Software Programming for Performance - Assignment 1 Report

Task 0

perf

Perf can be used to visualise which part of the code takes how much percentage of the runtime using flame graphs. It is convenient to see which part of the code has the largest room for optimisation and reduction in run time.

gprof

Gprof is similar to perf, except that it doesn't generate graphs and thus can be used just through the command line (and thus on abacus).

cachegrind

Cachegrind is a tool of valgrind. It simulates the working of the caches on our system and estimates the amount of cache misses it is going to make.

clock_gettime

This function in C finds us the system time. It can be used to calculate the run time of our programs.

Matrix Chain Multiplication

Baseline Code

In the baseline code, I first take the input in a 3D array named `arr` , then find the order in which i should multiply the matrices using an $O(N^3)$ DP. I store the ordering in the array name `orderings`. Now, to multiply them in the given order, I recursively divide the 5 matrices. For the base case, when i reach a single matrix (ie, where `i==j` is true), I return the matrix itself. In other cases, I multiply the 2 arrays recieved from the recursions. Here is the baseline code:

```
#include <stdio.h>
#include <stdlib.h>

int n;
int *x, *y;
long long ***arr;
int **orderings;

void input()
{
    scanf("%d", &n);
    x = (int *)malloc(n * sizeof(int));
    y = (int *)malloc(n * sizeof(int));
    arr = (long long ***)malloc(n * sizeof(long long **));
    orderings = (int **)malloc(n * sizeof(int *));
    for (int a = 0; a < n; a++)
    {
        scanf("%d%d", &x[a], &y[a]);
        arr[a] = (long long **)malloc(x[a] * sizeof(long long *));
        for (int b = 0; b < x[a]; b++)
        {
            arr[a][b] = (long long *)malloc(y[a] * sizeof(long long));
            for (int c = 0; c < y[a]; c++)
                scanf("%lld", &arr[a][b][c]);
        }
        orderings[a] = (int *)malloc(n * sizeof(int));
    }
}

void setOrder() // n^3 == 125 operatiosn at most
{
    long long cost[n][n];
    for (int a = 0; a < n; a++)
        cost[a][a] = 0;
    for (int len = 2; len <= n; len++)
        for (int i = 0; i < n - len + 1; i++)
        {
            int j = i + len - 1;
            cost[i][j] = 1e18;
            orderings[i][j] = 1e9;
            for (int k = i; k <= j - 1; k++)
            {
                long long here = cost[i][k] + cost[k + 1][j] + (long long)(x[i] * y[k] * y[j]);
                if (here < cost[i][j])
                {
                    cost[i][j] = here;
                    orderings[i][j] = k;
                }
            }
        }
}

long long **mat_mul(long long **arr1, long long **arr2, int n, int m, int k)
{
    long long **res = (long long **)malloc(n * sizeof(long long *));
    for (int a = 0; a < n; a++)
    {
        res[a] = (long long *)malloc(k * sizeof(long long));
        for (int b = 0; b < k; b++)
            res[a][b] = 0;
    }
    for (int a = 0; a < n; a++)
        for (int c = 0; c < k; c++)
            for (int b = 0; b < m; b++)
                res[a][c] += arr1[a][b] * arr2[b][c];
    for (int a = 0; a < n; a++)
        free(arr1[a]);
    for (int b = 0; b < m; b++)
        free(arr2[b]);
    free(arr1);
    free(arr2);
    return res;
}

long long **rec_mul(int i, int j)
{
    if (i == j)
        return arr[i];
    printf("%d %d\n", i, j);
    return mat_mul(rec_mul(i, orderings[i][j]), rec_mul(orderings[i][j] + 1, j), x[i], y[orderings[i][j]], y[j]);
}

int main()
{
    input();
    if (n == 1)
    {
        printf("%d %d\n", x[0], y[0]);
        for (int a = 0; a < x[0]; a++)
        {
            for (int b = 0; b < y[0]; b++)
                printf("%lld ", arr[0][a][b]);
            printf("\n");
        }
        return 0;
    }
    setOrder();
    long long **res = rec_mul(0, n - 1);
    printf("%d %d\n", x[0], y[n - 1]);
    for (int a = 0; a < x[0]; a++)
    {
        for (int b = 0; b < y[n - 1]; b++)
            printf("%lld ", res[a][b]);
        printf("\n");
    }
    return 0;
}
```

Memory Optimisations

1. Convert static arrays to dynamic arrays (1.5x speed up)
2. Convert 3D array of size `[5][1000][1000]` to 2D array of size `[5][1000 * 1000]`.

Cache Optimisations

1. When we are multiplying a matrix A with another matrix B using the loops

   ```
   for (int a = 0; a < n; a++)
           for (int c = 0; c < k; c++)
               for (int b = 0; b < m; b++)
                   res[a][c] += arr1[a][b] * arr2[b][c];
   ```

   It is beneficial to have the `arr2` stored in column major format instead of row major format
   To do this, we pass another parameter to our `rec_mul` function which indicates whether we want the resultant array in row major format or column major format

With these optimisations, I was able to reduce my **D1 level cache misses to 5%.**

After doing these optimisations, we have the code:

```
#include <stdio.h>
#include <stdlib.h>

int n;
int x[15], y[15];
int orderings[5][5];
long long arr[15][1000 * 1000];
int used = 5;

void input()
{
    scanf("%d", &n);
    for (int a = 0; a < n; a++)
    {
        scanf("%d%d", &x[a], &y[a]);
        for (int b = 0; b < x[a]; b++)
            for (int c = 0; c < y[a]; c++)
                scanf("%lld", &arr[a][b * y[a] + c]);
    }
}

void setOrder() // n^3 == 125 operatiosn at most
{
    long long cost[n][n];
    for (int a = 0; a < n; a++)
        cost[a][a] = 0;
    for (int len = 2; len <= n; len++)
        for (int i = 0; i < n - len + 1; i++)
```

```
        {
            int j = i + len - 1;
            cost[i][j] = 1e18;
            orderings[i][j] = 1e9;
            for (int k = i; k <= j - 1; k++)
            {
                long long here = cost[i][k] + cost[k + 1][j] + (long long)(x[i] * y[k] * y[j]);
                if (here < cost[i][j])
                {
                    cost[i][j] = here;
                    orderings[i][j] = k;
                }
            }
        }
}

void mat_mul(int idx_arr1, int idx_arr2, int idx_ret, int direction)
{
    int n = x[idx_ret] = x[idx_arr1];
    int k = y[idx_ret] = y[idx_arr2];
    int m = y[idx_arr1];
    for (int a = 0; a < n; a++)
        for (int b = 0; b < k; b++)
            arr[idx_ret][a * k + b] = 0;
    for (int a = 0; a < n; a++)
        for (int c = 0; c < k; c++)
        {
            long long x = 0;
            for (int b = 0; b < m; b++)
                x += arr[idx_arr1][a * m + b] * arr[idx_arr2][b + c * m];
            if (direction == 0)
                arr[idx_ret][a * k + c] += x;
            else
                arr[idx_ret][a + n * c] += x;
        }
}

int rec_mul(int i, int j, int direction)
{
    if (i == j)
    {
        int here = i;
        if (direction == 1)
        {
            here = used++;
            x[here] = x[i];
            y[here] = y[i];
            for (int a = 0; a < x[i]; a++)
                for (int b = 0; b < y[i]; b++)
                    arr[here][b * x[i] + a] = arr[i][a * y[i] + b];
        }
        return here;
    }
    int ret = used++;
    int arr1 = rec_mul(i, orderings[i][j], 0);
    int arr2 = rec_mul(orderings[i][j] + 1, j, 1);
    mat_mul(arr1, arr2, ret, direction);
    return ret;
}

int main()
{
    input();
    if (n == 1)
    {
        printf("%d %d\n", x[0], y[0]);
        for (int a = 0; a < x[0]; a++)
        {
            for (int b = 0; b < y[0]; b++)
                printf("%lld ", arr[0][a * y[0] + b]);
            printf("\n");
        }
        return 0;
    }
    setOrder();
    int idx = rec_mul(0, n - 1, 0);
    printf("%d %d\n", x[idx], y[idx]);
    for (int a = 0; a < x[idx]; a++)
    {
        for (int b = 0; b < y[idx]; b++)
            printf("%lld ", arr[idx][a * y[idx] + b]);
        printf("\n");
    }
    return 0;
}
```

Another memory optimisations

1. After converting to a 1D array, accessing the memory with `arr[idx][i * y[idx] + j]` can take a lot of time. To make this even more optimized, I store the pointer towards `arr[idx][i*y[idx]]` in a separate optimised variable (discussed next) `arr1` and access it as `arr1[j]` to improve runtime.

Compiler optimsations

1. `register` variables: Register variables are stored in fast access registers. However, there can only be a limited number of register variables (as there are only a limited number of registers). So, we need to carefully pick variables that are repeatedly used and hint the compiler to store them in these registers.
2. Repeated calculations: While storing the arrays in `[5][1000 * 1000]` arrays reduced our cache misses, to correctly access indices `[i][j]` we now need to do a calculations of `i * y + j` repeatedly. However, we can notice that for any given `i`, if we calculate and store `i * y` in a register variable, our computational overhead is significantly reduced.
3. Loop direction: The system we have been provided with possibly has a special faster instruction for decrement and compare with 0. I claim this because the program got a slight speed up on reversing loops of type

   ```
   for (int b = 0; b < m; b++)
                   x += arr[idx_arr1][a * m + b] * arr[idx_arr2][b + c * m];
   ```

   to

   ```
   while (--b)
       x += arr[idx_arr1][AM + b] * arr[idx_arr2][b + CM];
   x += arr[idx_arr1][AM + b] * arr[idx_arr2][b + CM];
   ```

With this, we get to our submitted code

Comparisons

Self generated testcases

Since the given testcases only had square matrices, I generated 10 cases with `n` as 5, `x[i]` and `y[i]` as random numbers between `(600, 1000)` and all matrices value as random numbers between `-10` and `10` to test the efficiency of my program

Program	Total runtime on given test cases	Total runtime on generated test cases
Baseline	150.836	243.0081
Static memory	70.6682	225.501
Cache optimisation	40.4194	79.6675
Compiler optimisations	19.4952	47.9309
GCC O2 compilation optimsations	12.4292	29.2782

Other optimisations

Cache optimisations with dynamic memory

Converting the arrays that are the right hand operand in matrix multiplication from row major to column major with dynamic arrays instead of static arrays.

Total time on given test cases: 42.293

Tiling with dynamic memory

Instead of converting the right hand operand to column major, we tile our arrays in such a way that they fit into the cache of the system and reduce our cache misses.

Total time with stride 32 on given test cases: 45.6091

Total time with stride 64 on given test cases: 58.1033

Total time with stride 128 on given test cases: 46.2168

Floyd Warshall

Baseline

```
#include <stdio.h>
#include <stdlib.h>

int min(int i, int j)
{
    if (i < j)
        return i;
    return j;
}

const int inf = 1e9;

int V, E;
int **edges;
int **dist;

void getinput()
{
    scanf("%d%d", &V, &E);
    edges = (int **)malloc(E * sizeof(int *));
    dist = (int **)malloc(V * sizeof(int *));
    for (int a = 0; a < E; a++)
    {
        edges[a] = (int *)malloc(3 * sizeof(int));
        scanf("%d%d%d", &edges[a][0], &edges[a][1], &edges[a][2]);
        edges[a][0]--;
        edges[a][1]--;
    }
    for (int a = 0; a < V; a++)
    {
        dist[a] = (int *)malloc(V * sizeof(int));
        for (int b = 0; b < V; b++)
            dist[a][b] = inf;
        dist[a][a] = 0;
    }
}

void FW()
{
    for (int a = 0; a < E; a++)
        dist[edges[a][0]][edges[a][1]] = min(dist[edges[a][0]][edges[a][1]], edges[a][2]);
    for (int k = 0; k < V; k++)
        for (int i = 0; i < V; i++)
            for (int j = 0; j < V; j++)
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
}

int main()
{
    getinput();
    FW();
    for (int a = 0; a < V; a++)
    {
        for (int b = 0; b < V; b++)
            printf("%d ", dist[a][b] < inf ? dist[a][b] : -1);
        printf("\n");
    }
    return 0;
}
```

Memory optimisations

1. Static memory
2. 2D array to 1D array

Cache optimisations

Unfortunately, I couldn't find a way to optimise the cache misses for this algorithm!

Compiler optimisations

1. After analysing the perf result of my code, the easiest are for optimisation seemed the `min` function. Instead of defining `min` as a function of its own, I improved the runtime by defining it as a macro

   ```
   #define min(a, b) (a > b ? b : a)
   ```

2. `register` variables: Register variables are stored in fast access registers for storing the array pointers and loop variables.
3. Repeated calculations: While storing the arrays in `[2500 * 2500]` arrays reduced our cache misses, to correctly access indices `[i][j]` we now need to do a calculations of `i * y + j` repeatedly.

However, we can notice that for any given `i`, if we calculate and store `i * y` in a register variable, our computational overhead is significantly reduced.

4. Loop direction: The system we have been provided with possibly has a special faster instruction for decrement and compare with 0. I claim this because the program got a slight speed up on reversing loops of type

```
for (int j = 0; j < V; j++)
    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
```

to

```
while (--j)
    dist[vi + j] = min(dist[vi + j], x + dist[vk + j]);
dist[vi + j] = min(dist[vi + j], x + dist[vk + j]);
```

Comparisions

Optimisations	Runtime
Baseline	247.692
Recursive (discussed below)	172.829
Optimised code	48.6697

Other optimisations tried

1. Bit Hack
   I tried calculating `min` with the following bit hack:

```
#define min(x, y) y ^ ((x ^ y) & -(x < y)))
```

   This did not provide any improvements, on the other hand, slowed down the program a little bit

2. Recursive Floyd Warshall
   Adding empty nodes to the graph to have 2^x nodes and performing a recursive Floyd Warshall by continually dividing it into 4 sub matrices. While the approach provided some speed up, addition of edges proved too costly to get results with this method

Recursive Function

```
FW_Recursive(int *A, int *B, int *C, int n)
{
  if(basecase){
    FW_Iterative(A, B, C, n);
    return;
  }
  FW_Recursive(A11, B11, C11, n/2);
  FW_Recursive(A12, B11, C12, n/2);
  FW_Recursive(A21, B21, C11, n/2);
  FW_Recursive(A22, B21, C12, n/2);
  FW_Recursive(A22, B22, C22, n/2);
```

```
    FW_Recursive(A21, B22, C21, n/2);
    FW_Recursive(A12, B12, C22, n/2);
    FW_Recursive(A11, B12, C21, n/2);
  }
```

Division of sub matrices

$A = \left[\begin{smallmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{smallmatrix}\right]$, $B = \left[\begin{smallmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{smallmatrix}\right]$, $C = \left[\begin{smallmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{smallmatrix}\right]$

Iterative Function

```
FW_Iterative(int *A, int *B, int *C, int n)
{
    for (register int c = 0; c < n; c++)
        for (register int a = 0; a < n; a++)
            if (B[a][c] < inf)
                for (register int b = 0; b < n; b++)
                    A[a][b] = min(A[a][b], B[a][c] + C[c][b]);
}
```

Initial call

The function is initially called with all 3, `A`, `B` and `C` being pointers to `dist[0][0]`.