矩阵乘法的高性能实现

摘要：

高性能的稠密矩阵乘法关键在于，注意处理器的多级存储并进行 Blocking，即希望读进每一级存储的数据被用于计算的次数尽可能多，降低从下一级存储访问数据的频率。常见的 Blocking 有针对各级缓存的。此外，由于寄存器的访问速度与缓存相比又有巨大的提升，针对寄存器的 Blocking 可以把处理器的 Performance 推向接近峰值。对于 CPU 和 KNL，本实验直接利用 x86 intrinsic 编写矩阵乘法的核心部分来实现寄存器 Blocking，并在 CPU 上达到了和 MKL 高性能计算库相近的效果。对于 GPU，主要利用了 shared memory 和寄存器进行 Blocking。

1. CPU

服务器的 CPU 是 24 核 Haswell 体系结构，每核支持两个超线程。向量化宽度为 256 个比特。每核的 L2 缓存大小为 256KB 左右。针对这些特点，主要进行了如下优化。

1.1 Cache Blocking

Haswell 的各级存储 Latency 如表 1 所示。

表 1: Haswell 各级存储的 Latency[1]

存储	Latency（时钟周期）
L1 Cache	4-5
L2 Cache	12
L3 Cache	43-58
RAM	62

从上表可以看出，L1 缓存和 L2 缓存的带宽差别并不是很大。而 L2 和 L3 之间存在较大差距。因此针对 L2 缓存作 Blocking 能得到较大的性能提升。在计算每个 Block 的时候需要存储两个输入矩阵和输出矩阵的一共 3 个 Block。计算得到合适的正方形 Block 边长为：$\sqrt{\frac{256K}{8\times 3}} \approx 103$，为了方便寄存器 Blocking（见 1.2），取边长为 96。

1.2 Register Blocking

Haswell 一共有 16 个 256bit 宽的 ymm 寄存器，ymm0-ymm15。对于矩阵乘法 $C = A\times B$，矩阵 C 的元素需要不停地被累加，因此把 C 的元素保存在寄存器中比保存 A 和 B 的元素更可以降低读写内存的开销。选定 register block 的大小为 4 x 12 个元素。即每次 C 矩阵的 4 x 12 个元素存储在寄存器中，然后遍历 A 的每列 4 个元素和 B 的每行 12 个元素，如图 1 所示

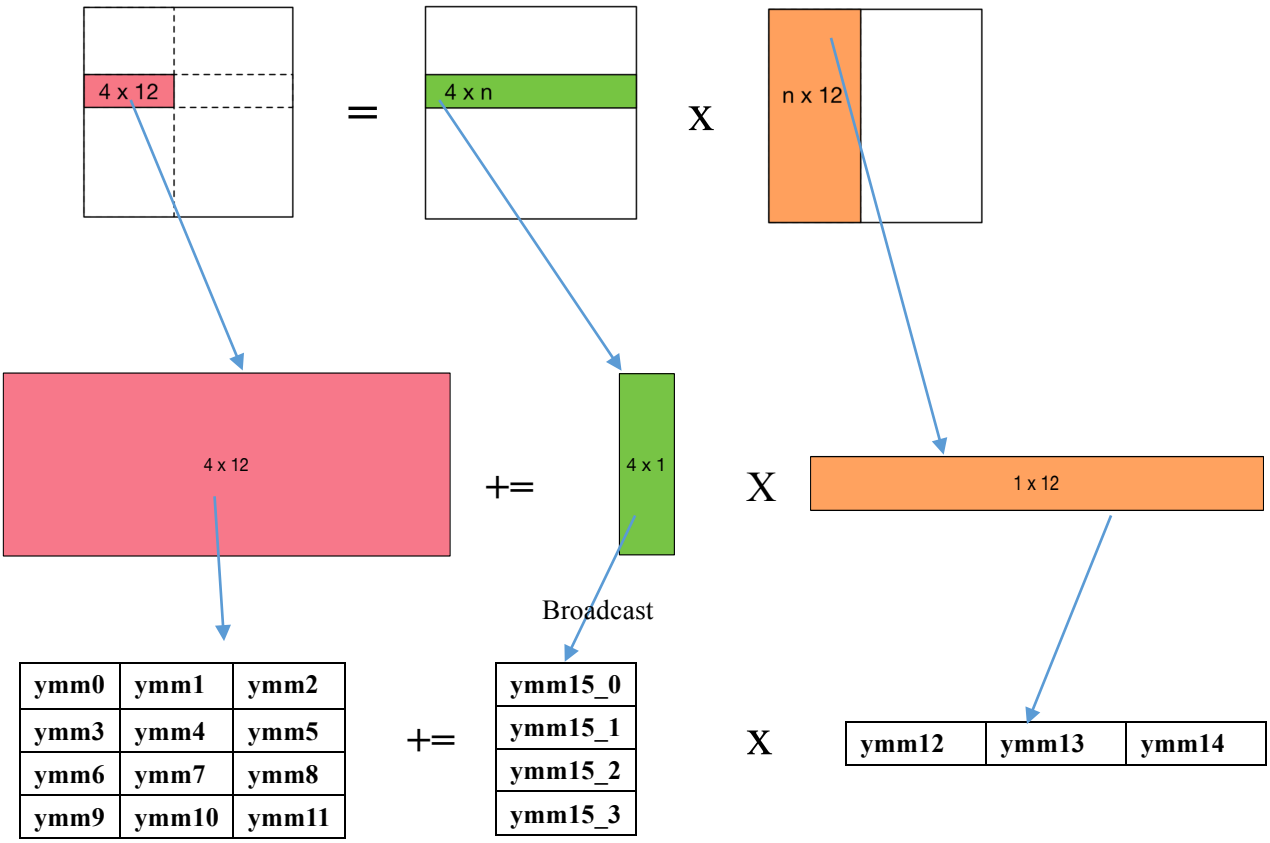

图 1 Register Blocking

如图 1，对于 C 中每一个 4x12 的小块，都依次遍历 A 中对应的 4x1 的列和 B 中对应的 1x12 的行。把 C 的 4x12 小块保存在 ymm0-ymm11 中。依次取出 A 中 4x1 的列和 B 中 1x12 的行到寄存器中，把它们相乘之后加到 C 中。由于寄存器数量有限，在做这个乘法的时候，需要特别注意寄存器的分配。首先，把 B 的 1x12 的行存入 ymm12-ymm14 中。然后读入 A 的 4x1 列第 1 个元素，broadcast 到 ymm15 中。此时 ymm15 存有 4 份 A 的 4x1 列的第 1 个元素，把 ymm15 与 ymm12-ymm14 依次相乘，结果加到 ymm0-ymm2 中，这一步使用 fma 指令。然后再读入 A 的 4x1 列的第二个元素，再 broadcast 到 ymm15 中，此时 ymm15 存有 4 个 A 的 4x1 列的第 2 个元素。把 ymm15 与 ymm12-ymm14 一次相乘，结果加到 ymm3-ymm5 中。对 A 的 4x1 列的每个元素重复这个步骤，就可以使用刚好 16 个寄存器。

由于 register blocking 是在 L2 Cache Blocking 的基础上进行的，因此，为了方便处理 L2 Cache 的 Block 长度应该选为 12 的倍数，这也是我们在 1.1 中选取 L2 Block 的大小为 96 x 96 的原因。

1.3 数据对齐

每个矩阵都存储在 c++的 2 维 STL vector 中。每一行的第一个元素都是 64-bit aligned。因为我们的 register block 的长宽都是 4 的倍数，因此进行对齐之后，

代码里的所有 load 和 store 操作都是对齐的，可以直接使用_mm256_load_pd 和 _mm256_store_pd。对齐使得 register blocking 的效果进一步提升。

1.4 性能测试

在单台机器(cn002)上使用 24 个线程的测试结果如表 2。正确性检验是直接与 MKL 的计算结果逐个元素比较，误差设置为 1e-6。（由于之前在集群上自己的目录底下安装了 mkl 库，所以 Makefile 里面指定的 MKL_ROOT 路径都是 2017211384 目录底下的 mkl。为了避免在其他地方编译的麻烦，矩阵乘法代码和 Makefile 中所有的与 mkl 测试有关的部分都注释掉了。但是已经测试过所有程序，可以保证正确性）。

表 2: CPU 性能测试

矩阵规模	我的 Gflops	MKL Gflops	我的时间(秒)	MKL 时间(秒)
1000 x 1000	264.27	222.69	0.00756788	0.00898099
2000 x 2000	258.87	480.54	0.061806	0.0332961
3000 x 3000	298.98	120.27	0.180609	0.448984
5000 x 5000	288.71	280.88	0.865921	0.890059
10000 x 10000	304.11	624.52	6.57659	3.20245

2. GPU

GPU 的实现，主要参考了[2]和[3]，实现了 shared memory blocking 和 register blocking。每个线程块负责计算矩阵 C 的一个 Block，计算由两层循环构成，外循环负责把 A 和 B 的 Block 依次读入 share memory，并把这两个 Block 的乘积累加起来，即进行 shared memory blocking。内循环使用 register blocking 计算 A 和 B 的 Block 的乘积。

2.1 Shared Memory Blocking

集群上 GPU 的计算能力为 3.5，每个 SM 的 shared memory 大小为 48KB。给每个线程块分配 2 个 64 x 16 大小的 shared memory，一共 16KB，分别用于存放矩阵 A 和 B 的元素。每个线程块有 8 x 16 个线程，负责计算矩阵 C 的 64 x 64 个元素，其中第(0,0)号线程负责计算 C 中对应的(0, 0), (0, 16), (0,32), (0,48), (8, 0), (8,16), (8,32), (8,48),…,(56, 0), (56, 16), (56,32), (56,48)号元素,一共 32 个。其他线程依次类推。

这一算法是根据[3]改进的，图 2 引自[3]，其中 N_{TX} 和 N_{TY} 分别表示线程块的 BlockDim.x 和 BlockDim.y，因此[3]的算法每个线程块有 16 x 16 个线程。图 2 中矩阵 C 的小点表示线程(0,0)需要计算的元素。在这样的分配方式下，[3]提出的算法是针对 Fermi GPU 的，而集群的 Kepler GPU 每个线程可以使用的寄存器

数量为 255 个，是 Fermi 的 4 倍。为了充分利用寄存器，把线程块的线程数目变成 8 x 16 个, 这样每个线程计算的元素数目翻倍。经测试，4 x 16 线程的线程块表现不如 8 x 16 的，因此实验中均使用 8 x 16 的线程块。

每个线程块的外循环依次读入 64 x 16 个 A 矩阵元素和 16 x 64 个 B 矩阵元素，直到遍历 A 的对应行所有元素和 B 的对应列所有元素，如图 2 所示。

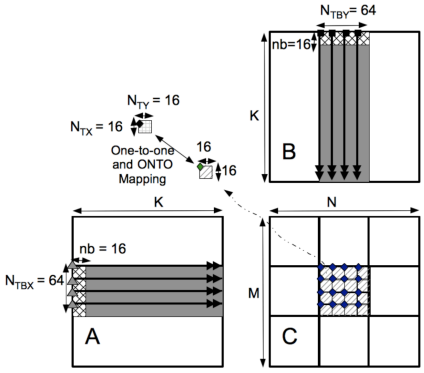

图 2[3]

2.2 Register Blocking

如 2.1 中描述的，8 x 16 个线程需要计算 64 x 64 个元素，因此每个线程负责计算 8 x 4 个元素。下面以线程(0,0)为例。首先，(0,0)在寄存器中分配 8 x 4 个元素来累加计算结果。然后读取 shared memory 存的矩阵 A 的 Block 的第(0, 0),(0, 8), (0, 16),…,(0, 56)号元素到寄存器中，共 8 个，看成一个列向量 a。再读取 shared memory 中矩阵 B 的 Block 的第(0, 0), (0, 16), (0,32), (0,64)号元素到寄存器中，共 4 个，看成一个行向量 b。计算 a 和 b 的乘积，得到 8 x 4 的矩阵，累加到线程保存的矩阵 C 的 8 x 4 个元素中。然后再取 shared memory 中 A 的下一列和 B 的下一行，计算乘积并累加。这样，每个 A 中的元素读入寄存器之后都被利用了 4 次，B 的每个元素读入寄存器之后都被利用了 8 次。

2.3 性能测试

在单个 GPU 上的性能如表 3 所示。正确性检验是直接与 CPU 上 MKL 的计算结果逐个元素比较，误差设置为 1e-6。其中时间记录的是纯计算部分。

表 3: GPU 性能测试

矩阵规模	Gflops	时间（秒）
1000 x 1000	620.31	0.00322422
2000 x 2000	655.40	0.0244124
3000 x 3000	703.60	0.0767486
5000 x 5000	689.88	0.362384
10000 x 10000	707.89	2.82531

3. KNL

KNL 的实现与 CPU 的大体类似。不同的是 KNL 每个核有 1MB 的 L2 Cache，因此，L2 Block 的大小需要调整为 192 x 192。此外，KNL 有 48 个 512bit 的寄存器，因此 Register Blocking 的规模也要做调整，从 1.2 中的 4 x 12 调整为 4 x 48。实现仍然使用 x86 intrinsic。用 24 个 512bit 的寄存器累加矩阵 C 的结果，再用 6 个 512bit 的寄存器依次读取矩阵 B 的行，然后用 1 个 512bit 的寄存器依次 broadcast 读取矩阵 A 的元素，一共用到 32 个寄存器。使用 72 个线程。性能测试结果如下：

表 4: KNL 性能测试(DDR)

矩阵规模	我的 Gflops	MKL Gflops	我的时间	MKL 时间
1000 x 1000	252.30	21.76	0.00792694	0.091917
2000 x 2000	195.37	63.38	0.081897	0.252449
3000 x 3000	136.97	49.67	0.394259	1.08716
5000 x 5000	140.80	211.22	1.77552	1.18362
10000 x 10000	150.00	1329.50	13.3331	1.50433

表 5: KNL 性能测试(MCDRAM)

矩阵规模	我的 Gflops	MKL Gflops	我的时间	MKL 时间
1000 x 1000	255.94	36.19	0.00781417	0.0552649
2000 x 2000	257.65	35.57	0.062098	0.44976
3000 x 3000	240.00	68.72	0.225004	0.785756
5000 x 5000	221.25	353.82	1.12993	0.706577
10000 x 10000	230.17	1156.52	8.68935	1.72932

CPU 使用 I0

使用 IO 计算的算法大致如下。设方阵 A 和 B 的维度为 n。将矩阵 A 按列拆解，$A = [a_1, a_2, \dots, a_m]$, 其中$a_i$是 n 乘 k 大小的子块，对应地将矩阵 B 拆解为 k 乘 n 大小的子块, $b_1, \dots, b_m$，k 的选取使得b_i可以放入内存。则$AB = \sum_{i=1}^{m} a_i b_i$。设算法在第 t 步的结果为$C_t$，则$C_{t+1} = C_t + a_t b_t$。可以把$C_t$存放在文件里面，然后计算$C_{t+1}$的时候依次逐行读入$C_t$,再把新的结果逐行写出。由于矩阵是按行存储的，为了计算$a_t b_t$，需要把矩阵 A 全部读取一遍，因此 IO 的开销是很大的。实验的 k 选取为 192。性能测试如下表：

表 6: CPU-IO 性能测试

矩阵规模	Gflops	我的时间
100 x 100	0.0295	0.067754
200 x 200	0.0394	0.405634
300 x 300	0.0998	0.541133

500 x 500	0.1096	2.28086
1000 x 1000	0.0888	22.5108

参考资料：

[1] http://www.7-cpu.com/cpu/Haswell.html

[2] Tan G, Li L, Triechle S, et al. Fast implementation of DGEMM on Fermi GPU[C]//Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, 2011: 35.

[3] Nath R, Tomov S, Dongarra J. An improved MAGMA GEMM for Fermi graphics processing units[J]. The International Journal of High Performance Computing Applications, 2010, 24(4): 511-515.