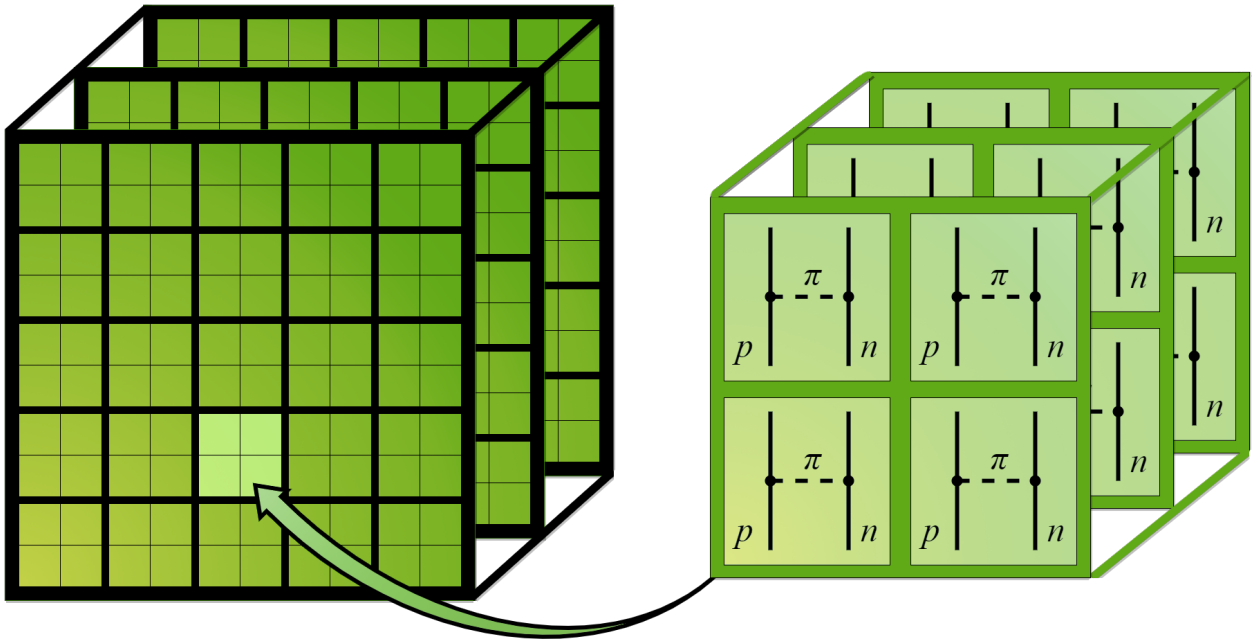
CHALMERS
UNIVERSITY OF TECHNOLOGY



Snabba beräkningar av elastisk proton-neutronspridning med en grafikprocessor

Kandidatarbete inom teknisk fysik

Erik Brusewitz, Alexander Körner, Joseph Löfving,
Hanna Olvhammar, Alfred Weddig Karlsson

Institutionen för fysik
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2021

Kandidatarbete 2021: TIFX04–21–05

Snabba beräkningar av elastisk proton-neutronspridning med en grafikprocessor

GPU Accelerated Calculations of Elastic Proton-Neutron Scattering

Erik Brusewitz
Alexander Körner
Joseph Löfving
Hanna Olvhammar
Alfred Weddig Karlsson

CHALMERS

Institutionen för fysik
Avdelningen för subatomär, högenergi- och plasmafysik
Chalmers tekniska högskola
Göteborg, Sverige 2021

Snabba beräkningar av elastisk proton-neutronspridning med en grafikprocessor

Erik Brusewitz
Alexander Körner
Joseph Löfving
Hanna Olvhammar
Alfred Weddig Karlsson

Handledare: Andreas Ekström, Sean Miller, Institutionen för fysik
Examinator: Lena Falk, Institutionen för fysik

Kandidatarbete 2021: TIFX04–21–05
Institutionen för fysik
Avdelningen för subatomär, högenergi- och plasmafysik
Chalmers tekniska högskola
412 96 Göteborg, Sverige
Telefon +46 (0)31-772 10 00

Framsida: Figuren illustrerar hur en växelverkansprocess beräknas i olika trådar på en grafikprocessor, där många trådar exekveras parallellt. Trådarna fördelas i block (till höger), och block fördelas i grid (till vänster).

Göteborg, Sverige 2021

Sammandrag

För att beskriva den starka kärnkraften med effektiv fältteori krävs noggrann kalibrering av kopplingskonstanterna i motsvarande potentialmodeller. Det ger upphov till ett flerdimensionellt inferensproblem som i sin tur kräver snabba numeriska beräkningar. Vi har därför undersökt möjligheten att genomföra effektiva beräkningar av spridningsfasskift genom att lösa Lippmann-Schwingerekvationen för elastisk proton-neutronspridning på en grafikprocessor (GPU). För att utnyttja parallelliseringsförmågan hos en GPU används gränssnittet CUDA i C++. Den numeriska lösningsmetoden, som baseras på upprepad lösning av en matrisekvation, har redan implementerats på en centralprocessor (CPU). Därför jämförs den totala exekveringstiden för CPU- och GPU-programmen, såväl som exekveringstiden per beräknat fasskift. Vi fann att GPU-programmet är snabbare än CPU-programmet vid beräkning av många fasskift samtidigt och att det därför finns goda möjligheter för mer effektiv kalibrering av kopplingskonstanterna med en GPU. Koden för att beräkna potentialmodellen är skriven för en CPU och har inte utvecklats i det här projektet. För att öka effektiviteten i våra GPU-beräkningar krävs dock effektivare hantering av potentialen. Vi drar slutsatsen att fortsatt optimering av vår GPU-kod samt anpassning för specifik hårdvara, som grafikkortet Nvidia Tesla V100, kan möjliggöra ännu snabbare beräkningar av elastisk proton-neutronspridning och därmed bidra till framsteg för att beskriva den starka kärnkraften.

Abstract

Describing the strong nuclear force with effective field theory demands careful calibration of the coupling constants in corresponding potential models. This generates a multidimensional inference problem which requires fast numerical calculations. Thus, we have examined the possibility to perform efficient calculations of scattering phase shifts by solving the Lippmann-Schwinger equation for elastic proton-neutron scattering on a graphics processing unit (GPU). To utilize parallelization on a GPU, we use the interface CUDA in C++. The numerical method, which is based on repeatedly solving a matrix equation, has already been implemented on a central processing unit (CPU). Therefore, the total time to execute the GPU and CPU programs as well as the time per calculated phase shift were compared. We found that the GPU program is faster than the CPU program when many phase shifts are calculated simultaneously, hence we believe it is possible to obtain more efficient calibration of the coupling constants using a GPU. The code for calculating the potential model is written for a CPU and has not been developed in this project. However, to increase the efficiency of our GPU calculations the potential needs to be managed more efficiently. We conclude that continued optimization of our GPU code as well as hardware specific adaptations, e.g. for an Nvidia Tesla V100 GPU, could enable even faster calculations of elastic proton-neutron scattering and thereby contribute to advancements in describing the strong nuclear force.

Nyckelord: Lippmann-Schwinger, nukleon-nukleonspridning, växelverkan, starka kärnkraften, CUDA, GPU, parallellisering

Keywords: Lippmann-Schwinger, nucleon-nucleon scattering, interaction, strong nuclear force, CUDA, GPU, parallelization

Innehåll

1
Inledning

Under 1900-talet gjordes revolutionerande framsteg för förståelsen av universums minsta beståndsdelar; kvantmekaniken formulerades varpå ett flertal nya forskningsfält etablerades. Inom partikelfysiken utvecklades standardmodellen, som beskriver stark, svag och elektromagnetisk växelverkan mellan elementarpartiklar. Kärnfysiken har dock inte kunnat beskrivas med en lika enhetlig teori, och det saknas en rigorös koppling till standardmodellen. Den starka kärnkraften, det vill säga stark växelverkan mellan nukleoner, är komplicerad att beskriva teoretiskt och har därför visat sig vara svår att modellera. De senaste årtiondena har det främst forskats kring hur den starka kärnkraften kan beskrivas med hjälp av effektiv fältteori, och det är aktuellt även idag [1].

Växelverkansmodeller som baseras på effektiv fältteori innehåller vanligtvis 20–30 kopplingskonstanter, vars numeriska värden behöver kalibreras genom att reproducera experimentell data från bland annat nukleon-nukleonspridning. Eftersom kvantmekanisk spridningsteori är väletablerad kan spridningstvärsnitt noggrant beräknas för givna värden på kopplingskonstanterna. På så sätt kan kopplingskonstanterna kalibreras med experimentell data.

Spridningstvärsnittet kan erhållas ur lösningar till Lippmann-Schwingerekvationen [2], varför kalibrering av kopplingskonstanterna leder till ett inferensproblem. Det kräver att Lippmann-Schwingerekvationen löses numeriskt en gång för varje uppsättning av numeriska värden på de kopplingskonstanter som undersöks. Därför behöver Lippmann-Schwingerekvationen lösas väldigt många gånger för att kopplingskonstanterna skall kalibreras, vilket kan ta exponentiellt lång tid att genomföra på en vanlig centralprocessor (CPU).

Det finns olika matematiska metoder för att lösa Lippmann-Schwingerekvationen numeriskt; i det här projektet ställs ekvationen upp på matrisform för att sedan lösas som ett ekvationssystem [3]. Lösningsmetoden har tidigare implementerats för beräkningar på en CPU, men det är möjligt att ett större antal beräkningar kan utföras snabbare på en grafikprocessor (GPU). Även om en CPU kan utföra beräkningarna parallellt finns det utrymme att parallellisera beräkningarna i mycket högre utsträckning på en GPU och därmed lösa Lippmann-Schwingerekvationen många gånger samtidigt på kort tid. Det är inte säkert att grafikprocessorns parallelliseringsförmåga möjliggör en snabbare lösning i det aktuella fallet, således syftar detta projekt till att avgöra huruvida en GPU verkligen är snabbare för lösning av Lippmann-Schwingerekvationen. Skulle GPU-beräkningarna visa sig vara snabbare öppnar det upp för många möjligheter att effektivt kalibrera modeller av den starka kärnkraften.

För att undersöka utrymmet att effektivisera lösningen av Lippmann-Schwingerekvationen skrivs först ett program för CPU-beräkningar i C++, varefter ett analogt program skrivs för GPU-beräkningar med Nvidias gränssnitt CUDA [4]. GPU-programmets körningstid jämförs sedan med ett motsvarande väloptimerat CPU-program. Förhoppningen är att projektet resulterar i ett värdefullt verktyg i form av ett effektivt GPU-program för att kalibrera och bättre analysera modeller av den starka kärnkraften.

I projektet görs ett antal avgränsningar. Vi undersöker endast elastisk spridning, vil-

ket innebär att den totala rörelseenergin är bevarad och att inga nya partiklar bildas i spridningen. Dessutom undersöks endast tvånukleonspridning, då komplexiteten för att beskriva spridningsprocessen ökar drastiskt med antalet växelverkande nukleoner. Speciellt undersöks endast proton-neutronspridning på grund av den bidragande komplexitet som de elektromagnetiska krafterna vid proton-protonspridning tillför. Anledningen till att neutron-neutronspridning inte undersöks är bristen på experimentell data, vilket gör att det finns få resultat att kalibrera potentialmodeller med.

Det görs också avgränsningar i den programmeringstekniska aspekten av arbetet. Tillhandahållen, färdigskriven kod används för att evaluera potentialen för den starka kraften. Dessutom görs analysen av programmets körtid rent numeriskt; teoretiska komplexitetsberäkningar genomförs inte.

Den fullständiga koden bakom programmen som utvecklas finns tillgängliga i ett GitHub-projekt på följande länk: https://github.com/JosephLofving/kandidat/.

Samhälleliga och etiska aspekter

Det här grundforskningsprojektet omfattar utveckling och undersökning av en ny implementering av en väletablerad numerisk metod för att lösa Lippmann-Schwingerekvationen. Nyttan av projektet består främst av tids- och energibesparingar och mer effektiv tillämpning av datorresurser. Att utnyttja hårdvara så effektivt som möjligt bidrar till billigare beräkningar i form av fler FLOPS per krona eller per energienhet. Det tillåter även mer sofistikerade beräkningar på enklare och mer lättillgängliga datorsystem, vilket i bästa fall kan avlasta superdatorer och beräkningskluster. Detta skulle frigöra mer tid för andra ändamål på dessa, vilket är eftertraktat av forskare inom många områden. Dessutom minskar slitage, och därav behovet av att ersätta komponenter som innehåller dyra och sällsynta metaller.

2

Teoretisk lösningsmetod för Lippmann-Schwingerekvationen

Syftet med det här kapitlet är att presentera den numeriska lösningsmetod för Lippmann-Schwingerekvationen som används i det här arbetet. Först introduceras rörelsemängden för två nukleoner i olika inertialsystem i avsnitt 2.1. Därefter härleds Lippmann-Schwingerekvationen i avsnitt 2.2 samtidigt som relevant notation introduceras. I avsnitt 2.3 redogörs för bevarade kvanttal och partialvågsformalism, följt av en introduktion till den underliggande potentialmodellen för den starka kraften mellan två nukleoner i avsnitt 2.4. En matematisk omskrivning av Lippmann-Schwingerekvationen till en matrisekvation genomförs sedan i avsnitt 2.5. Det avslutande avsnittet 2.6 redogör för hur spridningsfasskift kan beräknas utifrån lösningar till Lippmann-Schwingerekvationen.

2.1 Rörelsemängd och inertialsystem

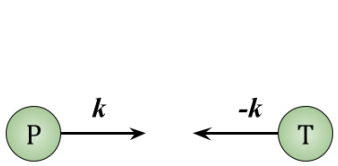
Nukleon-nukleonspridning studeras lämpligast i två ekvivalenta inertialsystem; masscentrums inertialsystem (”CM-systemet” hädanefter) samt laboratoriets inertialsystem (”labbsystemet” hädanefter). Vid tvånukleonspridning brukar den ena partikeln benämnas P för ”projectile” och den andra T för ”target”. I CM-systemet har nukleonerna P och T rörelsemängderna $\boldsymbol{k}$ respektive $-\boldsymbol{k}$ innan växelverkan; efter växelverkan har de rörelsemängderna $\boldsymbol{k'}$ respektive $-\boldsymbol{k'}$, se figur 2.1 (a) och (b). Vid elastisk spridning, som studeras här, är rörelsemängden lika stor innan som efter spridningen, det vill säga $k = k' \equiv k_0$. Rörelsemängden k_0 är av central betydelse i lösningen av Lippmann-Schwingerekvationen.

I labbsystemet färdas nukleon P mot nukleon T, som är i vila. Se figur 2.1 (c) och (d). Labbsystemet är användbart för att beskriva experimentella mätningar, exempelvis den kinetiska energin T_{labb} för nukleon P. Med Lorentzinvarians kan därmed rörelsemängden k_0 i CM-systemet beräknas, vilket ger

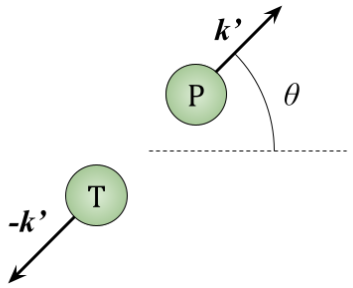
$$k_0^2 = T_{\text{labb}} m_{\text{T}}^2 \frac{T_{\text{labb}} + 2m_{\text{P}}}{(m_{\text{P}} + m_{\text{T}})^2 + 2m_{\text{T}} T_{\text{labb}}}. \tag{2.1}$$

I fallet då $m_{\text{P}} = m_{\text{T}} \equiv m$ gäller $k_0^2 = T_{\text{labb}} m/2$. Den maximala kinetiska energin för elastisk proton-neutronspridning är runt 290 MeV, vilket är tröskelenergin för att en pion skapas i spridningen.

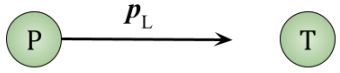
Endast proton-neutronspridning studeras i det här projektet, och i labbets inertialsystem görs valet att protonen har rörelseenergi T_{labb} och neutronen är i vila.



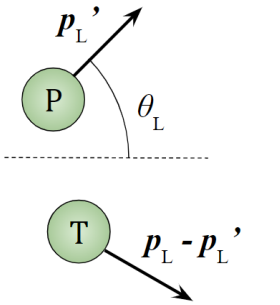
(a) CM-systemet innan växelverkan.



(b) CM-systemet efter växelverkan.



(c) Labbsystemet innan växelverkan.



(d) Labbsystemet efter växelverkan.

Figur 2.1: Nukleon-nukleonspridning i CM-systemet respektive labbsystemet innan och efter växelverkan. Här står P för ”projectile” och T för ”target”. Rörelsemängder betecknas i CM-systemet med $\boldsymbol{k}$ och i labbsystemet med $\boldsymbol{p}_{\mathrm{L}}$. Spridningsvinkeln är θ respektive θ_L.

2.2 Härledning av Lippmann-Schwingerekvationen

Betrakta elastisk tvånukleonspridning med icke-relativistisk rörelsemängd $\boldsymbol{k}$ i CM-system enligt figur 2.1 (a) och (b). CM-systemet utnyttjar potentialens translationsinvarians för en enklare matematisk formulering av problemet och används därför vid samtliga beräkningar. Labbsystemet som syns i figur 2.1 (c) och (d) används istället vid experimentella mätningar.

Låt $|\phi\rangle$ beteckna den ostörda vågfunktionen för de två nukleonerna, vilket är en planvåg, och låt $|\psi\rangle$ beteckna vågfunktionen då de växelverkar. Den tidsoberoende Schrödingerekvationen för det störda systemet är då

$$\left(\hat{H}_0 + \hat{V}\right)|\psi\rangle = E\,|\psi\rangle\,, \tag{2.2}$$

där $\hat{H}_0$ är Hamiltonoperatorn för fria partiklar och $\hat{V}$ är potentialen för växelverkan mellan de två nukleonerna, se avsnitt 2.4. Energin ges av $E = k^2/2\mu$, där μ är systemets reducerade massa. En omskrivning av ekvation (2.2) ger

$$\left(E - \hat{H}_0\right)|\psi\rangle = \hat{V}\,|\psi\rangle\,, \tag{2.3}$$

och med antagandet att operatorn $E - \hat{H}_0$ är inverterbar fås

$$|\psi\rangle = \left(E - \hat{H}_0\right)^{-1}\hat{V}\,|\psi\rangle\,. \tag{2.4}$$

Lösningen för $|\psi\rangle$ behöver uppfylla att $|\psi\rangle \to |\phi\rangle$ då $\hat{V} \to 0$, och eftersom potentialen har kort räckvidd inträffar det då avståndet mellan nukleonerna är stort. Därmed ges en möjlig lösning av

$$|\psi\rangle = |\phi\rangle + \left(E - \hat{H}_0\right)^{-1} \hat{V} |\psi\rangle \,. \tag{2.5}$$

Låt $|\phi'\rangle$ vara en annan planvåg. Om operatorn $\langle\phi'| \hat{V}$ appliceras på båda led i ekvation (2.5) fås

$$\left\langle \phi' \middle| \hat{V} \middle| \psi \right\rangle = \left\langle \phi' \middle| \hat{V} \middle| \phi \right\rangle + \left\langle \phi' \middle| \hat{V} \left(E - \hat{H}_0\right)^{-1} \hat{V} \middle| \psi \right\rangle . \tag{2.6}$$

Dessa termer kan identifieras som matriselement.

Definiera en operator $\hat{T}$ för övergången mellan $|\phi\rangle$ och $|\psi\rangle$ enligt $\hat{T}|\phi\rangle \equiv \hat{V}|\psi\rangle$. Inför även $\hat{G} \equiv (E - \hat{H}_0)^{-1}$. Ekvation (2.6) kan då skrivas om till

$$\left\langle \phi' \middle| \hat{T} \middle| \phi \right\rangle = \left\langle \phi' \middle| \hat{V} \middle| \phi \right\rangle + \left\langle \phi' \middle| \hat{V}\hat{G}\hat{T} \middle| \phi \right\rangle , \tag{2.7}$$

eller på operatorform

$$\hat{T} = \hat{V} + \hat{V}\hat{G}\hat{T}. \tag{2.8}$$

Det här är Lippmann-Schwingerekvationen på operatorform. Ekvation (2.8) kan även skrivas om enligt

$$\left(\hat{1} - \hat{V}\hat{G}\right)\hat{T} = \hat{V}. \tag{2.9}$$

För att beräkna T-matrisen skrivs detta om till en integralekvation som därefter löses numeriskt, vilket genomförs i avsnitt 2.5.

2.3 Bevarade kvanttal och partialvågor

Rumsdelen av den ostörda vågfunktionen $|\phi\rangle$ för de två partiklarna är i CM-systemet en planvåg med rörelsemängd $\boldsymbol{k}$. Planvågor är egentillstånd till operatorn för kinetisk energi, $\hat{H}_0$, och bildar en fullständig bas som betecknas $\{|\boldsymbol{k}\rangle\}$. För att underlätta lösningen av Lippmann-Schwingerekvationen kommer planvågstillstånden att beskrivas i en bas av sfäriska vågor, betecknad $\{|k\ell m\rangle\}$. Här är k beloppet av rörelsemängden. Bankvanttalet ℓ används för att beräkna egenvärdet till $\hat{\boldsymbol{L}}^2$-operatorn och det magnetiska kvanttalet m, ofta betecknat m_ℓ, används för att beräkna egenvärdet till $\hat{L}_z$-operatorn. Varje tillstånd $|k\ell m\rangle$ är ett egentillstånd till operatorerna $\hat{H}_0$, $\hat{\boldsymbol{L}}^2$ och $\hat{L}_z$ samtidigt. Ett planvågstillstånd kan uttryckas som en superposition av denna bas enligt

$$|\boldsymbol{k}\rangle = \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{\ell} C Y^*_{\ell m}(\boldsymbol{k}) \, |k\ell m\rangle \,, \tag{2.10}$$

där C är en normaliseringskonstant och $Y_{\ell m}$ är klotytefunktioner [5, kap. 6.4]. Metoden kallas partialvågsuppdelning och vågorna, som med superposition bygger den ursprungliga vågfunktionen, kallas partialvågor.

Hittills har endast rumsdelen av vågfunktionen betraktats, vilken efter partialvågsuppdelningen består av sfäriska vågor i rörelsemängdsrummet. För att beskriva den totala vågfunktionen beräknas tensorprodukten av spinn-, isospinn- och rumsdelarna av vågfunktionen. Därefter kan de kvanttal som specificerar nukleon-nukleonspridningen att undersökas. De relevanta kvanttalen är bankvanttalet ℓ, det totala spinnet S, det totala rörelsemängdsmomentet J samt det totala isospinnet T för två nukleoner. Av intresse är också den totala isospinnprojektionen T_z. Isospinnprojektionen t_z för en nukleon avgör om

den är en proton ($t_z = +1/2$) eller en neutron ($t_z = -1/2$). Därmed avgör värdet på T_z vilka nukleoner som ingår i spridningen: $T_z = -1$ för neutron-neutronspridning, $T_z = 0$ för proton-neutronspridning och $T_z = +1$ för proton-protonspridning. I tabell 2.1 redovisas de relevanta kvanttalen för spridningsprocessen.

Tabell 2.1: De tillåtna kvanttalen för två nukleoner.

Banrörelsemängdsmoment	$\ell = 0, 1, 2, \ldots$
Totalt spinn	$S = 0, 1$
Totalt rörelsemängdsmoment	$J = \lvert\ell - S\rvert,\ \ell,\ \ell + S$
Total isospinnprojektion	$T_z = -1,\ 0,\ +1$
Totalt isospinn	$T = 0, 1$

Nukleoner är fermioner och de tillåtna partialvågorna måste följa Paulis uteslutningsprincip. Enligt Pauliprincipen kan två fermioner inte befinna sig i samma tillstånd, vilket leder till att den totala vågfunktionen måste vara antisymmetrisk under utbyte av de två nukleonerna. Rumsdelen av vågfunktionen är symmetrisk då ℓ är jämnt och antisymmetriskt då ℓ är udda. Spinn- och isospinndelen av vågfunktionen är symmetriska var för sig då S respektive T är lika med 1 och antisymmetriska då de är lika med 0. Den totala vågfunktionens symmetri kan ses som en produkt av dess beståndsdelars symmetri, där symmetri ges av +1 och antisymmetri av -1. Den totala vågfunktionens symmetri P ges då av

$$P = (-1)^{\ell+S+T}. \tag{2.11}$$

Vanligtvis betraktas partialvågor med hjälp av spektroskopisk notation, enligt

$$^{2S+1}\ell_J. \tag{2.12}$$

Kvanttalen $\ell = 0, 1, 2, 3 \ldots$ ersätts av bokstäverna S, P, D, F ... i spektroskopisk notation. Som exempel uttrycks tillståndet för $\ell = 0$ och $S = 0$ som $^1\mathrm{S}_0$. Dessa partialvågor behandlas separat och kommer hädanefter att kallas kanaler.

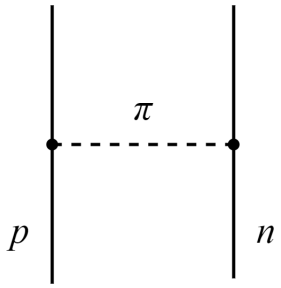
Vid elastisk spridning via stark kärnkraft är kvanttalen S, T och J bevarade. Däremot kan ℓ ändras så att $|\ell - \ell'| = 2$, där ℓ' betecknar bankvanttalet efter spridning. Detta sker endast då $S = 1$ och tillhörande kanaler kopplar ihop olika värden på ℓ och ℓ'. På grund av bevaring av paritet har vi ej fallet då $|\ell - \ell'| = 1$.

2.4 Potentialmodell för den starka kärnkraften

Den starka kärnkraften beskrivs av en komplicerad växelverkanspotential. Det här arbetet syftar till att möjliggöra snabb kalibrering av potentialen utgående från en given växelverkansmodell, som i det här fallet ges av en så kallad första ordningens kiral effektiv fältteori för enpionsutbyte [1]. Potentialen implementeras dock inte som en del av projektet, istället används tillhandahållen kod. Eftersom potentialen är av central betydelse i lösningen av Lippmann-Schwingerekvationen är det dock av intresse att ha en uppfattning om potentialmodellen.

I en första ordningens kiral effektiv fältteori för enpionsutbyte ges potentialen av

$$V(\boldsymbol{k}, \boldsymbol{k}') = -\frac{g_A}{4f_\pi}\boldsymbol{\tau}_1 \cdot \boldsymbol{\tau}_2 \frac{(\boldsymbol{\sigma}_1 \cdot \boldsymbol{q})(\boldsymbol{\sigma}_2 \cdot \boldsymbol{q})}{q^2 + m_\pi^2}, \tag{2.13}$$



Figur 2.2: Ett Feynmandiagram för proton-neutronväxelverkan med en potentialmodell som bygger på första ordningen i en kiral effektiv fältteori med enpionsutbyte.

där $\boldsymbol{q} = \boldsymbol{k'} - \boldsymbol{k}$ i CM-systemet, $\boldsymbol{\sigma}_{1,2}$ är spinnoperatorer, $\boldsymbol{\tau}_{1,2}$ är isospinnoperatorer, g_A är en kopplingskonstant och f_π är sönderfallskonstanten för pionen [1]. Här syns den translationsinvarians som nämndes i avsnitt 2.2, eftersom potentialen inte är rumsberoende.

I det här arbetet undersöks proton-neutronväxelverkan via enpionsutbyte enligt processen i figur 2.2, och det är huvudsakligen potentialen från ekvation (2.13) som används i lösningen av Lippmann-Schwingerekvationen (2.9).

2.5 Numerisk lösning av Lippmann-Schwinger-ekvationen

På grund av partialvågsuppdelningen och potentialens uppbyggnad kan varje tillstånd beskrivas med rörelsemängden k och kvanttalen T, T_z, S, J och ℓ [1]. I det okopplade fallet är alla kvanttal bevarade under spridning. Notationen $T(k', k)$ införs för matrisen T. För varje partialvåg kan då ekvation (2.8) skrivas som

$$T(k', k) = V(k', k) + \frac{2}{\pi} \int_0^\infty \frac{p^2 V(k', p) T(p, k)}{E - E_p + i\varepsilon} \mathrm{d}p, \tag{2.14}$$

där $E = k_0/2\mu$ för elastisk spridning, $E_p = p/2\mu$ och $i\varepsilon$ är en infinitesimal term [6, kapitel 18.3].

Eftersom $E - E_p$ ger upphov till de (reella) polerna $p_0 = \pm k_0$ adderas termen $i\varepsilon$ i nämnaren i ekvation (2.14). Integralen är definierad på positiva reella axeln, vilket innebär att endast polen $p = k_0$ är av betydelse. Det är lämpligt att dela upp integralen i en principalvärdesintegral (betecknad med $\mathcal{P}$) och en imaginär term, vilket kan göras med en standardomskrivning som utnyttjar variabelsubstitution och deformation i komplexa talplanet [6, kapitel 18.3]. I allmänhet ger det

$$\int_{-\infty}^{\infty} \frac{f(p)}{p - p_0 \pm i\varepsilon} \mathrm{d}p = \mathcal{P} \int_{-\infty}^{\infty} \frac{f(p)}{p - p_0} \mathrm{d}p \mp i\pi f(p_0). \tag{2.15}$$

Det här kan användas för att skriva om ekvation (2.14). I ekvation (2.14) motsvaras p_0 av k_0, och $f(p)$ av

$$\frac{2}{\pi} \frac{p^2 V(k', p) T(p, k)}{(k_0 + p)/2\mu}, \tag{2.16}$$

vilket gör att ekvation (2.14) kan skrivas om till

$$T(k', k) = V(k', k) + \frac{2}{\pi} \mathcal{P} \int_0^\infty \frac{p^2 V(k', p) T(p, k)}{(k_0^2 - p^2)/2\mu} \mathrm{d}p - 2i\mu k_0 V(k', k_0) T(k_0, k). \tag{2.17}$$

För att beräkna principalvärdet av integralen numeriskt används resultatet

$$\mathcal{P}\int_0^\infty \frac{f(p)}{k_0^2-p^2}\mathrm{d}p = \int_0^\infty \frac{f(p)-f(k_0)}{k_0^2-p^2}\mathrm{d}p. \tag{2.18}$$

Eftersom integralen i högerledet inte är singulär går den att beräkna numeriskt. Nu kan ekvation (2.17) skrivas som

$$\begin{aligned} T(k',k) =& V(k',k) + \frac{2}{\pi}\int_0^\infty \frac{p^2V(k',p)T(p,k) - k_0^2V(k',k_0)T(k_0,k)}{(k_0^2-p^2)/2\mu}\mathrm{d}p \\ &- 2i\mu k_0 V(k',k_0)T(k_0,k). \end{aligned} \tag{2.19}$$

För att beräkna integralen numeriskt kan Gausskvadratur användas [6, kapitel 18.3]. Med Gausskvadratur approximeras integralen till en summa över N stycken kvadraturpunkter k_j, där $j = 1, 2, ..., N$, med vikter w_j enligt

$$\int_0^\infty g(k)\mathrm{d}k \approx \sum_1^N w_j g(k_j). \tag{2.20}$$

Hur kvadraturpunkterna och deras vikter beräknas beskrivs i avsnitt 3.3.

Eftersom endast elastisk spridning är av intresse behöver T-matrisen $T(k', k)$ bara lösas för den kolonn som innehåller rörelsemängden k_0, vilken ges av $T(k, k_0)$. Då kan ekvation (2.19), tillsammans med approximationen av integralen, skrivas som

$$\begin{aligned} T(k,k_0) =& V(k,k_0) + \frac{2}{\pi}\sum_{j=1}^N \frac{k_j^2V(k,k_j)T(k_j,k_0)w_j}{(k_0^2-k_j^2)/2\mu} \\ &- \frac{2}{\pi}\sum_{j=1}^N \frac{w_j}{(k_0^2-k_j^2)/2\mu}k_0^2V(k,k_0)T(k_0,k_0) - 2i\mu k_0V(k,k_0)T(k_0,k). \end{aligned} \tag{2.21}$$

Den här ekvationen innehåller $N+1$ okända, nämligen kvadraturpunkterna k_j samt punkten k_0 som motsvarar den kinetiska energin för den inkommande nukleonen enligt ekvation (2.1). Låt därför k_i definieras enligt

$$k_i = \begin{cases} k_j, & i = j = 1, \ldots, N \quad \text{(kvadraturpunkter)}, \\ k_0, & i = 0. \end{cases} \tag{2.22}$$

För att förenkla notationen, låt kolonnvektorn vara $T_i \equiv T(k_i, k_0)$ och den motsvarande kolonnen i potentialen vara $V_i \equiv V(k_i, k_0)$. Den fullstora potentialmatrisen betecknas med $V_{ij} \equiv V(k_i, k_j)$. Vi har då $N + 1$ linjära ekvationer för vektorn T_i enligt

$$T_i = V_i + \frac{2}{\pi}\sum_{j=1}^N \frac{k_j^2V_{ij}T_jw_j}{(k_0^2-k_j^2)/2\mu} - \frac{2}{\pi}\sum_{j=1}^N \frac{w_j}{(k_0^2-k_j^2)/2\mu}k_0^2V_iT_0 - 2i\mu k_0V_iT_j. \tag{2.23}$$

För att lösa dem införs en ny vektor [6, kapitel 18.3],

$$D_i = \begin{cases} +\dfrac{2}{\pi}2\mu k_i^2\dfrac{w_j}{k_0^2-k_i^2}, & \text{för } i = 1, 2, ..., N, \\ -\dfrac{2}{\pi}2\mu k_0^2\displaystyle\sum_{j=1}^N \dfrac{w_j}{k_0^2-k_j^2} - 2\mu i k_0, & \text{för } i = 0, \end{cases} \tag{2.24}$$

som motsvarar operatorn $\hat{G}$ i ekvation (2.8). Ekvation (2.23) reduceras då till

$$T_i - \sum_{j=0}^{N} V_{ij} D_i T_j = V_i. \tag{2.25}$$

På matrisform kan alltså ekvation (2.25) uttryckas som

$$[F][T] = [V], \tag{2.26}$$

där

$$F_{ij} = \delta_{ij} - V_{ij} D_j. \tag{2.27}$$

Observera att $V_{ij}D_j$ inte summeras över j, istället sker elementvis multiplikation av matrisen V och vektorn D. I ekvation (2.26) är F en $(N+1)\times(N+1)$-matris och T och V är kolonnvektorer med $N+1$ element. I ekvation (2.27) används den fullstora $(N+1)\times(N+1)$-matrisen V. Ekvation (2.26) går att använda för att beräkna T-matrisen numeriskt, och det är denna matrisekvation som kommer lösas på en GPU.

2.6 Fasskift och tvärsnitt

När Lippmann-Schwingerekvationen är löst behöver information extraheras ur T-matrisen. I synnerhet kommer så kallade fasskift, som används för att parametrisera spridningens tvärsnitt, beräknas ur den så kallade S-matrisen som i sin tur kan fås av T-matrisen. Definiera S-matrisen som den matris som tar vågfunktionen från sitt initial- till sluttillstånd. Eftersom sannolikhetsflöde måste vara bevarat i spridningsförloppet, och varje partialvåg betraktas som ett separat spridningsförlopp, behöver S vara unitär i varje partialvåg ℓ, det vill säga att $|S_\ell| = 1$ för alla ℓ. Alltså kan S-matrisen som mest bidra med förändring i fas. Det går därför att parametrisera S-matrisen med fasskiften, men okopplade och kopplade kanaler kräver olika tillvägagångssätt.

I okopplade kanaler, där ℓ bevaras, behöver S_ℓ endast bestå av en skalär. Detta gäller inte för de kopplade kanalerna, där ℓ förändras till antingen $|J-1|$ eller $J+1$, se avsnitt 2.3. Sluttillståndet består då av en blandning av två olika partialvågor, vilket kräver att S_ℓ i detta fall ges av en 2×2-matris. Nedan följer en beskrivning av dessa två fall i mer detalj.

2.6.1 Okopplade kanaler

Ett välkänt resultat är att tvärsnittet σ ges av

$$\sigma = \int |f(\theta)|^2 \mathrm{d}\Omega, \tag{2.28}$$

där Ω är en rymdvinkel och $f(\theta)$ är spridningsamplituden i riktningen θ. Spridningsamplituden kan tolkas som sannolikhetsamplituden av att spridningen sker i en riktning med polvinkel θ. Tvärsnittet bestäms alltså av spridningsamplituden, som i sin tur kan bestämmas ur vågfunktionens fasskift.

Eftersom S_ℓ-matrisen för en okopplad kanal är en skalär, och dessutom är unitär, kan den parametriseras med partialvågsfasskiftet δ_ℓ enligt

$$S_\ell = e^{2i\delta_\ell}. \tag{2.29}$$

Den spridda vågfunktionen är en superposition av den ostörda partialvågen som färdas i sin originalriktning och den spridda partialvågen med spridningsamplitud f_ℓ, därmed gäller

$$S_\ell = 1 + 2ik_0 f_\ell \tag{2.30}$$

där första termen motsvarar den ostörda vågen och den andra termen motsvarar den spridda. Därmed fås sambandet

$$f_\ell = \frac{S_\ell - 1}{2ik_0} = \frac{e^{2i\delta_\ell} - 1}{2ik_0}, \tag{2.31}$$

och med f_ℓ bestämd kan totala spridningsamplituden f erhållas genom viktad summering över partialvågorna, varpå ekvation (2.28) kan användas för att bestämma tvärsnittet. För mer detalj, se exempelvis [5, kap. 6].

All information om spridningsförloppet fås från T-matrisen, varför fasskiftet δ_ℓ kan kopplas till elementet $T_\ell = T(k_0, k_0)$. Enligt [6, ekv. (8.27)] gäller för varje partialvåg ℓ att

$$T_\ell = \frac{e^{i\delta_\ell}\sin(\delta_\ell)}{-2\mu k_0} = \frac{e^{i\delta_\ell}(e^{i\delta_\ell} - e^{-i\delta_\ell})}{-4i\mu k_0} = \frac{e^{2i\delta_\ell} - 1}{-4i\mu k_0}. \tag{2.32}$$

Det gäller alltså att

$$e^{2i\delta_\ell} = 1 - 4i\mu k_0 T_\ell \tag{2.33}$$

och fasskiftet löses ut enligt

$$\delta_\ell = \frac{-i}{2}\ln(1 - 4i\mu k_0 T_\ell) \tag{2.34}$$

där principalgrenen av den komplexa logaritmen används. Vid okopplad spridning med tillräckligt låg energi kommer fasskiftet i högre ordningens partialvågor att avta väldigt snabbt och därför behöver endast $\ell = 0$-vågen användas. Det samband som är intressant är alltså

$$T_0 = \frac{1 - e^{2i\delta_0}}{4i\mu k_0}. \tag{2.35}$$

2.6.2 Kopplade kanaler

Eftersom S_ℓ är en symmetrisk 2×2-matris behövs tre parametrar: de två fasskiften δ_- respektive δ_+ och en så kallad mixingvinkel ε_J. Det finns två vanligt förekommande konventioner för dessa parametrar: Blatt-Biedenharn-konventionen (BB) [7]

$$S_{\text{BB}} = \begin{bmatrix} \cos(\varepsilon_J) & \sin(\varepsilon_J) \\ -\sin(\varepsilon_J) & \cos(\varepsilon_J) \end{bmatrix} \begin{bmatrix} e^{2i\delta_-} & 0 \\ 0 & e^{2i\delta_+} \end{bmatrix} \begin{bmatrix} \cos(\varepsilon_J) & -\sin(\varepsilon_J) \\ \sin(\varepsilon_J) & \cos(\varepsilon_J) \end{bmatrix}, \tag{2.36}$$

och Stapp-konventionen (också kallad ”bar”) [8] vars parametrar brukar markeras med överstreck,

$$S_{\text{bar}} = \begin{bmatrix} e^{2i\bar{\delta}_-}\cos(2\bar{\varepsilon}_J) & ie^{2i(\bar{\delta}_- + \bar{\delta}_+)}\sin(2\bar{\varepsilon}_J) \\ ie^{2i(\bar{\delta}_- + \bar{\delta}_+)}\sin(2\bar{\varepsilon}_J) & e^{2i\bar{\delta}_+}\cos(2\bar{\varepsilon}_J) \end{bmatrix}. \tag{2.37}$$

Enligt Stapp [8] kan båda konventionernas parametrar relateras till varandra enligt

$$\begin{aligned} \delta_+ + \delta_- &= \bar{\delta}_+ + \bar{\delta}_-, \\ \sin\bigl(\bar{\delta}_- - \bar{\delta}_+\bigr) &= \frac{\tan(2\bar{\varepsilon}_J)}{\tan(\varepsilon_J)}. \end{aligned} \tag{2.38}$$

Haftel och Tabakin [3] beskriver en metod som använder den så kallade R-matrisen

$$R_\ell = \begin{bmatrix} R_{11} & R_{12} \\ R_{12} & R_{22} \end{bmatrix} \tag{2.39}$$

för att beräkna BB-parametrarna. I praktiken är R-matrisen endast en alternativ, dock reell, formalism som är ekvivalent med T-matrisen och innehåller samma information. Ur R-matrisen extraheras BB-parametrarna enligt följande:

$$\begin{aligned} \tan(2\varepsilon_J) &= \frac{2R_{12}}{R_{11} - R_{22}}, \\ \tan(\delta_\pm) &= \frac{1}{2}\rho\left(R_{11} + R_{22} \mp \frac{(R_{11} - R_{22})}{\cos(2\varepsilon_J)}\right) = \frac{1}{2}\rho\left(R_{11} + R_{22} \mp \frac{2R_{12}}{\sin(2\varepsilon_J)}\right). \end{aligned} \tag{2.40}$$

där $\rho = 2\mu k_0$. R- och T-matriserna kan relateras till varandra med sambandet

$$R = (I - i\rho T)^{-1} T. \tag{2.41}$$

För varje R_ℓ och T_ℓ ger detta

$$\begin{aligned} R_{11} &= \frac{T_{11} + i\rho(T_{12}^2 - T_{11}T_{22})}{(1 - i\rho T_{11})(1 - i\rho T_{22}) + \rho^2 T_{12}^2}, \\ R_{12} &= \frac{T_{12}}{(1 - i\rho T_{11})(1 - i\rho T_{22}) + \rho^2 T_{12}^2}, \\ R_{22} &= \frac{T_{22} + i\rho(T_{12}^2 - T_{11}T_{22})}{(1 - i\rho T_{11})(1 - i\rho T_{22}) + \rho^2 T_{12}^2}. \end{aligned} \tag{2.42}$$

Genom att sätta in sambanden (2.42) i ekvationerna för BB-parametrarna (2.40) kan man visa att parametrarna för varje partialvåg ℓ kan erhållas från T_ℓ enligt

$$\begin{aligned} 2\varepsilon_J &= \arctan\left(\frac{2T_{12}}{T_{11} - T_{22}}\right) \\ \delta_\pm &= \frac{i}{2}\ln\left(1 - i\rho\left(T_{11} + T_{22} \mp \frac{2T_{12}}{\sin(2\varepsilon_J)}\right)\right). \end{aligned} \tag{2.43}$$

Dessa tre parametrar kan, precis som fasskiftet i okopplade kanaler, användas för att bygga upp spridningsförloppets tvärsnitt. I de kopplade kanalerna är beräkningarna däremot mer omfattande och därför utelämnas de ur denna rapport.

3

Implementering av teorin på CPU

För att lösa Lippmann-Schwingerekvationen många gånger så snabbt som möjligt kommer teorin från kapitel 2 implementeras med GPU-kod i programspråket C++. Anledningen till att programmet skrivs för en GPU snarare än en CPU är att beräkningarna kan parallelliseras och möjligen utföras snabbare. Innan programmet effektiviseras är det dock värdefullt att realisera och testa teorin med CPU-kod, eftersom GPU-programmering medför utmaningar som inte nödvändigtvis är relaterade till teorin. På så sätt kan GPU-utvecklingen vara centrerad enbart kring effektivisering utan att överflödiga teorikomplikationer uppstår.

I avsnitt 3.1 ges en överblick av CPU-programmets upplägg, följt av detaljbeskrivningar av huvudfilerna i avsnitten 3.2–3.5. Där diskuteras speciellt viktiga delar av koden; den fullständiga koden för CPU-programmet återfinns i CPU-grenen av projektets GitHub-projekt [a]. I avsnitt 3.6 verifieras att koden fungerar och ger korrekt resultat, och i avsnitt 3.7 genomförs en analys av Gausskvadraturens precision.

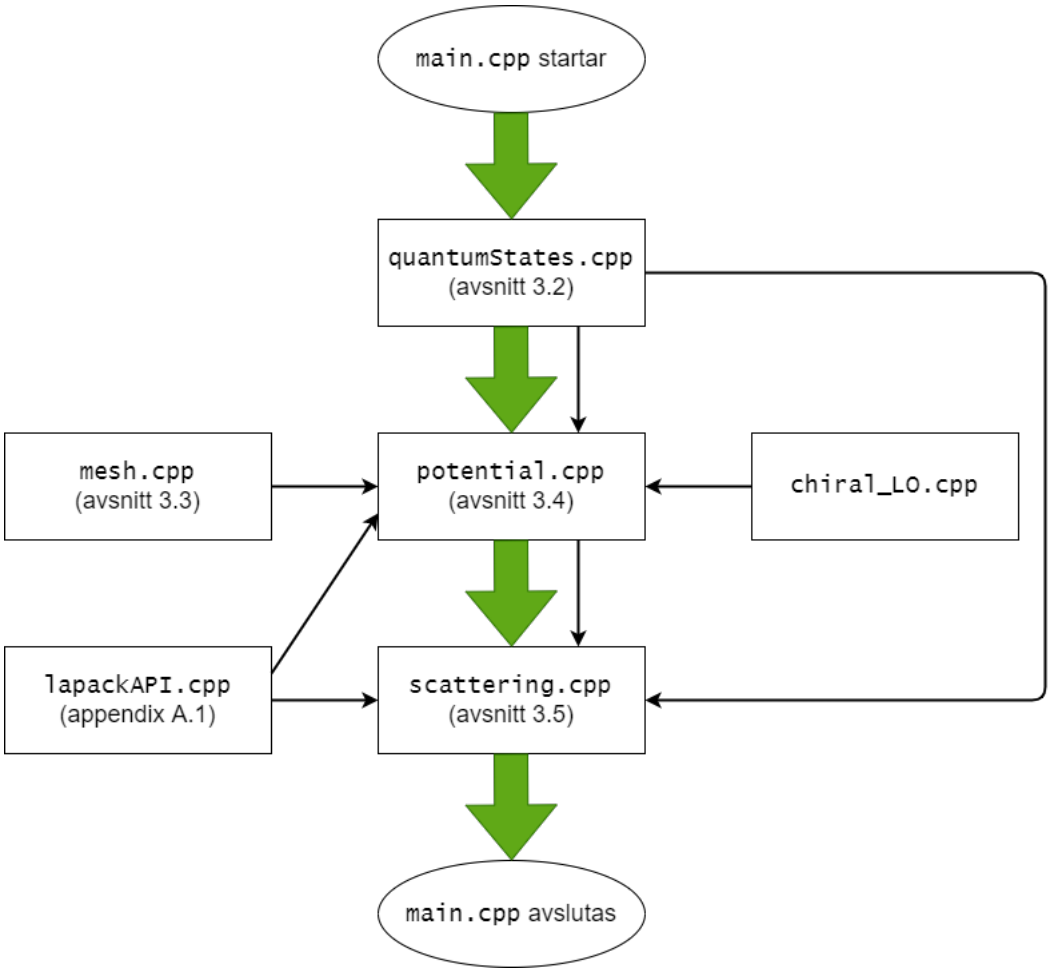
3.1 Överblick av CPU-programmets upplägg

För att lösa Lippmann-Schwingerekvationen med CPU-kod skapades ett program i C++ bestående av flera filer. I det här avsnittet ges först en översiktlig genomgång av programmets upplägg, och i de följande avsnitten beskrivs utförandet i de relevanta filerna i detalj. Varje fil har ett särskilt syfte för att Lippmann-Schwingerekvationen skall kunna lösas, och filerna beror på varandra i olika mån. För att illustrera programmets uppbyggnad används flödesschemat i figur 3.1. Där syns filernas namn, tillhörande avsnitt för detaljer samt deras beroende av varandra.

Hela programmet baseras på att en fil `main.cpp` anropar funktioner från andra filer; programmet börjar alltså när `main.cpp` börjar exekveras och avslutas när `main.cpp` avslutas. De tre huvudfilerna som anropas från `main.cpp` är `quantumStates.cpp` (avsnitt 3.2), `potential.cpp` (avsnitt 3.4) och `scattering.cpp` (avsnitt 3.5). Dessa syftar till att ställa upp kvanttillstånden, konstruera växelverkanspotentialen respektive lösa matrisekvationen (2.26). Utöver de tre huvudfilerna finns ytterligare tre relevanta filer. Filen `lapackAPI.cpp` lägger grunden för all matrismanipulation med hjälp av linjär algebrabiblioteken LAPACK och BLAS. Detta beskrivs utförligt i Appendix A.1. Där definieras bland annat klassen `LapackMat`, som abstraherar matriser och matrisoperationer. För att ställa upp potentialen används filen `chiral_LO.cpp`[b]. Dessutom används filen `mesh.cpp` (avsnitt 3.3) som konstruerar kvadraturpunkterna och tillhörande vikter som behövs för att lösa Lippmann-Schwingerekvationen numeriskt.

[a] https://github.com/JosephLofving/kandidat/tree/cpu

[b] Filen `chiral_LO.cpp` tillhandahölls i projektets början och är skriven av vår handledare Sean Miller.



Figur 3.1: En grafisk representation av CPU-programmet. De breda pilarna visar anropningskedjan av de tre huvudfilerna från `main`-filens start till slut. De smala pilarna visar vilka filer som beror på varandra. En smal pil från en fil A till en annan fil B representerar att A används av B.

3.2 `quantumStates.cpp` – Uppställning av kvanttillstånd

Vid programmets början initialiseras de aktuella tillstånden i partialvågsbasen som introducerades i avsnitt 2.3. Detta sker i filen `quantumStates.cpp`, där objektet `QuantumState` definieras. Objektet sparar alla de kvanttal som specificerar tillståndet i en lista vid namn `state`. Funktionen `setupBasis` skapar ett antal sådana tillstånd baserat på det önskade intervallet på kvanttalet J, det vill säga det totala rörelsemängdsmomentet. Därefter testas alla kombinationer av kvanttalen i tabell 2.1; de tillstånd som uppfyller Pauliprincipen enligt ekvation 2.11 läggs till i listan `basestate` tillsammans med pariteten av tillståndet samt isospinnprojektionen $T_z = 0$ (se avsnitt 2.3).

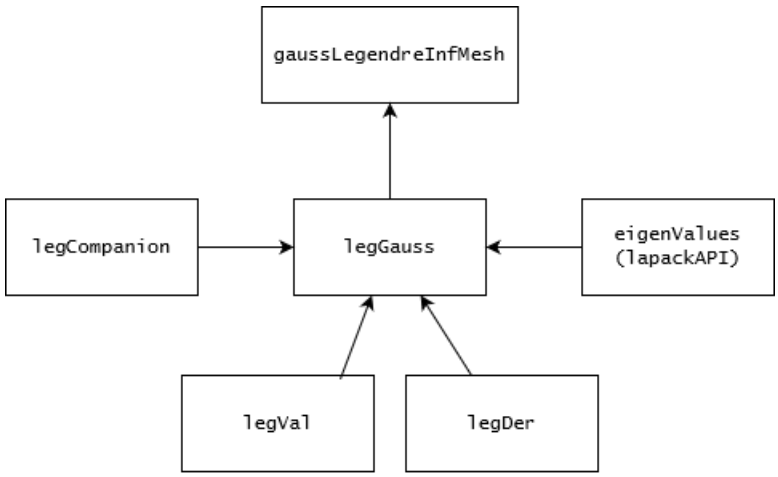
```
1  std::vector<QuantumState> setupBasis(int Jmin, int Jmax){
2    std::vector<QuantumState> basis;
3    for (int J = Jmin; J <= Jmax; J++){
4      for (int S = 0; S < 2; S++){
5        for (int l = abs(J - S); L <= J + S; l++){
6          for (int T = 0; T < 2; T++){
```

```
7           if ((l + S + T) % 2 != 0) {
8             QuantumState basestate;
9             basestate.addQuantumNumber("Tz", 0);
10            basestate.addQuantumNumber("l", L);
11            basestate.addQuantumNumber("pi", pow(-1, L));
12            basestate.addQuantumNumber("S", S);
13            basestate.addQuantumNumber("J", J);
14            basestate.addQuantumNumber("T", T);
15
16            basis.push_back(basestate);
17            }}}}}
18    return basis;
19  }
```

Därefter grupperas kvanttillståndet i kanaler, där en kanal består av de tillåtna tillstånden före och efter spridningen. För de okopplade fallen har en kanal samma tillstånd innan och efter spridningen, eftersom alla kvanttal då är bevarade. Vid de kopplade fallen innehåller en kanal de fyra olika tillstånden där $|\ell - \ell'| = 0, 2$.

3.3 `mesh.cpp` – Konstruktion av kvadraturpunkter och vikter

Filen `mesh.cpp` skapar kvadraturpunkter och vikter enligt en Gauss-Legendrekvadratur på ett valfritt intervall, se ekvation 2.20. Filen är strukturerad enligt flödesschemat i figur 3.2.



Figur 3.2: En grafisk representation av `mesh.cpp`. Pilarna visar vilka filer som beror på varandra, där en pil från en funktion A till en annan funktion B representerar att A används av B. Funktionerna är namngivna enligt Pythonpaketet `numpy.polynomial.legendre`.

Den funktion som anropas från `main.cpp` är `gaussLegendreInfMesh`. Denna tar in två parametrar: `N` samt `scale`, där `N` är antalet kvadraturpunkter för Gauss-Legendremeshen, och `scale` bestämmer avståndet mellan dessa punkter. Dessa parametrar väljs så att kvadraturen löser integraler med tillräcklig noggrannhet. Högre antal kvadraturpunkter ger mer precision men också markant längre körtid av programmet. Det beror på att matriserna i `scattering.cpp` är av storlek $(N + 1) \times (N + 1)$, antal operationer

i matrisinvertering skalar med N^3, samt att antalet operationer skalar linjärt med tiden. Variabeln `scale` påverkar inte körtiden, utan behöver endast sättas till ett lämpligt värde. I vårt fall har 100,0 MeV visat sig vara lämpligt. Funktionen returnerar vikterna `w[j]` och kvadraturpunkterna `k[j]` från ekvation 2.20.

Kodens huvudberäkningar genomförs i funktionen `legGauss`. Här implementaras den Gaussiska kvadraturen som nämns ekvation 2.20, och mer specifikt Gauss-Legendrekvadratur [9]. Kvadraturpunkterna k_j är rötterna till Legendrepolynomet av grad N ($L_N(x)$) och vikterna fås enligt

$$w_j = \frac{C}{(L'_N(k_j) \cdot L_{N-1}(k_j)}, \tag{3.1}$$

där C är en reell konstant som bestäms via välkända värden för integrering sådan att integraler på intervallet $[-1, 1]$ blir korrekta [10].

`legGauss` anropar först metoden `legCompanion`, som beräknar följeslagarmatrisen till Legendrepolynomet av grad N. Nollställena till ett polynom ges av egenvärdena till dess följeslagarmatris [11], och således är dessa egenvärden kvadraturpunkterna. Egenvärdena beräknas med funktionen `eigenValues` i `lapackAPI.cpp`, se appendix A.1. Sedan tillämpades även en iteration av Newtons metod för att ge ytterligare beräkningsprecision till nollställena. Vikterna beräknas till sist via applicering av ekvation (3.1). Funktionerna `legVal` och `legDer` är hjälpfunktioner till `legGauss`, där `legVal` evaluerar värdet på ett givet Legendrepolynom i vissa punkter, medan `legDer` deriverar ett givet Legendrepolynom.

Enligt teorin i avsnitt 2.5 skall intervallet $[0, \infty)$ integreras, men `legGauss` implementerar Gauss-Legendrekvadratur på intervallet $[-1, 1]$. För att överrensstämma med teorin utökas i `gaussLegendreInfMesh` slutligen integrationsintervallet till $[0, \infty)$ genom följande operationer på kvadraturpunkterna och vikterna respektive

$$k_j = C \cdot \tan\left[\pi(k_j^{[-1,1]} + 1)/4\right], \quad w_j = C \cdot w_j^{[-1,1]} \cdot \frac{\pi}{4\cos^2\left[\pi(k_j^{[-1,1]} + 1)/4\right]}. \tag{3.2}$$

3.4 `potential.cpp` – Bildandet av potentialmatrisen

När de tillåtna kvanttillstånden är grupperade i kanaler och kvadraturpunkterna är skapade kan potentialmatrisen V fyllas, `VMatrix` i koden. Filen som gör det är `potential.cpp` och den använder tillstånden för den aktuella kanalen, kvadraturpunkterna samt energin T_{labb}. Till att börja med används T_{labb} för att beräkna rörelsemängden k_0 enligt ekvation (2.1), varpå k_0 läggs till i slutet av listan med kvadraturpunkter (vektorn `k` i koden). Elementen i V-matrisen beräknas med första ordningens kiral effektiv fältteori, se avsnitt 2.4. Argumenten som behövs är rörelsemängderna k och k' (`kIn` respektive `kOut` i koden), kvanttalen S, J, T och T_z samt en tom lista `VArray` att fylla med potentialvärden. Listan har sex platser som fylls med olika värden beroende på kanalen. Index som anger vilken plats i listan som ska användas bestäms med funktionen `getArrayIndex`, vilket kan resultera i något av värdena i tabell 3.1

Kvadraturpunkterna representerar rörelsemängder och alla kombinationer av k och k' beräknas för alla kvanttillstånd i kanalen. Värdet från potentialen läggs in i potentialmatrisen `VMatrix`, där variablerna `rowIndex` och `colIndex` hanterar förskjutningen och placeringen av elementen för de kopplade kanalerna. Potentialmatrisen för de kopplade kanalerna sorteras enligt följande matris, där ℓ' är `ll`.

Tabell 3.1: Index från funktionen `getArrayIndex` och dess innebörd.

`arrayIndex`	Kvanttal
0	$S = 0,\ L = LL = J$
1	$S = 1,\ L = LL = J$
2	$S = 1,\ L = J + 1,\ LL = J + 1$
3	$S = 1,\ L = J - 1,\ LL = J - 1$
4	$S = 1,\ L = J + 1,\ LL = J - 1$
5	$S = 1,\ L = J - 1,\ LL = J + 1$

$$\begin{array}{c} \\ \texttt{l} = \texttt{J} - 1 \\ \texttt{l} = \texttt{J} + 1 \end{array} \begin{array}{c} \texttt{ll} = \texttt{J} - 1 \quad \texttt{ll} = \texttt{J} + 1 \\ \begin{pmatrix} \texttt{VArray[3]} & \texttt{VArray[5]} \\ \texttt{VArray[4]} & \texttt{VArray[2]} \end{pmatrix} \end{array}.$$

Till slut fås en $(N+1)\times(N+1)$-matris för de okopplade kanalerna, eller en motsvarande $2(N+1)\times 2(N+1)$-matris för de kopplade kanalerna, där N är antalet kvadraturpunkter. Plus ett tillkommer eftersom k_0 lades till i vektorn `k`. Matrisen fylls enligt koden nedan.

```
1  LapackMat potential(...)
2  ...
3      for (int kIn = 0; kIn < k.size(); kIn++) {
4        for (int kOut = 0; kOut < k.size(); kOut++) {
5          potentialClassPtr->V(k[kIn], k[kOut], coupled, S, J, T, Tz, VArray);
6          VMatrix.setElement(kIn+rowIndex*k.size(),
7            kOut+colIndex*k.size(),factor*VArray[arrayIndex]);
8        }
9      }
10     return VMatrix;
11 }
```

3.5 `scattering.cpp` – Lösning av matrisekvationen samt fasskiftsberäkning

När alla tillstånd, kvadraturpunkter och potentialen är uppställda används filen `scattering.cpp` för att lösa Lippmann-Schwingerekvationen. Filen bygger helt på avsnitt 2.5 och 2.6. Först ställs D-vektorn upp enligt ekvation (2.24), som innehåller den reducerade massan μ, kvadraturpunkterna k_i, de tillhörande vikterna w_i samt rörelsemängden k_0. Samma notation används i koden. Ur ekvationen observeras att samma uttryck används för elementen $i = 1, 2, \ldots, N$ medan ett annat används för element $i = 0$. Vid implementering i C++ är det dock enklare att placera detta element i slutet av vektorn, vilket ger upphov till en förskjutning av index i koden. Det gjordes redan i `potential.cpp` då k_0 lades till i slutet av k-vektorn (se avsnitt 3.4). Följande är ett utdrag ur den funktion som ställer upp D-vektorn.

```
1  std::vector<std::complex<double>> setupDVector(...) {
2      ...
3      double sum = 0;
4      /* Beräkna D med ekvation (2.24) */
5      for (int i = 0; i < N; i++) {
```

```
      D[i] = -twoOverPi * twoMu * pow(k[i], 2) * w[i] / (pow(k0, 2) - pow(k[i], 2));
      sum += w[i] / (pow(k0, 2) - pow(k[i], 2));
    }
    D[N] = twoOverPi * twoMu * pow(k0, 2) * sum + twoMu * I * k0;
    return D;
}
```

När D-vektorn är uppställd kan den så kallade VD-matrisen skapas, vilken skrivs som $V_{ij}D_j$ i ekvation (2.27). VD-matrisen är alltså elementvis multiplikation av matrisen V och vektorn D. I det kopplade fallet förenklas beräkningarna med några kodrader vars syfte är att bifoga en kopia av D-vektorn till sig själv. Huruvida spridningen är kopplad eller inte avgörs av en funktion `isCoupled`. D-vektorn är därför dubbelt så lång i det kopplade fallet. VD-matrisen skapas därefter med elementvis multiplikation med funktionerna `setElement` och `getElement` ur `lapackAPI.cpp`, se appendix A.1.2.

```
LapackMat setupVDKernel(...) {
    ...
    /* Avgör om kanalen är kopplad */
    if (isCoupled(channel)) {
      D.insert(std::end(D), std::begin(D), std::end(D));
    }

    /* Beräkna VD-matrisen från ekvation (2.27) */
    LapackMat VD = LapackMat(D.size());
    for (int row = 0; row < D.size(); row++) {
      for (int column = 0; column < D.size(); column++) {
        VD.setElement(row, column, V.getElement(row, column) * D[column]);
      }
    }
    ...
    return VD;
}
```

Med VD-matrisen kan ekvationen för T-matrisen till slut lösas. Då används definitionen av F i ekvation (2.27) för att sedan direkt tillämpa ekvation (2.26). Det görs med funktionen `solveMatrixEq` från `lapackAPI.cpp`, som löser ekvationen $FT = V$ med syntaxen `solveMatrixEq(F,V)` (se appendix A.1.4).

I `scattering.cpp` fås fasskiftet δ för det okopplade fallet direkt ur det sista elementet i T-matrisen med ekvation (2.35) (observera indexförskjutningen; i ekvation (2.35) fås fasskiftet med T-matrisens första element medan det i programmet fås ur T-matrisens sista element). Ekvationen implementeras med följande kodrader.

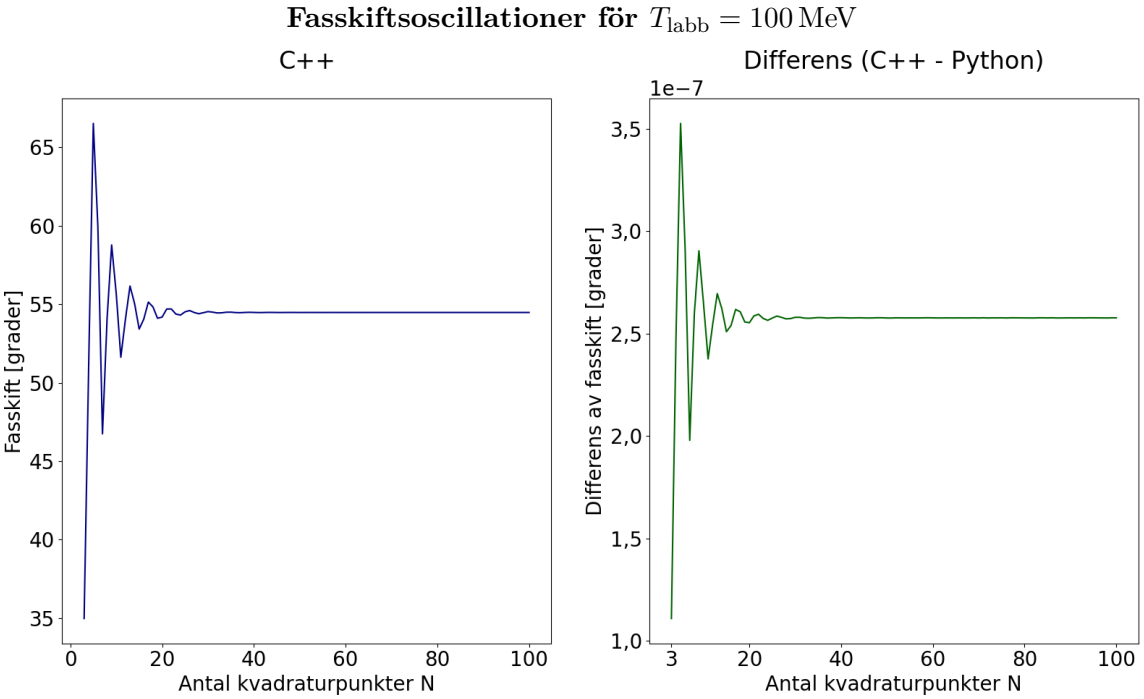
```
std::vector<std::complex<double>> computePhaseShifts(...) {
    ...
    /* Det okopplade fallet */
    std::complex<double> T0 = T.getElement(N, N);
    std::complex<double> argument = 1.0 - 2.0 * I * rho * T0;
    std::complex<double> delta = (-0.5 * I) * std::log(argument) * constants::rad2deg;
    ...
}
```

I det kopplade fallet används Blatt-Biedenharns konvention enligt ekvation (2.43) för att bestämma mixingvinkeln och de två fasskiften med hjälp av T-matrisen. Därefter omvandlas variablerna till Stapps konvention enligt ekvation (2.38).

3.6 Verifiering av CPU-programmet

Samtliga kontroller och beräkningar i detta samt nästföljande avsnitt är alla utförda på samma datorsystem samt på tillståndet $^1\mathrm{S}_0$. En befintlig och testad kod skriven i Python [12], som i rapporten refereras till som ”Pythonkoden”, användes i detta avsnitt för att verifiera resultaten från C++-koden.

$T_{\text{labb}} = 100\,\text{MeV}$ fixeras, och fasskift studeras som funktion av antalet kvadraturpunkter N. Sedan beräknas skillnaden mellan resultaten från C++ och Python för att kvantifiera exakt hur mycket dessa skiljer sig, se figur 3.3.



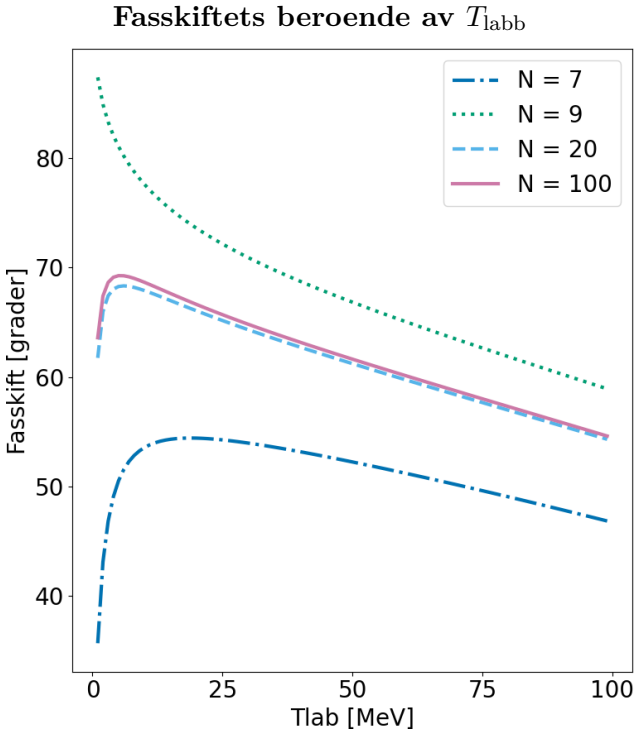
Figur 3.3: Fasskift mot N för C++ till vänster samt skillnaden mellan C++ och Pythonkoden till höger. $T_{\text{labb}} = 100\,\text{MeV}$ för alla mätningar. Differensen av fasskiftet konvergerar mot $2{,}58 \cdot 10^{-7}$. Med logaritmisk skala erhölls samma form.

Notera i högra delen av figur 3.3 att för alla N är skillnaden i fasskift mellan C++ och Python i storleksordning 10^{-7}. Denna skillnad härstammar från att Pythons `float` tillämpar singelprecision, vars högsta precision är 10^{-8}. C++ tillämpar dock dubbelprecision där den högsta precisionen är 10^{-16}. Det är således väntat att se brus i flyttalen vid storleksordning 10^{-7} i Pythonkoden. Skillnaden i resultat mellan programmeringsspråken beror alltså på detta brus nära den numeriska nollan i Python. Den beräknade skillnaden är den minsta som kan åstadkommas mellan programmeringsspråken, och metoderna antas därmed i fortsättningen vara ekvivalenta i sina resultat. Således används enbart C++-programmet i den följande analysen.

3.7 Konvergens med avseende på antal kvadraturpunkter

Syftet med detta avsnitt är att för C++-koden i $^1S_0$ bestämma vilka värden på antalet kvadraturpunkter N som är lämpliga att använda. Då $N \to \infty$ ökar precisionen i resultatet upp till dubbelprecisionsgränsen på 10^{-16}, men programmets körtid ökar (som tidigare diskuterats i avsnitt 3.3) enligt N^3. En bra balans mellan precision och körtid eftersträvas till senare då effektiva och tillräckligt noggranna beräkningar önskas utföras på en GPU. Vi anser att en felmarginal på 10^{-3} för fasskiftet är tillräckligt noggrant för programmets ändamål.

Ett enkelt test görs där fasskiftet studeras som funktion av T_{labb} för olika värden på antalet kvadraturpunkter N, se figur 3.4. Fasskiftsoscillationen för högre antal kvadraturpunkter visas i figur 3.5.

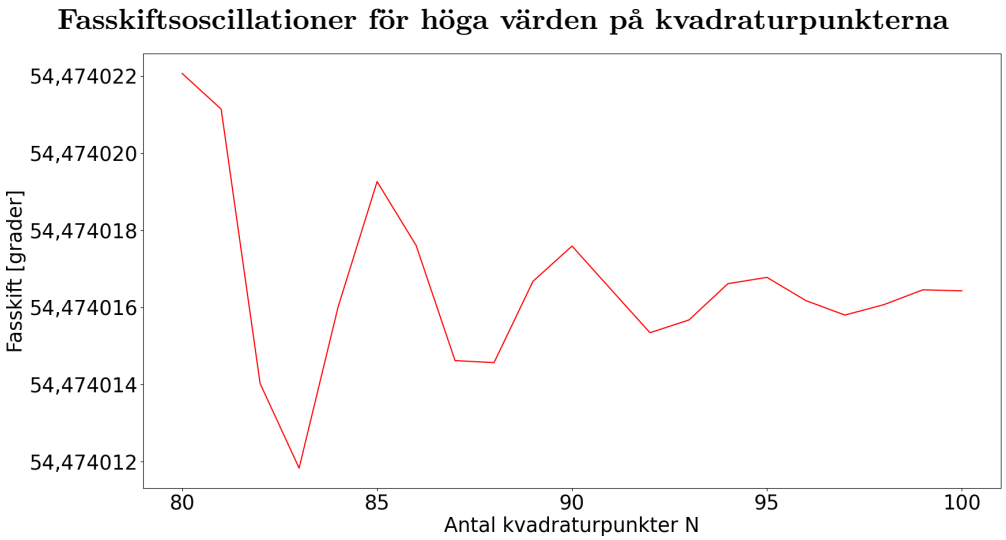


Figur 3.4: Fasskift mot T_{labb}, för olika N. Det syns att högre antal kvadraturpunkter ger bättre precision för fasskiftet.

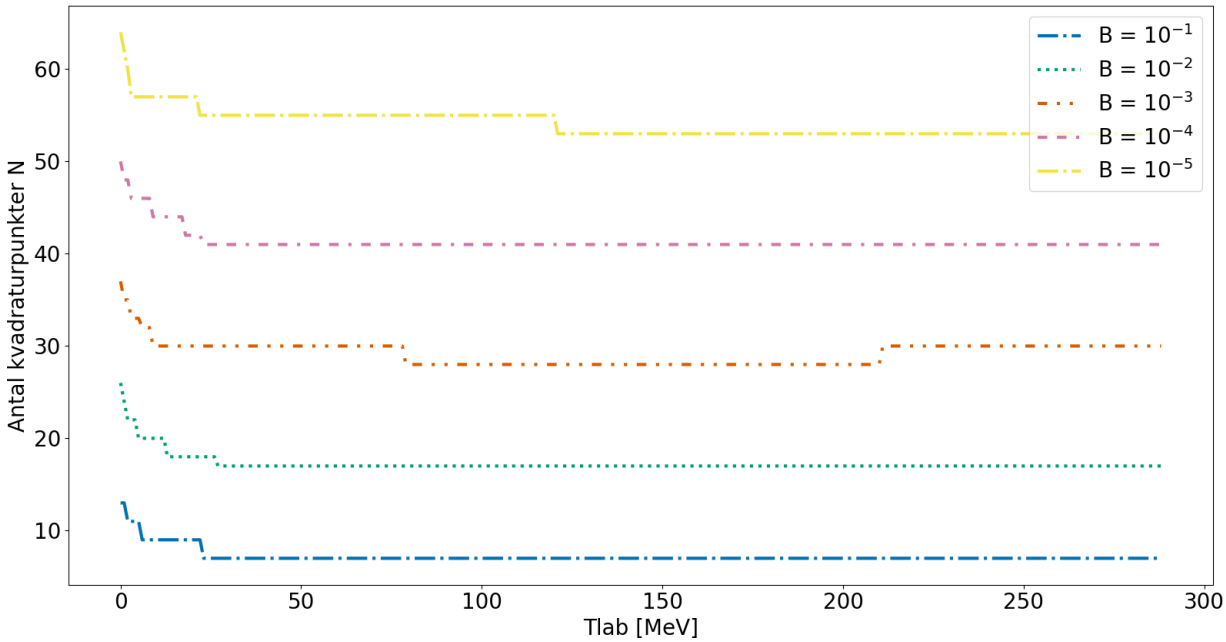
Notera i figur 3.5 att oscillationen av fasskiftet i C++ har en amplitud på mindre än 10^{-5}. Då den eftersökta precisionen på 10^{-3} är två storleksordningar lägre än detta kommer fasskiftet vid $N = 100$ anses vara ”konvergerat”. I resterande del av avsnittet studeras storleken på fasskiftets metodfel för $N < 100$, och jämförs med det ”konvergerade” värdet av fasskiftet vid $N = 100$.

Fasskiftet anses konvergerat för ett visst brytvärde B vid ett kvadraturpunktsantal N_0 om oscillationsamplituden är mindre än B för alla $N > N_0$. Detta brytvärde väljs helt arbiträrt, där det enda kravet är att det måste vara större än 10^{-5}. Detta eftersom 10^{-5} har ansetts vara försumbart litet, som följd av att fasskiftet anses vara konvergerat då det oscillerar med amplituden 10^{-5}, se figur 3.5.

Den önskade precisionen på programmet var 10^{-3}. Kurvan för $B = 10^{-3}$ i figur 3.6 befinner sig helt under linjen $N = 40$, så beräkningar vid $N = 40$ kommer ha en precision av minst 10^{-3} för alla T_{labb}. I följande avsnitt kommer således $N = 40$ användas i GPU-



Figur 3.5: Fasskift mot N, för $T_{\text{labb}} = 100\,\text{MeV}$. Notera att skalan på y-axeln är av storleksordning 10^{-5}. Variationen av fasskiftet är även på denna skala väldigt liten för kvadraturpunkter nära $N = 100$.



Figur 3.6: Antal kvadraturpunkter N som en funktion av T_{labb}. N är det lägsta antalet kvadraturpunkter för vilket alla större antal kvadraturpunkter skiljer sig från det konvergerade värdet med maximalt B.

koden för alla beräkningar till $^1\text{S}_0$.

4

Parallelliserade beräkningar på GPU

I det här kapitlet beskrivs vår metod för att lösa Lippmann-Schwingerekvationen många gånger parallellt på en GPU. Metoden baseras på teorin i kapitel 2 och CPU-programmet i kapitel 3. Först introduceras principerna för parallellisering med CUDA i avsnitt 4.1. Därefter ges en överblick av GPU-programmet i avsnitt 4.2, följt av detaljbeskrivningar av huvudfilerna i avsnitt 4.3, 4.4 och 4.5 med fokus på det som skiljer från CPU-programmet. Fullständig kod för GPU-programmet återfinns i huvudgrenen av GitHub-projektet[a].

4.1 Principer för parallellisering

CUDA är ett programmeringsgränssnitt utvecklat av Nvidia för användning av deras grafikprocessorer för allmänna beräkningar, vilket gör det möjligt att utnyttja en grafikprocessors parallelliseringsförmåga till andra ändamål än grafikberäkningar. Källkodsfiler för C++ som använder CUDA har filändelsen `.cu` och kompileras med Nvidias kompilator `nvcc`. Ett krav för att kunna använda CUDA är att datorn innehåller en kompatibel grafikprocessor från Nvidia. I CUDA används ofta ordet ”device” för GPU:n, och ”host” för CPU:n.

Inför utvecklingen av GPU-programmet var det viktigt att förstå grundprinciperna för parallellisering. Det inkluderar bland annat användningen av pekare (avsnitt 4.1.1), funktioner som kan köras parallellt på en GPU (avsnitt 4.1.2), särskild minneshantering (avsnitt 4.1.3) samt hur parallella beräkningar implementeras i funktioner (avsnitt 4.1.4). För att lösa Lippmann-Schwingerekvationen i ekvation (2.26) behövs dessutom ett verktyg för matrismanipulation, mer specifikt biblioteket cuBLAS som implementerar beräkningsmetoder för linjär algebra (avsnitt 4.1.5).

4.1.1 Introduktion till pekare i C++

Ett centralt koncept i C++ är pekare; en variabel som kan sägas peka på en minnesadress. I CPU-programmet abstraherades de flesta pekarna via den egenutvecklade objekttypen `LapackMat` (se appendix A.1) och med objekttypen `std::vector` för ökad läslighet. I GPU-programmet är dock prestanda av större vikt, varför den ooptimerade klassen `LapackMat` inte används. Vidare finns inte `std::vector` tillgängligt på GPU:n. Fält och pekare används därmed explicit i GPU-programmet.

Variabler som initieras med exempelvis `int x = 5` finns i de flesta programmeringsspråk. En pekare till `x`, som initialiseras enligt `int* ptr_x = &x`, innehåller istället minnesadressen till variabeln `x`. En utskrift av `x` i terminalen skulle visa siffran fem, medan en utskrift av `ptr_x` skulle visa den hexadecimala minnesadress där variabeln `x` lagras. En pekare innehåller alltså information om exakt var en variabel lagras i minnet.

[a] https://github.com/JosephLofving/kandidat/

På liknande sätt kan en pekare till ett fält skapas. Betrakta ett fält som initieras enligt `int y[3] = {4,5,6}`. En pekare till `y` kan då skapas med `int* ptr_y = &y[0]`, så att pekaren innehåller minnesadressen för fältets första element (notera att indexeringen börjar på 0 i C++). Eftersom minnesadresserna för varje element i ett fält placeras i följd räcker det med minnesadressen till det första elementet samt fältstorleken för att fältet skall kunna nås enbart genom pekare. De här egenskaperna är inbyggda i C++ och element i fältet kan enkelt fås direkt genom pekaren; utskrift av `ptr_y[2]` visar exakt samma som utskrift av `y[2]`, det vill säga siffran sex.

För att lagra matriser som endimensionella fält, som kan nås via pekare, kan matriselementen lagras antingen radvis eller kolonnvis; biblioteket cuBLAS (se avsnitt 4.1.5) lagrar matriser i fält kolonnvis. Elementen sparas då kolonn efter kolonn, vilket gör att exempelvis en 3×3-matris bildar ett endimensionellt fält enligt

$$\begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} \implies \begin{bmatrix} a & b & c & d & e & f & g & h & i \end{bmatrix}. \tag{4.1}$$

För en `m`×`n`-matris, där `m` är antalet rader och `n` är antalet kolonner, ges matriselementet med index `[i, j]` då av index `[i + j*m]` i fältet. För att lagra ett tredimensionellt objekt som ett endimensionellt fält nås elementet med index `[i, j, k]` i objektet genom index `[i + j*m + k*m*n]` i fältet.

En viktig egenskap för pekare är hur de hanteras som funktionsargument. När variabler används som argument kan det ses som att de förändras inom funktionen, men återställs när funktionen avslutas. Om en pekare används som argument återställs dock inte dess innehåll, utan behåller förändringen. Det beror på att pekaren är densamma men innehållet vid minnesadressen har ändrats. Den här egenskapen utnyttjas flitigt i GPU-programmet, eftersom vissa GPU-funktioner inte kan returnera variabler.

4.1.2 Kernels och devicefunktioner

Ett CUDA-program består i grunden av CPU-kod som anropar funktioner som utförs parallellt på en GPU. Sådana funktioner kallas kernels och deklareras i källkoden med tillägget `__global__` ovanför den vanliga funktionsdeklarationen. Ordet ”global” syftar på att funktionen kan kallas från både CPU och GPU. Väl inne i en kernel kan även andra GPU-funktioner anropas, vilka kallas devicefunktioner. Dessa deklareras med tillägget `__device__` ovanför den vanliga funktionsdeklarationen.

Kernels kan bara läsa från GPU-minnet och returnerar alltid `void`, det vill säga inga variabler alls, varför pekare ofta används som funktionsargument. Devicefunktioner kan returnera i princip alla typer, med begränsningen att de endast kan anropas från kernels. Innan en kernel anropas måste alla funktionsargument existera i deviceminnet.

4.1.3 Minneshantering

En GPU har en egen minnesenhet (”deviceminne”) som är avskild från det vanliga minnet en CPU annars arbetar med (”hostminne”). Hanteringen av deviceminnet sker främst i tre steg: allokering, kopiering från och till hostminnet, samt frigöring av färdiganvänt minne. För att sammanfatta processen ges ett illustrativt exempel i kodavsnittet nedan.

```
1 /* Den här exempelkoden belyser stegen för minneshantering vid GPU-användning */
2 int N = 100;
3 double* x_h = new double[N];
```

```
4  ... // x_h manipuleras
5  double* x_d;
6  cudaMalloc((void**)&x_d, N*sizeof(double));
7  cudaMemcpy(x_d, x_h, N*sizeof(double), cudaMemcpyHostToDevice);
8  ... // Kernels som ändrar x_d anropas
9  cudaDeviceSynchronize();
10 cudaMemcpy(x_h, x_d, N*sizeof(double), cudaMemcpyDeviceToHost);
11 ... // x_h används
12 delete[] x_h;
13 cudaFree(x_d);
```

Först allokeras hostminne med operatorn `new`, där hostvariablerna ofta namnges med ändelsen `_h`. Således deklareras en tom variabel, ofta en pekare till ett tomt fält av given längd. Sedan deklareras pekare till devicevariablerna, som ofta namnges med ändelsen `_d`. Därefter allokeras deviceminne med `cudaMalloc`, som tar som argument en pekare samt önskad storlek på det allokerade minnet. Pekaren kommer då att peka på det allokerade deviceminnet.

Nästa steg är att kopiera över variablerna till deviceminnet så att de är tillgängliga för kernels. Kopieringen sker med funktionen `cudaMemcpy`, som kräver en pekare till hostvariabeln, en pekare till devicevariabeln, storleken av minnet som kopieras samt ett argument som anger huruvida kopieringen sker från host- till deviceminne eller vice versa. I det här stadiet används argumentet `cudaMemcpyHostToDevice`.

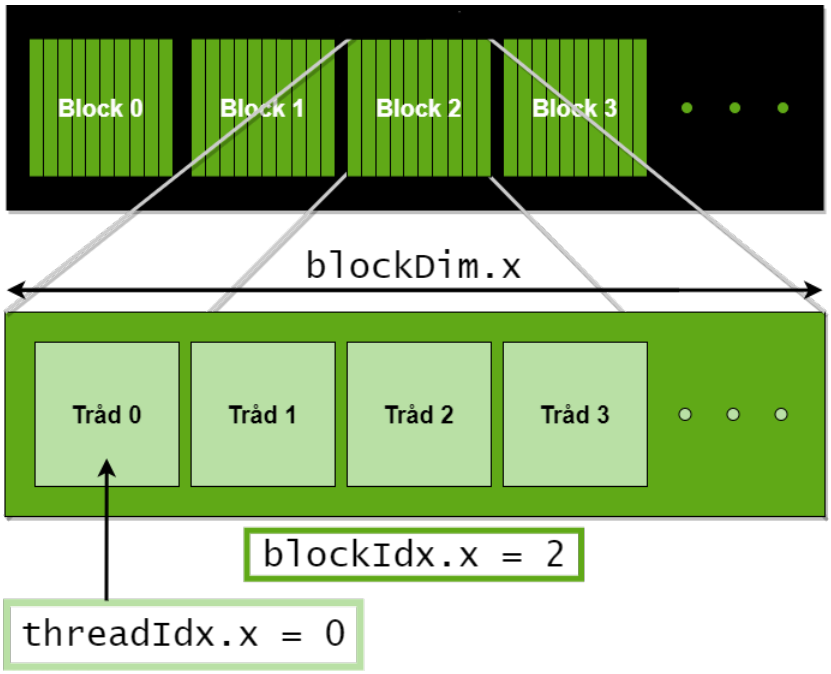
Därefter kan hostkoden anropa kernels med devicevariabler som argument. När alla kernels har exekverats bör alla parallella processer på GPU:n inväntas med `cudaDeviceSynchronize`. Därefter kan devicevariablernas innehåll nås från CPU:n genom att kopiera dem från deviceminne till hostminne, vilket görs genom att använda argumentet `cudaMemcpyDeviceToHost` i `cudaMemcpy`.

Avslutningsvis behöver såväl host- som deviceminne frigöras med `delete` respektive `cudaFree` när variablerna är färdiganvända för att undvika minnesläckor.

4.1.4 Att anropa en kernel

På en GPU kan beräkningar utföras parallellt med ett stort antal så kallade trådar, där varje tråd utför sekventiella beräkningar. På en CPU utförs beräkningar främst sekventiellt, exempelvis genom att fylla element efter element av en vektor med en `for`-loop, eller på ett mycket färre antal trådar. Parallellisering införs då beräkningarna delas upp på flera trådar som kan exekveras samtidigt, och vanligtvis oberoende av varandra. Då kan exempelvis flera vektorelement fyllas samtidigt om beräkningen för varje element sker på varsin tråd. I CUDA-programmering finns tre mjukvarunivåer som motsvarar uppdelningen i hårdvaran: Trådar fördelas i grupper om 32 som kan exekveras samtidigt på en CUDA-kärna, eller strömningsprocessor. Fördelningen av sådana grupper till CUDA-kärnor sker genom en indelning i så kallade block, som hanteras av strömningsmultiprocessorer. Blocken delas i sin tur in i så kallade grid, som hanteras på GPU:n.

I CUDA finns inbyggda variabler som ger information om bland annat trådindex, blockindex samt antalet trådar i varje block (blockdimensionen). Varje variabel finns i de tre dimensionerna `x`, `y` och `z`. I en kernel kan alltså variabeln `threadIdx.x` ge information om i vilken tråd kerneln exekveras i `x`-dimensionen. På samma sätt innehåller `blockIdx.x` i vilket block kerneln exekveras, och `blockDim.x` antalet trådar i blockens `x`-dimension. I figur 4.1 visas uppdelningen av grid, block och trådar samt de inbyggda variablerna schematiskt i en dimension.



Figur 4.1: Parallellisering delas upp i tre nivåer hårdvarumässigt, och behandlas därför i tre motsvarande nivåer programmeringsmässigt: Ett grid innehåller flera block (överst), och ett block innehåller flera trådar (nederst). I figuren visas `x`-dimensionen för grid, block och trådar samt hur de indexeras med CUDA:s inbyggda variabler `threadIdx.x` och `blockIdx.x`. Variabeln `blockDim.x` anger hur många trådar varje block innehåller.

I kodutdraget nedan ges ett exempel på hur en kernel som fyller ett fält kan anropas och exekveras. För att kalla på en kernel behövs information om antalet block per grid samt antalet trådar per block, vilka ges som en typ av funktionsargument på rad 22.

```
1  /* Här ges ett exempel på hur en kernel anropas och exekveras */
2  __global__
3  void fillArray(double* a, int N) {
4      /* Bestäm fältindex utifrån nuvarande tråd och block */
5      int index = blockIdx.x*blockDim.x + threadIdx.x;
6      /* Säkerställ att indexen inte överskrider fältstorleken */
7      if (index < N) {
8          a[index] = 1;
9      }
10 }
11
12 int main() {
13     N=10000;
14     ...
15     double* a_d;
16     cudaMalloc((void**)&a_d, N*sizeof(double));
17     /* Sätt upp 1024 trådar per block */
18     threadsPerBlock = 1024;
19     /* Sätt upp N/1024 block per grid och avrunda uppåt, dvs 10 block per grid */
20     blocksPerGrid = ceil(double(N) / double(threadsPerBlock);
21     /* Fyll hela fältet med ettor */
```

```
22      fillArray<<<blocksPerGrid, threadsPerBlock>>>(a_d, N);
23      ...
24  }
```

Kerneln `fillArray` i exemplet ovan fyller 10 000 element i ett fält `a` med talet 1. I ett vanligt CPU-program skulle det ske med en `for`-loop, men i exempelkoden innebär anropet av kerneln att funktionen exekveras en gång per tråd. Så många block som hårdvaran har kapacitet för utförs samtidigt, där varje block exekverar 1024 trådar parallellt. Funktionen loopar automatiskt över blocken tills alla trådar exekverats. Det enda som krävs är att begränsa antalet trådar som exekveras så att de inte överstiger fältets storlek, vilket möjliggörs med variabeln `index` som initialiseras på rad 5. Eftersom antalet trådar per block är 1024 är `blockDim.x` = 1024. Om kerneln befinner sig på tråd 600 i block 5 är `index` $= 5 \cdot 1024 + 600 = 5720$, vilket är mindre än 10 000. Med `if`-satsen på rad 7–9 kan därmed `a[index]` sättas till 1. Skulle däremot `index` vara större än 10 000 ignoreras rad 8–10 tills alla de tio blocken är exekverade. Notera att en `if`-sats i en kernel inte är tidskrävande, till skillnad från `if`-satser i stora `for`-loopar, eftersom varje tråd bara utför den en gång.

4.1.5 cuBLAS

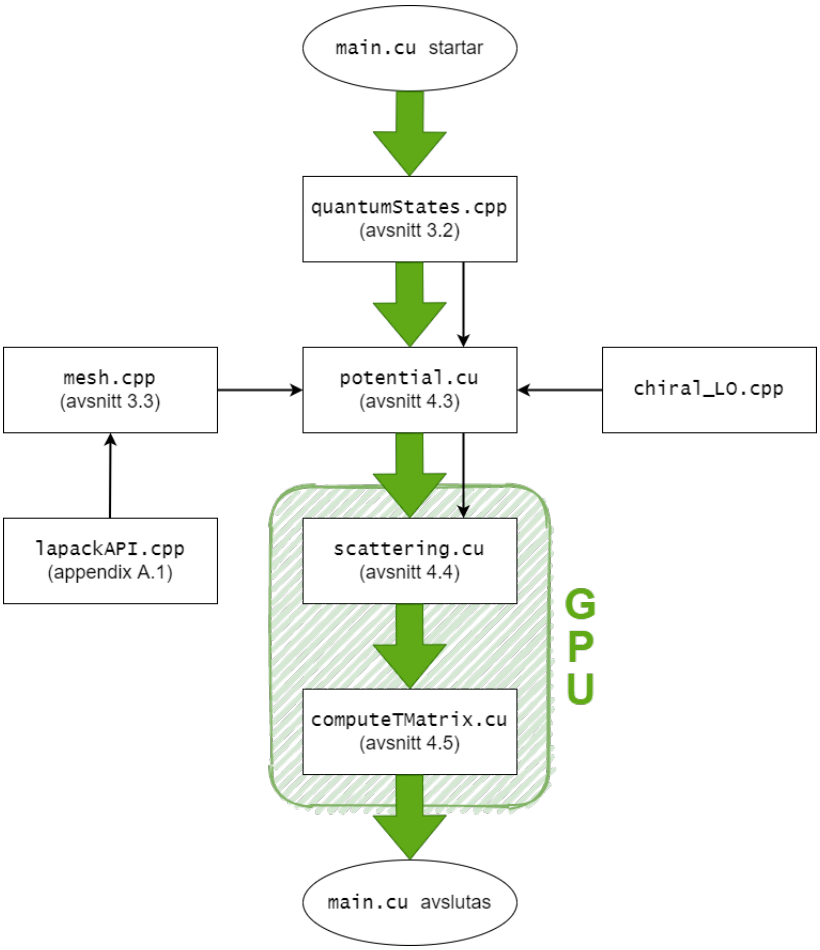
Namnet cuBLAS står för ”CUDA Basic Linear Algebra Subroutine library” och är en GPU-implementation av BLAS i CUDA-gränssnittet [13]. Som namnet antyder används detta bibliotek för linjär algebra. I GPU-programmet används cuBLAS som en parallelliserad motsvarighet till LAPACK och BLAS.

De funktioner som tillhandahålls av cuBLAS hanterar allokeringen av trådar och block internt, men kräver – likt egenskrivna CUDA-kernels – att användaren själv allokerar deviceminne på samma sätt som beskrivs i avsnitt 4.1.3. Vidare kräver alla cuBLAS-funktioner ett så kallat ”handle” som parameter. Detta objekt innehåller bland annat information om GPU:n som används och behöver initialiseras av användaren själv. Att objektet behöver initialiseras av användaren och användas som argument för varje cuBLAS-funktion ger större kontroll över hur cuBLAS används i system med flera GPU:er.

4.2 Överblick av GPU-programmets upplägg

Utvecklingen av GPU-programmet baserades till stor del på CPU-programmet från kapitel 3, men behövde anpassas för att kunna beräkna T-matrisen för många olika potentialer V och energier T_{labb}. Med ekvation (2.1) inses att en förändring av T_{labb} leder till en förändring av k_0, vilket innebär att funktionerna `setupDVector` och `computePhaseShifts` (se avsnitt 3.5) kan parallelliseras. Vidare påverkar en förändring av V funktionerna `setupVDKernel`, `computeTMatrix` och `computePhaseShifts` (avsnitt 3.5).

Variation av T_{labb} och V påverkar alltså majoriteten av alla funktioner i de två filerna `potential.cpp` (avsnitt 3.4) och `scattering.cpp`, varför dessa filer byttes ut mot de CUDA-kompatibla filerna `potential.cu` och `scattering.cu` som beskrivs i detalj i avsnitt 4.3 respektive 4.4. Notera dock att filerna `quantumStates.cpp` och `mesh.cpp` inte påverkas av variationer av T_{labb} och V, varför dessa filer behölls från CPU-programmet. En omstrukturering i GPU-programmet ledde också till att filen `scattering.cu` inte längre anropade `quantumStates.cpp`. I figur 4.2 syns en sammanfattning av programmets upplägg enligt samma princip som i figur 3.1, med tillägget av ett skuggat område som markerar vilka filer som körs på GPU:n.



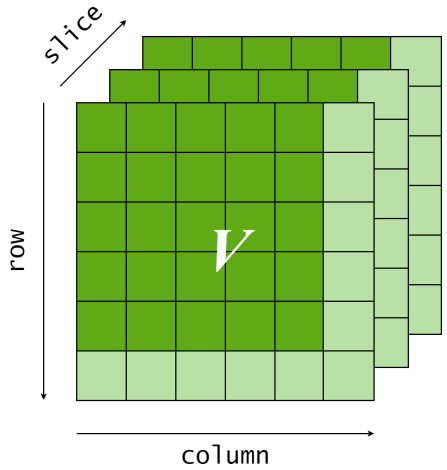
Figur 4.2: En grafisk representation av GPU-programmets upplägg, med samma notation som i figur 3.1. Den skuggade delen av diagrammet uppmärksammar att funktionerna i dessa filer utförs på GPU:n. Till skillnad från CPU-programmet finns en ny fil `computeTMatrix.cu`, och den omarbetade filen `scattering.cu` beror inte på vare sig `lapackAPI.cpp` eller `quantumStates.cpp`. Filerna `mesh.cpp`, `lapackAPI.cpp`, `quantumStates.cpp` och `chiral_LO.cpp` är oförändrade från CPU-programmet.

En väsentlig skillnad från CPU-programmet är att CUDA domineras av användning av pekare (avsnitt 4.1.1). Direkt minneshantering är dessutom nödvändig eftersom överföringar mellan host- och deviceminne måste hanteras manuellt (avsnitt 4.1.3). Vidare skapades filen `computeTMatrix.cu` som genomför matrisekvationslösning med cuBLAS-biblioteket, vilket ersatte de motsvarande funktionerna i `lapackAPI.cpp`. Filen `computeTMatrix.cu` beskrivs i detalj i avsnitt 4.5. För att fokusera på effektiviseringen i GPU-programmet gjordes valet att begränsa beräkningarna till $^1S_0$-kanalen, som är okopplad.

4.3 `potential.cu` – Omstrukturering av potentialfunktionen

Filen `potential.cu` körs fortfarande på CPU:n, eftersom den centrala koden i `chiral_LO.cpp` inte är GPU-anpassad. Den största skillnaden från CPU-programmet (se avsnitt 3.4) är att flera spridningsenergier nu behandlas samtidigt, vilket innebär att den tidigare tvådimensionella matrisen `VMatrix` nu har har en tredje dimension, `slice`, för spridningsenergin. Dessutom har stödet för kopplade kanaler tagits bort.

Elementen $V(k_i, k_j)$ i potentialmatrisen, där k_i och k_j är kvadraturpunkterna, exklusive rörelsemängden k_0, förändras inte för olika spridningsenergier T_{labb}. Denna del av matrisen, som representeras av de mörka elementen i figur 4.3, beräknas därför bara en gång för att sedan kopieras för varje `slice`. De resterande elementen, $V(k_0, k_j)$ samt $V(k_i, k_0)$, förändras däremot för varje T_{labb} eftersom k_0 beror på spridningsenergin. Dessa element finns i de sista raderna och kolonnerna i `VMatrix` och representeras av de ljusa rutorna i figur 4.3. De behöver beräknas för varje spridningsenergi T_{labb}.



Figur 4.3: Schematisk bild av potentialmatrisen. De sista raderna och kolonnerna, ljusa i figuren, motsvarar de element som är beroende av T_{labb}. De mörka beror inte av T_{labb} och behöver därför bara beräknas en gång. Dimensionerna `row`, `column` och `slice` definieras i figuren.

4.4 `scattering.cu` – Implementering av kernels

Samtliga funktioner i `scattering.cu` ersattes med kernels och devicefunktioner (se avsnitt 4.1.2). I `main.cu` används `scattering.cu` först då kerneln `setupDVector` anropas. De centrala delarna i kodutdraget nedan är att `if`-satser används med block och trådar istället för en `for`-loop, vilket beror på att loopar sker automatiskt med parallellisering. Principerna från avsnitt 4.1.4 gäller för hela GPU-programmet, utvidgat till de tre dimensionerna `row`, `column` och `slice` som syns i figur 4.3. Intressanta delar av koden nedan är alltså raderna 4–9. Fältet `D` har en dimension för vanliga vektorelement (`column` i koden) och en för olika energier T_{labb} (`slice` i koden). I GPU-programmet är alltså D ett tvådimensionellt objekt och indexeras enligt principen i avsnitt 4.1.1, vilket syns på bland annat rad 10 nedan.

```
1  __global__
2  void setupDVector(cuDoubleComplex* D, ...) {
3    ...
4    /* Inför index för att kontrollera hur D fylls */
5    int column = blockIdx.x * blockDim.x + threadIdx.x;
6    int slice = blockIdx.z * blockDim.z + threadIdx.z;
7
8    /* Bestäm D med ekvation (2.24) */
9    if (column < quadratureN && slice < TLabLength) {
10     D[column + slice * matLength] = make_cuDoubleComplex(twoOverPi * twoMu *
```

```
11          k[column] * k[column] * w[column] / (k0[slice] * k0[slice] -
12          k[column] * k[column]), 0);
13      /* Bestäm sista elementet av D */
14      D[quadratureN + slice * matLength] = make_cuDoubleComplex(-twoOverPi *
15          twoMu * k0[slice] * k0[slice] * sum[slice], -twoMu * k0[slice]);
16      }
17    }
```

På rad 15 syns en variabel `sum`, som är värdet av summan i ekvation (2.24). I CPU-programmet beräknades summan i `setupDVector`; i GPU-programmet beräknas summan istället parallelliserat i en egen kernel, och ges som argument till funktionen.

För att använda komplexa tal i CUDA används typen `cuDoubleComplex` eftersom typen `std::complex<double>` inte är tillgängligt på GPU:n. Det innebär bland annat att nya komplexa tal skapas med `make_cuDoubleComplex`, som syns på rad 10 och 14 ovan, samt att addition och multiplikation av komplexa tal sker med funktionerna `cuCadd` respektive `cuCmul`.

Med `D`-fältet kan VD-matrisen skapas i kerneln `setupVDKernel`. Kerneln har två skillnader från CPU-motsvarigheten: `for`-loopen har bytts ut mot en `if`-sats som hanterar block och trådar, och F-matrisen initialiseras här istället för i CPU-funktionen `computeTMatrix`. Det senare beror på att F-matrisen enklare kan initialiseras i samma kernel som VD när de båda är pekare, samt att motsvarigheten av CPU-funktionen `computeTMatrix` ligger i den nya filen `computeTMatrix.cu` istället för i `scattering.cu` (se avsnitt 4.5). Både VD och F är tredimensionella objekt i GPU-programmet, och på bland annat rad 11 och 16 nedan ges exempel på den indexering som introducerades i avsnitt 4.1.1.

```
1   __global__
2   void setupVDKernel(cuDoubleComplex* VD, ...) {
3     ...
4     /* Inför index för att kontrollera hur VD fylls */
5     int row = blockIdx.y * blockDim.y + threadIdx.y;
6     int column = blockIdx.x * blockDim.x + threadIdx.x;
7     int slice = blockIdx.z * blockDim.z + threadIdx.z;
8
9     if (row < matLength && column < matLength && slice < TLabLength) {
10      /* Beräkna VD */
11      VD[row + column * matLength + slice * matLength * matLength] =
12              cuCmul(V[row + column * matLength + slice * matLength * matLength],
13              D[column + slice * matLength]);
14      /* Bestäm diagonalelementet för F enskilt, med ekvation (2.27) */
15      if (row == column) {
16        F[row + row * matLength + slice * matLength * matLength] = cuCadd(
17          make_cuDoubleComplex(1, 0), cuCmul(make_cuDoubleComplex(-1, 0),
18          VD[row + row * matLength + slice * matLength * matLength]));
19      }
20      /* Bestäm resten av F med med (2.27) */
21      else {
22        F[row + column * matLength + slice * matLength * matLength] =
23          cuCmul(make_cuDoubleComplex(-1, 0), VD[row + column * matLength +
24          slice * matLength * matLength]);
25      }
```

```
  }
  ...
}
```

Variabeln `matLength` har värdet $N+1$ och används eftersom alla ingående matriser har storlek $(N+1)\times(N+1)$, och alla vektorer längd $N+1$. Variabeln `TLabLength` anger antalet energier som funktionen beräknas för.

Både `setupDVector` och `setupVDKernel` kallas direkt från `main.cu`. Deras utdata skickas sedan till `computeTMatrix.cu`. När Lippmann-Schwingerekvationen är löst för T-matrisen kan sedan fasskiftet beräknas genom att skicka T-matrisen till kerneln `computePhaseShifts` i `scattering.cu`. Även här har koden parallelliserats.

```
__global__
void computePhaseShifts(...) {
    ...
  int slice = blockIdx.x * blockDim.x + threadIdx.x;
  if (slice < TLabLength) {
    rhoT[slice] = 2 * mu * k0[slice];
    const cuDoubleComplex I = make_cuDoubleComplex(0.0, 1.0);

    cuDoubleComplex T0 = (T[(quadratureN)+(quadratureN * matLength) +
        slice * matLength * matLength ]);
    argument[slice] = make_cuDoubleComplex(1,0) - 2.0 * I * rhoT[slice] * T0;
    phases[slice] = -0.5 * I * constants::rad2deg *
          logCudaComplex(argument[slice]);

  }
}
```

4.5 `computeTMatrix.cu` – Lösning av matrisekvationen

Filen `computeTMatrix.cu` innehåller funktionen `computeTMatrixCUBLAS` som löser den centrala matrisekvationen, se ekvation (2.26). Lösningen genomförs parallellt för många matriser samtidigt och görs i två steg: först LU-faktoriseras F-matriserna med pivotering, och sedan används denna faktorisering för att beräkna T-matriserna genom att lösa ekvationssystemen. Denna tvåstegsmetod ger generellt sett snabbare och mer numeriskt stabila beräkningar än vanlig matrisinversion [14].

De två stegen genomförs med hjälp av separata funktioner från cuBLAS-biblioteket: `cublasZgetrfBatched` och `cublasZgetrsBatched`, vilka beskrivs i mer detalj i appendix A.2. Nedan följer koden för detta, men med felhantering och minneshantering dold för läsbarhetens skull.

```
void computeTMatrixCUBLAS(cuDoubleComplex* T_d, cuDoubleComplex* F_d, int matLength,
                    int TLabLength, cublasStatus_t status, cublasHandle_t handle) {
    ...
  // Skapa pekarfält för matriser
  for (int i = 0; i < TLabLength; i++) {
    Fptr_array_h[i] = F_d + (i * matLength * matLength);
```

```
7       Tptr_array_h[i] = T_d + (i * matLength);
8     }
9       ...
10    // Genomför LU-faktorisering
11    status = cublasZgetrfBatched(handle, matLength, Fptr_array_d, matLength, pivotArray_d,
12                  trfInfo_d, TLabLength);
13
14    // Beräkna T-matrisen
15    status = cublasZgetrsBatched(handle, CUBLAS_OP_N, matLength, 1, Fptr_array_d,
16                  matLength, pivotArray_d, Tptr_array_d, matLength, &trsInfo_d,
17                  TLabLength);
18      ...
19  }
```

Lägg särskilt märke till pekarfälten, såsom `Fptr_array_h`. Dessa fält innehåller pekare till de fält som innehåller de relevanta matriserna. Lägg vidare märke till att T-matrisen, `T_d` i koden, dyker upp som ett argument till funktionen. Innan funktionen genomförs innehåller `T_d` V-matrisen i ekvation (2.26) vilken cuBLAS sedan fyller med resultatet från beräkningen. Dessa två matriser har samma dimension och biblioteket undviker att allokera minne för extra matriser i onödan med denna metod.

5

Resultat av tidsprofilering och förslag till vidareutveckling

I det här kapitlet redovisas och analyseras tidseffektiviteten av vårt GPU-program. De uppmätta körtiderna för GPU-programmet presenteras i avsnitt 5.1, och en jämförelse med ett referensprogram [15] i avsnitt 5.2. Referensprogrammet är ett väloptimerat program vars kod ej har varit tillgänglig för oss.[a] Slutligen presenteras förslag för vidareutveckling av koden i avsnitt 5.3.

5.1 Tidsprofilering av GPU-programmet

För att skapa en tidsprofil av GPU-programmet användes C++-funktionen `chrono::high_resolution_clock` för att mäta körtiden för olika delar av programmet med millisekundsprecision. Programmet kördes upprepade gånger, och fasskift beräknades för upp emot tusentals energier i $^1S_0$-kanalen; i en mer vetenskapligt användbar tillämpning av programmet skulle parametrarna i potentialen varieras för att täcka in ett helt parameterrum. Då skulle ett stort antal fasskift beräknas vid hundratals olika spridningsenergier och för flera olika kanaler. Variationen i spridningsenergier ansågs dock räcka för att bedöma GPU-programmets hastighet.

Tabell 5.1 visar tidsprofiler från mätningar där 700 energier i intervallet 1 MeV–300 MeV evaluerades med 40 och 80 kvadraturpunkter; enligt analysen i avsnitt 3.7 ger 40 kvadraturpunkter en felmarginal på maximalt 10^{-3}, vilket ansågs tillräckligt. Mätningen där antalet kvadraturpunkter sattes till 80 gjordes för att uppskatta körtiden för kopplade kanaler, se avsnitt 3.4. Mätningarna kördes 1000 gånger var och mätdatan användes för att skapa en normalfördelning för körtiden för programmets delar med MATLAB-funktionen `fitdist`. I tabellen visas medelvärdet μ samt standardavvikelsen σ i mikrosekunder.

I tabellen syns det att körtiden klart domineras av funktionen `potential`. Denna funktion bygger dock på en ooptimerad betakod för populeringen av potentialmatrisen då den optimeringen ligger utanför projektets omfattning. Vidare inkluderar de uppmätta körtiderna för referensprogrammet inte skapandet av potentialmatrisen. För att få en värdefull jämförelse med referensprogrammet utesluts därmed körtiden för `potential` från jämförelserna i avsnitt 5.2. I en mer vetenskapligt intressant användning av GPU-programmet väntas potentialkoden ersättas med en mer optimerad metod.

Vidare är det tydligt att de två näst mest tidskrävande funktionerna – `cublasCreate` och `cudaFree(0)` – har konstant körtid oavsett antal kvadraturpunkter. Funktionen `cublasCreate` är en nödvändig initialisering inför användningen av cuBLAS-funktioner, medan det effektlösa funktionsanropet `cudaFree(0)` används för att initialisera CUDA utan att påverka tidsprofileringen för andra funktioner. Dessa två funktioner behöver bara anropas en gång vardera per körning och har alltid ungefär samma körtid. När antalet fasskift som beräknas ökar har dessa två så kallade ”overheadfunktioner” en obetydlig

[a] Mätdatan kommer från vår handledare Andreas Ekström.

Tabell 5.1: Körtidens medelvärde μ och standardavvikelse σ för olika delar av koden. Värdena beräknades med $N = 40$ respektive $N = 80$ för 700 energier efter 1000 körningar.

Del av kod/funktion	$N = 40$ Tidsandel	$N = 80$ Tidsandel	$N = 40$ $\mu \pm \sigma$ [µs]	$N = 80$ $\mu \pm \sigma$ [µs]
`potential`	64,81 %	77,03 %	416 220 ± 8699,13	881 091 ± 8740,85
`cublasCreate`	22,30 %	12,71 %	143 220 ± 2778,02	145 405 ± 2276,78
`cudaFree(0)`	11,02 %	6,53 %	70 757 ± 6954,1	74 654 ± 3058,1
`computeTMatrixCUBLAS`	1,06 %	2,35 %	6812 ± 233,7	26 824 ± 378,31
Minnesfrigöring	0,30 %	0,63 %	1929 ± 113,7	7159 ± 107,1
`gaussLegendreInfMesh`	0,08 %	0,13 %	496 ± 16,1	1489 ± 37,41
`cudaMalloc`	0,04 %	0,03 %	257 ± 23,7	351 ± 22,0
`cudaMemcpy`	<0,01 %	<0,01 %	38 ± 42	47 ± 38
`setupDVectorSum`	<0,01 %	<0,01 %	16 ± 2,2	16 ± 2,6
Minnesallokering CPU	<0,01 %	<0,01 %	11 ± 1,4	12 ± 1,3
`getK0`	<0,01 %	<0,01 %	9 ± 1	9 ± 2
`computePhaseShifts`	<0,01 %	<0,01 %	4 ± 1	5 ± 1
`setupDVector`	<0,01 %	<0,01 %	3 ± 0,5	3 ± 0,5
`setupVDKernel`	<0,01 %	<0,01 %	2 ± 0,3	2 ± 0,3
Totalt	100 %	100 %	642 220 ± 12 801,2	1 143 880 ± 11 844,4

körtid i jämförelse med resten av programmet. På grund av detta exkluderas även dessa från jämförelserna i avsnitt 5.2.

Slutligen kan det noteras att körtiden för `computeTMatrixCUBLAS` ökar med en faktor 3,9 när antalet kvadraturpunkter fördubblas. Den överlägset mest tidskrävande delen av funktionen är LU-faktoriseringen (se avsnitt 4.5), som teoretiskt har en komplexitet på $2N^3/3$ för stora N [14], vilket vid en fördubbling av N medför en ökning på 5,3. Att det teoretiska värdet är såpass mycket högre än det uppmätta beror sannolikt på att N inte är stort nog att uppnå sin komplexitetsgräns.

5.2 Jämförelse med referensprogram på CPU

För att få ett mått på GPU-programmets prestanda jämförs dess körtid med referensprogrammet för olika antal fasskiftsberäkningar. Jämförelsen presenteras i tabell 5.2. I tabellen redovisas medelvärden av körtiden utifrån 100 körningar för olika antal fasskiftsberäkningar för GPU-koden och 20 körningar för referenskoden. De medelvärden som redovisas är för GPU-koden med och utan overheadfunktionerna samt för referenskoden. Liknande analys gjordes för ökande antal kvadraturpunkter. Denna aspekt bedömdes inte lika intressant då det finns en gräns där ökad precision inte är nödvändig och där ett ökat antal kvadraturpunkter påverkar körtiden för mycket, se avsnitt 3.7.

Jämförelsen av tiderna mellan GPU- och CPU-programmen är inte helt rättvis då de genomfördes på olika datorer (se tabell 5.3). Det är dock svårt att göra jämförelsen ob-

Tabell 5.2: Medelvärden av körtiden för GPU-programmet och referensprogrammet efter 100 respektive 20 körningar för olika antal fasskift. Antalet kvadraturpunkter, N, var 40 i alla mätningar. GPU-programmets körtid redovisas både med och utan overhead. Vi ser att GPU-koden blir snabbare än CPU-koden redan vid åtta fasskiftsberäkningar när `potential` och overhead exkluderas, medan GPU-koden blir snabbare för 2048 beräkningar med overhead medräknat.

Antal fasskift	GPU med overhead: μ [ms]	GPU: μ [ms]	CPU: μ [ms]
4	206,5	2,435	1,525
8	206,7	2,460	2,780
16	205,4	2,512	3,915
32	207,0	2,784	13,42
64	205,4	3,222	19,19
128	209,5	4,255	23,91
256	210,1	5,921	47,16
512	217,0	9,270	102,7
1024	224,7	16,64	174,4
2048	238,7	30,99	326,9
4096	267,2	57,58	626,7

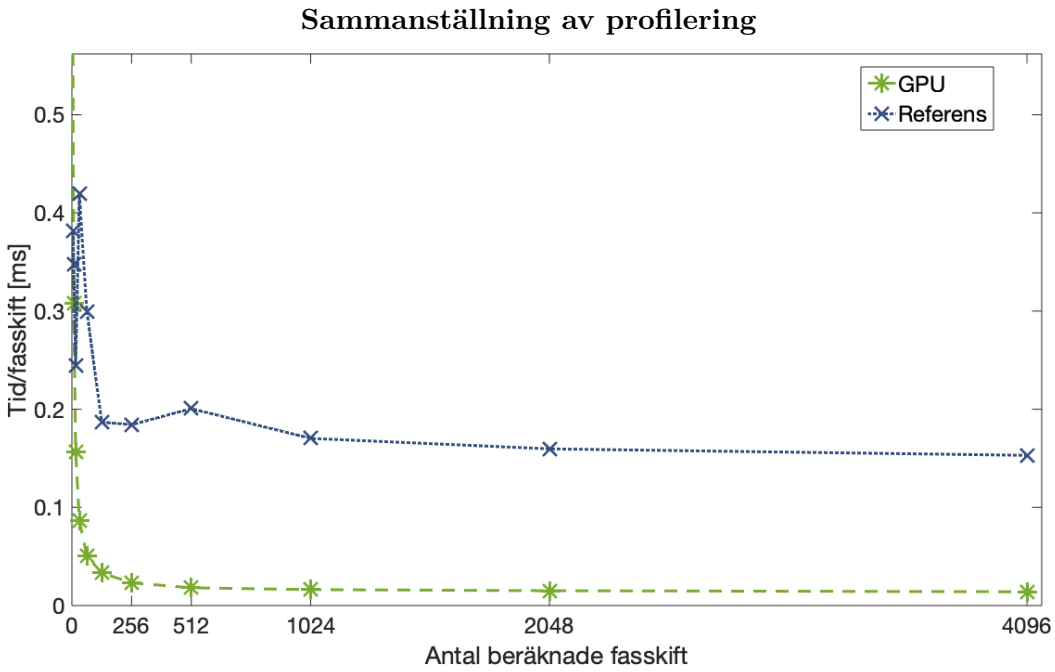
jektivt rättvis ur den aspekten eftersom huvuddelen av programmen kör på fundamentalt olika hårdvarukomponenter; det finns ingen klar GPU-motsvarighet till en given CPU.

Tabell 5.3: Specifikationer för datorer där CPU- och GPU-kod har evaluerats. GPU:n för referensprogrammet är irrelevant då programmet är helt CPU-baserat.

Komponent	GPU-program	Referensprogram
CPU	AMD Ryzen 3600X 6-Core Processor @ 3.80 GHz	Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60 GHz
GPU	MSI GeForce GTX 1070 8GB Founders Edition	Irrelevant

I figur 5.1 visas en graf över beräkningstid per fasskift för olika antal fasskift för GPU-programmet samt för referensprogrammet. Enligt grafen verkar körtiden per beräknat fasskift konvergera för GPU-koden redan vid 2048 fasskiftsberäkningar; värdet vid 2048 är 0,015 ms/fasskift vilket är relativt nära värdet vid det högsta antalet energier som mättes, 32 768 som är 0,010 ms/fasskift[b]. Referensprogrammet har vid 4096 energier ett värde på 0,153 ms/fasskift, alltså runt tio gånger långsammare än GPU-koden. Kodens tänkta tillämpningsområde innefattar ett väldigt stort antal fasskiftsberäkningar, eventuellt så många som 10^9 fasskift åt gången. Därav förväntas tiden per fasskift alltid vara inom det konvergerade området.

[b]Mätdata samlades in för högre antal fasskiftsberäkningar än vad som redovisas i figur 5.1 och tabell 5.2. Dessa exkluderades då motsvarande data inte fanns för referensprogrammet.



Figur 5.1: En sammanställning av mätningar där antal beräknade fasskift successivt ökades. På y-axeln visas den totala körtiden dividerat på antal beräknade fasskift och på x-axeln visas antalet beräknade fasskift. Den streckade linjen med stjärnor motsvarar GPU-programmet och den prickade linjen med kryss motsvarar referensprogrammet. Antalet kvadraturpunkter var 40.

Sammanfattningsvis anses resultaten av de sammanställda körtiderna och jämförelsen med referensprogrammet tyda på att ett parallellt GPU-program är snabbare vid beräkningar av ett stort antal fasskift. Följaktligen tros ett GPU-accelererat program vara värdefullt för vidare forskning om den starka kärnkraften. För att få ut det mesta av denna typ av parallell GPU-kod krävs dock att potentialmatriserna genereras effektivt.

5.3 Förslag till vidareutveckling

Vi kan se i tabell 5.1 att GPU-programmets största flaskhals är funktionen `potential`. För att kunna utnyttja GPU-acceleration fullt ut behövs snabba, effektiva metoder för att generera många användbara potentialmatriser, förslagsvis med en parallelliserad lösning.

Det finns också utrymme att beräkna observabler i GPU-programmet. Programmets utdata är fasskift, vilket i sig inte är en fysikalisk observabel. Syftet med fasskift är istället att parametrisera spridningsamplituder. För att beräkna observabler i form av spridningstvärsnitt krävs en implementering av fasskiftsberäkning för kopplade kanaler. Implementeringen av dessa beräkningar låg utanför det här projektets omfattning och därför gavs endast en liten överblick av tvärsnittsberäkning i avsnitt 2.6.1. En möjlig framtida vidareutveckling av programmet är därför att implementera tvärsnittsberäkning, så att programmet kommer närmare en fullständig simulering av spridningsexperiment.

Slutligen skulle en tidsprofil skapad med bättre hårdvara vara av intresse, då denna typ av kod förväntas köra på kraftfullare beräkningskluster snarare än personliga datorer. Ett par mätningar utfördes på beräkningsklustret Vera [16], med grafikkortet Tesla V100 [17]. Kötiderna var dock för långa för att hinna testa den slutgiltiga versionen av koden. För att kunna utnyttja kraftfullare datorer maximalt lär hårdvaruspecifik optimering krävas.

Referenser

[1] R. Machleidt och D. R. Entem, "Chiral effective field theory and nuclear forces," *Physics Reports-Review Section Of Physics Letters*, volym 503, nr 1, s. 1–70, 2010. DOI: https://doi.org/10.1016/j.physrep.2011.02.001.

[2] B. A. Lippmann och J. Schwinger, "Variational Principles for Scattering Processes. I," *Phys. Rev.*, volym 79, s. 469–480, 3 aug. 1950. DOI: 10.1103/PhysRev.79.469.

[3] M. I. Haftel och F. Tabakin, "Nuclear saturation and the smoothness of nucleon-nucleon potentials," *Nuclear Physics A*, volym 158, s. 1–42, juli 1970. DOI: https://doi.org/10.1016/0375-9474(70)90047-3.

[4] *CUDA-Tools*, https://developer.nvidia.com/cuda-toolkit, Hämtad: 2021-02-05.

[5] J. J. Sakurai och J. J. Napolitano, *Modern Quantum Mechanics*, 2. utg. San Francisco, CA: Addison-Wesley, 2011, ISBN: 978-0-8053-8291-4.

[6] R. H. Landau, *Quantum Mechanics II: A Second Course in Quantum Theory*, 2. utg. Wiley-VCH, 1995, ISBN: 978-0-471-1 1608-0.

[7] J. M. Blatt och L. C. Biedenharn, "Neutron-Proton Scattering with Spin-Orbit Coupling. I. General Expressions," *Phys. Rev.*, volym 86, s. 399–404, 3 maj 1952. DOI: 10.1103/PhysRev.86.399.

[8] H. P. Stapp, T. J. Ypsilantis och N. Metropolis, "Phase-Shift Analysis of 310-Mev Proton-Proton Scattering Experiments," *Phys. Rev.*, volym 105, s. 302–310, 1 jan. 1957. DOI: 10.1103/PhysRev.105.302.

[9] M. Hjorth-Jensen, "Computational Physics - Lecture Notes Fall 2013," okt. 2013.

[10] *numpy.polynomial.legendre.leggauss*, https://numpy.org/doc/stable/reference/generated/numpy.polynomial.legendre.leggauss.html, Hämtad: 2021-05-13.

[11] N. Higham, *What Is a Companion Matrix?* https://nhigham.com/2021/03/23/what-is-a-companion-matrix/, Hämtad: 2021-05-13, mars 2021.

[12] A. Ekström och S. Miller, *priv com.*

[13] *CUDA Toolkit Documentation – cuBLAS*, https://docs.nvidia.com/cuda/cublas/index.html, Hämtad: 2021-04-30.

[14] I. Gustafsson, *Numerisk analys*. Stockholm: Liber, 2016, s. 104–105, 108, ISBN: 978-91-47-11246-3.

[15] B. D. Carlsson, A. Ekström, C. Forssén m. fl., "Uncertainty Analysis and Order-by-Order Optimization of Chiral Nuclear Interactions," English, *Physical Review X*, volym 6, nr 1, s. 011 019–23, febr. 2016. DOI: 10.1103/PhysRevX.6.011019.

[16] *Vera*, https://www.c3se.chalmers.se/about/Vera/, Hämtad: 2021-02-05.

[17] *Tesla V100*, https://www.nvidia.com/en-gb/data-center/tesla-v100/, Hämtad: 2021-05-02.

A

Bibliotek för linjär algebra

A.1 LAPACK

I projektet användes biblioteket LAPACK. LAPACK utvecklades först för FORTRAN 77, men har översatts till C. C-implementeringen innehåller ett stort antal egenheter som gick rakt emot de programmeringskonventioner som användes i övrigt i projektet. För att underlätta programmeringen och kodens läsbarhet utvecklades ett enkelt LAPACK-API som passade bättre med koden i övrigt än vad den vanliga C-implementeringen gjorde. Nedan redogörs för API:ets innehåll.

A.1.1 Konstruktorer

LAPACK erbjuder inga klasser för matriser. Istället nyttjas vanligtvis fält eller vektorer, vars innehåll får motsvara matriselementen i kolumnvis ordning, på detta vis:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \Longrightarrow \begin{bmatrix} a_{11} & a_{21} & a_{31} & a_{12} & a_{22} & a_{32} & a_{13} & a_{23} & a_{33} \end{bmatrix}. \tag{A.1}$$

Matrisens dimensioner lagras separat, som vanliga heltal.

För att underlätta programmeringen skapades klassen `LapackMat`, vilken lagrar matrisens innehåll som en vektor, samt matrisens dimensioner. Tre olika konstruktorer skapades: en som tar matrisens dimensioner och innehållsvektor som argument och skapar motsvarande matrisobjekt, en som tar matrisens dimensioner och skapar en nollfylld matris med givna dimensioner, och en som tar sidlängden för en kvadratisk matris och skapar motsvarande identitetsmatris. Dessa tre konstruktorer kallade alla på en funktion vid namn `init` med lämpliga argument. Detta för att hålla koden mer koncis.

```
1  void LapackMat::init(int x, int y, std::vector<std::complex<double>> z) {
2      width = x;
3      height = y;
4      contents = z;
5  }
6
7  LapackMat::LapackMat(int x, int y, std::vector<std::complex<double>> z) {
8    LapackMat::init(x, y, z);
9  }
10
11 LapackMat::LapackMat(int x, int y) {
12   LapackMat::init(x, y, std::vector<std::complex<double>>(x*y, 0.0));
13 }
14
15 LapackMat::LapackMat(int x) {
```

```
  LapackMat::init(x, x, std::vector<std::complex<double>>(x*x, 0.0));

  for (int i = 0; i < x; i++) {
    this->setElement(i, i, 1.0);
  }
}
```

A.1.2 Elementoperationer

Två enkla funktioner skapades för att utläsa eller ändra ett element vid ett givet index för ett `LapackMat`-objekt.

```
std::complex<double> LapackMat::getElement(int row, int col) {
  if (row >= this->height) {
    std::cout << "Invalid row: " << row << " (Column: " << col << ")" << std::endl;
    abort();
  }
  if (col >= this->width) {
    std::cout << "Invalid column: " << col << " (Row: " << row << ")" << std::endl;
    abort();
  }
  return contents[row + col*height];
}

void LapackMat::setElement(int row, int col, std::complex<double> value) {
  if (row >= this->height) {
    std::cout << "Invalid row: " << row << " (Column: " << col << ")" << std::endl;
    abort();
  }
  if (col >= this->width) {
    std::cout << "Invalid column: " << col << " (Row: " << row << ")" << std::endl;
    abort();
  }
  contents[row + col*height] = value;
}
```

A.1.3 Matrisalgebra

Matrisalgebran i LAPACK utgår från kommandot `zgemm_`, vilket i grund och botten genomför en manipulation av en matris C enligt

$$C = \alpha \cdot \mathrm{op}(A) \cdot \mathrm{op}(B) + \beta \cdot C, \tag{A.2}$$

där A och B är matriser, operatorn op() antingen är en transponering eller en identitetsoperator och α och β är skalärer. I våra egenskrivna algebrafunktioner kopieras ofta matrisen C innan funktionen genomförs eftersom `zgemm_` manipulerar den; utan kopiering skulle den ursprungliga matrisen förloras. `zgemm_` definieras som följer:

```
extern "C" {
  void zgemm_(char* TRANSA, char* TRANSB, int* M, int* N, int* K,
```

```
    std::complex<double>* ALPHA, std::complex<double>* A, int* LDA,
    std::complex<double>* B, int* LDB, std::complex<double>* BETA,
    std::complex<double>* C, int* LDC);
}
```

Argumenten förklaras i tabell A.1

Tabell A.1: Argumenten i `zgemm_` och deras innebörd.

`TRANSA`	`'T'` för att transponera `A`. `'N'` annars.
`TRANSB`	`'T'` för att transponera `B`. `'N'` annars.
`M`	Radantal i `A` och `C`.
`N`	Kolonnantal i `B` och `C`.
`K`	Kolonnantal i `A` och radantal i `B`.
`ALPHA`	α i ekvation (A.2).
`A`	A i ekvation (A.2).
`LDA`	Ledande dimension i `A`. Används för att operera på undermatriser till `A`. Sätts lika med `M` för att operera på hela matrisen.
`B`	B i ekvation (A.2).
`LDB`	Ledande dimension i `B`. Används för att operera på undermatriser till `B`. Sätts lika med `K` för att operera på hela matrisen.
`BETA`	β i ekvation (A.2).
`C`	C i ekvation (A.2).
`LDC`	Ledande dimension i `C`. Används för att operera på undermatriser till `C`. Sätts lika med `M` för att operera på hela matrisen.

```
LapackMat operator+(LapackMat A, LapackMat B) {
  LapackMat dummyB   = LapackMat(B.width, B.height, B.contents);
  LapackMat identity = LapackMat(A.height);
  char TRANS = 'N';
  std::complex<double> ALPHA = 1;
  std::complex<double> BETA  = 1;

  zgemm_(&TRANS, &TRANS, &A.height, &identity.width, &A.width, &ALPHA,
      A.contents.data(), &A.height, identity.contents.data(), &A.width, &BETA,
      dummyB.contents.data(), &A.height);

  return dummyB;
}

LapackMat operator-(LapackMat A, LapackMat B) {
  LapackMat dummyB   = LapackMat(B.width, B.height, B.contents);
  LapackMat identity = LapackMat(A.height);
  char TRANS = 'N';
  std::complex<double> ALPHA = 1;
  std::complex<double> BETA  = -1;

  zgemm_(&TRANS, &TRANS, &A.height, &identity.width, &A.width, &ALPHA,
      A.contents.data(), &A.height, identity.contents.data(), &A.width, &BETA,
```

```
      dummyB.contents.data(), &A.height);

  return dummyB;
}

LapackMat operator*(std::complex<double> scalar, LapackMat A) {
  LapackMat B = LapackMat(A.height);
  LapackMat C(A.width, A.height);
  char TRANS = 'N';
  std::complex<double> BETA = 0;

  zgemm_(&TRANS, &TRANS, &A.height, &B.width, &A.width, &scalar, A.contents.data(),
      &A.height, B.contents.data(), &A.width, &BETA, C.contents.data(), &A.height);

  return C;
}

LapackMat operator*(LapackMat A, std::complex<double> scalar) {
  return scalar*A;
}

LapackMat operator*(LapackMat A, LapackMat B) {
  LapackMat C(A.height, B.width);
  char TRANS = 'N';
  std::complex<double> ALPHA = 1;
  std::complex<double> BETA  = 0;

  zgemm_(&TRANS, &TRANS, &A.height, &B.width, &A.width, &ALPHA, A.contents.data(),
      &A.height, B.contents.data(), &A.width, &BETA, C.contents.data(), &A.height);

  return C;
}
```

A.1.4 Matrisekvationer

För att lösa den centrala matrisekvationen (se ekvation (2.26)) används en tvåstegsmetod: först skapas en LU-faktorisering av F-matrisen med pivotering och sen löses ekvationen med hjälp av denna faktorisering. Detta ger ökad beräkningshastighet och större numerisk stabilitet än en matrisinversion [14].

De två stegen motsvaras av LAPACK-funktionerna `zgetrf_` respektive `zgetrs_`. De definieras som följer:

```
extern "C" {
  void zgetrf_(int* M, int *N, std::complex<double>* A, int* LDA, int* IPIV,
      int* INFO);
  void zgetrs_(char* TRANS, int* N, int* NRHS, std::complex<double>* A, int* LDA,
      int* IPIV, std::complex<double>* B, int* LDB, int* INFO);
}
```

Argumenten förklaras i tabell A.2 och A.3.

Tabell A.2: Argumenten i `zgetrf_` och deras innebörd.

`M`	Radantal i `A`.
`N`	Kolonnantal i `A`.
`A`	Matrisen som ska faktoriseras. Efter körning innehåller `A`:s nedre vänstra halva L-matrisen (dock utan ettorna på diagonalen, men dessa är implicita), och dess övre högra halva innehåller U-matrisen.
`LDA`	Ledande dimension i `A`. Används för att operera på undermatriser till `A`. Sätts lika med `M` för att operera på hela matrisen.
`IPIV`	Fält som fylls med heltal som motsvarar pivoteringen i faktoriseringen med cykelnotation.
`INFO`	Är 0 efter körning om körningen lyckades och nollskild annars.

Tabell A.3: Argumenten i `zgetrs_` och deras innebörd.

`TRANS`	`'T'` för att transponera `A`. `'N'` annars.
`N`	Ordning av `A`.
`NRHS`	Antalet högerled i matrisekvationen (NRHS = Number of Right Hand Sides), det vill säga antalet kolonner i `B`.
`A`	A-matrisen i ekvationen $AX = B$ efter att ha LU-faktoriserats av `zgetrf_`.
`LDA`	Ledande dimension i `A`. Används för att operera på undermatriser till `A`. Sätts lika med `N` för att operera på hela matrisen.
`IPIV`	Pivoteringsfält som ges av `zgetrf_`.
`B`	B-matrisen i ekvationen $AX = B$. Efter körning innehåller den X för samma ekvation; det är alltså här lösningen plockas ut.
`LDB`	Ledande dimension i `B`. Används för att operera på undermatriser till `B`. Sätts lika med `N` för att operera på hela matrisen.
`INFO`	Är 0 efter körning om körningen lyckades och nollskild annars.

Dessa funktioner implementeras i den egenskrivna funktionen `solveMatrixEq`. Eftersom `zgetrf_` modifierar `A`-matrisen och `zgetrs_` modifierar `B`-matrisen kopieras dessa först och deras kopior används i beräkningen; på detta vis undviker man att förlora de ursprungliga matriserna ifall de skulle behövas senare.

```
1  LapackMat solveMatrixEq(LapackMat A, LapackMat B) {
2    LapackMat dummyA = LapackMat(A.width, A.height, A.contents);
3    LapackMat dummyB = LapackMat(B.width, B.height, B.contents);
4
5    int INFO;
6    char TRANS = 'N';
7    std::vector<int> IPIV(std::min(A.width, A.height));
8
9    zgetrf_(&A.height, &A.width, dummyA.contents.data(), &A.height, IPIV.data(),
10       &INFO);
11   zgetrs_(&TRANS, &A.height, &B.width, dummyA.contents.data(), &A.width,
12       IPIV.data(), dummyB.contents.data(), &A.height, &INFO);
13
14   return dummyB;
15 }
```

A.1.5 Egenvärden

För egenvärdesberäkning används funktionen `zheev_`. Denna funktion kan bara beräkna egenvärden för hermiteska matriser, men detta krav uppfylls av de relevanta matriserna i programmet. Funktionen definieras som följer:

```
extern "C" {
  void zheev_(char* JOBZ, char* UPLO, int* N, std::complex<double>* A, int* LDA,
      double* W, std::complex<double>* WORK, int* LWORK, double* RWORK,
      int* INFO);
}
```

Funktionens argument redovisas i tabell A.4.

Tabell A.4: Argumenten i `zheev_` och deras innebörd.

`JOBZ`	`'N'` för att enbart beräkna egenvärden. `'V'` för att beräkna både egenvärden och egenvektorer.
`UPLO`	Då `A` är hermitesk behövs bara ena halvan; den andra är implicit. `'U'` för att funktionen ska använda den övre triangeln av matrisen och `'L'` för att den lägre ska användas.
`N`	Ordningen av `A`.
`A`	Den hermiteska matrisen vars egenvärden ska beräknas. Innehåller egenvektorer till `A` efter körning om `JOBZ = 'V'`. Om `JOBZ = 'U'` raderas den triangeln i `A` som angavs av `UPLO`, inklusive diagonalen.
`LDA`	Ledande dimension i `A`. Används för att operera på undermatriser till `A`. Sätts lika med `N` för att operera på hela matrisen.
`W`	Fält som innehåller de beräknade egenvärdena efter körning.
`WORK`	”Arbetsfält”; används internt av programmet vid beräkningar, men har ingen särskild betydelse för användaren varken innan eller efter körning.
`LWORK`	Anger dimension på `WORK`. Kan med fördel sättas till `2*N-1` för att säkert vara tillräckligt stor.
`RWORK`	Ytterligare arbetsfält.
`INFO`	Är 0 efter körning om körningen lyckades och nollskild annars.

`zheev_` implementeras i den egenskrivna funktionen `eigenValues`, som följer:

```
std::vector<double> eigenValues(LapackMat A) {
  LapackMat dummyA = LapackMat(A.width, A.height, A.contents);

  char JOBZ = 'N';
  char UPLO = 'U';
  std::vector<double> W(A.width);
  int LWORK = 2*A.width-1;
  std::vector<double> RWORK(3*A.width-2);
  std::vector<std::complex<double>> WORK(LWORK);
  int INFO = 0;

  zheev_(&JOBZ, &UPLO, &A.width, dummyA.contents.data(), &A.width, W.data(),
      WORK.data(), &LWORK, RWORK.data(), &INFO);

```

```
15    return W;
16  }
```

Notera att `A` kopieras och att kopian används i `zheev_` då matrisen modifieras av funktionen.

A.2 cuBLAS

A.2.1 `cublasZgetrfBatched`

Funktionen `cublasZgetrfBatched` är en GPU-anpassad, parallell version av `zgetrf_` i LAPACK, vilken beskrivs i avsnitt A.1.4. Parametrarna beskrivs i tabell A.5. De största skillnaderna är att ett cuBLAS-handle behövs, att `M`-parametern är exkluderad (matriserna antas vara kvadratiska), att `A` har ersatts med ett fält av pekare till `A`-matriserna och att det finns en parameter för antalet matriser.

Tabell A.5: Argumenten i `cublasZgetrfBatched` och deras innebörd.

`handle`	cuBLAS-handle som kopplar till cuBLAS-biblioteket. Innehåller info om hårdvaran bland annat.
`n`	Ordningen av `A`-matriserna.
`Aarray`	Fält av pekare till `A`-matriserna.
`lda`	Ledande dimension i `A`-matriserna. Används för att operera på undermatriser till `A`-matriserna. Sätts lika med `N` för att operera på hela matriserna.
`PivotArray`	Fält som fylls med heltal som motsvarar pivoteringarna i faktoriseringen med cykelnotation.
`infoArray`	Fält där varje element motsvarar en matris. Elementet som motsvarar en matris är 0 efter körning om körningen lyckades och nollskild annars.
`batchSize`	Antalet matriser.

A.2.2 `cublasZgetrsBatched`

Funktionen `cublasZgetrsBatched` är en GPU-anpassad, parallell version av `zgetrs_` i LAPACK, vilken beskrivs i avsnitt A.1.4. Parametrarna beskrivs i tabell A.6. De största skillnaderna är att ett cuBLAS-handle behövs, att `A` och `B` har ersatts med ett fält av pekare till `A`-matriserna respektive `B`-matriserna och att det finns en parameter för antalet matriser.

Tabell A.6: Argumenten i `cublasZgetrsBatched` och deras innebörd.

`handle`	cuBLAS-handle som kopplar till cuBLAS-biblioteket. Innehåller info om hårdvaran bland annat.
`trans`	`CUBLAS_OP_T` för att transponera matriserna. `CUBLAS_OP_N` annars.
`n`	Ordningen av `A`-matriserna.
`nrhs`	Antalet högerled i matrisekvationen (NRHS = Number of Right Hand Sides), det vill säga antalet kolonner i `B`.
`Aarray`	Fält av pekare till `A`-matriserna.
`lda`	Ledande dimension i `A`-matriserna. Används för att operera på undermatriser till `A`-matriserna. Sätts lika med `N` för att operera på hela matriserna.
`devIpiv`	Fält från `cublasZgetrf` med heltal som motsvarar pivoteringarna i faktoriseringen med cykelnotation.
`Barray`	Fält av pekare till `B`-matriserna.
`ldb`	Ledande dimension i `B`-matriserna. Används för att operera på undermatriser till `B`-matriserna. Sätts lika med `N` för att operera på hela matriserna.
`info`	0 om alla matrislösningar lyckades, nollskild annars.
`batchSize`	Antalet matriser.