Rapport optimalisering av FFT

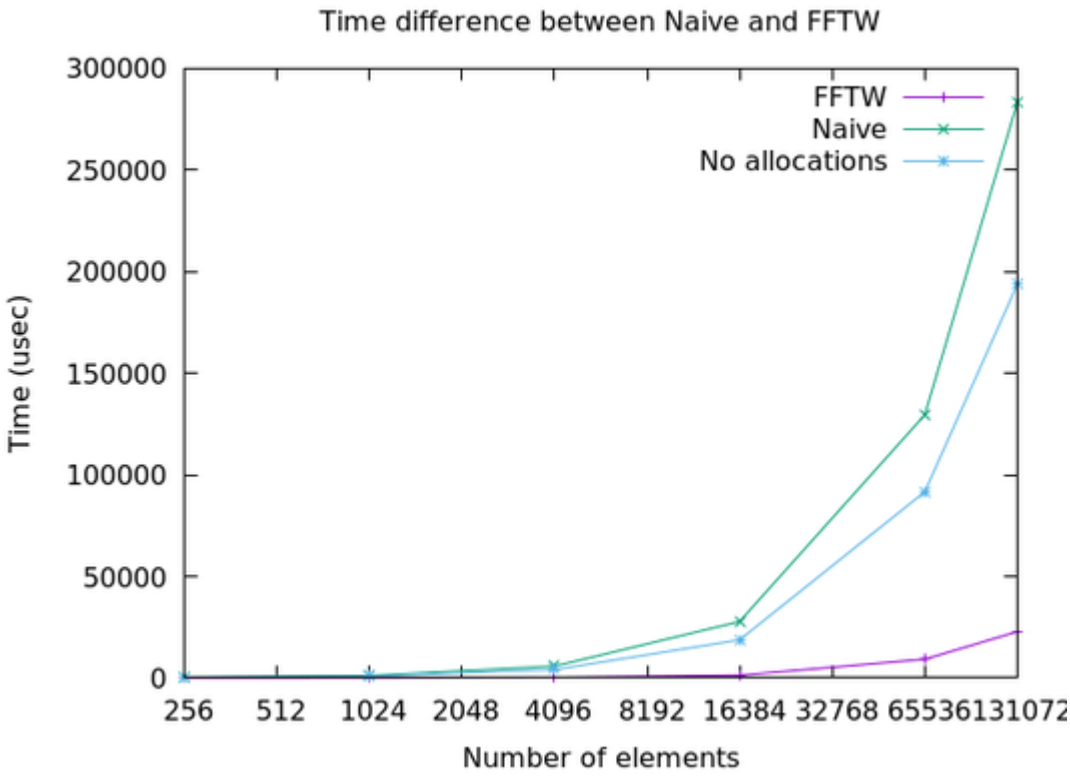
Vi valgte å forbedre den naive koden ved å fjerne minne- allokeringene, samt regne ut twiddle- faktoren på forhånd. Vi så også på muligheten for å parallellisere programmet, men har ikke implementert dette.

Uten allokering

I den naive koden blir det brukt hjelpemetoder for å overføre partall- indekser og oddetall- indekser fra "in" - arrayet til "out" - arrayet. Dette gjøres via fire nye arrayer som det allokeres minne til. I den forbedrede koden, innføres det en ny parameter, *stride*, som brukes til å flytte minne- pekeren ved å legge til *stride*. Parameteren dobles ved hvert rekursive kall, som vil si at minne- pekeren vil flyttes med det dobbelte ved hvert kall på funksjonen. Det vil være lite endring i algoritmen fra den naive koden, men ved å bruke dette tallet som parameter til funksjonen, unngår vi allokering og de- allokering av minne. Dette vil man helst unngå, ettersom det er tidkrevende å finne plass (allokere) minne, fordi allokerings- algoritmen bruker tid på å finne en ledig blokk for allokering.

Kjøretiden svarte til forventningen om at utrydding av malloc og free gjør programmet at programmet kjører raskere. Se *figur 1* nedenfor, hvor No allocations er kjøretiden på programmet ved bruk av stride i stedet for minne- allokering.

Fjerning av allokering av minne og implementasjon av *stride* er implementert ved å følge pseudokoden fra wikipedia- artikkelen på Cooley- Tukey fft- algoritmen.



Figur 1

Utregning av twiddle på forhånd

Etter vi hadde fått lagt inn stride-parameteren i algoritmen på en måte slik at vi unngikk minne-allokasjonene i hver rekursjon, så gikk vi inn for å regne ut twiddlefaktoren i w på forhånd, slik at vi kunne unngå å måtte regne ut den komplekse eksponenten hver runde i for-loopen i hver rekursjon

```
for(int i = 0; i < half; ++i) {
    const complex e = even_out[i];
    const complex o = odd_out[i];
    const complex w = cexp(0 - (2. * M_PI * i) / n * I);
    out[i]        = e + w * o;
    out[i + half] = e - w * o;
}
```

Figur 2

Her ser vi altså hvordan det først var implementert når vi regner ut twiddle-faktoren hver eneste loop.
Vi ser at størrelsen til twiddle-faktoren er påvirket av størrelsen til i og n, de andre tallene er statiske. Å legge inn i pluss resten av tallene inn i en egen metode er enkelt nok, da kjører vi bare en for-loop n/2 antall ganger, og får med det riktige i-verdier. n

variabelen er et større problem, som derimot lar seg løse ved hjelp av stride-parameteren vår. Ideen var først å lage et dobbelt array, som da ble laget ved en dobbelt for-loop som kjørte log2(n) antall ganger ytterst, men vi innså at det ble fort veldig komplisert og lite effektivt. Derfor tok vi et steg tilbake og så på hva som egentlig skjer med twiddle-faktorene hver rekursjon. Vi kom frem til et par ting:

1. antall ganger de kjører gjennom for-loopen halverte seg for hver rekursjon, som betyr at vi vil aldri trenge en høyere i-verdi enn i det første kallet på metoden, altså n/2.
2. For hver rekursjon blir n, som vi deler resten av twiddle-faktoren på, dobbelt så liten, altså dette tilsvarer at twiddle-faktorene dobles for hver rekursjon.

Det er her stride parameteren kommer inn, for den dobles også for hver rekursjon, så med det kom vi frem til at vi istedenfor å gi twiddle-faktorene en ny n for hver rekursjon som resten må deles på, så beholder vi hele tiden den samme n-en, og ganger heller twiddle-faktorene med s for hver rekursjon. Da kan vi nemlig bare regne ut twiddle-faktorene på forhånd med en enkel metode som kjører gjennom og returnerer et array med en twiddle for hver i-verdi, slik som dette:

```
complex* precalc_twiddles(int n){
    int half = n/2;
    complex mult = (2. * M_PI * I)/n;
    complex* twiddles = malloc(sizeof(complex)*half);

    for(int i = 0; i < half; ++i){
        twiddles[i] = cexp(0 - mult * i);
    }
    return twiddles;
}
```

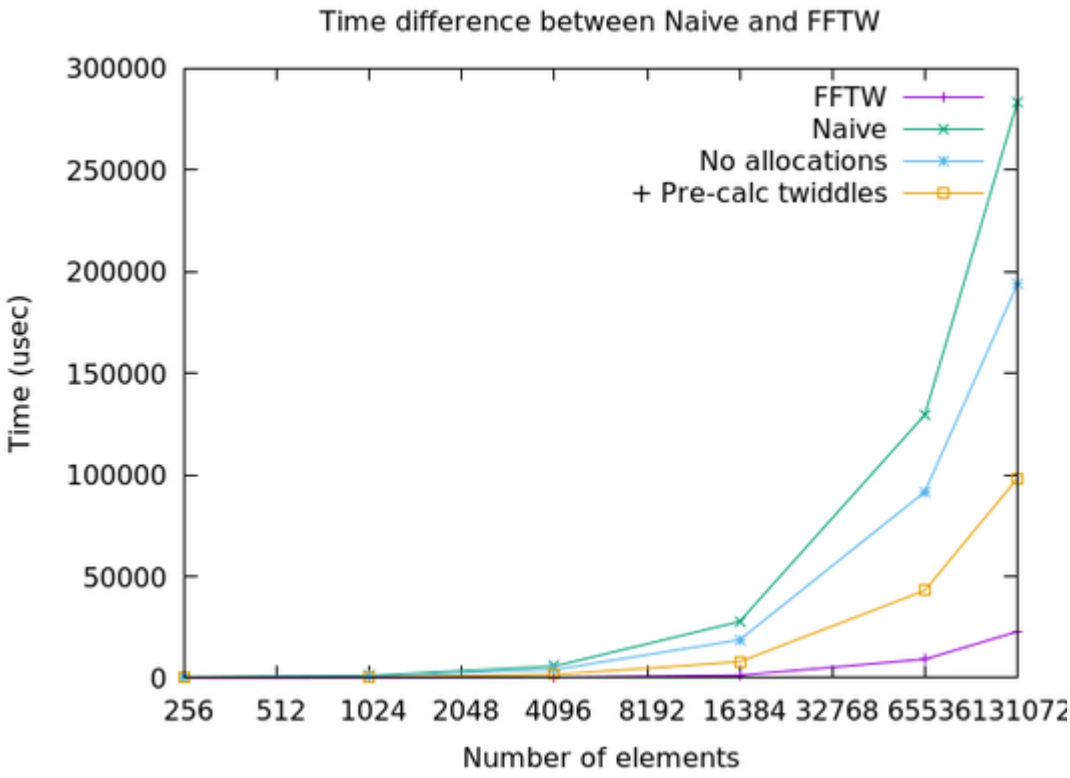
Figur 3

Så må vi bare gange disse verdiene med den gjeldende s-verdien hver gang vi henter frem en. Vi henter jo alltid frem riktig verdi, ettersom at vi regnet ut twiddle-verdiene i en for-loop like stor, og i samme rekkefølge som den hvor vi bruker de. Det ser slik ut:

```
for(int i = 0; i < half; ++i){
    const complex e = out[i];
    const complex wo = twiddles[i * s]*out[i+half];
    out[i]        = e + wo;
    out[i + half] = e - wo;
}
```

Figur 4

Denne ganske enkle, men litt abstrakte, forbedringen ga oss en stor kjøretids-forbedring, hvor vi gikk fra å bruke i underkant av 200 000 mikrosekund i figur 1 til å bruke rett i underkant av 100 000 mikrosekund, se figur 2. Det ga oss altså nesten en 100% forbedring når vi gjorde dette oppå forbedringen ved å fjerne alle allokasjoner ved hjelp av stride-parameteren.



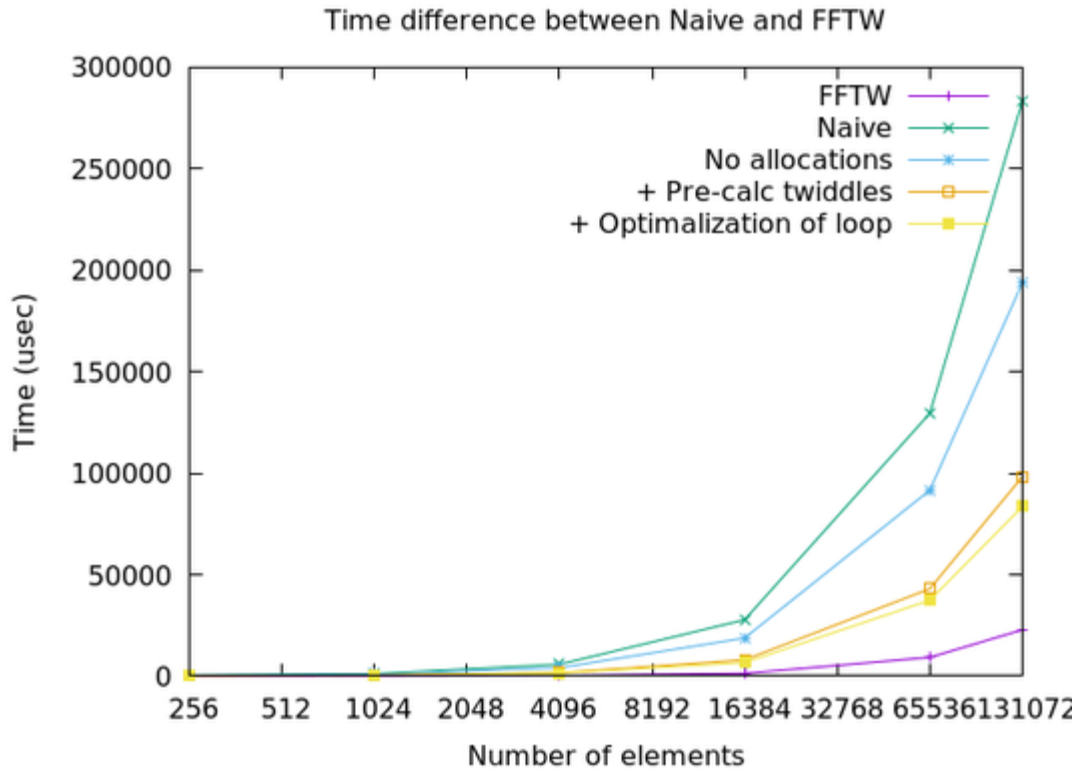
Figur 5

Baksiden ved denne forbedringen er jo det at vi krever litt tid til å regne ut alle disse twiddle-verdiene på forhånd, altså det tar lenger tid før den faktiske fft-algoritmen får startet, og vi har lenger ikke en algoritme uten noen allokasjoner, så det betyr at så lenge algoritmen kjører, så er en del av minnet avholdt til å ta vare på twiddle-verdiene vi har lagret i arrayet før start, helt frem til slutten og vi er ferdige med algoritmen. Det betyr jo også at vi ikke bruker verdiene i arrayet før rekursjonen er på vei opp igjen, altså det er en periode av algoritmen hvor arrayet funker som dødvekt, og brukes ikke til noe. Det er dessverre sånn det må være da vi skal regne ut twiddle-verdiene på forhånd. Et alternativ ville ha vært og regnet ut twiddle-faktorene når vi nådde bunn-tilfellet i rekursjonen, det eneste da er at det blir relativt komplisert, ettersom at for et n langt input-array, så vil vi ha n antall bunntilfeller. Så det er jo noe å tenke på, men alt i alt, så er det en veldig effektiv forbedring som gjør mye mer bra for effektiviteten av kjøringa enn den ødelegger.

Optimalisering av loop

Etter å ha suksessfullt implementert både stride-parameteren og forkalkulasjon av twiddle-faktorene, så gikk vi over og fant enkelte ting vi kunne forenkle eller fjerne for å optimalisere algoritmen videre. Dette vil mest ikke ha spesielt store utslag dersom det er utenfor en løkke, men ting som er i en løkke kjøres jo antall rekursjoner * antall iterasjoner, som da blir n ganger flere operasjoner. Vi fikk funnet og fikset opp i

hovedsaklig én ting som var litt ueffektivt i koden vår. Vi hadde nemlig en egen variabel som aksesserte siste halvdel av output-arrayet. Denne halvdelen skal brukes til å gange med twiddle-faktorene, og dette produktet legges til eller trekkes fra første halvdel av output-arrayet. Vi fikset dette ved å la være å lage denne variabelen, men ganget heller verdien rett inn med twiddle-faktoren, slik at vi kun trengte å legge til eller trekke fra et tall, istedenfor å finne produktet 2 ganger, først når det skal adderes, så når det skal subtraheres. Vi fikk en synlig forbedring i kjøretid i forhold til hva vi hadde i figur 2, se figur 3.



Figur 6

Før optimaliseringen så for-løkken i den rekursive metoden ut som følgende

```
for(int i = 0; i < half; ++i){
    const complex e = out[i];
    const complex o = out[i+half];
    const complex w = twiddles[i * s];
    out[i]        = e + w*o;
    out[i + half] = e - w*o;
}
```

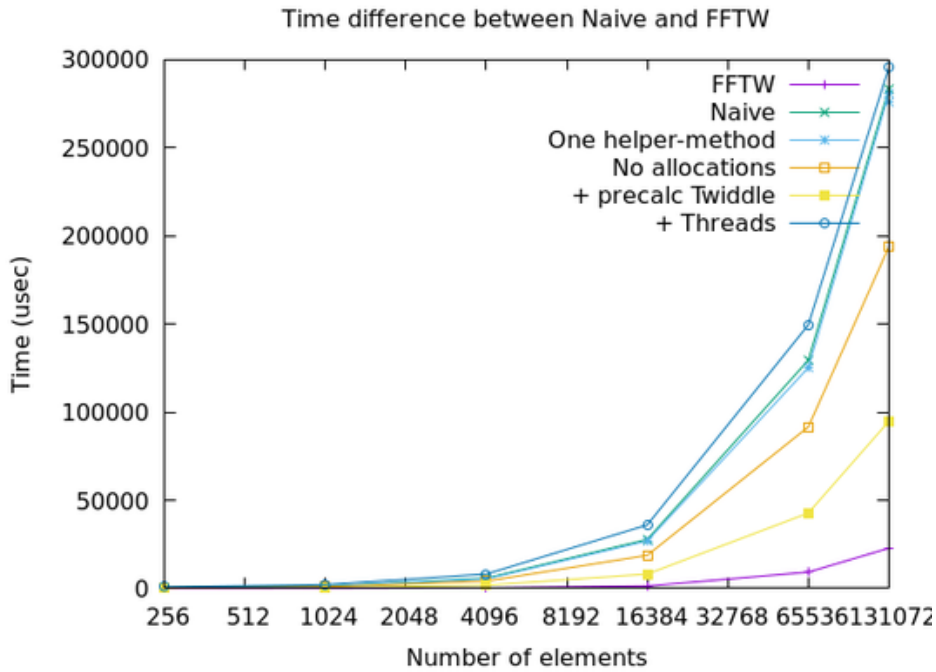
Her ser man klart og tydelig at man danner variabelen o, og ganger denne med w 2 ganger per iterasjon, istedenfor å gange den rett inn med verdien til w, og da bare regne ut produktet 1 gang per iterasjon slik som vi gjør her

```
for(int i = 0; i < half; ++i){
    const complex e = out[i];
    const complex wo = twiddles[i * s]*out[i+half];
    out[i]        = e + wo;
    out[i + half] = e - wo;
}
```

Andre mulige forbedringer

Et annet forslag til forbedring av den naive koden er parallellisering. Dette kunne i teorien effektivisert programmet. Likevel er parallellisering ikke uten kostnad, og i vårt tilfelle ville det mest sannsynlig ikke vært verdt det. Mye fordi eventuell implementasjon av tråder hadde medført mye venting og organisering. I tillegg er det ikke alle deler av programmet som kan parallelliseres.

Vi forsøkte å implementere tråder, men kom ikke noen særlig vei med dette, da kjøretiden ble betydelig mye tregere, mest sannsynlig som følge av en dårlig implementasjon. Etter å ha fjernet denne implementasjonen, så glemte vi å fjerne importen av biblioteket til openMP, og da oppdaget vi at kjøretiden var tilnærmet like sen som den var ved implementasjonen vår på plass (se *Figur 7)*. Dette antyder at vår forsøkte implementasjon feilet, men viser også noe av kostnaden vår bruk av tråder.



Figur 7, oversikt over alle forskjellige kjøretider, "+" viser at det er i tillegg til implementasjonene over.