Loongson3A/3B上linpack简介

Chaos (xuchao@loongson.cn)

February 25, 2014 Rev. 1.0

Contents

A 概述

我们在这一章介绍如何快速的编译和使用linpack，所有的代码包括pmon kernel都放在Git库中。简单介绍下Git库中的目录结构：

1. demo 包括3A 3B上可以运行的动态xhpl程序以及对应的HPL.dat文件
2. document 主要是朱海涛和何颂颂关于3A 3B的论文，以及Goto关于GotoBLAS库的介绍的文档
3. loongson3a 包括hpl-2.0以及GoToBLAS库，编译方法可以见代码中的简单指引
4. loongson3b 3B上的hpl-2.0和GotoBLAS库[1]
5. mpich2 mpich2-1.2 mpich2-sync 都是MPI库，前者是binary，用在3A上，后者是souce code，最后是用在3B上的binary
6. system 因为目前公共git树上的pmon和kernel在3B上都存在一些问题，因此单独维护两个包，并提供可供测试的pmon和kernel的二进制文件
7. toolchain 3A和3B上的工具链[2]

根据如上的目录结构，将PATH指向对应的toolchain，可以很快的编译出相应的xhpl可执行程序。

```
//compile 3A linpack
    cd loongson3a/GotoBLAS2
    ./quickbuild.64bit
    cd hpl-2.0
    make arch=goto_us

//compile 3B linpack
    cd loongson3b/Gotoblas
    cd interface
    ./compile
    cd ../3c_ori
    ./compile
    cd hpl-2.0
    make arch=zhangming
```

[3]

根据Wiki上的内容，我们对HPL.dat的参数简单的说明：

1. HPLinpack benchmark input file
2. Innovative Computing Laboratory, University of Tennessee
3. HPL.out output file name (if any)
4. 6 device out (6=stdout,7=stderr,file)，定义输出路径，

[1] libgoto2.a其实由两部分组成，3c_ori中compile和interface中的compile，可以看看脚本用了那些源文件

[2] CentOS6上正常使用，6.4multilib会有库相关的问题

[3] 这里将3b的Gotoblas的两个目录中的源文件生成a文件，但是删除a文件重新生成会有问题，目前只能更新两个目录中的源文件然后修改merge到a文件中，TODO

5. 1 # of problems sizes (N)，说明需要测试的矩阵个数，和下一行的数字对应，如为1,则只测试下一行第一个矩阵
6. 58000 60000 30 34 35 Ns，定义矩阵规模的大小
7. 1 # of NBs，7 8行说明求解矩阵分块的大小NB
8. 128 NBs
9. 1 PMAP process mapping (0=Row-,1=Column-major) 选择处理器阵列是按列的排列方式还是按行的排列方式[4]
10. 4 # of process grids (P x Q)
11. 2 Ps
12. 2 Qs

第6行定义了矩阵的规模，矩阵的规模N越大，有效计算所占的比例也越大，系统浮点处理性能也就越高，但与此同时，矩阵规模N的增加会导致内存消耗量的增加，一旦系统实际内存空间不足，使用缓存，性能会大幅度降低。因此，对于一般系统而言，要尽量增大矩阵规模N的同时，又要保证不使用系统缓存。考虑到操作系统本身需要占用一定的内存，除了矩阵A(N×N)之外，HPL还有其它的内存开销，另外通信也需要占用一些缓存(具体占用的大小视不同的MPI而定)。一般来说，矩阵A占用系统总内存的80%左右为最佳，NxNx8=系统总内存x80%。这是一个参考值，具体N 的最优选择还与实际的软硬件环境密切相关

为提高数据的局部性，从而提高整体性能，HPL采用分块矩阵的算法。分块的大小对性能有很大的影响，NB的选择和软硬件许多因数密切相关。NB 值的选择主要是通过实际测试得到最优值。但NB的选择上还是有一些规律可寻，如：NB不可能太大或太小，一般在256以下；NBx8一定是Cache line的倍数等等。NB大小的选择还跟通信方式、矩阵规模、网络、处理器速度等有关系。一般通过单节点或单CPU测试可以得到几个较好的NB值，但当系统规模增加、问题规模变大，有些NB取值所得性能会下降。所以最好在小规模测试是选择三个左右性能不错的NB，在通过大规模测试检验这些选择。

按列的排列方式适用于节点数较多、每个节点内CPU数较少的瘦系统；而按行的排列方式适用于节点数较少、每个节点内CPU数较多的胖系统。在集群系统上，按列的排列方式的性能远好于按行的排列方式。在HPL文档中，其建议采用按行的排列方式，可能和MPI任务递交的不同方式有关。

PxQ = 进程数，这是HPL的硬性规定；PxQ = 系统CPU数= 进程数；P≤Q，这是一个经验值，一般来说，P的值尽量取得小一点；P=2n，即P最好选择2的幂，即P=1 2 4 8 16...。

我们在3b的测试过程中还需要用到hugepage的特性，即16M的page size来降低TLB miss带来的影响，首先确保kernel是支持hugepage的，然后使用proc文件系统提供的接口使能

```
umount /mnt
mount -t hugetlbfs none /mnt
echo 3 >/proc/sys/vm/drop_caches
echo 80> /proc/sys/vm/nr_hugepages
```

line3用来丢弃kernel缓存的页面，清理出更多的可用page，line4用来设置大页。linpack可用的大页由nr_hugepages*16M计算得出。执行命令cat/proc/sys/vm/nr_hugepages，验证设置是否生效，程序执行时可以使用命令cat/proc/meminfo，观察hugepages_free可以判断使用了多少hugepage。测试过程中，根据物理内存实际大小，需要多次试验，最后确定合适的大页。

[4]这个参数从HPL.dat中读入后传入，决定了HPL_pddriver.c中的选择对应的处理函数

A.1 联机测试

在CentOS的联机测试过程中主要配置两点，hostname和免密码的ssh访问。其中hostname的设置

```
//edit /etc/sysconfig/network
    HOSTNAME=node1
//touch /etc/hostname
    node1
    hostname -F /etc/hostname
```

接着需要在hosts解析的过程中将hostname和IP对应起来，比如在/etc/hosts中添加对应项，然后将所有的hostname写入到一个文件，在进行多机测试时候作为mpdboot的输入文件。

```
10.20.43.30 node1
10.20.43.35 node2
```

设置好如上两步后，还需要配置mpd.conf文件

```
echo set secreword=loogson > /etc/mpd.conf
chmod 600 /etc/mpd.conf
```

配置完成后，在主节点启动mpd，即可用mpdboot -n 主机数目-f mpd.host来启动联机系统，比如mpdboot -n 3 -f mpd.hosts。

一些debug的TIPS：

- mpd只需要在一台机器上启动
- 使用mpdtrace -l来检查所有的机器是否已经连接上线

HPL.dat的配置和单机遵循相同的规则。

B 硬件单元

B.1 3A Vector

根据矩阵乘法的特征，3A上有独立的乘加指令[5]。并且从SPEC上出来的结果，可执行代码段中的大部分指令是ld branch store，因此加快访存也是另一种可行的思路[6]。因此3A上有128位访存指令，思路上跟SIMD有点类似。因此，总体来看3A上对浮点计算的硬件支持非常有限，但是使用128位的访存指令对编译器的要求会低一点，更方便的应用在普遍的场景下。

B.2 3B上DMA

3B上的情况比3A会复杂一点，硬件所采取的策略非常激进。和当前Intel的Nehalem相比，浮点计算能力非常客观！但是这样的激进策略导致目前很难在编译器层面进行自动化的支持。

3B上的主要支持即新的乘加指令，这个可以参考朱的论文。并且在访存上使用了DMA类的设计，寄存器堆方面使用了128个256位向量寄存器。简单的概括下3B上矩阵乘法的做法：

1. 锁住3/4的cache，将矩阵划块，将部分矩阵搬运到cache lock所指向的位置

[5]减少了指令数量和寄存器的消耗
[6]缓解存储墙效应，这样看来低频多核的优势略大于高频单核？

2. 配置DMA，将DMA指向的数据load进寄存器堆，使用两个对称区域实现计算和DMA快速切换[7]

3. 在core上，使用乘加指令对寄存器堆中的数进行乘加运算

需要格外注意的是，DMA Engine配置需要大约4拍。根据内存写模式，一般需要在写之前进行读操作，这个时间占据了大部分的启动时间，约30拍，因此在考虑使用DMA的时候需要仔细斟酌其消耗。根据CPU设计的内部设计，4个core挂在AMBA[8]的AXI交叉开关上，这个特征可能是访存路径上的瓶颈。

C 矩阵乘法

GotoBLAS中的矩阵乘法在Goto的文章中有详细的介绍，可以在Git库的document目录下找到，同时朱海涛博士的文章也进行了详细的分析。

D MPI库

MPI库是Intel开发的并行库，主流的超级计算机在进行linack浮点计算的一般采用该库。其被设计成环状的消息交换网络，内部自带一些测试功能，比如消息环路的延迟测试。在使用的过程中，MPI库本身的暂时不存在性能上的瓶颈。MPI的API设计也都在其头文件中，应用到实际的环境中也比较方便。Git库中提供MPI相关的代码和二进制文件。

- 可以使用Git库上的已经编译好的MPI库
- 根据源代码重新编译
- 在CentOS6.0和CentOS6.4中都已经集成mpich2的库，使用yum安装即可

E 3A上linpack

前文已经介绍MPI，另外两个组件是GotoBLAS库和HPL，HPL不需要格外的关注，实现前端和MPI的消息处理，然后调用后端的GotoBLAS库进行矩阵乘法计算。

简单介绍3A上linpack的代码，从hpl-2.0/testing/ptest/HPL_pddriver.c中的main()开始。在代码中详细解释了HPL.dat中的各项数字的意义，关注：NS Ps Qs，一般情况下PxQ的值为CPU的node数目[9]，NS用来控制矩阵的规模。根据我们的HPL.dat中的配置我们需要额外关注下面的三个func。

```
    algo.pffun = HPL_pdpanllT;
    algo.rffun = HPL_pdrpanllT;
    algo.upfun = HPL_pdupdateTT;

    HPL_pdtest(&test, &grid, &algo, val[in], vbval[inb]);
        HPL_pdgesv();
            HPL_pdgesv0();
    HPL_degemm();
#define     HPL_dgemm       cblas_dgemm
#define     HPL_dtrsm       cblas_dtrsm
```

[7]DMA的特性可以分成DRA和DCA，首先进行DCA，后进行DRA。具体的配置方法可以参见向量DMA的手册
[8]AMBA是ARM开放的总线协议，对应的有Intel的QPI和AMD的HT
[9]node数目，根据在Kernel中的算法是：core/4，即8core为2node，可能是因为AMBA的链接的原因导致4core为一个node

在GotoBLAS库中的时候需要关注一下他组织代码的的一个小技巧，可以看到在driver/level3/Makefile中将多种实现map到gemm_new.c中的CNAME的同一个实现上。同样的做法在level3_new.c和gemm_thread_n_new.c中也可以看到。而这些都是我们需要特别注意的部分，我们需要修改和调整的都分布在这三个文件中，他们的表现直接影响到最后的分值。

F 3B上linpack

3B上的linpack比3A上稍微复杂一点，因为3B上非常激进的设计。期望用硬件DMA功能来消除访存所带来的延迟，涉及到内存控制器的性能，但是最后的结果是，从实际计算数值和理论峰值来看，还远没有达到这一目标。相比较Intel的Nehalem来说，比如说i7的4core，峰值可以达到12GFlops左右，实际测试过程中访存并不是瓶颈所在，真正的瓶颈在于浮点计算能力，但是如果i7具有和3B同样的浮点计算能力，在访存方面应该也是比较捉襟见肘的。

截取部分的3B上linpack的代码来进行分析，因为经过很多人的努力，这部分已经趋于稳定。如果要做性能上微调大部分的工作还是集中在如下的几个部分[10]。

```
dgemm_dma_NN_4t()
```

G 调优

调优一般性的办法就是使用oprofile，别的方法就是因为是芯片原厂的有时可以有仿真器去模拟运行linpack，然后观察部件的协调性。

G.1 oprofile

最常用的办法，因为linpack在计算的过程中采用每core每thread的办法，在采样的过程中主要针对CPU的core进行采样。因为cache lock的原因我们不对cache的效率进行格外的评估，主要分析core上的运行时间片。

oprofile的代码可以在wiki上找到，loongson上可用的CPU event可以在同一页面上找到，如下的命令可以对单core的event进行采样。运行完成后，shutdown掉oprofile，使用opreport将结果重定向文件进行分析。

```
opcontrol --init
opcontrol --reset
opcontrol --no-vmlinuxx
opcontrol --setup --sepatate=cpu --event=CPU_CLK_UNHALTED:100000:0
opcontrol --start
```

截取部分的例子看看主要的时间花费在那里，通过代码可知count即为核心中的计算部分，通过addr2line即可将count的地址对应到具体的代码，可以看到3B上8core，对应8组数据，所获得的sample数和占用的百分比。因此我们可以推测计算占用了多长时间。并且接下来的指令是bc2f，即等待DMA传输完成的类test bit的等待指令，理论上来说，DMA的根本目的在于消除访存带来的延迟，即在一组计算完毕后另外一组数据已经在寄存器堆中，不需要bc2f等待多拍即可开始计算，但是实际的profile出来的结果是，还是会在两组不同的计算间有一段时间的等待。访存的延迟并没有完全消除。

[10]在代码中有一些统计各个部分的时间的打印，可以将这些打印开关打开，会发现在进行大规模的计算时候时间都去了哪儿

```
0000000120058524 <count>:
/* count total: 2078559 12.50772047146 34.82872053229 35.20102050694 34.86512103309
                        12.47222093139 34.93092085283 34.74642097683 35.2782 */
 81822  0.4924 72429  1.2323 71810  1.2311 72012  1.2243
 82264  0.4878 73486  1.2264 73392  1.2229 73561  1.2371
 :    120058524: bc2f     120058524 <count>
```

那么该如何解释这一现象？

下面是我们在3B的机器上用stream来测试内存带宽的数据：

```
Function      Rate (MB/s)   Avg time     Min time     Max time
Copy:         1233.5621       0.0263       0.0259       0.0270
Scale:        1242.5727       0.0262       0.0258       0.0268
Add:          1483.4950       0.0330       0.0324       0.0334
Triad:        1474.5204       0.0328       0.0326       0.0331
```

根据DMA功能部件的设计可以分析得出DMA的搬运数据时间除了依赖配置使用，各大的部分还是取决于内存控制器的带宽。我们可以根据如上的带宽计算出在计算时间内可以搬运多少个双精度浮点数。并且实际的情况可能还会比理论数值更低，如果考虑到在高负载的情况下多个DMA channel在内存控制器上产生的race condition会产生堵塞，这个需要怎么去验证？

G.2 Beyond Software

可能这就是芯片厂商的优势所在，我不清楚别的软件厂商是如何做到的，但是观察大部分跑linpack的场景，多半是芯片厂商，自然会引入仿真器分析，并且linpack作为前期处理器性能评估和验证必不可少的大型软件。