办PerfXLab澎峰科技，为计算机视觉、语音识别等公司提供一体化性能优化方案。雷锋网(公众号：雷锋网)【新智造】频道此前曾采访并报道了PerfXLab澎峰科技。

课程内容

- OpenBLAS项目介绍
- 矩阵乘法优化算法
- 一步步调优实现

以下为公开课完整视频，共64分钟：

以下为公开课内容的文字及 PPT 整理。

雷锋网的朋友们大家好，我是张先轶，今天主要介绍一下我们的开源矩阵计算库OpenBLAS以及矩阵乘法的优化。

雷锋网
AI研习社

什么是BLAS

- Basic Linear Algebra Subprograms
- 基本线性代数子程序
 - BLAS3级：矩阵-矩阵
 - BLAS2级：矩阵-向量
 - BLAS1级：向量-向量
- 科学计算的核心之一

A

x

y

A₁₁

A₁₂

A₁₃

A₁₄

A₁₅

A₂₁

A₂₂

A₂₃

A₂₄

A₂₅

A₃₁

A₃₂

A₃₃

A₃₄

A₃₅

A₄₁

A₄₂

A₄₃

A₄₄

A₄₅

A₅₁

A₅₂

A₅₃

A₅₄

A₅₅

x₁₁

x₂₁

x₃₁

x₄₁

x₅₁

y₁₁

y₂₁

y₃₁

y₄₁

y₅₁

首先，什么是BLAS？

2017 深度学习框架发展大盘

基于典型相关分析的词向量

中科院孙冰杰博士：基于网络数据表示学习的重叠社区发现 | 分享总结

热门搜索

手机

漏洞

移动互联网新闻

中国移

DIY

ARM

HTC ONE

任天堂

谷歌地图

虚拟运营商

移动安全

https://www.leiphone.com/news/201704/Puevv3ZWxn0heoEv.html

2/17

BLAS是 Basic Linear Algebra Subprograms (基本线性代数子程序) 的首字母缩写，主要用来做基础的矩阵计算，或者是向量计算。它分为三级：

- BLAS 1级，主要做向量与向量间的dot或乘加运算，对应元素的计算；
- BLAS 2级，主要做矩阵和向量，就类似PPT中蓝色部分所示，矩阵A*向量x，得到一个向量y。除此之外，可能还会有对称的矩阵变形；
- BLAS 3级，主要是矩阵和矩阵的计算，最典型的是A矩阵*B矩阵，得到一个C矩阵。由矩阵的宽、高，得到一个m*n的C矩阵。


为什么BLAS是一个非常重要的库或者接口，是因为它是很多科学计算的核心之一。每年做超级计算机的排行榜，都要做LINPACK测试，该测试很多部分就是做BLAS 3级矩阵和矩阵的计算。此外，还有很多科学和工程的模拟，在转换后都变成了一种矩阵上的操作。如果你把矩阵优化的特别好的话，对整个应用的提升，都是非常有帮助的。

雷锋网
AI研习社

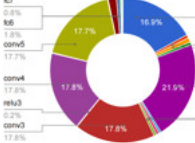
BLAS与Deep Learning

- Alexnet
 - Conv layer ->BLAS
 - FC layer -> BLAS

下载
雷锋网App
可观看本次课程回放

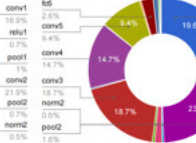


GPU Forward Time Distribution



Layer	Percentage
fc7	0.8%
fc6	1.8%
conv6	17.7%
conv4	17.8%
conv5	17.8%
conv3	17.8%
conv2	17.8%
conv1	17.8%

CPU Forward Time Distribution



Layer	Percentage
fc6	0.8%
fc7	1.8%
conv6	17.7%
conv4	17.8%
conv5	17.8%
conv3	17.8%
conv2	17.8%
conv1	17.8%

BLAS与 Deep Learning 的关系，深度学习这几年火了之后，比如大家非常了解的Alexnet，如果做具体的性能划分，PPT上的这张图来源于某篇论文，cut down之后看每个部分花了多少时间，发现它大部分的时间花费在卷积层（Conv Layer），另外不少时间花在了全连接层（FC layer）。卷积层目前通用的实现是展成矩阵，变成矩阵与矩阵的乘法，就是BLAS 3级。而全连接层一般是变成一个矩阵和向量的乘法，也落成了BLAS操作。

也就是说，基于矩阵类学习的深度学习，有90%或者更多的时间是通过BLAS来操作的。当然，随着新的算法出现，卷积层对3*3的卷积核有专门的算法，或者用FFT类算法也可以做，但是在通用上，展矩阵来做也非常广泛。

雷锋网
AI研习社

BLAS实现

- 商业
 - Intel MKL
 - AMD ACML
 - NVIDIA CUBLAS
 - IBM ESSL
- 开源
 - GotoBLAS (2010年中止开发)
 - ATLAS
 - OpenBLAS
 - BLIS

下载
雷锋网App
可观看本次课程回放

目前，BLAS只是一个定义了的实现接口，但是具体的实现其实有很多种。从商业的角度来讲，存在很多商业版本，比如说 Intel、AMD、NVIDIA、IBM公司等，基本上为了搭配自己的硬件，对其做了更优的优化，出了商业版本。

针对开源的而言，有如下几种，之前比较知名的是GoToBLAS，和OpenBLAS有很深的渊源，但是在2010年终止开发了，有时间在给大家分析其背后的原因，主力的开发人员后藤，离开了UT Austin的研究组，进入了公司，就终止了开发。ATLAS是美国一个学校做的，OpenBLAS是我们基于GotoBLAS做的，BLIS是后藤走了之后，基于GotoBLAS扩展出来的一个项目，目前还处在相对早期的阶段，成熟度还差一些。

雷锋网
AI研习社

OpenBLAS简介

- 2011年，forked from GotoBLAS2
- 全球最好的开源矩阵计算库
- 2016 中国计算机学会科技进步二等奖
- 进入主流Linux发行版
- 进入OpenHPC套件



Ubuntu
openblas package



debian



fedora

OpenBLAS历史已经有几年了，从2011年初开始进入，最初的原因是GotoBLAS放弃了，我们重新fork了一个社区发行版，继续开发和维护，它的维护不是一个简单修BUG的过程，如果想要获得比较好的性能，需要不停跟着硬件走，比如说新出一种新的硬件架构，或者适配更广的硬件架构，都要进行一定的优化，来获得比较好的加速效果。OpenBLAS算是目前全球最好的开源矩阵计算库，在去年的时候得到了中国计算机学会科技进步二等奖，同时也进入了很多主流的Linux安装包，比如说Ubuntu里面就有我们的OpenBLAS Package，你能想到的Linux发行版几乎都进去了，但这不是我们主动去做的。还有一个OpenHPC的套件，也是最近做高性能计算的一个源。



雷锋网
AI研习社

OpenBLAS简介


- 支持主流CPU处理器
 - Intel, AMD
 - ARM, AArch64
 - MIPS, 龙芯
 - IBM POWER
- 支持常见操作系统
 - Linux
 - Windows
 - Mac OSX
 - FreeBSD
 - Android



目前OpenBLAS的进展是，支持几乎全部的主流CPU处理器，同时都能达到比较好的优化性能。从操作系统来说，基本上常见主流的OS都支持。整体上，从适配的处理器范围和支持的操作系统，在开源库中算是最广的实现。

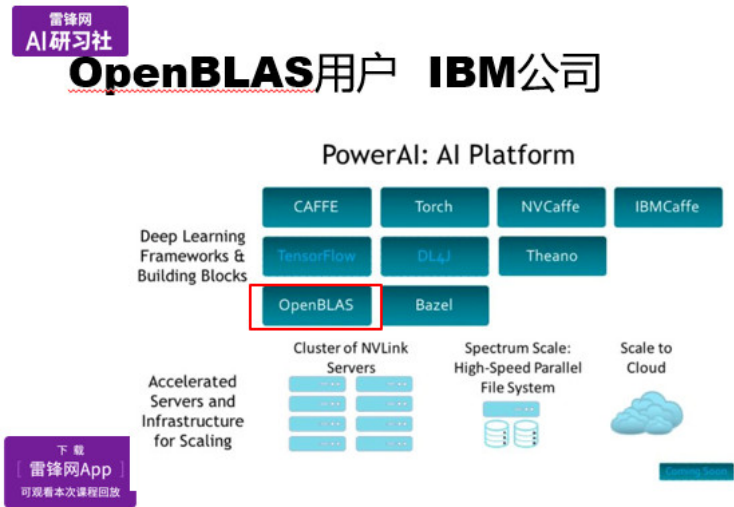
雷锋网
AI研习社

OpenBLAS用户

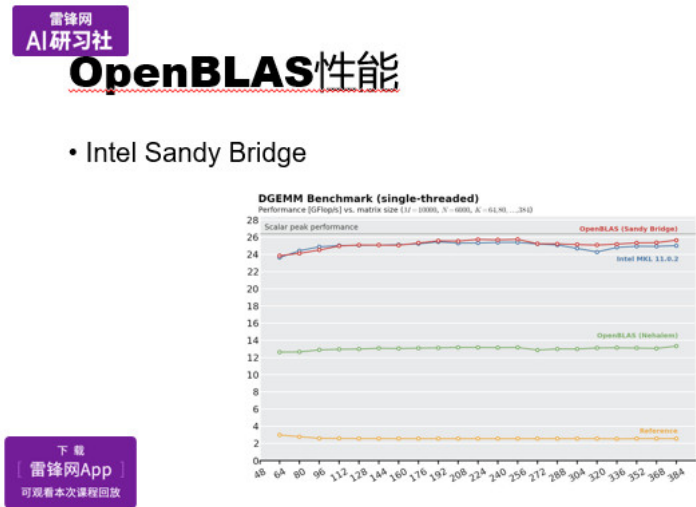


下载
雷锋网App
可观看本次课程回放

因此，OpenBLAS的用户也是比较多的。比如有开源项目Julia语言、GNU octave等；深度学习方面有大家熟悉的mxnet、Caffe都可以选OpenBLAS，作为CPU端的计算backend；IBM公司、ARM公司也都在他们的产品里边使用了OpenBLAS，NVIDIA公司在做一些跟CPU的对比测试时，把OpenBLAS列为了一个基准。其他还有一些做编译器的以色列创业公司，还有国内的一些互联网公司，比如搜狗。前段时间还和搜狗公司的人聊过，我们的库在线上已经稳定运行一年多以上的时间，所以说它的工程质量上还是还是可以的。



IBM前段时间，因为深度学习非常火，做了一个Power AI的软件框架，可以看到，最上面一层是一些开源的框架，底层的计算中就有我们的OpenBLAS。当然是为了搭载他的服务器。



简要的看一下性能，BLAS库的性能是越高越好。在Intel的 Sandy Bridge平台上，相比MKL的性能，基

本上是重合在一起的，达到了一个相当的性能。

雷锋网
AI研习社

OpenBLAS性能

• 龙芯3A

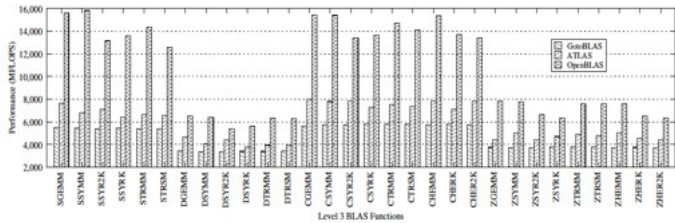


Figure 11. Multi-threaded Level 3 BLAS Performance (NP=4)

下载
雷锋网App
可观看本次课程回放

这张图展示了在龙芯上做的一个结果，测得比较全，整体的BLAS多线程的，性能全测试了，性能比较高的都是我们，提高了一倍到两倍。这是因为我们针对龙芯3A做了优化，所以取得了非常好的效果。

雷锋网
AI研习社

矩阵乘法GEMM优化

参考资料

- <https://github.com/flame/how-to-optimize-gemm/wiki>
- <https://github.com/flame/blislab>
- <http://apfel.mathematik.uni-ulm.de/~lehn/sghpc/gemm/>

下载
雷锋网App
可观看本次课程回放

刚才主要介绍了OpenBLAS的性能和效果，我们在GitHub上做了托管，欢迎对矩阵乘法或优化感兴趣的同学们能加入进来，贡献代码，我们公司愿意拿出一笔钱来支持这个项目继续往前走。接下来会开始一些技术类的干货，主要讲一下大家对优化比较感兴趣的部分，我参考了矩阵乘法的这几篇教程，UT Austin Flame组做的教程。我把他的内容基本上是抠出来了，一步步带着大家过一下，如果我们从最简单的矩阵乘法实现，到一个高性能的矩阵乘法实现，大概是几步，怎么来的？或者是为什么优化，每一步能获得多少性能收益。这样大家对于一些优化的其他程序，希望能提供一些帮助。

雷锋网
AI研习社

矩阵乘法GEMM优化

```
for ( i=0; i<m; i++ ){          /* Loop over the rows of C */
  for ( j=0; j<n; j++ ){        /* Loop over the columns of C */
    for ( p=0; p<k; p++ ){      /* Update C( i,j ) with the inner
                                product of the ith row of A and
                                the jth column of B */
      C( i,j ) = C( i,j ) + A( i,p ) * B( p,j );
    }
  }
}
```

下载
雷锋网App
可观看本次课程回放

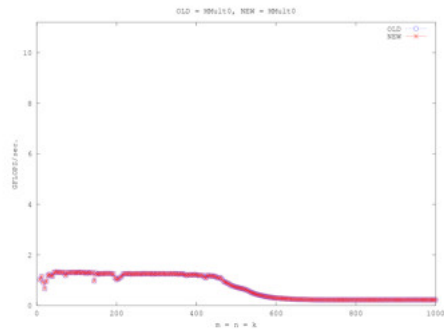
我们首先看一下基本实现。

我想只要学过《线性代数》之类的，这种矩阵乘法，是一个非常简单的问题，如果转换成C代码来做的话，就是一个三重循环，我在这张图里列出了一个【i j k】的三重循环，这里面矩阵乘法的代码就已经是，它实现的功能就是矩阵A*矩阵B，加到矩阵C里面，C是结果矩阵，这里面C的代码，和在课本上看到的累加的公式是一样的。找到行，对应这个位置的结果C，把i行的每个元素，都取出来乘以B列，对应的乘，然后加起来就可以得到这个结果。但是这种实现，如果你放到现在的处理器上跑性能的话，和优化后的BLAS库的实现，性能会差很多倍，甚至会差10倍。



矩阵乘法GEMM优化

• 性能



为什么呢，我们就做了一下最后的性能测试

这张图也是截自教程里，代表了一个性能图，越高越好。这里的测试平台是Intel core i5，只是测了单线程，没管多线程的事情。这种初始实现可能是1 GFlop/s。随着规模变大，矩阵的性能在下降是为什么呢？因为在实现的过程中，没有考虑到cache的原因，当矩阵比较小的时候，速度还能快一些，当矩阵大了的时候，一定会跌下去，所以图里就有一个下滑的过程。

这个是非常原始、基础的实现。



矩阵乘法GEMM优化

```
for ( j=0; j<n; j+=4 ){          /* Loop over the columns of C, unrolled by 4 */
  for ( i=0; i<m; i+=1 ){          /* Loop over the rows of C */
    /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
       one routine (four inner products) */

    AddDot1x4( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc );
  }
}
```

AddDot 1x4

```
for ( p=0; p<k; p++ ){
  C( 0, 0 ) += A( 0, p ) * B( p, 0 );
  C( 0, 1 ) += A( 0, p ) * B( p, 1 );
  C( 0, 2 ) += A( 0, p ) * B( p, 2 );
  C( 0, 3 ) += A( 0, p ) * B( p, 3 );
}
```

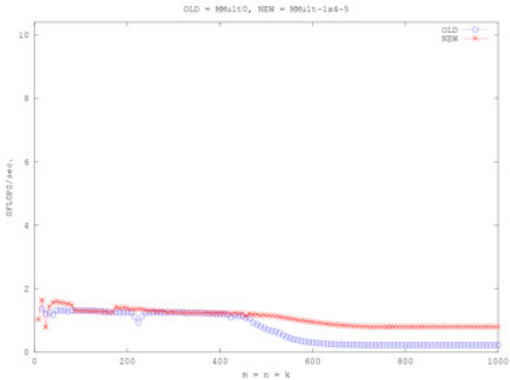


再往上走一步，怎么样才能再稍微优化一下。我们需要把矩阵的乘法顺序调一下，我们在这里做了一个小的分块，把p单独提到了一个函数里，以点乘的形式写出来，每次做一个1*4的结果，单独提出来变成一个函数。p的这一步，要把计算顺序稍微换一下，把i放到里面，j放到外面，这块背景为什么要换一下，实际上是因为我们假设矩阵在存储的时候是以列优先存储的，在列项的数值是连续存储，行之间是有间隔的，这对于访存更有优势。变成这样的实现之后，对整体的性能其实没什么帮助，那为什么换成这种形式来写呢，是为了之后的优化，留下一定的空间。

雷锋网
AI研习社

矩阵乘法GEMM优化

- 性能
 - p unrolled
 - A(0, p)复用，只从内存读取一次



下载
雷锋网App
可观看本次课程回放

当矩阵比较小，在cache里面的时候，性能基本是没什么变化的，但是超过cache的时候，它的性能稍微好了一点，从刚才非常低的值，也达到了接近1GFlop/s主要的原因是对A(0,p)做了一定的复用，它省了一些cache。另外一方面，它本身在做循环的利用来说，相当于比这部分做了一定循环的展开，所以在效率上得到了一定的提升。

这块的复用，只从内存读取一次，所以对超过cache的情况有了一定改善。

雷锋网
AI研习社

矩阵乘法GEMM优化

- 优化AddDot1x4
 - 使用寄存器变量

```
for ( p=0; p<k; p++){
    C( 0, 0 ) += A( 0, p ) * B( p, 0 );
    C( 0, 1 ) += A( 0, p ) * B( p, 1 );
    C( 0, 2 ) += A( 0, p ) * B( p, 2 );
    C( 0, 3 ) += A( 0, p ) * B( p, 3 );
}
```



```
register double
/* hold contributions to
   C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ) */
c_00_reg, c_01_reg, c_02_reg, c_03_reg,
/* holds A( 0, p ) */
a_0p_reg;

c_00_reg = 0.0;
c_01_reg = 0.0;
c_02_reg = 0.0;
c_03_reg = 0.0;

for ( p=0; p<k; p++){
    a_0p_reg = A( 0, p );

    c_00_reg += a_0p_reg * B( p, 0 );
    c_01_reg += a_0p_reg * B( p, 1 );
    c_02_reg += a_0p_reg * B( p, 2 );
    c_03_reg += a_0p_reg * B( p, 3 );
}

C( 0, 0 ) += c_00_reg;
C( 0, 1 ) += c_01_reg;
C( 0, 2 ) += c_02_reg;
C( 0, 3 ) += c_03_reg;
```

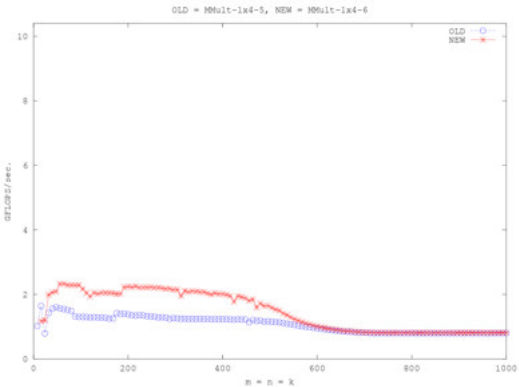
下载
雷锋网App
可观看本次课程回放

在这个基础上，我们就需要看一下有什么更好的方法来做优化。我们的基准就是，AddDot1*4的基准上怎么做，我们想到第一点做的是，我们可不可以用寄存器变量来做，而不是操作内存。我可以申请一堆C00，01这样的寄存器变量，在C语言中是register double，还有矩阵A的部分，也用寄存器变量。

雷锋网
AI研习社

矩阵乘法GEMM优化

- 性能
 - 降低cache的开销



下载
雷锋网App
可观看本次课程回放

剩下的操作都是一些寄存器的变量，当然B还是之前的方式，最后再写回C里面。它完成的流程基本跟与之前的实习一样，只是我们引入了寄存器变量，让更多的数据保存到寄存器里，而不是放到cache缓存里，来减轻cache的压力，这也能获得一部分性能的提升。

可以看到，红色的线是我们优化后的性能，达到了2GFlop/s，蓝色的线是之前的性能，这里做的优化就是利用寄存器降低cache的消耗，取得了50%左右的性能提升。完成了这一步之后，我们还可以再怎么样做优化呢？

雷锋网
AI研习社

矩阵乘法**GEMM**优化

- 优化AddDot1x4
 - B访存，使用指针

下载
雷锋网App
可观看本次课程回放

```
register double
/* hold contributions to
   C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ) */
c_00_reg, c_01_reg, c_02_reg, c_03_reg,
/* holds A( 0, p ) */
a_0p_reg;

c_00_reg = 0.0;
c_01_reg = 0.0;
c_02_reg = 0.0;
c_03_reg = 0.0;

for ( p=0; p<k; p++){
    a_0p_reg = A( 0, p );

    c_00_reg += a_0p_reg * B( p, 0 );
    c_01_reg += a_0p_reg * B( p, 1 );
    c_02_reg += a_0p_reg * B( p, 2 );
    c_03_reg += a_0p_reg * B( p, 3 );
}

C( 0, 0 ) += c_00_reg;
C( 0, 1 ) += c_01_reg;
C( 0, 2 ) += c_02_reg;
C( 0, 3 ) += c_03_reg;
```

→

```
bp0_pntr = &B( 0, 0 );
bp1_pntr = &B( 0, 1 );
bp2_pntr = &B( 0, 2 );
bp3_pntr = &B( 0, 3 );

c_00_reg = 0.0;
c_01_reg = 0.0;
c_02_reg = 0.0;
c_03_reg = 0.0;

for ( p=0; p<k; p++){
    a_0p_reg = A( 0, p );

    c_00_reg += a_0p_reg * *bp0_pntr++;
    c_01_reg += a_0p_reg * *bp1_pntr++;
    c_02_reg += a_0p_reg * *bp2_pntr++;
    c_03_reg += a_0p_reg * *bp3_pntr++;
}

C( 0, 0 ) += c_00_reg;
C( 0, 1 ) += c_01_reg;
C( 0, 2 ) += c_02_reg;
C( 0, 3 ) += c_03_reg;
```

我们刚才在A、C的部分，已经用寄存器做了替换，那么B访存这部分，我们有没有可能也做一些优化。在之前实现的时候，B部分每次的坐标计算都需要算出来，B访问每个位置都要算一次，这样就引入了额外的开销，其实是可以避免的。

我们使用指针的方式，一开始先把对应的指针位置指好，每次计算的时候只要指针连续移动就好，而不是每次读一个位置重新算一遍，这样速度就会快一些。

雷锋网
AI研习社

矩阵乘法**GEMM**优化

- 性能
 - 降低B index开销

下载
雷锋网App
可观看本次课程回放

我们看一下，做完这个小优化之后，降低了B index的消耗之后，从刚才的2G F...达到了4G的性能。这块的改善对于小矩阵有效果，大矩阵全都超出了cache范围，就不会有效果的。所以假设矩阵都在cache里面，这块也获得了不小的性能提升。

雷锋网
AI研习社

矩阵乘法**GEMM**优化

- 优化AddDot1x4
 - Unroll loop p

下载
雷锋网App

可观看本次课程回放

```
bp0_ptr = &b( 0, 0 );
bp1_ptr = &b( 0, 1 );
bp2_ptr = &b( 0, 2 );
bp3_ptr = &b( 0, 3 );

c_00_reg = 0.0;
c_01_reg = 0.0;
c_02_reg = 0.0;
c_03_reg = 0.0;

for ( p=0; p<k; p++) {
    a_0p_reg = A( 0, p );

    c_00_reg += a_0p_reg * *bp0_ptr++;
    c_01_reg += a_0p_reg * *bp1_ptr++;
    c_02_reg += a_0p_reg * *bp2_ptr++;
    c_03_reg += a_0p_reg * *bp3_ptr++;
}

C( 0, 0 ) += c_00_reg;
C( 0, 1 ) += c_01_reg;
C( 0, 2 ) += c_02_reg;
C( 0, 3 ) += c_03_reg;
```

→

```
for ( p=0; p<k; p+=4 ){
    a_0p_reg = A( 0, p );

    c_00_reg += a_0p_reg * *bp0_ptr;
    c_01_reg += a_0p_reg * *bp1_ptr;
    c_02_reg += a_0p_reg * *bp2_ptr;
    c_03_reg += a_0p_reg * *bp3_ptr;

    a_0p_reg = A( 0, p+1 );

    c_00_reg += a_0p_reg * *(bp0_ptr+1);
    c_01_reg += a_0p_reg * *(bp1_ptr+1);
    c_02_reg += a_0p_reg * *(bp2_ptr+1);
    c_03_reg += a_0p_reg * *(bp3_ptr+1);

    a_0p_reg = A( 0, p+2 );

    c_00_reg += a_0p_reg * *(bp0_ptr+2);
    c_01_reg += a_0p_reg * *(bp1_ptr+2);
    c_02_reg += a_0p_reg * *(bp2_ptr+2);
    c_03_reg += a_0p_reg * *(bp3_ptr+2);

    a_0p_reg = A( 0, p+3 );

    c_00_reg += a_0p_reg * *(bp0_ptr+3);
    c_01_reg += a_0p_reg * *(bp1_ptr+3);
    c_02_reg += a_0p_reg * *(bp2_ptr+3);
    c_03_reg += a_0p_reg * *(bp3_ptr+3);

    bp0_ptr+=4;
    bp1_ptr+=4;
    bp2_ptr+=4;
    bp3_ptr+=4;
}
```

当你完成这一部分的时候，你可以想，我把A矩阵做了寄存器替换，B矩阵通过index改进，我们下一步还能怎么做？

其实就是一个比较常用的方式，做展开。

在最里层循环，是不是可以展开成4次，在做这个的时候，我们可以降低整个循环这部分的开销，而且让它流水的情况更好。

雷锋网
AI研习社

矩阵乘法**GEMM**优化

- 性能
 - 相对初始实现

下载
雷锋网App

可观看本次课程回放

这部分也会对性能有一些改善。这张图比较的当初在中间阶段的时候比起开始阶段，得到了多少提升。通过使用寄存器变量，使用了指针，在做了一定的底层循环展开之后，达到了红色线的性能，已经比蓝色线有了明显的提升，但是这个还不算完，只是一个基础。但是在1*4的层面，已经没什么油水可挖了，所以我们需要在更上层找一些方法。

雷锋网
AI研习社

矩阵乘法**GEMM**优化

- 计算C 4x4

下载
雷锋网App

可观看本次课程回放

```
for ( j=0; j<n; j+=4 ){ /* Loop over the columns of C, unrolled by 4 */
    for ( i=0; i<m; i+=1 ){ /* Loop over the rows of C */
        /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
           one routine (four inner products) */
        AddDot1x4( k, &A( i,0 ), lda, &b( 0,j ), ldb, &C( i,j ), ldc );
    }
}
```

↓

```
for ( j=0; j<n; j+=4 ){ /* Loop over the columns of C, unrolled by 4 */
    for ( i=0; i<m; i+=4 ){ /* Loop over the rows of C */
        /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
           one routine (four inner products) */
        AddDot4x4( k, &A( i,0 ), lda, &b( 0,j ), ldb, &C( i,j ), ldc );
    }
}
```

在1*4的时候，A的值产生了一些重用，但是如果块再放一点，比如说变成4*4时，也就是说每次计算的时候算的是一个方块，而不是列。这个对于整个的优化来说，可以复用你的访存，而且能够更充分的发挥计算能力。



矩阵乘法GEMM优化

- AddDot4x4
 - 重复AddDot1x4优化



```
for ( p=0; p<k; p++){
    a_0p_reg = A( 0, p );
    a_1p_reg = A( 1, p );
    a_2p_reg = A( 2, p );
    a_3p_reg = A( 3, p );

    /* First row */
    c_00_reg += a_0p_reg * *b_p0_ptr;
    c_01_reg += a_0p_reg * *b_p1_ptr;
    c_02_reg += a_0p_reg * *b_p2_ptr;
    c_03_reg += a_0p_reg * *b_p3_ptr;

    /* Second row */
    c_10_reg += a_1p_reg * *b_p0_ptr;
    c_11_reg += a_1p_reg * *b_p1_ptr;
    c_12_reg += a_1p_reg * *b_p2_ptr;
    c_13_reg += a_1p_reg * *b_p3_ptr;

    /* Third row */
    c_20_reg += a_2p_reg * *b_p0_ptr;
    c_21_reg += a_2p_reg * *b_p1_ptr;
    c_22_reg += a_2p_reg * *b_p2_ptr;
    c_23_reg += a_2p_reg * *b_p3_ptr;

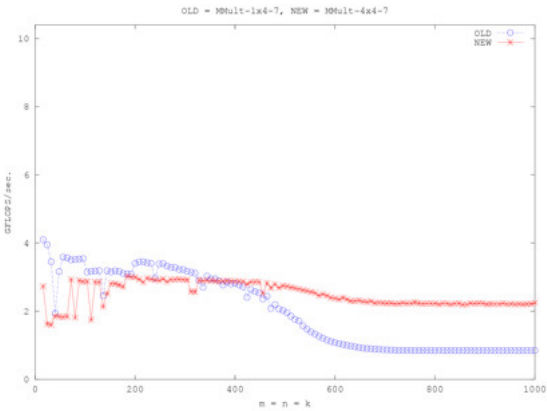
    /* Four row */
    c_30_reg += a_3p_reg * *b_p0_ptr;
    c_31_reg += a_3p_reg * *b_p1_ptr;
    c_32_reg += a_3p_reg * *b_p2_ptr;
    c_33_reg += a_3p_reg * *b_p3_ptr;
}
```

当我们变成小的这种4*4的方块时，AddDot函数也要写成这个模式。当然，这部分也要用刚才做过的那些1*4的方法，A这边之前是1个值，现在是4个值，用寄存器的变量，C部分已经是4*4共有16个，也全都是寄存器变量，B的部分全部用指针来优化。



矩阵乘法GEMM优化

- 性能
 - 相对1x4



但这样做的话，对于整体的性能提升是比较有限的。因为这只是一个初始的结果，可以看到，对于小矩阵，在cache范围内，没有什么起色。但是超过cache后，对于大规模的矩阵，是有了一定性能提升。



矩阵乘法GEMM优化

- AddDot4x4
 - 向量化指令



```
for ( p=0; p<k; p++){
    a_0p_a_1p_vreg.v = _mm_load_pd( (double *) &A( 0, p ) );
    a_2p_a_3p_vreg.v = _mm_load_pd( (double *) &A( 2, p ) );

    b_p0_vreg.v = _mm_loadup_pd( (double *) b_p0_ptr++ );
    b_p1_vreg.v = _mm_loadup_pd( (double *) b_p1_ptr++ );
    b_p2_vreg.v = _mm_loadup_pd( (double *) b_p2_ptr++ );
    b_p3_vreg.v = _mm_loadup_pd( (double *) b_p3_ptr++ );

    /* First row and second rows */
    c_00_c_10_vreg.v += a_0p_a_1p_vreg.v * b_p0_vreg.v;
    c_01_c_11_vreg.v += a_0p_a_1p_vreg.v * b_p1_vreg.v;
    c_02_c_12_vreg.v += a_0p_a_1p_vreg.v * b_p2_vreg.v;
    c_03_c_13_vreg.v += a_0p_a_1p_vreg.v * b_p3_vreg.v;

    /* Third and fourth rows */
    c_20_c_30_vreg.v += a_2p_a_3p_vreg.v * b_p0_vreg.v;
    c_21_c_31_vreg.v += a_2p_a_3p_vreg.v * b_p1_vreg.v;
    c_22_c_32_vreg.v += a_2p_a_3p_vreg.v * b_p2_vreg.v;
    c_23_c_33_vreg.v += a_2p_a_3p_vreg.v * b_p3_vreg.v;
}
```

在以4*4的结果优化基础上，我们可以再做进一步的优化和开发。为什么要转换成4*4的优化，是因为我们现在CPU的处理器，基本上想获得高的性能，必须要用向量化指令，不管是老的SSE2，AVX或者AVX 2.0等，对于CPU的优化，如果想达到高性能，必须要用到单指令多数据的向量化指令。

雷锋网
AI研习社

矩阵乘法**GEMM**优化

- 性能
 - 向量化
 - 相对于1x4

下载
雷锋网App

可观看本次课程回放

做了4*4的分块之后，在这种情况下，会更有利于向量指令。在这里以向量指令重写了这部分循环的内容，当然这部分指令我没有任何的内嵌汇编或者纯汇编的操作，我就直接用了Intel Intrinsic的形式来写，可以看到这部分写的就是一个128位的sse，这是做一个双精度浮点double的一个矩阵，数据都是double的，从A里把这两个值load进来。后两个load进这个向量寄存器里，B部分每次都要用load复制的这种指令load进去，剩下的这块基本都是用向量的Intrinsic的变量来做了操作，在这块跟之前看起来差不多，所以在编译的时候都变成了向量化的指令。这两行就算前部分C的值，这部分就算后部分C的值。

通过这种向量寄存器的指令使用后，他的性能提升非常明显，从刚才4G可以达到超过6G，而且这一部分是一个整体的变化，cache内向量加速效果是非常明显的，基本上是翻了一倍。

下一步需要解决的是这个cache的问题，问题是没有做大的分块，超过cache大小之后性能就会下滑，要解决这个问题，需要在更上一层做Blocking。

雷锋网
AI研习社

矩阵乘法**GEMM**优化

- Blocking
 - 解决超过L2 Cache的性能

下载
雷锋网App

可观看本次课程回放

```
void MV_MMult( int m, int n, int k, double *a, int lda,
               double *b, int ldb,
               double *c, int ldc )
{
    int i, j, p, pb, kb;

    /* This time, we compute a mc x n block of C by a call to the InnerKernel */

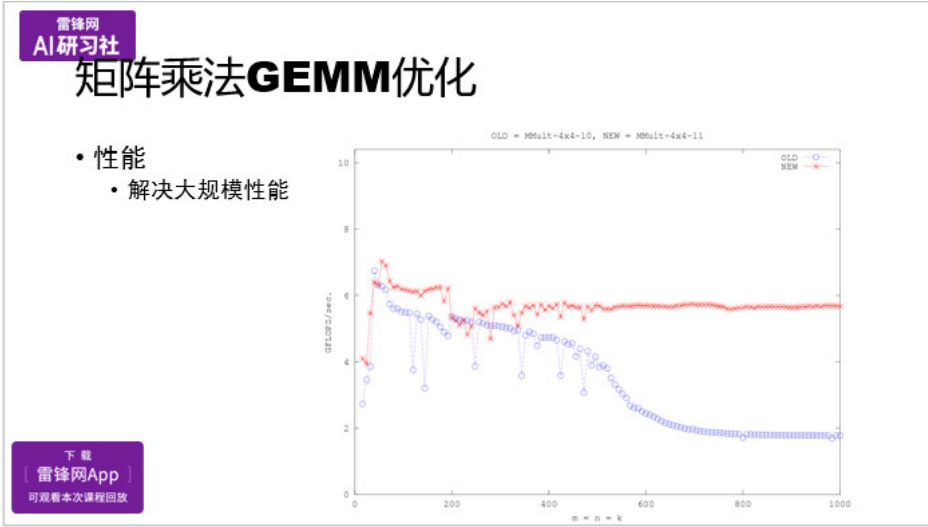
    for ( p=0; p<n; p+=kc ){
        pb = min( k-p, kc );
        for ( i=0; i<n; i+=mc ){
            kb = min( m-i, mc );
            InnerKernel( kb, n, pb, &a( i,p ), lda, &b( p, 0 ), ldb, &c( i,0 ), ldc );
        }
    }

    void InnerKernel( int m, int n, int k, double *a, int lda,
                     double *b, int ldb,
                     double *c, int ldc )
    {
        int i, j;

        for ( j=0; j<n; j+=4 ){
            /* Loop over the columns of C, unrolled by 4 */
            for ( i=0; i<n; i+=4 ){
                /* Loop over the rows of C */
                /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
                 one routine (four inner products) */

                AddDot4x4( k, &a( i,0 ), lda, &b( 0,j ), ldb, &c( i,j ), ldc );
            }
        }
    }
}
```

转换成代码的话，在这一层做一个K的切分，下面一层做一个m的切分，至于kc和mc都是参数。这些参数都是可调的，都要根据L2 cache的大小进行调整，然后每次做的是一个小块c的矩阵乘，相当于一个子问题，这个子问题的实现基本和刚才4*4的实现是一样的。



这一部分blocking做完的性能如图所示，蓝色的线是没有做Blocking的性能，红色线是做过之后。当问题规模在cache内，它的性能改善基本比较小，但是当大规模的矩阵，通过做了这次Blocking之后，可以看到获得了非常明显的提升，变快了2倍。

雷锋网
AI研习社

矩阵乘法**GEMM**优化

- Packing A
 - 把A copy一份到连续空间

下载
雷锋网App
可观看本次课程回放

```
void InnerKernel( int m, int n, int k, double *a, int lda,
                 double *b, int ldb,
                 double *c, int ldc )
{
    int i, j;
    double
        packedA[ m * k ];

    for ( j=0; j<n; j+=4 ){          /* Loop over the columns of C, unrolled by 4 */
        for ( i=0; i<m; i+=4 ){      /* Loop over the rows of C */
            /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
               one routine (four inner products) */
            if ( j == 0 ) PackMatrixA( k, &A( i, 0 ), lda, &packedA[ i*k ] );
            AddDot4x4( k, &packedA[ i*k ], 4, &B( 0,j ), ldb, &C( i,j ), ldc );
        }
    }
}
```

对于大矩阵，为了充分的利用cache，让子问题变小，提升它的数据局部性，在做其他问题优化的时候也很有必要的。下一步当我们做到blocking的时候，如果只是代码级别变化的时候，基本已经做完了。

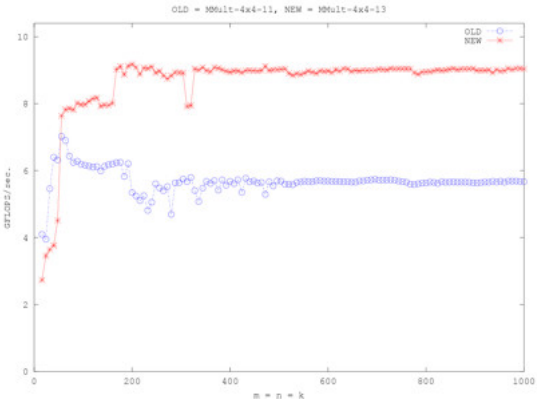
此后再进一步优化的点，引入引入一些操作。当我们分析程序存在的性能瓶颈，对于A的访存和B的访存是比较慢，很多访存在矩阵中是不连续的，所以访存性能就差了很多，一方面不能利用cache，一方面在TLB上也有影响，当然C部分也有一些影响，C矩阵往往很大，没有办法做packing，只能对A和B来做，packing的意思是说，我在这里有一部分连续的内存空间，m*k，对应前面的mc和kc，在这块内存空间，在每次做计算之前，我把所需要用到的A的矩阵，从原始矩阵读取出来，存放到连续的一块内存空间里。Packed Matrix A这个函数的具体实现非常简单，基本上就是从对应的位置取出来，放在连续的内存地址就结束。

为什么会做这步操作呢？这步操作的意义在于，通过pack之后，下一步AddDot4*4里读的元素就不是从A矩阵读，而是从pack后缓存区的位置读。一个好处是，A矩阵已经预热，放进CPU的cache里了；第二个好处是，你可以看到我在存储的时候，这种连续性的存储，读的时候也是连续性读取，效率会非常高，cache效率也非常高。加上通过pack这一步，对于性能的改善，是非常明显的。

雷锋网
AI研习社

矩阵乘法GEMM优化

- 性能
 - A的访存



下载
雷锋网App
可观看本次课程回放

这张图是上一步的操作，做了packing之后，除了极小矩阵规模没什么效果，或者引入了额外开销，效果还变差之外，绝大部分的性能提升是非常明显的，有50%以上，达到了9GFlop/s。

雷锋网
AI研习社

矩阵乘法GEMM优化

- Packing B
 - 把B copy一份到连续空间

```
void InnerKernel( int m, int n, int k, double *a, int lda,
                 double *b, int ldb,
                 double *c, int ldc, int first_time )
{
    int i, j;
    double
        packedA[ m * k ];
    static double
        packedB[ kc*nb ]; /* Note: using a static buffer is not thread safe... */

    for ( j=0; j<n; j+=4 ){ /* Loop over the columns of C, unrolled by 4 */
        if ( first_time )
            PackMatrixB( k, &b( 0, j ), ldb, &packedB[ j*k ] );
        for ( i=0; i<m; i+=4 ){ /* Loop over the rows of C */
            /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
               one routine (four inner products) */
            if ( j == 0 )
                PackMatrixA( k, &a( i, 0 ), lda, &packedA[ i*k ] );
            AddDot4x4( k, &packedA[ i*k ], a, &packedB[ j*k ], k, &c( i,j ), ldc );
        }
    }
}
```

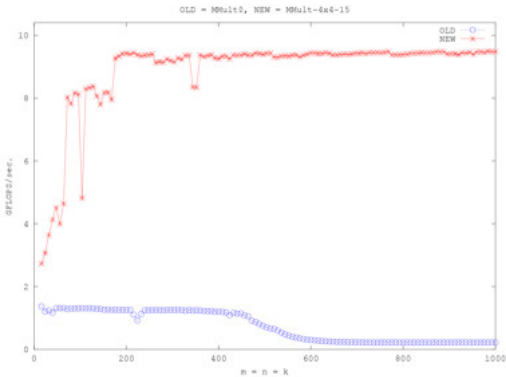
下载
雷锋网App
可观看本次课程回放

对于矩阵乘法实现的话，packing矩阵是一个非常重要的优化方式。再往后大家会想，对于A来说做了Packing，对于B是不是也能做Packing，同样道理也是可以的，就是把它拷贝到一个连续空间。B部分的Packing操作和A部分类似，也是把它的数从原始矩阵里读出来，然后放到一个连续空间里，使它的内存访问做连续的访存。当然这部分，因为B访存是个流式的访存，所以在这部分的改进会稍微小一点，相比A只有大概10%左右的提升。

雷锋网
AI研习社

矩阵乘法GEMM优化

- 性能
 - 相比初始



下载
雷锋网App
可观看本次课程回放

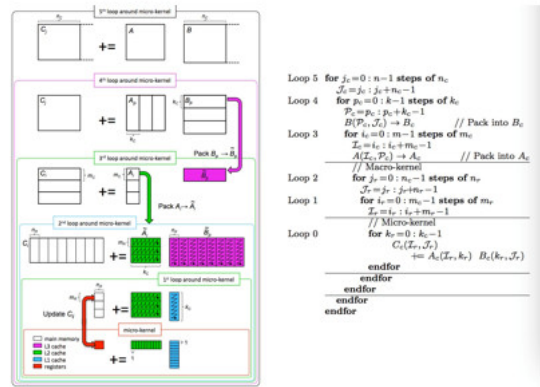
对于矩阵乘法实现的话，packing矩阵是一个非常重要的优化方式。再往后大家会想，对于A来说做了Packing，对于B是不是也能做Packing，同样道理也是可以的，就是把它拷贝到一个连续空间。B部分的

Packing操作和A部分类似，也是把它的数据从原始矩阵里读出来，然后放到一个连续空间里，使它的内存访问做连续的访存。当然这部分，因为B访存是个流式的访存，所以在这部分的改进会稍微小一点，相比A只有大概10%左右的提升。

雷锋网
AI研习社

矩阵乘法GEMM算法

下载
雷锋网App
可观看本次课程回放



当你完成到这一步的时候，相比最开始三重循环的性能改进，你的矩阵乘法的性能已经有很明显的提升了。你如果想做的更好的话，内部的核心可能不止要写intrinsic的指令，还要写内嵌汇编，重排流水线，使硬件资源能够发挥更多，可能还会提升10%。当然这部分对实现BLAS比较重要，会抠的比较细。

我们再整体回顾一下矩阵乘法的算法，我把算法的这部分放到最后，从开始一步步实现之后，做到最后大家可能看的比较清楚一些。A矩阵*B矩阵得到C矩阵，对应的是最外层的循环，每一步往下的时候，其实都是在做分块，做分块的原因刚才有提到，就是为了配合硬件结构，因为memory、cache等都是分层的，它是越来越小的，做分块实际上是提高了cache的各层的利用率。

今天就分享到这里，谢谢大家。

问答解答

问题1：什么是访存优化？

张先轶：访存优化解决的是处理器读取数据的性能。从计算上来说，是相对好优化的，但是优化访存会非常困难，稠密矩阵乘法的数据还是相对规整的，读数据的顺序是有规则的，更容易优化一些。但是我们也做过很多稀疏矩阵的优化，比如稀疏矩阵乘向量的优化，这个对访存来说更困难一些，因为没有办法预测到下一次访存在什么位置，这造成了优化的困难。

问题2：OpenBLAS和其他矩阵库有什么关系？

张先轶：OpenBLAS和其他BLAS实现其实都是完成了接口，BLAS只是接口的定义，具体来说可以有多种实现。我们认为我们OpenBLAS在开源的实现是非常好的。如果是标准BLAS，有参考实现，只是一个非常简单的Fortran实现，性能很差的，我们要比他们快很多。MKL是Intel公司自己做的BLAS，我们跟他们相当。Engien我们没有完全测过，它号称自己做的很好，但是他们的做法在X86的平台可能有些效果，但是对其他平台的效果我表示怀疑。不过我没有具体做对比测试，所以发言权不大。

问题3：从入门到精通需要多久？

张先轶：如果我指导的话，几个月时间就可以上手做一些事情。欢迎大家。

问题4：比起高通的库表现如何？

张先轶：说实话高通的库没有测过，我觉得它号称比较快，是因为在32位的ARM上，我们OpenBLAS没有做向量化优化，高通的那个部分做了，所以它可能会比我们快一些，但是在我们公司内部的PerfBLAS是优化了的。

问题5：分块的目的是什么？

张先轶：就是优化访存，通过分块之后让访存都集中在一定区域，能够提高了数据局部性，从而提高cache利用率，性能就会更好。

问题6：硬件不给力能玩神经网络么？

张先轶：我们给出的一个数据是，我们在ARM CortexA57的平台上，4核1.7GHz，跑AlexNet模型，一张图是150ms，这个速度还算比较快。另一方面我们还在做一些其他的模型，做了精简优化，再配合底层库的优化。在某个小模型下，在跑一张小图片的inference只用了50ms。所以，在ARM的处理器上，还是可以做到实时本地化的神经网络inference。

问题7：内部版本和开源版本差别大么？

张先轶：内部版本是针对深度学习做了一些差异化处理，性能高的可能会到1倍多，这部分的优化，一部分是矩阵的规模，和刚才讲的做方阵不一样，深度学习的矩阵大部分是中小型，某个维度会比较小，要用到的优化策略，或者分块策略会不一样，甚至有时候特别小根本不用分块或packing，直接做可能更好一些。

雷锋网原创文章，未经授权禁止转载。详情见[转载须知](#)。

雷锋网 2017年度特辑

早鸟4折价 ¥79
(限量400份, 售完即涨)

全套
12辑

7人收藏

分享：

相关文章

OpenBLAS

张先轶

矩阵乘法



OpenBLAS项目与矩阵乘法优化
AI研习社 04-10 20:00



从开源到商业，他用了七年时间做出深度学习定制库



OpenBLAS项目与矩阵乘法优化
AI研习社 04-10 20:00



从开源到商业，他用了七年时间做出深度学习定制库



迅雷“内讧”始末：原迅雷高级副总裁於菲被指利益输送



惠普被指擅自在用户电脑上安装“间谍软件”，收集用户数据



腾讯《绝地求生》官方手游不止一款？《光荣使命》正在内测



Velodyne正式推出128线激光雷达VLS-128，探测距离更远

文章点评：

我有话要说.....

☐ 同步到新浪微博

提交

热门关键字

热门标签 人工智能 机器人 机器学习 深度学习 金融科技 未来医疗 智能驾驶 自动驾驶 计算机视觉 激光雷达 图像识别 智能音箱 区块链 智能投顾 医学影像 物联网 IoT 微信小程序平台 微信小程序在哪 CES 2017 CES 2016年最值得购买的智能硬件 2016 互联网 小程序 微信朋友圈 抢票软件 智能手机 智能家居 智能手环 智能机器人 智能电视 360智能硬件 智能摄像机 智能硬件产品 智能硬件发展 智能硬件创业 黑客 白帽子 大数据 云计算 新能源汽车 无人驾驶 无人机 大疆 小米无人机 特斯拉 VR游戏 VR电影 VR视频 VR眼镜 VR购物 AR 直播 扫地机器人 医疗机器人 工业机器人 类人机器人 聊天机器人 微信机器人 微信小程序 移动支付 支付宝 P2P 区块链 比特币 风控 高盛 人脸识别 指纹识别 黑科技 谷歌地图 谷歌 IBM 微软 乐视 百度 三星s8 腾讯 三星Note8 小米MIX 小米Note 华为 小米 阿里巴巴 苹果 MacBook Pro iPhone Face ID GAIR IROS 双创周 云栖大会 优施 智能硬件公司 智能硬件 QQ红包 支付宝红包 敬业福 机器学习 任平 坚果投影仪 tensorflow安装 谷歌nexus 6p wifi汉枫 light phone 开源无人机 iphone7plus双摄像头有什么用 fdd oculus 官网 g flex2 z3compact 小米 血压计 腾讯ai lab 更多

联系我们 关于我们 加入我们 意见反馈 投稿