ECE 4822: Engineering Computation IV

Homework No. 2: Fast Algorithms in C++

Shahzad Khan

P01:

Problem

The goal for this part was to redo our HW_01 program and basically optimize it.

Originally my program had run with a time complexity of $O(N^3)$. This is because

Of my original naive approach where we did the matrix calculation with 3 for loops

```
bool multiMatrix(float* mat3, float* mat2, float* mat1, long nrows, long ncols){

 for (long i = 0; i < nrows; i++) {
  for (long j = 0; j < nrows; j++) {
   float sum = 0;
   for (long k = 0; k < ncols; k++) {
    sum += mat1[i * ncols + k] * mat2[k * nrows + j];
   }
   mat3[i * nrows + j] = sum;
  }
 }
 return true;
}
```

From the code above we see that since we are looping through the matrix

3 times, which isn't computationally efficient. Which is the reason for such a high

$O(N)$ value.

Solution

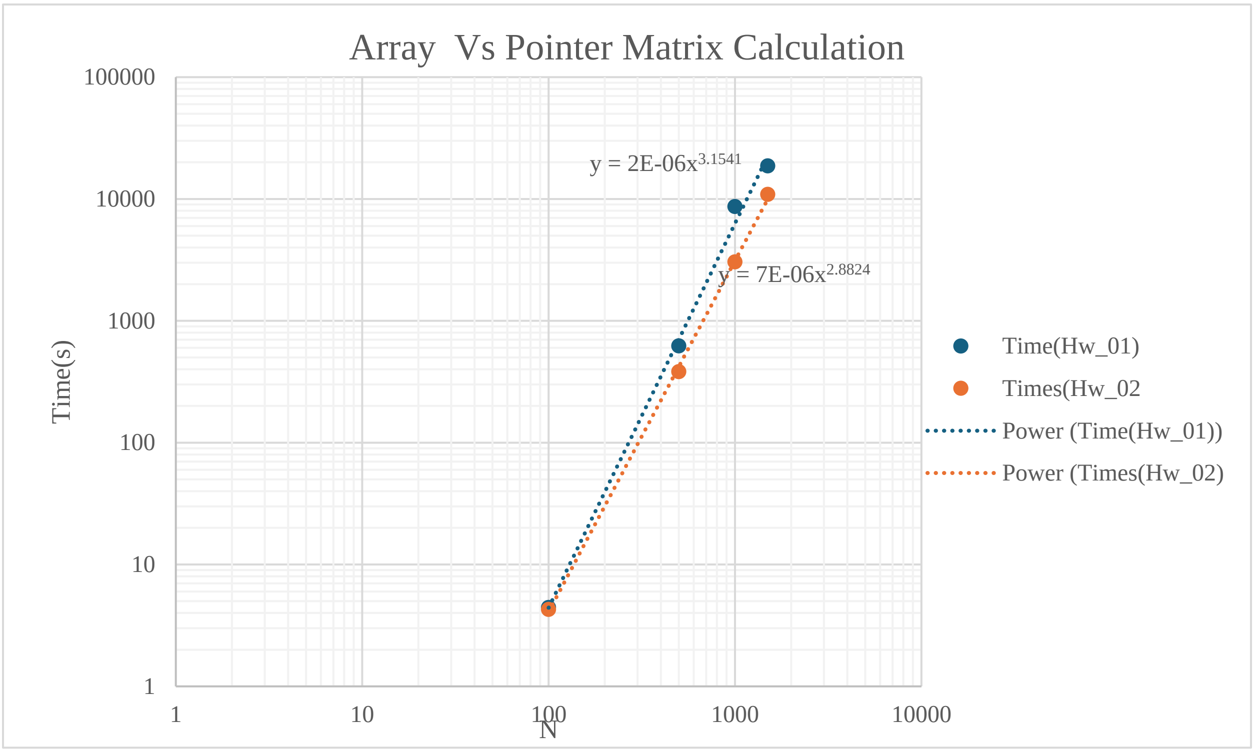
There are many solutions to a problem like this, the solution I had went with was

To optimize performance, I redesigned the program to be more cache-friendly. The key idea was to access data contiguously in memory, reducing cache misses. Since C++ stores rows sequentially, reordering the loops allowed the program to traverse memory in a way that aligns with row-major storage. This improved spatial locality and reduced wasted memory accesses.

```
for(long i = 0; i < nrows; i++){
 /*
  Setting the pointers to the rows in our result matrix
  as well as our first matrix
  */
 float* c_row = mat3 + i * nrows;
 float* a_row = mat1 + i * ncols;
 for(long k =0; k < ncols; k++){
  /*
   A is going to equal "A[i, k]"
   Well we are setting b_rows to be the rows for our other matrix mat2
   */
  float a = a_row[k];
  float* b_row = mat2 + k * nrows;
  float* c_ptr = c_row;
  float* b_ptr = b_row;

  /*
   This is the matrix multiplication, we are making the
   c_row = to a_row * b_row with pointers and we advance c and b
   */
  for(long j = 0; j < nrows; j++){
   *c_ptr += a * (*b_ptr);
   c_ptr++;
   b_ptr++;

  }
 }
}
```



Old and Revised P01 Graph

Above we see the graph and our two times plotted. Orange being our new code with blue being the previous assignment. Plotted on log/log scale we can see that our new code is slightly lower than our previous. With time and complexity of $O(N^{2.8})$ this is a huge step up in how fast our program ran. Our previous code calculated a $1000x1000$ matrix in $\mathbf{0.864s}$ well our new code calculated it in $\mathbf{0.3036s}$

The best time complexity as of January 2024 is $O(n^{2.37})$ computed by a galactic algorithm, which are algorithms that aren't practical because they use matrices that are too large to handle on present day computers.

P02

Problem

Our problem was implementing an autocorrelation function into C.*" a statistical measure that shows the similarity between a time series and a lagged version of itself, indicating the correlation between observations at different time intervals."* So basically, we are shifting the signal and comparing it to itself. This allows you to find similarities within a signal, like if it's periotic you can tell using the function.

```
// --------------------------------------------------
// Compute autocorrelation function
// R[k] = sum_{n=0}^{N-1-k} x[n] * x[n+k],   k=0..K-1
// --------------------------------------------------
bool autocor(float* R, float* x, long N, long K) {
  if (!R || !x) {
    return false;
  }

  for (long k = 0; k < K; k++) {
    float sum = 0.0f;
    for (long n = 0; n < N - k; n++) {
      sum += x[n] * x[n + k];
    }
    R[k] = sum;
  }

  return true;
}
```

This code has a time complexity of $O(NK)$. This is because even though he has 2 for loops in the function, one of them is simply looping through K which doesn't change at all. Making it a constant, and after that we just have a loop that goes through N times. Thus, leaving us with a time complexity of $O(NK)$. This by itself is quite fast, my original idea was that we can't any faster. But using Tukey's FFTs autocorrelation function we can fix that.

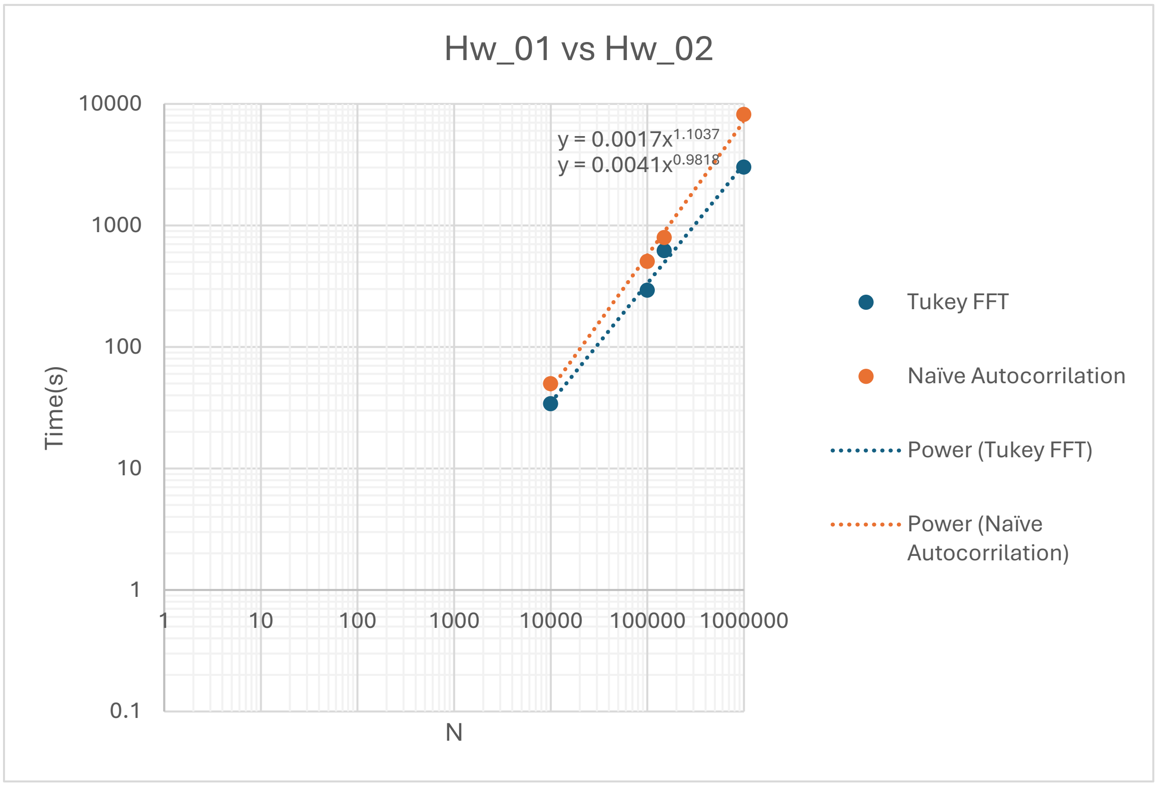
Solution

```
// Recursive FFT
void fft(Complex* a, int n, int invert) {
 if (n == 1) return;
 Complex *a0 = (Complex*) malloc(n/2 * sizeof(Complex));
 Complex *a1 = (Complex*) malloc(n/2 * sizeof(Complex));
 for (int i = 0; 2*i < n; i++) {
  a0[i] = a[2*i];
  a1[i] = a[2*i+1];
 }
 fft(a0, n/2, invert);
 fft(a1, n/2, invert);



 double ang = 2*M_PI/n * (invert ? -1 : 1);
 Complex w = {1,0}, wn = {cos(ang), sin(ang)};
 for (int i = 0; 2*i < n; i++) {
  Complex t = c_mul(w, a1[i]);
  a[i] = c_add(a0[i], t);
  a[i+n/2] = c_sub(a0[i], t);
  if (invert) {
   a[i].re /= 2; a[i].im /= 2;
   a[i+n/2].re /= 2; a[i+n/2].im /= 2;
  }
  w = c_mul(w, wn);
 }
 free(a0); free(a1);
}
```

The code above is an implementation of Tukey FFT . Usually when we want to do a DFT the implementation would be in $O(N^2)$ which would be extremely slow for our case. But what Tukey does is it splits the DFT into smaller transforms of evens and odds. Allowing us to half N and repeats over and over again until the size is 1.



Now we see the graph our HW_01 graph is around $0.0017x^{1.1037}$ which is around $O(NK)$. Although it is fast, we can go faster. With our hw_02 FFT we get $0.0041x^{0.9818}$ the FFT reduces the computational complexity from roughly $O(NK)$ to around $O\big(Nlog(N)\big)$

Conclusion

Through this assignment, I explored how algorithm design directly affects computational efficiency. In P01, restructuring the matrix multiplication loops to be cache-friendly significantly reduced runtime, lowering the time complexity. This optimization demonstrates how careful memory access patterns can have a large impact on performance.

In P02, I implemented an autocorrelation function with a time complexity of $O(NK)$While this approach is efficient for moderate input sizes, further optimization using the Fast Fourier Transform (FFT) reduced the complexity to approximately $O\left(Nlog(N)\right)$The FFT achieves this by recursively splitting the DFT into smaller transforms of even and odd indices, reusing computations, and combining results efficiently through the butterfly operation. Although the FFT provides huge benefits for large datasets, practical overhead can make it slower for smaller input sizes.

Overall, this assignment highlighted the importance of considering both asymptotic complexity and real-world performance when designing fast algorithms. By comparing naive, cache-optimized, and FFT-based implementations, it becomes clear that algorithmic efficiency and memory access patterns together determine practical runtime.