SPP Assignment1 by Alapan Sau Mar 27, 2021

Question 1: Matrix Multiplication

Problem Statement:

We have to take n matrices as input and compute multiplication of the given matrices and return the product matrix given $1 < n < 5$. Each matric have a maximum dimension of $1000 * 1000$.

Optimisation Approaches:

The baseline algorithm of matrix multiplication is almost taking 38 secs on abacus for 5 matrices of each of dimensions $1000 * 1000$. The various optimisation approaches I made and their corresponding run time values are provided for a matrix of size $1000 * 1000$.

1. Changing Loop order:

The order of looping was changed to i, k, j from i, j, k was made to enhance the utility of **cache spatial locality**. This change brought the run time to $30 secs$.

2. Divide and Conquer Algorithm:

Next I implemented a cache oblivious divide and conquer algorithm, to reduce the time furthur. However with. a coarsening constant of 1, the function overhead being significantly high, the runtime almost went up to 50 secs. On some trial and error, the best coarsening constant was found to be 32 which gave a runtime of almost 12 secs. The divide and conquer algorithm used is given as follows:

Inputs: matrices A of size n × m, B of size m × p.
Base case: if $max(n, m, p)$ is below some threshold, use an unrolled version of the iterative algorithm.
Recursive cases:
If $max(n, m, p) = n$, split A horizontally:

$$C = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix}$$

$$C = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix}$$

Else, if $max(n, m, p) = p$, split B vertically:

$$C = A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix} C = A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix}$$

Otherwise, $max(n, m, p) = m$. Split A vertically and B horizontally:

$$C = \begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2$$

$$C = \begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2$$

3. Using a 2D Array instead of a 1D Array:

Inititally the Divide and Conquer algorithm was implemented by using 1D Arrays of size $(rowsize * columnsize)$. However the pointer access referencing over the array was causing a significant overhead. On using a 2D array, accessing rows became much faster, causin the runtime to decrease to ~7 secs.

4. Compiler level Optimisation

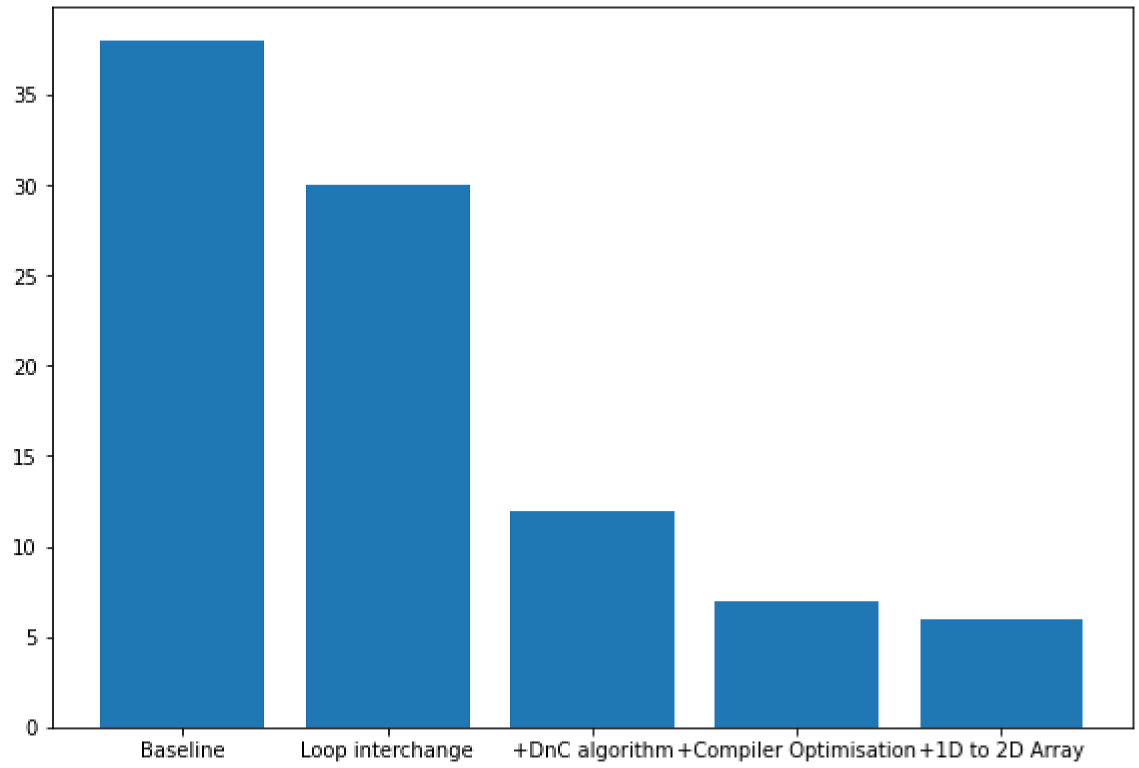
I read about the various compiler level optimisations. I used keywords like **register** to store the variables in register for faster access. Also, **restrict** to reduce some of the compiler checks. Also I used a functional macro for the max function. All these brought down the runtime to 6 secs.

```
import matplotlib.pyplot as plt

import matplotlib.pyplot as plt
fig = plt.figure(figsize=(15,5))
```

```
fig = plt.figure(figsize=(15,5))
ax = fig.add_axes([0,0,0.5,1])
opt = ["Baseline","Loop interchange","+DnC algorithm","+Compiler Optimisation", "+1D to 2D Array"]
tim = [38,30,12,7,6]
ax.bar(opt,tim)

plt.show()
```



Question 2: Floyd Warshall Algorithm

Problem Statement:

We have to calculate the multiple source shortest path using the Floyd Warshall Algorithm. The complexity of the algorithm is $O(n^3)$.

Optimisation Approaches

The baseline code took $60 secs$ for the testcase `t29`. The baseline loop order cannot be changed since k has to be the outermost loop because of dependencies. The various optimisation techniques used are as follows:

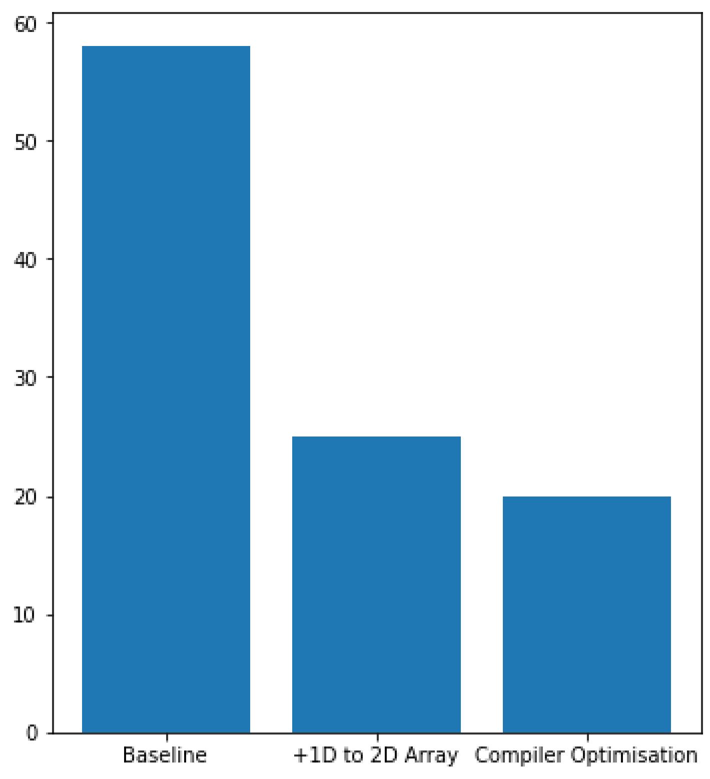
1. Using a 2D Array instead of a 1D Array:

Inititally the algorithm was implemented by using 1D Arrays of size $(rowsize * columnsize)$. However the pointer access referencing over the array was causing a significant overhead. On using a 2D array, accessing rows became much faster, causin the runtime to decrease to ~25 secs.

2. Compiler level Optimisation

I read about the various compiler level optimisations. I used keywords like **register** to store the variables in register for faster access. Also, **restrict** to reduce some of the compiler checks. Also, I used pointer reference instead of array reference. All these decreased the runtime to almost ~20 secs.

```
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(15,5))
ax = fig.add_axes([0,0,0.3,1])
opt = ["Baseline", "+1D to 2D Array", "Compiler Optimisation"]
tim = [58,25,20]
ax.bar(opt,tim)
plt.show()
```



60
50
40
30
20
10
0
Baseline
+1D to 2D Array
Compiler Optimisation