CSDS 451: Designing High Performant Systems for AI
Fall 2023
PA 1
MANSIH SRI SAI SURYA ROUTHU (mxr809)

Writing code that works properly is very difficult, however tuning that code to make it run as fast as possible can also be challenging. Most of deep learning can be broken down into a series of basic operations performed on multidimensional data like matrix multiplication. Although a seemingly simple concept, decades of research has been put into reducing the amount of time (and number of operations) it takes to multiply two matrices. This assignment will have you explore how to use these tuned operations and see how much faster they can be when compared to a naive approach.

NOTE: Please comment your code appropriately. We will review and compile the code that you write and verify that it works ourselves. **Also, everything must be able to be compiled on the HPC, so please let us know which versions of libraries/compilers that you are using.**

Submission: This assignment consists of two parts: **a) code** and your **b) write-up**.

Code: You should provide all source code/data files in a zip file. Please do not include PDFs in the zip so that they may be viewed using Canvas.

Write-up: All responses to questions that are not written code should be typed and compiled to PDF. There is no requirement for what text editing tool that you use, however we recommend using LaTeX or Markdown. **Hand-written submissions (including using a tablet), outside of drawn diagrams for certain problems, will not be graded.**

1) An Introduction to BLAS: BLAS stands for **Basic Linear Algebra Subprograms** and is a library originally from 1979 written in Fortran with highly-tuned functions for many different routines. These operations are broken down into three levels: level 1 are only vector operations, level 2 are vector-matrix operations, and level 3 are matrix-matrix operations. This problem focuses on a level 3 routine.

a) Go on the HPC and figure out which BLAS library is installed and choose one to use. Then, go online and find the routine for matrix multiplication. We will use the the d-variant here, so please use **DGEMM** for double precision. **Do not use another variant of the matrix multiplication.**
   - What inputs does the routine take? **(1 pt)**

Answer:

In BLAS, the DGEMM routine takes following inputs:

i. A,B – two input matrices for multiplication

ii. M,K,N – Dimensions of input matrices (A – M x K, B – K x N)
iii. (α, β) – Scaling factors of the matrices A and B
iv. Pointers which locate to the memory locations of input and output matrices
v. Leading dimensions for matrices A, B, and C

---

b) **BONUS: 1 point to your overall grade. Answer should be detailed and a minimum of 2 paragraphs, you will not receive credit if you do not cite sources.**

- What techniques can be used to speed up matrix multiplication? Topic ideas include data-entry order (caching), SIMD, and algorithmic advances.

Answer:

Techniques used to speed up matrix multiplications are as below:

i. SIMD(Single Instruction multiple data) – Performing multiple operations on data in parallel can speed up the matrix multiplication.
Source: https://en.wikipedia.org/wiki/Single_instruction,_multiple_data
SIMD (Single Instruction, Multiple Data) is a parallel processing approach where multiple processing units execute identical operations on multiple data elements concurrently. It is essential for enhancing performance in multimedia tasks like image and audio processing. SIMD can be implemented in hardware or accessed through an instruction set architecture (ISA). Unlike true concurrency, where different instructions are executed simultaneously, SIMD units perform the same instruction simultaneously on diverse data. This Flynn's taxonomy concept is found in modern CPUs and accelerates data-intensive operations by applying a single operation to multiple data points, significantly improving computational efficiency.

ii. Data-entry Order (Caching) – Here the order in which the data is stored in the memory is changed. Reordering the data in memory for the improvisation in the utilization of cache if a good idea. Column-major style is one of the ideas of such ordering.

iii. Algorithmic Advances: More advanced and efficient algorithms have been developed until recent times. It's a good idea to implement these algorithms. They are as follows:

a. Strassen's algorithm: $O(n^{2.807})$
Source: https://en.wikipedia.org/wiki/Strassen_algorithm
The Strassen algorithm, named after Volker Strassen, is a matrix multiplication technique in linear algebra. It offers improved efficiency over the standard method for large matrices due to its better asymptotic complexity. However, for smaller matrices, the naive approach may be more efficient. While Strassen's algorithm is not the fastest for extremely large matrices, it's highly practical for real-world applications, as the fastest-known algorithms for such matrices are not feasible. Strassen's method is applicable to various mathematical structures, like rings with addition and multiplication, but not all, such as semirings like min-plus or Boolean algebra, where the naive approach remains effective.

b. Schonhage algorithm: $O(n^{2.522})$
Source: https://saturncloud.io/blog/understanding-the-schnhagestrassen-algorithm-a-comprehensive-guide-to-huge-integer-multiplication/

The Schönhage–Strassen algorithm, developed by Arnold Schönhage and Volker Strassen in 1971, is an efficient multiplication method for large integers. It employs a recursive application of the Fast Fourier Transform (FFT) modulo 2n+1. The algorithm's bit complexity for multiplying two n-digit numbers is O(nlog nlog log n). It remained the fastest known method for large integer multiplication from 1971 until 2007 when Martin Fürer introduced a faster algorithm. Although theoretical algorithms with even better complexity exist, their impractical constant factors make them unsuitable for real-world applications. Schönhage–Strassen finds use in diverse areas, including prime number searches, π approximations, and factorization via Kronecker substitution.

c. Alman, Williams: $O(n^{2.3728596})$
d. Coppersmith-Winograd algorithm: $O(n^{2.371552})$

Source: https://epubs.siam.org/doi/10.1137/1.9781611975031.67

Recent advancements in matrix multiplication complexity analysis, focusing on Coppersmith-Winograd tensor powers, have led to improved rectangular matrix multiplication algorithms. These developments establish a lower bound denoted as $\alpha > 0.31389$ for $n \times n^{\alpha}$ matrix multiplication by $n^{\alpha} \times n$ matrices, enhancing prior bounds and offering faster algorithms for various problems dependent on rectangular matrix multiplication. This includes tasks like computing shortest paths in weighted directed graphs, where it serves as a crucial bottleneck.

When multiplying two numbers, if integers are too large we can follow the Karatsuba algorithm – which reduces the complexity of multiplication to $O(n^{1.59})$ from $O(n^2)$ where n is the length of the integer.

---

c) How many floating point operations are required to multiply two square matrices, both of side length 1,024? **(1 pt)**

Answer:

Total number of multiplications while multiplying two matrices:

for the multiplication of two N x N matrices the number of multiplications = $N^2 \times N$

Explanation – N^2 is the number of elements in the output matrix

N is the number of multiplications for every element in the output matrix.

Similarly, number of additions = $N^2 \times (N - 1)$

Explanation - N^2 is the number of elements in the output matrix.

N-1 is the number of addition operations needed to add N numbers.

Total operations = $N^2 \times N + N^2 \times (N - 1)$

$= N^2 \times (2N-1)$

Given, N = 1,024

Hence, the number of operations = $(1024)^2 \times (2 \times 1024 - 1)$

$= 2,146,432,072$

---

d) Create two square matrices of side length 2,048 using random doubles (this program does not need to be turned in). Save the contents to a text file so that this is reproducible. In a separate program, read in the contents of the text files to matrices, then use the **DGEMM** operation to multiply them. Save the output matrix to a text file as well. Time the operation, but make sure to only capture the routine's time (not the reading of the files nor the writing). (**HINT: Look up how to create a timer in C++. Make sure it doesn't only count CPU cycles, but is converted to unit of time.**

- Now that we have time, how many FLOPS did the routine achieve? **(3 pts)**

Answer:

Two square matrices of side length 2048 are created and stored in **matrix_1.txt** and **matrix_2.txt**. The code file **matrix_mul_q1_d.cpp** is executed and the result of the matrix multiplication is stored in **result_matrix.txt.**

Execution time :7.20973 Sec

Number of FLOPS = $(N^2 \times (2N-1)) /$ Execution time

$= (2048^2 \times (2 \times 2048 - 1))/ 7.20973$

$= 17175674880/7.20973$

$= 2382290998.414642434$ FLOPS

```
[mxr809@hpc5 ~]$ module load blas
[mxr809@hpc5 ~]$ g++ matrix_mul_q1_d.cpp -lcblas
[mxr809@hpc5 ~]$ ./a.out
Matrix multiplication routine time: 7.20973 seconds.
[mxr809@hpc5 ~]$
```

---

2) A Naive Approach: Now that you have successfully implemented a BLAS routine, let's take a step back and have you create your own basic matrix multiplication program.

a)Write a program that reads in the two matrices you created for **Problem #1** and multiplies them. Write the resulting matrix to another text file.

- This should be the most basic implementation you can think of, **do not** incorporate optimizations like parallelization or tiling. **(3 pts)**

Answer:

Program file name: matrix_mul_q2_a.cpp
Result_file: result_matrix_naive.cpp

```
[mxr809@hpc5 ~]$ g++ matrix_mul_q2_a.cpp
[mxr809@hpc5 ~]$ ./a.out
Matrix multiplication completed and result saved to result_matrix_naive.txt.
Matrix multiplication routine time: 253.179 seconds.
[mxr809@hpc5 ~]$
```

b)Write a program that reads in the result from BLAS and your result. Iterate through both matrices comparing each value to make sure that they are identical to two decimal places. **(1 pt)**

Answer:

Program file name: matrix_mul_q2_b.cpp
Result – The matrices are equal (considering precision up to 2 numbers after decimal)

```
[mxr809@hpc5 ~]$ g++ matrix_mul_q2_b.cpp
[mxr809@hpc5 ~]$ ./a.out
Matrices are identical to two decimal places.
[mxr809@hpc5 ~]$
```

c)Time only the matrix multiplication operation (not reading or writing of files).

- How many FLOPS do you achieve with your implementation? **(1 pt)**

Answer:

Execution time :253.179 Sec

Number of FLOPS = $(N^2 \times (2N-1)) / \text{Execution time}$

$= (2048^2 \times (2 \times 2048 - 1))/ 253.179$

$= 17175674880/253.179$

$= 67840045.5$ FLOPS

3) LAPACK: LAPACK stands for **Linear Algebra Package** and was initially released in 1992. Like BLAS, it provides routines for linear algebra operations. You'll be using it

for this question to find the LU decomposition of the resulting matrix from your BLAS matrix multiplication.

a) Find a version of LAPACK on the HPC that you will use. Then go online and figure out which routine you will use for the factorization. What function did you find? If you are unsure that you have the correct one, then reach out to a TA for some guidance.
(1 pt)

Answer:

LAPACK 3.7.0 and 3.9.1 modules are available in the HPC

```
[mxr809@hpc6 ~]$ module spider lapack/3.9.1

----------------------------------------------------------------------------------------------------------------------------------------------------------------
  lapack: lapack/3.9.1
----------------------------------------------------------------------------------------------------------------------------------------------------------------
    Description:
      provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value
      problems. The associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur) are also provided, as are related computations such as reordering of the Schur
      factorizations and estimating condition numbers. Dense and banded matrices are handled, but not general sparse matrices. In all areas, similar functionality is provided for
      real and complex matrices, in both single and double precision.

    You will need to load all module(s) on any one of the lines below before the "lapack/3.9.1" module is available to load.

      gcc/7.3.0


[mxr809@hpc6 ~]$ module load gcc/7.3.0

The following have been reloaded with a version change:
  1) gcc/6.3.0 => gcc/7.3.0     2) openmpi/2.0.1 => openmpi/3.1.6

[mxr809@hpc6 ~]$ module load lapack/3.9.1
```

I chose the 3.9.1 version

Here I chose to use **LAPACE_dgetrf** function for generating L and U matrices

b) Read in your resulting matrix from the matrix multiplication and run the LU decomposition on it.

- How long does it take to run (only the routine, not reading in the files)? Is it faster than the matrix multiplication? **(3 pts)**

Answer:

Program file: **matrix_mul_q3_b.cpp**
Resultant Matrices: **matrix_L.txt**, **matrix_U.txt**
Execution Time: 1.83 Sec

```
[mxr809@hpc6 ~]$ g++ matrix_mul_q3_b.cpp -llapacke
[mxr809@hpc6 ~]$ ./a.out
LU decomposition completed in 1.83 seconds.
[mxr809@hpc6 ~]$
```

c) Write a program that runs the routine on the following matrix:

- $\begin{bmatrix} 4.0 & 3.0 \\ 6.0 & 3.0 \end{bmatrix}$

- What are the resulting **L** and **U** matrices? **(1 pt)**

Answer:

Program file: **matrix_mul_q3_c.cpp**

Result: The resultant matrices L and U are shown in the below image

```
[mxr809@hpc6 ~]$ g++ matrix_mul_q3_c.cpp -llapacke
[mxr809@hpc6 ~]$ ./a.out
Enter the size of the square matrix (n): 2
Enter the elements of the matrix:
4.0
3.0
6.0
3.0
LU decomposition successful. The L and U matrices are:
L matrix:
1 0
0.666667 1
U matrix:
6 3
0 1
[mxr809@hpc6 ~]$
```

---

4) Faster Matrix Multiplication: As we've seen above, the basic triple for loop implementation for matrix multiplication is vastly outperformed by the BLAS routine. One method of making a faster function is called blocking/tiling. You will create your own version here that will operate on the same matrices that you created in **Problem #1** and used in **Problem #2**.

a) How does blocking work in matrix multiplication? What does it do that helps to accelerate the operation? **(3 pts)**

Answer :

Blocking, also referred to as loop blocking or tiling, is a technique employed in matrix multiplication to enhance the efficiency of the operation, particularly in relation to cache utilization and memory access patterns. Its purpose is to break down large matrices into smaller blocks or tiles, enabling matrix multiplication to be performed on these smaller units. The following explanation outlines the functioning of blocking and its contribution to accelerating matrix multiplication:

Matrix Partitioning: In the conventional matrix multiplication algorithm, each element of the resulting matrix is obtained by accumulating the sum of products of corresponding elements from the input matrices. Blocking involves partitioning the input matrices (A and B) as well as the output matrix (C) into smaller submatrices or blocks.

Submatrix Multiplication: Instead of conducting matrix multiplication on the entire matrices A and B, it is performed on these smaller submatrices, commonly known as blocks. For instance, if a large matrix A of size N x N is being processed, it can be partitioned into smaller blocks of size M x M, where M is chosen to effectively fit within the cache size.

Cache Miss Reduction: By working with smaller blocks, it becomes possible to efficiently load these blocks into the cache. The advantage lies in the fact that once a block is loaded into the cache, numerous multiplications and additions can be performed with minimal data movement between the cache and main memory. This reduction in cache misses, which are detrimental to performance, proves to be advantageous.

Enhanced Data Locality: Blocking promotes improved data locality by ensuring that the data required for a specific block multiplication is located close together in memory. Consequently, the data in the cache is reused more effectively, resulting in reduced time spent on fetching data from main memory.

Parallelization: Blocking also facilitates parallelism. Since the work is divided into smaller blocks, it becomes feasible to distribute the workload across multiple processors or threads, leading to significant speedup on multi-core processors.
Loop Structure Optimization: When implemented correctly, blocking can optimize the loop structure, resulting in improved performance.

---

b) Read in the two random matrices that you previously created and write a program that performs blocked matrix multiplication on them. Time the actual operation time (not reading or writing) and record the output to a text file. **(5 pts)**

- **Make sure to comment your code.**
- How many FLOPS do you achieve using this version?
- Is it close to the BLAS implementation's level? Why/why not? **Do some research if you need to.**

Answer:

Program File: **matrix_mul_q4_b.cpp**
Result: **result_matrix_blocked.txt**
Execution Time: 139.79 Sec
Number of Flops: 1.228988e+08 FLOPS

```
[mxr809@hpc6 ~]$ module load gcc/6.3.0

Due to MODULEPATH changes the following have been reloaded:
  1) openmpi/2.0.1

[mxr809@hpc6 ~]$ g++ matrix_mul_q4_b.cpp
[mxr809@hpc6 ~]$ ./a.out
Blocked matrix multiplication completed.
Time taken for blocked matrix multiplication: 139.79 seconds.
FLOPS achieved: 1.22898e+08 FLOPS/second
[mxr809@hpc6 ~]$
```

BLAS is faster than Blocking technique and the reason may be some of the following:
1.As BLAS algorithms are developed in 1960's lot of development has happened over time and BLAS may be having optimizations for specific hardware.
2. The BLAS libraries have been specifically crafted to optimize cache efficiency, a critical factor in matrix operations. These libraries meticulously regulate memory access patterns to reduce cache misses, a feature that is especially vital for larger matrices, where cache misses can significantly impede performance.
3. Parallelism, such as multi-threading or utilizing SIMD (Single Instruction, Multiple Data) instructions in BLAS.
4. The performance advantage of BLAS routines becomes more pronounced as the matrix size increases.
5.The BLAS libraries have been specifically engineered to optimize the utilization of memory bandwidth. If memory bandwidth poses a constraint within your system, BLAS has the potential to deliver superior performance owing to its optimized memory access patterns.

Blocking does not have all these advantages as it is not using any library and is coded and hence because of lack performance optimizations latencies are obvious.

---

c)Run the blocked output and the BLAS result through your comparing program to make sure that the resulting matrices match. **(2 pts)**

Answer:

Program file: **matrix_mul_q4_c.cpp**
Result: Matrices are identical

```
[mxr809@hpc6 ~]$ g++ matrix_mul_q4_c.cpp
[mxr809@hpc6 ~]$ ./a.out
Matrices are identical to two decimal places.
[mxr809@hpc6 ~]$
```