CS 301
High-Performance Computing

Lab 3 - B1

Matrix Multiplication using Transpose

Aditya Nawal (202001402)
Divya Patel (202001420)

February 27, 2023

Contents

1 Introduction

This report investigates three different approaches to matrix multiplication. In Problem A-1, we explore the conventional matrix multiplication algorithm and do it in six different ways to interchange the loops to optimize its performance. In Problem B-1, we study the use of transpose matrix multiplication as a way to reduce the computational cost of matrix multiplication. Finally, in Problem C-1, we analyze the block matrix multiplication algorithm, which uses a divide and conquer strategy to compute the product of two large matrices.

For each problem, we provide a detailed analysis of the algorithms, including their computational complexity. We also implement each algorithm and evaluate its performance on matrices of different sizes. The results of our experiments are presented in the form of graphs.

2 Hardware Details

2.1 Lab 207 PC

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 4
- On-line CPU(s) list: 0-3
- Thread(s) per core: 1
- Core(s) per socket: 4
- Socket(s): 1
- NUMA node(s): 1
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 60
- Model name: Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz
- Stepping: 3
- CPU MHz: 3300.000
- CPU max MHz: 3700.0000
- CPU min MHz: 800.0000
- BogoMIPS: 6585.38

- Virtualization: VT-x
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 6144K
- NUMA node0 CPU(s): 0-3
- Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm epb invpcid_single tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid xsaveopt dtherm ida arat pln pts

```
[student@localhost ~]$ ifconfig
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1  (Local Loopback)
        RX packets 60  bytes 5868 (5.7 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 60  bytes 5868 (5.7 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

p4p1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.100.64.86  netmask 255.255.255.0  broadcast 10.100.64.255
        inet6 fe80::b283:feff:fe97:d2f9  prefixlen 64  scopeid 0x20<link>
        ether b0:83:fe:97:d2:f9  txqueuelen 1000  (Ethernet)
        RX packets 32826  bytes 46075919 (43.9 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 8015  bytes 586362 (572.6 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

virbr0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 192.168.122.1  netmask 255.255.255.0  broadcast 192.168.122.255
        ether 52:54:00:3a:16:71  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figure 1: IP address of Lab PC

2.2 HPC Cluster

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 16
- On-line CPU(s) list: 0-15
- Thread(s) per core: 1
- Core(s) per socket: 8
- Socket(s): 2
- NUMA node(s): 2
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 63
- Model name: Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz
- Stepping: 2
- CPU MHz: 1976.914
- BogoMIPS: 5205.04
- Virtualization: VT-x
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 20480K
- NUMA node0 CPU(s): 0-7
- NUMA node1 CPU(s): 8-15

3 Problem B1

3.1 Description of the problem

Given two matrices A and B of dimensions $m \times n$ and $n \times p$, respectively, the standard matrix multiplication algorithm requires $O(mnp)$ operations to compute the product $C = AB$, where C is a matrix of dimensions $m \times p$. This computational cost is prohibitive for large matrices and, therefore, efficient algorithms for matrix multiplication are of great interest.

One possible approach to reduce the computational cost of matrix multiplication is to use the transpose of one of the matrices. In particular, we can compute the product $C = AB$ as follows:

$$C = A \cdot B^T, \tag{1}$$

where B^T is the transpose of B.

we convert $C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$ to $C_{ij} = \sum_{k=1}^{n} A_{ik} B_{jk}$ When you multiply two matrices A and B, you need to access the elements of B by columns, which are not stored contiguously in memory if you use row-major ordering. This can result in many cache misses and slow down the operation.

However, if you transpose B first, then you can access its elements by rows, which are stored contiguously in memory. This can reduce the number of cache misses and speed up the operation.

Of course, transposing B also requires accessing its elements non-sequentially, but this is done only once before the multiplication. The multiplication itself is done many times and benefits from cache locality.

3.2 Serial Complexity

This method requires same number of operations as the standard matrix multiplication algorithm, i.e., $O(n^3)$.

3.3 Profiling Information

Following are the snapshops taken while profiling.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.40      4.28     4.28                             main
  0.00      4.28     0.00        2     0.00     0.00  diff

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
       else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
       function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
       the function in the gprof listing. If the index is
       in parenthesis it shows where it would appear in
       the gprof listing if it were to be printed.


Copyright (C) 2012-2014 Free Software Foundation, Inc.
```

Figure 2: Screenshot of text file generated from profiling on Lab 207 PC using gprof

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.40      4.51     4.51                             main
  0.00      4.51     0.00        2     0.00     0.00  diff

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.


Copyright (C) 2012 Free Software Foundation, Inc.
```

Figure 3: Profiling on HPC cluster using gprof

3.4 Optimization Strategy

We are not optimizing algorithm for martix multiplication as here in this question we are supposed to do it in $\mathcal{O}(n^3)$ time complexity. We are optimizing little bits of code like writting C[i][j] += A[i][k] * B[k][j] instead of C[i][j] =C[i][j] + A[i][k] * B[k][j] to reduce the number of memory access.

The reason why C[i][j] += A[i][k] * B[k][j] has better performance than C[i][j] =C[i][j] + A[i][k] * B[k][j] is that it avoids an extra memory access for C[i][j]. In the first case, C[i][j] is only read once and updated once. In the second case, C[i][j] is read twice and updated once. This can make a difference when dealing with large matrices and cache memory.

3.5 Graph of Problem Size vs Algorithm Runtime

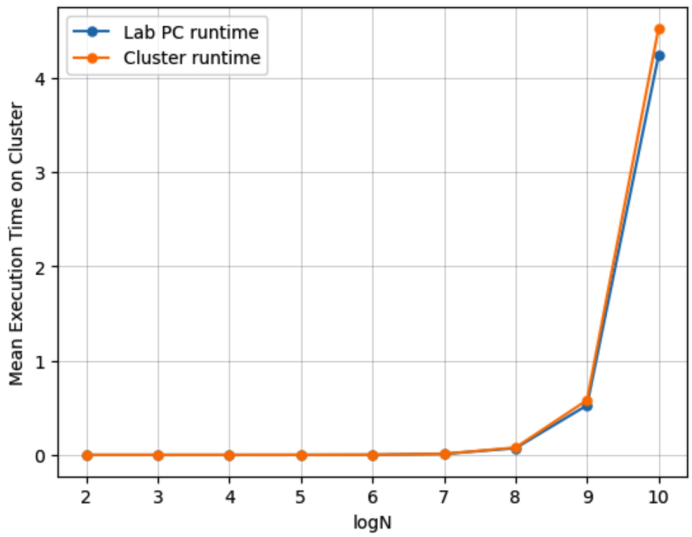

Figure 4: Graph of Problem Size vs Algorithm Runtime for Lab PC and HPC cluster

3.6 Graph of Problem Size vs End-to-End Runtime

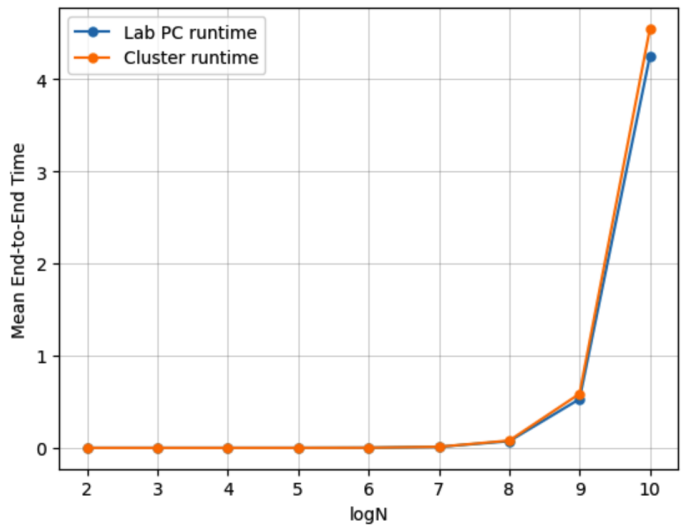

Figure 5: Graph of Problem Size vs End-to-End Runtime for Lab PC and HPC cluster

3.7 Discussion

In this report, we have explored the idea of transposing a matrix before multiplication as a way to improve the computational efficiency of the operation. We have seen that transposing a matrix can improve the cache locality of the data access and reduce the number of cache misses. We have also learned that transposing a matrix has some properties such as:

1. The transpose of the transpose of a matrix is equal to the original matrix.
2. The transpose of a scalar multiple of a matrix is equal to the scalar multiple of the transpose of the matrix.
3. The transpose of the product of two matrices is equal to the product of their transposes in reverse order.

These properties can help us manipulate matrices and simplify expressions involving transposes. However, we should noted that transposing a matrix before multiplication is not always faster than normal matrix multiplication, as it depends on various factors such as the size and shape of the matrices, the implementation of the algorithm, and the hardware architecture.