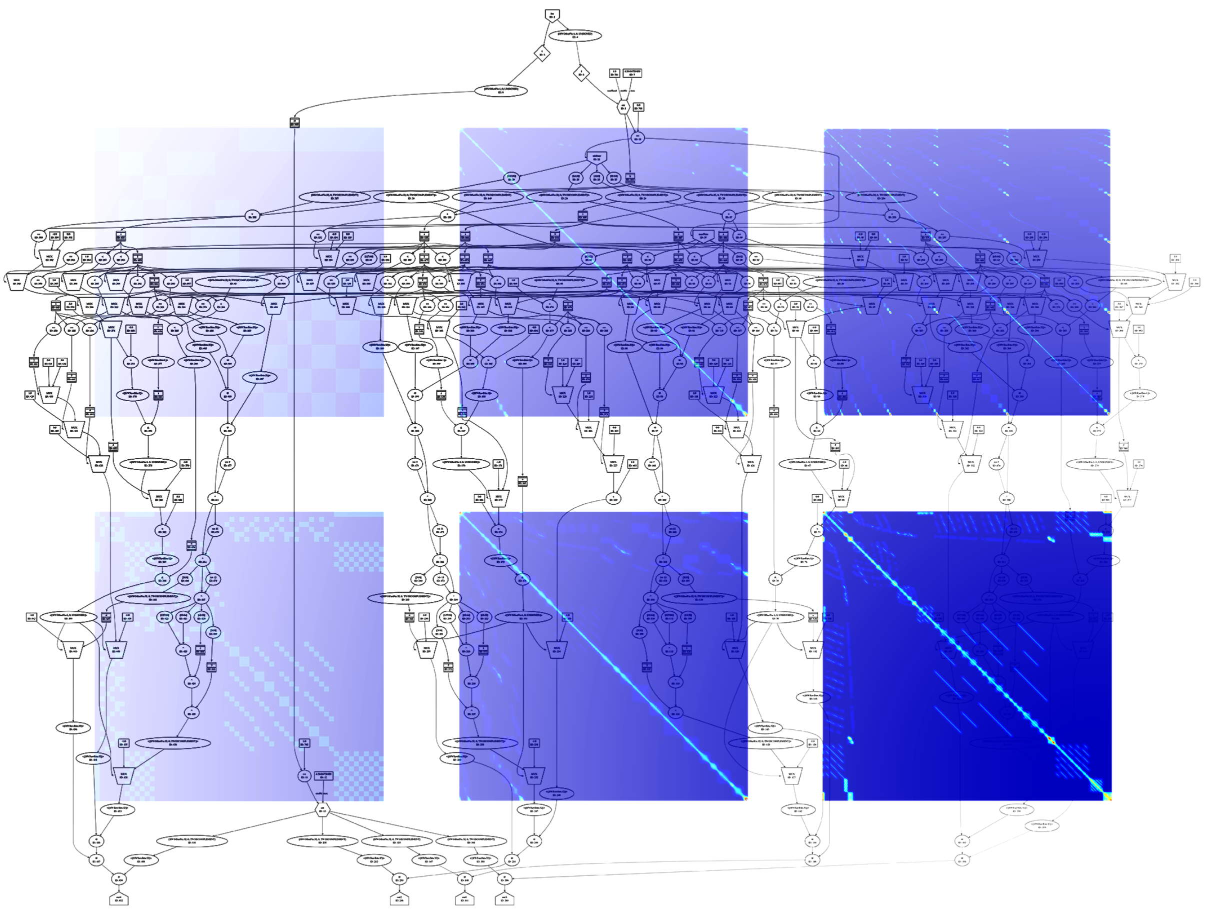
CHALMERS



Feasibility of FPGA-based Computations of Transition Densities in Quantum Many-Body Systems

Bachelor's thesis in Engineering Physics

ROBERT ANDERZÉN, MAGNUS RAHM, OLOF SALBERGER
JOAKIM STRANDBERG, BENJAMIN SVEDUNG, JONATAN WÅRDH

Department of Fundamental Physics
Division of Nuclear Theory
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2013
Bachelor's thesis FUFX02-13-02

BACHELOR'S THESIS IN ENGINEERING PHYSICS

Feasibility of FPGA-based Computations of Transition Densities in Quantum Many-Body Systems

ROBERT ANDERZÉN, MAGNUS RAHM, OLOF SALBERGER
JOAKIM STRANDBERG, BENJAMIN SVEDUNG, JONATAN WÅRDH

Department of Fundamental Physics
Division of Nuclear Theory
CHALMERS UNIVERSITY OF TECHNOLOGY

Göteborg, Sweden 2013

Feasibility of FPGA-based Computations of Transition
Densities in Quantum Many-Body Systems
ROBERT ANDERZÉN, MAGNUS RAHM, OLOF SALBERGER
JOAKIM STRANDBERG, BENJAMIN SVEDUNG, JONATAN WÅRDH

Bachelor's thesis FUFX02-13-02
ISSN 1654-4676
Department of Fundamental Physics
Division of Nuclear Theory
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31-772 1000

Cover:
Six matrices showing the distribution of connections between many-body states, for systems of different particle numbers and cutoff energies, overlaid on a graph of the kernel-design for generating the connections for a system of four particles.

Chalmers Reproservice
Göteborg, Sweden 2013

Abstract

This thesis presents the results from a feasibility study of implementing calculations of transition densities for quantum many-body systems on FPGA hardware. Transition densities are of interest in the field of nuclear physics as a tool when calculating expectation values for different operators. Specifically, this report focuses on transition densities for bound states of neutrons. A computational approach is studied, in which FPGAs are used to identify valid connections for one-body operators. Other computational steps are performed on a CPU. Three different algorithms that find connections are presented. These are implemented on an FPGA and evaluated with respect to hardware cost and performance. The performance is also compared to that of an existing CPU-based code, Trdens.

The FPGA used to implement the proposed designs was a *Xilinx Virtex 6*, built into Maxeler's MAX3 card. It was concluded that the FPGA was able to find the connections of a one-body operator in a fraction of the time used by Trdens, ran on a single CPU-core. However, the CPU-based conversion of the connections to the form in which Trdens presents them, was much more time-consuming. For FPGAs to be feasible, it is hence necessary to accelerate the CPU-based computations or include them into the FPGA-implementations. Therefore, we recommend further investigations regarding calculations of the final representation of transition densities on FPGAs, without the use of an off-FPGA computation.

Acknowledgements

We would like to express our very great appreciation to our physics supervisors Christian Forssén and Håkan Johansson for their continuous assistance and support throughout the project. For their time and knowledge invested in this project, and for ongoing advice and encouragement we are deeply grateful.

We would also like to offer our special thanks to our data supervisors Georgi Gaydadjiev and Ioannis Sourdis who helped us in understanding and using the FPGA and to Catalin Ciobanu who guided us in our struggles with the Maxeler machine.

Contents

Chapter 1

Introduction

Ever since the first steps of quantum mechanics were taken in the beginning of the 20th century, our understanding of physics on a microscopical scale has grown rapidly. Although the theory seems to reproduce experimental results, most systems are way too complicated to be solved analytically. One such example is many-body systems, e.g. atomic nuclei. In the last decades, as powerful computers have been developed, numerical computations have become an increasingly important tool to understand the physics. However, for *ab initio* many-body problems, not even the most powerful computers are efficient enough to perform the calculations in a reasonable amount of time. The desire to further push the boundaries of what can be calculated, calls for more efficient algorithms and hardware customized for the purpose.

The final goal in *ab initio* many-body computations is usually to calculate an expectation value for a given observable. The process involves several steps, one of which is to create matrix representations of either the Hamiltonian or another observable, with respect to a many-body basis. The accuracy of the results is highly dependent on the dimension of this basis. Already for quite small systems, such as the ^{10}B-core, a basis size of some billion elements is desirable, leading to a matrix representation with 10^{18} elements.

A useful representation of the problem is given by using the second quantization formalism. With the aid of this formalism, the construction of the matrix becomes a highly regular procedure of determining whether elements of the type

$$\langle \nu_j | a_\alpha^\dagger a_\beta | \nu_k \rangle \tag{1.1}$$

are 0, 1 or -1, i.e. if $|\nu_j\rangle$ and $|\nu_k\rangle$ connect. After doing so, non-zero elements are summed up using eigenstates of the Hamiltonian, forming transition density matrix elements which contain the major part of the computational effort needed to calculate expectations. Based on the transition densities, we can also calculate expectation values corresponding to a multitude of different operators, with few additional computations.

This thesis investigates the feasibility of using FPGAs for the computation of transition densities. An FPGA (*Field Programmable Gate Array*) is a type of hardware built out of many small logic blocks with programmable interconnections, allowing pipelining and massive parallelism. Such features are highly desirable when attempting to accelerate a computation in which a lot of data is subject to the same operations. Currently, the calculation of transition densities is mainly performed in parallel on CPU-based clusters. However, the regularity of the computation indicates that it might be suitable for implementation on an FPGA. Therefore, the main objective of the current thesis is to investigate if the calculation of transition densities can be adapted to FPGA-implementation, and whether such an implementation could be more efficient than CPU-based computations.

1.1 Specific aims

As stated above, the goal of this project has been to investigate the feasibility of FPGA-implementation of the calculation of transition densities. To do this, we have aimed at developing a number of algorithms to calculate transition densities for one-body operators, corresponding to fermionic systems with a single kind of particle. The performance of these have then been studied and compared to a CPU-based computational approach. The conclusions are summarized in a recommendation, suggesting whether FPGA-implementation is feasible for calculation of transition densities and how the developed algorithms can be generalized to wider classes of operators and systems.

1.2 Method

Initially, literature studies were conducted in order to find how transition densities are expressed in terms of second quantization. Also, properties and programming of FPGAs were investigated, specifically a programming interface by Maxeler, for their MAX3 FPGA. Following this, we developed various algorithms for how the FPGA could be used to calculate transition densities. Finally, these were implemented and benchmarked with regards to their performances, and then compared quantitatively to the performance of an existing CPU-based code.

1.3 Structure of the thesis

In chapter 2, we explain the quantum mechanical formalism, specifically the theory of second quantization, which constitutes a cornerstone of many-body theory. This is followed by chapter 3, where we give a more precise formulation of what is meant by calculating a transition density, providing a numerical example to clarify the concepts. The transition density matrix elements described in this chapter are of a non-reduced form. However, we are ultimately interested in a slightly different, reduced, representation, which in a final step needs to be calculated. The description of the theory for this belongs to chapter 4.

Chapter 5 serves as an introduction to the properties and computational possibilities of FPGAs. We describe how this leads to specific considerations in algorithm designs and give an overview of the Maxeler programming interface. Thereafter, chapter 6 describes and evaluates a number of different strategies using FPGAs to calculate transition densities. We also describe the methodology of how the reduced matrix representation can be calculated, from FPGA-generated output.

In chapter 7 we evaluate the performance of the algorithms described in chapter 6. We verify the correctness of the computation and determine which algorithm is the most efficient. Also, we analyze performance bounding factors and compare to the performance of an available CPU-based code. Finally, in chapter 8 we discuss the results, and outline possible generalizations of the implementations, leading up to chapter 9 in which we give recommendations and conclusions of whether FPGAs are suitable for the calculation of transition densities.

Chapter 2

Quantum many-body theory

This chapter serves to describe the quantum mechanics needed to accurately formulate the concepts and calculation of transition densities, i.e. the model problem in this project. Specifically, we will describe many-body theory and the second quantization formalism. The starting point will be to introduce the many-body basis for a system of k particles, giving a brief description of the consequences of the fact that the particles are indistinguishable. Thereafter, we use this basis to introduce the second quantization formalism, involving concepts such as the Fock space and creation/annihilation operators. For completeness, we will treat both bosons and fermions, but the latter will be the main topic of this study.

2.1 Symmetric and anti-symmetric bases

To describe the Hilbert space for a system of interacting particles, we need a basis for the state space of these particles. Such a basis can be constructed using a single-particle basis, i.e. a basis for the state space of a single particle. The single-particle basis is chosen as the energy eigenstates of some appropriately chosen single-particle potential.

To introduce some formalism, we will denote the Hilbert space for a single particle by $\mathcal{H}$ and the single-particle basis by $\{|\alpha\rangle\}$. We assume that $|\alpha\rangle$ are chosen to be orthonormal as this can always be done for eigenstates of a Hermitian operator.

The Hilbert space for the system of k particles, $\mathcal{H}^{\otimes k}$, is formed as the k-fold tensor product of single-particle Hilbert spaces, $\mathcal{H}_i$:

$$\mathcal{H}^{\otimes k} = \mathcal{H}_1 \otimes \mathcal{H}_2 \otimes ... \otimes \mathcal{H}_k. \tag{2.1}$$

This construction is adjoined with a natural basis containing product states:

$$|\alpha_1\alpha_2...\alpha_k) := |\alpha_1\rangle \otimes |\alpha_2\rangle \otimes ... \otimes |\alpha_k\rangle. \tag{2.2}$$

Note that the characteristic soft parenthesis $|...)$ is used when denoting product states. These states are orthonormal which could be written as[1]

$$(\alpha_1\alpha_2...\alpha_k|\alpha_1'\alpha_2'...\alpha_k') = \delta_{\alpha_1,\alpha_1'}\delta_{\alpha_2,\alpha_2'}...\delta_{\alpha_k,\alpha_k'}. \tag{2.3}$$

[1] $\delta_{i,j}$ denotes Kronecker's delta.

As we will see, these product states are not appropriate to describe physical states. However, we will use them to construct basis states that indeed are physically valid and will constitute our many-body basis.

2.1.1 Identical particles

In quantum mechanics, particles of the same type are indistinguishable entities. Suppose we have two states $|\alpha_1\rangle$ and $|\alpha_2\rangle$ describing two different particles of the same type. If $|\alpha_1\alpha_2)$ would be a physically valid state for the combined system of both particles, it would necessarily be the same as $|\alpha_2\alpha_1)$, due to that they are indistinguishable. However, we can only regard many-body states that make no difference between particles as physically valid. We conclude that neither $|\alpha_1\alpha_2)$ nor $|\alpha_2\alpha_1)$ can represent a physically valid state.

In order to extract physically valid states, we introduce the permutation operator P_{12} whose action upon a state $|\alpha_1\alpha_2)$ is to interchange the quantum numbers of particle 1 and 2, i.e.

$$P_{12}|\alpha_1\alpha_2) = |\alpha_2\alpha_1). \tag{2.4}$$

From the fact that all $|\alpha_i\alpha_j)$ are orthonormal we conclude that the matrix elements of P_{12} will be real. Hence P_{12} is Hermitian. Furthermore, when acting with P_{12} twice we get the same state back. This implies that

$$P_{12}^2 = P_{12}^\dagger P_{12} = P_{12}^{-1} P_{12} = 1, \tag{2.5}$$

and hence that P_{12} is unitary with eigenvalues $p = \pm 1$.

Now, consider two operators $A(1)$ and $A(2)$ being the same operator but acting on different single-particle Hilbert spaces. Let $|\alpha_i\rangle$ be an eigenstate of the operator A, having the eigenvalue a_i. This means that

$$\begin{aligned} A(1)|\alpha_1\alpha_2) &= a_1|\alpha_1\alpha_2), \\ A(2)|\alpha_1\alpha_2) &= a_2|\alpha_1\alpha_2). \end{aligned} \tag{2.6}$$

Now, we let P_{12} act on $A(1)|\alpha_1\alpha_2)$,

$$P_{12}A(1)|\alpha_1\alpha_2) = P_{12}a_1|\alpha_1\alpha_2) = a_1|\alpha_2\alpha_1) = A(2)|\alpha_2\alpha_1). \tag{2.7}$$

Recalling that $P_{12}^{-1}P_{12} = 1$, (2.7) can further be written as

$$P_{12}A(1)|\alpha_1\alpha_2) = P_{12}A(1)P_{12}^{-1}P_{12}|\alpha_1\alpha_2) = P_{12}A(1)P_{12}^{-1}|\alpha_2\alpha_1). \tag{2.8}$$

Since $|\alpha_1\alpha_2)$ is arbitrary, we deduce that $P_{12}A(1)P_{12}^{-1} = A(2)$. This can be interpreted as a change of labelling of the particles, making the operators act on the other particle. However, knowing that particles are indistinguishable we should only construct Hamiltonians and other operators where the indices of the particles appear symmetrically. Otherwise we would have given certain particles a certain meaning. Expressed more formally, we demand that $P_{12}HP_{12}^{-1} = H$ since the left-hand side only switches the Hilbert spaces of the particles on which the Hamiltonian acts.

The properties of P_{12} extend to permutation operators P_{ij} acting on arbitrary indices i and j in Hilbert spaces with k particles. We can define a general permutation operator

P as a product of P_{ij}s. Then P performs an arbitrary permutation of the single-particle states when acting on a product state (2.2). This operator is also Hermitian and unitary. By repeatedly invoking $[H,P_{ij}]$ we see that the Hamiltonian commutes with the general permutation operator,

$$[H,P] = 0. \tag{2.9}$$

In Section 2.3, we will motivate that this implies that we are confined to either the symmetric $p = +1$, or the anti-symmetric $p = -1$ eigenspace of P. Particles belonging to the respective eigenspaces are referred to as *bosons* and *fermions*. When dealing with a specific kind of particle, one only needs to consider states in one of the eigenspaces of P. This will be a cornerstone when developing the many-body basis, which then can be specified for bosons and fermions separately.

2.1.2 Bosonic and fermionic eigenstates

In order to find many-body bases for bosons and fermions, we should search for eigenstates to P having the eigenvalues 1 and -1, respectively. In the case of two particles, the construction is quite simple.

Example. The state

$$|\alpha_1\alpha_2\rangle = \frac{1}{\sqrt{2}}\left[|\alpha_1\alpha_2) - |\alpha_2\alpha_1)\right] \tag{2.10}$$

will be a fermionic eigenstate, because when acting with the permutation operator, we get

$$P_{12}|\alpha_1\alpha_2\rangle = \frac{1}{\sqrt{2}}\left[P_{12}|\alpha_1\alpha_2) - P_{12}|\alpha_2\alpha_1)\right] = \frac{1}{\sqrt{2}}\left[|\alpha_2\alpha_1) - |\alpha_1\alpha_2)\right] = -|\alpha_1\alpha_2\rangle. \tag{2.11}$$

A bosonic eigenstate is constructed similarly, but with a plus, i.e.

$$|\alpha_1\alpha_2\rangle = \frac{1}{\sqrt{2}}\left[|\alpha_1\alpha_2) + |\alpha_2\alpha_1)\right]. \tag{2.12}$$

Notice that we denote the bosonic and fermionic eigenstates with a ket, as opposed to the soft paranthesis that was used for product states. Note also that we use a factor $2^{-1/2}$ to normalize the states. ■

In order to construct a symmetric or an anti-symmetric many-body basis for arbitrary k, we use the permutation operator to define symmetrizing and anti-symmetrizing operators,

$$\mathcal{S} = \frac{1}{k!}\sum_p P, \qquad \mathcal{A} = \frac{1}{k!}\sum_p (-1)^p P. \tag{2.13}$$

The sums run over all permutations p, even or odd. When acting with $\mathcal{S}$ or $\mathcal{A}$ on a product state of type (2.2), we construct a physically valid symmetric or anti-symmetric basis state for a bosonic or fermionic system, respectively. Thus, for bosons we have

$$|\alpha_1\alpha_2...\alpha_k\rangle = \sqrt{\frac{k!}{n_{\alpha'}!n_{\alpha''}!...}}\,\mathcal{S}|\alpha_1\alpha_2...\alpha_k) \tag{2.14}$$

where $n_{\alpha'}$ is the number of particles in the states α', serving to normalize the state[2]. From (2.14) it is easily seen that interchanging two particles leaves the state unchanged, i.e.

$$|\alpha_1\alpha_2...\alpha_k\rangle = |\alpha_2\alpha_1...\alpha_k\rangle. \tag{2.15}$$

In the case of fermions, we have

$$|\alpha_1\alpha_2...\alpha_k\rangle = \sqrt{k!}\mathcal{A}|\alpha_1\alpha_2...\alpha_k). \tag{2.16}$$

For fermions, the Pauli exclusion principle is incorporated thanks to the sign in the anti-symmetrizing operator, i.e. it induces a negative sign so that

$$|\alpha_1\alpha_2...\alpha_k\rangle = -|\alpha_2\alpha_1...\alpha_k\rangle. \tag{2.17}$$

When assigning the same quantum numbers to two particles, the state will therefore vanish.

A final point concerning the notation for indistinguishable particles, either bosons and fermions, is that states referred to by different ordering of the single-particle states exhibit no significant physical difference. It is therefore common practice to impose a specific *ordering*. In this thesis, we will choose the single-particle states to be ordered increasingly, with respect to some enumeration of the single-particle basis. For example, the state $|\alpha_2\alpha_1...\alpha_k\rangle$ will always be sorted to $|\alpha_1\alpha_2...\alpha_k\rangle$, possibly introducing a negative sign in the fermionic case.

2.2 Second quantization

In the second quantization formalism, we describe states by introducing creation and annihilation operators, which may be used to construct any physical state by acting on the vacuum state denoted by $|0\rangle$.

As a starting point, we can take a closer look at how many-body states were constructed in the previous section. For k particles, we described the state space as the direct product of individual state spaces, as seen in (2.1). However, at this point the number of particles was fixed. Contrary to this, we now want to introduce a state space which deals with any number of particles, i.e. a state space that can handle the creation and annihilation of particles. One way to construct such a space is to form the direct sum of spaces of the form (2.1), for all $k \geq 0$. This yields

$$\mathcal{F} = \bigoplus_{k=0}^{\infty} \mathcal{H}^{\otimes k}, \tag{2.18}$$

which is known as the *Fock space*. When dealing with fermions and bosons this space is unnecessarily large and we can restrict ourselves to symmetrical and anti-symmetrical Fock space, obtained by

$$\mathcal{S}(\mathcal{F}) = \bigoplus_{k=0}^{\infty} \mathcal{S}(\mathcal{H}^{\otimes k}) \quad \text{and} \quad \mathcal{A}(\mathcal{F}) = \bigoplus_{k=0}^{\infty} \mathcal{A}(\mathcal{H}^{\otimes k}), \tag{2.19}$$

[2]Here, we need to distinguish α', α'' etc. from $\alpha_{1,2,...}$ because for bosons, several α_i could refer to the same single-particle state. The notation will be used for fermions as well.

respectively.

Any state may be written as $|\alpha\rangle = a^\dagger_\alpha|0\rangle$ where $a^\dagger_\alpha$ is a *creation* operator. Going into bra form, we are led to expressions of the form $\langle\alpha| = \langle 0|a_\alpha$. Therefore, whenever we take scalar products and expectation values, the Hermitian conjugates of the creation operators are relevant. We call these the *annihilation* operators.

A different way to introduce the second quantization formalism is by first considering the occupation number representation, which is made possible by the particles being indistinguishable. Suppose that we have a complete basis of single-particle states $\{|\alpha\rangle\}$ that spans the entire single-particle state space. We may identify the many-body state representation used in the previous section by its equivalent occupation number representation,

$$|\alpha_1\alpha_2\alpha_3...\rangle = |n_{\alpha'}, n_{\alpha''}, n_{\alpha'''}, ...\rangle, \tag{2.20}$$

where $n_{\alpha'}$ denotes the number of particles in the state α'. For bosons each $n_{\alpha'}$ may be any non-negative integer, while for fermions it is restricted to 0 or 1 because of the Pauli exclusion principle. The occupation number representation is particularly useful in the case of bosons, though one needs to proceed with care in the fermionic case as one needs to keep track of signs. In the following sections, we will describe the specifics of fermions and bosons.

2.2.1 Fermion operators

For fermionic systems we seek operators that embody the anti-symmetry of fermion states. The fermion *creation* operator can be written as

$$a^\dagger_\alpha|\alpha_1\alpha_2...\alpha_k\rangle = |\alpha\alpha_1\alpha_2...\alpha_k\rangle. \tag{2.21}$$

We can write the states in terms of creation operators operating on the vacuum,

$$|\alpha_1\alpha_2...\alpha_k\rangle = a^\dagger_{\alpha_1}a^\dagger_{\alpha_2}...a^\dagger_{\alpha_k}|0\rangle. \tag{2.22}$$

In the same way as in (2.21) the *annihilation* operator can be written as

$$a_\alpha|\alpha\alpha_1\alpha_2...\alpha_k\rangle = |\alpha_1\alpha_2...\alpha_k\rangle. \tag{2.23}$$

As stated in Section 2.1.2, we want to have the many-body states ordered in a specific way. When sorting the state, a negative sign may appear, according to (2.17). Thus, (2.22) and (2.23) are written, in a more useful way, as

$$a^\dagger_{\alpha_i}|\alpha_1...\alpha_{i-1}\alpha_{i+1}...\alpha_k\rangle = (-1)^{i-1}|\alpha_1...\alpha_{i-1}\alpha_i\alpha_{i+1}...\alpha_k\rangle, \tag{2.24a}$$

$$a_{\alpha_i}|\alpha_1...\alpha_{i-1}\alpha_i\alpha_{i+1}...\alpha_k\rangle = (-1)^{i-1}|\alpha_1...\alpha_{i-1}\alpha_{i+1}...\alpha_k\rangle. \tag{2.24b}$$

As can be seen from (2.22), the creation operators must all anti-commute, to ensure the change of sign. To be able to go back and forth between bras and kets we need the annihilation operators to anti-commute too. To meet the properties of fermions we state the complete set of canonical anti-commutation relations[3]

$$\{a_{\alpha_i}, a^\dagger_{\alpha_j}\} = \delta_{\alpha_i\alpha_j} \tag{2.25a}$$

$$\{a_{\alpha_i}, a_{\alpha_j}\} = \{a^\dagger_{\alpha_i}, a^\dagger_{\alpha_j}\} = 0. \tag{2.25b}$$

[3] $\{a,b\} = ab + ba$ denotes the anti-commutator.

Combined with the relation $a_{\alpha_i}|0\rangle = 0$, it could be taken as the definition of the creation and annihilitation operators for fermions. The relations (2.25) ensures us not to violate any of the anti-symmetric properties.

Example. We can justify (2.25) by noting that it implies $a_{\alpha_2}|\alpha_1\alpha_2\alpha_3\rangle = -|\alpha_1\alpha_3\rangle$, i.e. a_{α_2} does indeed annihilate α_2, and the change of sign is accounted for. We have

$$\begin{aligned}&a_{\alpha_2}a^\dagger_{\alpha_1}a^\dagger_{\alpha_2}a^\dagger_{\alpha_3}|0\rangle = -a^\dagger_{\alpha_1}a_{\alpha_2}a^\dagger_{\alpha_2}a^\dagger_{\alpha_3}|0\rangle = -a^\dagger_{\alpha_1}(1-a^\dagger_{\alpha_2}a_{\alpha_2})a^\dagger_{\alpha_3}|0\rangle = \\ &-a^\dagger_{\alpha_1}a^\dagger_{\alpha_3}|0\rangle + a^\dagger_{\alpha_1}a^\dagger_{\alpha_2}a_{\alpha_2}a^\dagger_{\alpha_3}|0\rangle = -a^\dagger_{\alpha_1}a^\dagger_{\alpha_3}|0\rangle - a^\dagger_{\alpha_1}a^\dagger_{\alpha_2}a^\dagger_{\alpha_3}a_{\alpha_2}|0\rangle = \\ &-a^\dagger_{\alpha_1}a^\dagger_{\alpha_3}|0\rangle.\end{aligned} \tag{2.26}$$

In step three we have used (2.25a) which makes the creation operators of states 1 and 3 remain. In the last step, we also used $a_{\alpha_i}|0\rangle = 0$. ■

We note that if we try to annihilate a single-particle state that does not exist, the result will be zero. This can be seen from (2.25a) which implies that the annihilation operator will walk right through all creation operators (leaving only a difference in sign) and finally operate on the vacuum, where we use $a_{\alpha_i}|0\rangle = 0$. Further, since we are dealing with fermions, we can only have one particle in each state. This is ensured by (2.25b). If we try to create a state that already exists, the result will be zero. This is derived from

$$a^\dagger_{\alpha_1}a^\dagger_{\alpha_1}a^\dagger_{\alpha_2}|0\rangle = -a^\dagger_{\alpha_1}a^\dagger_{\alpha_1}a^\dagger_{\alpha_2}|0\rangle \tag{2.27}$$

which can be true only if $a^\dagger_{\alpha_1}a^\dagger_{\alpha_1}a^\dagger_{\alpha_2}|0\rangle = 0$.

2.2.2 Boson operators

When dealing with bosons it is useful to adopt the occupation number formalism. In the bosonic case, we will not have any change of sign when interchanging particles, but we will need normalization factors. In the occupation number representation, the bosonic creation and annihilation operators may be defined as

$$a^\dagger_\alpha|n_\alpha, n_{\alpha'}, n_{\alpha''}...\rangle = \sqrt{n_\alpha+1}|n_\alpha+1, n_{\alpha'}, n_{\alpha''}...\rangle, \tag{2.28a}$$
$$a_\alpha|n_\alpha, n_{\alpha'}, n_{\alpha''}...\rangle = \sqrt{n_\alpha}|n_\alpha-1, n_{\alpha'}, n_{\alpha''}...\rangle. \tag{2.28b}$$

They satisfy the canonical commutation relations[4],

$$[a_{\alpha_j}, a^\dagger_{\alpha_k}] = \delta_{\alpha_j\alpha_k} \tag{2.29a}$$
$$[a_{\alpha_j}, a_{\alpha_k}] = [a^\dagger_{\alpha_j}, a^\dagger_{\alpha_k}] = 0 \tag{2.29b}$$

with the addition of $a_{\alpha_j}|0\rangle = 0$, just as in the fermionic case.

We now observe that we can rewrite any state with occupation numbers as

$$|n_{\alpha'}, n_{\alpha''}, n_{\alpha'''}, ...\rangle = \frac{1}{\sqrt{n_{\alpha'}!n_{\alpha''}!n_{\alpha'''}!...}}a^\dagger_{\alpha_1}a^\dagger_{\alpha_2}...a^\dagger_{\alpha_k}|0\rangle. \tag{2.30}$$

[4]Here $\delta_{\alpha_j\alpha_k} = 1$ if α_j and α_k refer to the same state.

When dealing with fermions or bosons we can use the operator form (2.22) and (2.30) respectively. We are then ensured by the commuting relations that the appropriate properties for each particle will hold when operating on these states.

2.2.3 The number operator

As the perhaps simplest example of an operator, we have the number operator:

$$N_\alpha = a_\alpha^\dagger a_\alpha. \tag{2.31}$$

It is easily shown using (2.28) for the bosonic case and (2.24) for the fermionic case that it satisfies

$$N_\alpha |n_\alpha n_{\alpha'} n_{\alpha''} ...\rangle = n_\alpha |n_\alpha n_{\alpha'} n_{\alpha''} ...\rangle. \tag{2.32}$$

Thus, the number operator N_α counts how many particles are occupying the state α.

2.2.4 One-body operators in Fock space

We will now describe how operators can be expressed in the second quantization formalism. Let O be an Hermitian operator that acts on a particular Hilbert space of one particle, i.e. a one-body operator. In order to keep track of what Hilbert space it refers to, we could label it as $O(i)$, meaning that it operates on the i-th particle. If we let $O(i)$ operate on a product state we can write it as

$$O(i)|\alpha_1\alpha_2...\alpha_k) = |\alpha_1\rangle \otimes ... \otimes O|\alpha_i\rangle \otimes ... \otimes |\alpha_k\rangle. \tag{2.33}$$

On the right side we omitted the index, because it loses its meaning when we put it in front of a single-particle state of the Hilbert space i to which it refers. This captivates the fact that the action imposed by $O(i)$, is O on the i-th particle, and is the same for all different is.

We are not in general interested in $O(i)$ but rather the sum of O over all particles, which we denote by O_k,

$$O_k = \sum_{i=1}^{k} O(i). \tag{2.34}$$

The action of O_k on a product state can then be written as

$$O_k|\alpha_1\alpha_2...\alpha_k) = \sum_{i=1}^{k} |\alpha_1\rangle \otimes ... \otimes O|\alpha_i\rangle \otimes ... \otimes |\alpha_k\rangle. \tag{2.35}$$

To continue, we expand O in its complete eigenspace spanned by $\{|\lambda\rangle\}$ with eigenvalues λ, i.e.[5]

$$O = \sum_{\lambda\lambda'} |\lambda\rangle\langle\lambda|O|\lambda'\rangle\langle\lambda'| = \sum_{\lambda} \lambda|\lambda\rangle\langle\lambda|. \tag{2.36}$$

where we used the *resolution of identity*[6]. If we let O_k act on a product state of its own eigenstates, we yield

$$O_k|\lambda_1\lambda_2...\lambda_k) = \sum_{i=1}^{k} |\lambda_1\rangle \otimes ... \otimes O|\lambda_i\rangle \otimes ... \otimes |\lambda_k\rangle = \left(\sum_{i=1}^{k} \lambda_i\right) |\lambda_1\lambda_2...\lambda_k). \tag{2.37}$$

[5]Here we use "$'$" as in λ' to denote different indexing from the same set $\{|\lambda\rangle\}$.

[6]For a complete set $\{|\alpha\rangle\}$ it holds that $\sum_\alpha |\alpha\rangle\langle\alpha|=1$.

So far we have dealt with product states. In order to expand the concept to symmetric and anti-symmetric states, we see that $[O_k,\mathcal{A}] = [O_k,\mathcal{S}] = 0$ since O_k must be symmetric and $\mathcal{A}$, $\mathcal{S}$ are linear combinations of P. For $\mathcal{A}$, remembering that $|\lambda_1\lambda_2...\lambda_k\rangle = \sqrt{k!}\mathcal{A}|\lambda_1\lambda_2...\lambda_k)$, we have

$$O_k|\lambda_1\lambda_2...\lambda_N\rangle = \sqrt{k!}\mathcal{A}O_k|\lambda_1\lambda_2...\lambda_k) = \left(\sum_{i=1}^{k}\lambda_i\right)|\lambda_1\lambda_2...\lambda_k\rangle. \tag{2.38}$$

Thus, $|\lambda_1\lambda_2...\lambda_k\rangle$ is an eigenstate of O_k with eigenvalue $\sum_{i=1}^{k}\lambda_i$. Since λ_i could refer to the same single-particle state, it is instructive to embrace the number representation formalism, that is $|\lambda_1\lambda_2...\lambda_k\rangle = |n_\lambda, n_{\lambda'}, n_{\lambda'',...}\rangle$ where $\lambda, \lambda'...$ range over all possible unique λ, and n_λ is the occupation number of this state. Since $n_\lambda = 0$ for a state not present, we can write

$$\sum_{i=1}^{k}\lambda_i = \sum_{\lambda}\lambda n_\lambda \tag{2.39}$$

where the last sum runs over all eigenvalues λ. Inserted into (2.38), we get

$$O_k|\lambda_1\lambda_2...\lambda_k\rangle = \left(\sum_{\lambda}\lambda n_\lambda\right)|\lambda_1\lambda_2...\lambda_k\rangle. \tag{2.40}$$

We recall from Section 2.2.3 that $n_\lambda|\lambda_1\lambda_2...\lambda_k\rangle = N_\lambda|\lambda_1\lambda_2...\lambda_k\rangle$ where $N_\lambda = a_\lambda^\dagger a_\lambda$. Thus, we can express O_k as

$$O_k = \sum_{\lambda}\lambda a_\lambda^\dagger a_\lambda. \tag{2.41}$$

In general we are not interested in the action of O_k on the eigenspace of itself, but rather in another complete space spanned by $\{|\alpha\rangle\}$. By the use of the resolution of identity we have

$$|\lambda\rangle = \sum_{\alpha}|\alpha\rangle\langle\alpha|\lambda\rangle. \tag{2.42}$$

The fact that $|\lambda\rangle = a_\lambda^\dagger|0\rangle$, makes it reasonable to write [1]

$$a_\lambda^\dagger = \sum_{\alpha}a_\alpha^\dagger\langle\alpha|\lambda\rangle \quad \text{and} \quad a_\lambda = \sum_{\alpha}a_\alpha\langle\lambda|\alpha\rangle. \tag{2.43}$$

Inserted into (2.41), we get

$$\begin{aligned} O_k = \sum_{\lambda}\lambda a_\lambda^\dagger a_\lambda &= \sum_{\lambda}\lambda\sum_{\alpha\beta}a_\alpha^\dagger\langle\alpha|\lambda\rangle\langle\lambda|\beta\rangle a_\beta \\ &= \sum_{\alpha\beta}\langle\alpha|\left(\sum_{\lambda}\lambda|\lambda\rangle\langle\lambda|\right)|\beta\rangle a_\alpha^\dagger a_\beta \\ &= \sum_{\alpha\beta}\langle\alpha|O|\beta\rangle a_\alpha^\dagger a_\beta. \end{aligned} \tag{2.44}$$

Here we have also used β to denote states in the set $\{|\alpha\rangle\}$.

Note that the right side of (2.44) makes no reference to the number of particles. This makes it plausible that operators expressed in this form could operate on any number of particles. It can be shown that this is true [2] and we call it a Fock space operator. We will distinguish Fock space operators from other operators using a "hat". To conclude, a one-body Fock space operator is expressed as

$$\hat{O} = \sum_{\alpha\beta} \langle\alpha|O|\beta\rangle a_\alpha^\dagger a_\beta. \tag{2.45}$$

A matrix element for the operator (2.45) is written as

$$\langle\psi_f|\hat{O}|\psi_i\rangle = \sum_{\alpha\beta} \langle\alpha|O|\beta\rangle \langle\psi_f|a_\alpha^\dagger a_\beta|\psi_i\rangle. \tag{2.46}$$

We observe that for a given pair of states ψ_i and ψ_f, the factor $\langle\psi_f|a_\alpha^\dagger a_\beta|\psi_i\rangle$ is independent of the specific action of $\hat{O}$ on the states; i.e. it is the same for all one-body operators. Thus, if we calculate it, the job is done for all one-body operators. We call

$$\langle\psi_f|a_\alpha^\dagger a_\beta|\psi_i\rangle \tag{2.47}$$

a *transition density matrix element*, and those elements are the specific aim of the computations in this thesis.

2.2.5 Two- and three-body operators in Fock space

One-body operators relate to properties of non-interacting particles, e.g. momentum, kinetic energy or potential energy. To describe interaction between two particles, we have to turn to two-body operators. A generalization of the procedure in the previous section, yields the following expression for a two-body operator [2]:

$$\hat{V} = \frac{1}{2} \sum_{\alpha\beta\alpha'\beta'} (\alpha\beta|V|\alpha'\beta') a_\alpha^\dagger a_\beta^\dagger a_{\alpha'} a_{\beta'}. \tag{2.48}$$

The factor 1/2 derives from the fact that we only want to count each interaction once.

Furthermore, in some cases we need to account for interactions involving three particles. A three-body operator is written as [2]

$$\hat{W} = \frac{1}{6} \sum_{\alpha\beta\gamma\alpha'\beta'\gamma'} (\alpha\beta\gamma|W|\alpha'\beta'\gamma') a_\alpha^\dagger a_\beta^\dagger a_\gamma^\dagger a_{\alpha'} a_{\beta'} a_{\gamma'}. \tag{2.49}$$

2.3 Matrix representations for commuting operators

The quantum mechanical problem of finding the eigenspace of an operator could be stated as a matrix eigenvalue problem. This is done by expanding the operator in a complete basis. The computational difficulties of a matrix eigenvalue problem is highly dependent of the dimension of the basis. This section motivates how the dimension can be reduced when commutation relations between the operators of interest are known beforehand.

Consider two Hermitian operators A and B, and a complete basis $|a^{(i)}\rangle$, being a degenerate set of eigenstates for A, with n_a-fold degeneracy for each eigenvalue a,

$$A|a^{(i)}\rangle = a|a^{(i)}\rangle \quad \text{for} \quad i = 1,2,...,n_a. \tag{2.50}$$

Let us say we want to the find the eigenstates for B, as a linear combination of the states $|a^{(i)}\rangle$. In order to do so, we form the matrix elements $\langle a'^{(i)}|B|a''^{(j)}\rangle$. If A and B commute, $[A,B] = 0$, we only need to consider the subsets $|a^{(i)}\rangle$ for different a one at a time. This is ensured by the following theorem:

Theorem 2.3.1. *Let A and B be Hermitian, commuting operators, and let $|a^{(i)}\rangle$ be a complete basis of degenerate eigenstates of A, $A|a^{(i)}\rangle = a|a^{(i)}\rangle$ for $i = 1,2,...,n_a$. Then B will not couple states with different a, that is $\langle a'^{(i)}|B|a''^{(j)}\rangle = 0$ if $a' \neq a''$.*

Proof. We use the fact that A and B commute and are Hermitian,

$$\begin{aligned} 0 &= \langle a'^{(i)}|[A,B]|a''^{(j)}\rangle = \langle a'^{(i)}|AB - BA|a''^{(j)}\rangle = \langle a'^{(i)}|AB|a''^{(j)}\rangle - \langle a'^{(i)}|BA|a''^{(j)}\rangle = \\ &= a'\langle a'^{(i)}|B|a''^{(j)}\rangle - a''\langle a'^{(i)}|B|a''^{(j)}\rangle = (a' - a'')\langle a'^{(i)}|B|a''^{(j)}\rangle. \end{aligned}$$

Thus we see that $\langle a'^{(i)}|B|a''^{(j)}\rangle \neq 0$ only if $a' = a''$. □

We still need to consider all a to find all eigenstates of B but these sets do not couple to each other. In terms of linear algebra we say that $\langle a'^{(i)}|B|a''^{(j)}\rangle$ is a block diagonal matrix and we could write it as a direct sum of the independent matrices where the set of eigenvalues and eigenstates is the sum of eigenvalues and eigenstates of every constituent matrix. In this way, we will end up with a set of smaller matrices instead of one block diagonal, the former being computationally preferable. The principle can be generalized to a set of mutually commuting operators.

Usually, the eigenstates of interest are those of a Hamiltonian, which plays the role of B. If we construct many-body states being eigenstates to another operator, playing the role of A, we may treat eigenstates for different eigenvalues one at a time, provided that the operator commutes with the Hamiltonian. In Section 2.1.1, we saw one such operator, the permutation operator, implying that we can treat symmetric and anti-symmetric states one at a time.

In the following chapter, we will have a closer look at the specific problem and define a Hamiltonian commuting with several other operators, making further use of Theorem 2.3.1.

Chapter 3

Problem description

In this chapter we will describe the model problem, a quantum mechanical many-body problem, with the primary example being an atomic nucleus. In nuclear systems, many complex interactions are involved, some of which are not yet fully understood, and a number of Hamiltonian structures have been proposed. The models have been able to explain some experimental results, but not all.

However, in our study we will deal with so-called *ab initio* computations[1], employing the no-core shell model, NCSM. In such computations, we know what specific approximations we have done since calculated eigenstates are those of a specific chosen nuclear Hamiltonian. In principle, *ab initio* computations should be able to reproduce all experimental results, provided that the appropriate physics is incorporated in the Hamiltonian. In this way, we can evaluate the choice of Hamiltonian since we can compare it to experimental results. However, a drawback of the method is the monstrous dimensions arising, calling for heavy computational power.

Although the entire *ab initio* process involves several computationally intense steps, the focus in this project will be the calculation of transition density matrix elements for one-body operators,

$$\langle \psi_f | a_\alpha^\dagger a_\beta | \psi_i \rangle. \tag{3.1}$$

We have further made the restriction to only consider systems with one kind of particle, e.g. neutrons. The generalization to systems with both protons and neutrons is fairly straight-forward, and essentially involves the addition of an extra quantum number, isospin. The generalization is further described in Chapter 8.

In this chapter, we will outline a NCSM-computation step by step, and give a simple example for clarity. We will explain most of the important steps in the full computation, but focus on the parts that are of interest in the calculation of transition densities. But to begin with, we give a brief explanation of the nuclear Hamiltonian, whose properties are important for the behaviour of the studied system, and thus for the computations as well.

[1]Computations from first principles, i.e. basic physical laws.

3.1 The nuclear Hamiltonian

To describe a nuclear system of interacting protons and neutrons in *ab initio* NCSM computations, the starting point is a Hamiltonian that is chosen to be translationally invariant and have the general structure [3]

$$\hat{H} = \frac{1}{A}\sum_{i<j}^{A}\frac{(\vec{p}_i - \vec{p}_j)^2}{2m} + \sum_{i<j}^{A} V_{\mathrm{NN}}(i,j) + \sum_{i<j<k}^{A} V_{\mathrm{NNN}}(i,j,k). \tag{3.2}$$

Here m is the average mass of the neutron and proton. In this Hamiltonian, there are one-, two- and even three-body operators. V_{NN} is the nucleon-nucleon interaction and contains a part that is approximately isospin invariant, i.e. it is the same between proton-neutron, proton-proton and neutron-neutron. The three-body interaction terms are described by $V_{\mathrm{NNN}}(i,j,k)$.

The Hamiltonian (3.2) commutes with the total angular momentum, parity and isospin. Hence, in accordance with Theorem 2.3.1, this Hamiltonian will only couple states with the same parity Π, total angular momentum J, angular momentum projection M and isospin. Since we will only deal with neutrons, we will not consider the isospin quantum number. It will then be sufficient to choose a basis with good quantum numbers J, M, Π to express the eigenstates of the Hamiltonian.

Here, we will construct a many-body basis with good M and Π but not necessarily good J. This is called the M-scheme. The angular momentum coupled J-scheme, which uses states with both good J and M, turns out to be computationally difficult to set up. M- and J-schemes are further discussed in Appendix A. When using this basis, we will consider one-body operators $\hat{O}$, which also commute with the total angular momentum and parity.

3.2 *Ab initio* no-core shell model computations

The final goal of *ab initio* many-body computations is often to compute expectation values of Fock-space operators for energy eigenstates, or transitions matrix elements between such eigenstates. Expressing the eigenstates as linear combinations of many-body basis states $|\nu\rangle$, i.e. $|\psi_i\rangle = \sum_\nu c^i_\nu|\nu\rangle$ and $|\psi_f\rangle = \sum_\nu c^f_\nu|\nu\rangle$, and restricting ourselves to one-body operators, we can write the target of the calculations as[2]

$$\langle\psi_f|\hat{O}|\psi_i\rangle = \sum_{\alpha\beta}\langle\alpha|O|\beta\rangle\langle\psi_f|a^\dagger_\alpha a_\beta|\psi_i\rangle = \sum_{\alpha\beta}\sum_{\nu_j\nu_k}\langle\alpha|O|\beta\rangle c^f_{\nu_j}c^i_{\nu_k}\langle\nu_j|a^\dagger_\alpha a_\beta|\nu_k\rangle. \tag{3.3}$$

The computation of this expression using NCSM essentially involves 6 steps:

1. Choose a single-particle basis, i.e. the set of states $|\alpha\rangle$, $|\beta\rangle$ in (3.3).
2. Construct a truncated many-body basis, i.e. the states $|\nu\rangle$, from the single-particle states.

[2]Since the Hamiltonian is real and symmetric, we can choose the eigenvectors to be real, and we need not bother with complex conjugates.

3. Construct the matrix representation of the Hamiltonian, with respect to the truncation of the many-body basis.
4. Diagonalize the Hamiltonian to find the energy-eigenstates of the system, i.e. the amplitudes c_ν for all many-body states in the basis.
5. Compute the transition density matrix elements $\langle \psi_f | a^\dagger_{\alpha_j} a_{\alpha_k} | \psi_i \rangle$.
6. Compute matrix elements for $\hat{O}$ using the transition densities.

In what follows, we will describe the process step by step. Our study mainly investigates the feasibility of using FPGA hardware for step 5, given that the results of the previous steps are at hand, but the algorithms could be generalized to also include step 3, which follows the same basic principles. We will also make a thorough discussion of step 1-2, whilst step 4 and 6 are not covered here. We will use a small example of a nucleus consisting of three neutrons to explain and expose the detailed calculations of the various steps.

3.2.1 Choice of single-particle basis – the harmonic oscillator

To begin with, we need a single-particle basis. We use the three-dimensional harmonic oscillator whose Hamiltonian is written as

$$\hat{H}_{\text{HO}} = \frac{\hat{p}^2}{2m} + \frac{m\Omega^2\hat{r}^2}{2} \tag{3.4}$$

with eigenstates $|nlm_l\rangle$. We extend the Hilbert space to include spin, $\mathcal{H}_{\text{HO}} \otimes \mathcal{H}_{\text{spin}}$. As a basis for this space we choose the coupled states

$$|nljm_j\rangle = \sum_{m_l m_s} (lm_l s m_s | jm_j)\, |nlm_l\rangle \otimes |sm_s\rangle, \tag{3.5}$$

where $(lm_l s m_s|jm_j)$ are Clebsch-Gordan coefficients. The single-particle representation $|\alpha\rangle$ in Chapter 2 should be identified with the quantum numbers discussed here, i.e. $|\alpha\rangle = |nljm_j\rangle$. For nucleons $s = 1/2$, and we will get $j = |l \pm 1/2|$. The eigenenergies are given by [1]

$$E_{nl} = \hbar\Omega\left(N + \frac{3}{2}\right) \quad \text{where} \quad N = 2n + l, \tag{3.6}$$

and the parity is given by

$$\pi_l = (-1)^l = (-1)^{N-2n} = (-1)^N. \tag{3.7}$$

The harmonic oscillator basis has an infinite dimension. When performing calculations, it must be truncated. This is done by defining a maximum number $N_{\text{single,max}}$ of energy quanta. For all energies from 0 to $N_{\text{single,max}}$ we form all possible combinations of n, l, j and m_j.

Table 3.1: Single-particle basis for $N_{\text{single,max}} = 2$. The first column defines indices used to identify the states, but the ordering is arbitrary.

Index	N	n	l	j	m_j	Index	N	n	l	j	m_j
1	0	0	0	1/2	-1/2	11	2	0	2	5/2	-5/2
2	0	0	0	1/2	1/2	12	2	0	2	5/2	-3/2
3	1	0	1	3/2	-3/2	13	2	0	2	5/2	-1/2
4	1	0	1	3/2	-1/2	14	2	0	2	5/2	1/2
5	1	0	1	3/2	1/2	15	2	0	2	5/2	3/2
6	1	0	1	3/2	3/2	16	2	0	2	5/2	5/2
7	1	0	1	1/2	-1/2	17	2	0	2	3/2	-3/2
8	1	0	1	1/2	1/2	18	2	0	2	3/2	-1/2
9	2	1	0	1/2	-1/2	19	2	0	2	3/2	1/2
10	2	1	0	1/2	1/2	20	2	0	2	3/2	3/2

Example. For $N = 0$ we must have $n = 0$ and $l = 0$ since $N = 2n + l$ and both are non-negative. Furthermore, we have $j = |l \pm 1/2|$. Hence, for $l = 0$ the only possibility is $j = 1/2$. Finally, we have $m_j = -j, -j+1,...,j-1,j$. For $N = 0$ we end up with two possibilities, $|nljm_j\rangle = |00\tfrac{1}{2}, \pm\tfrac{1}{2}\rangle$. In Table 3.1 we show all single-particle states with $N \leq 2$, i.e. we have chosen $N_{\text{single,max}} = 2$. ■

Finally, in a real computation, we would have to choose a harmonic oscillator energy, $\hbar\Omega$, for the basis. It is true that this is an arbitrary choice and that in the limit of an infinite many-body basis, all $\hbar\Omega$ should produce the same results. But by optimizing $\hbar\Omega$, the number of needed many-body states may be reduced, simplifying the computations. In actual computations, however, one usually does several calculations using different $\hbar\Omega$ to verify independence [4]. Henceforth, we will forget about the choice of $\hbar\Omega$, and care about the quantum number representation $|nljm_j\rangle$ alone.

3.2.2 Construction of a many-body basis

Given the single-particle basis, we will construct a many-body basis out of the single-particle states, as described in Chapter 2. Thanks to the Pauli exclusion principle, a single-particle state may occur only once in each many-body state, and many-body states ordered in different ways represent the same physical state.

Since a complete many-body basis would have infinite dimension, this basis will be truncated. Here we choose to define a maximum energy constraint N_{max} to our many-body states, i.e. the sum of the energies of the constituent single-particle states must not exceed a certain limit,

$$\sum_{i=1}^{A} N_i \leq N_{\text{tot}} \equiv N_0 + N_{\text{max}}, \tag{3.8}$$

where A is the number neutrons. Here $N_i = 2n_i + l_i$ are the energies of the constituent single-particle states and N_0 is the lowest possible energy allowed by the Pauli exclusion

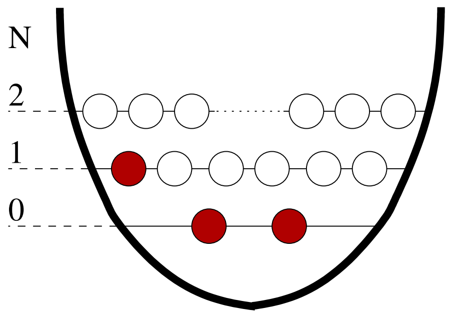


Figure 3.1: The potential well of an harmonic oscillator. To minimize the energy for three neutrons, two particles can have $N = 0$, but one must have $N = 1$, making $N_0 = 1$. For $N = 2$ there are 12 states, and for $N = 3$ there are 20.

principle. In general, N_0 depends on the number of protons and neutrons. The truncation (3.8) has been proven to induce fast convergence rates of expectation values [5]. Given the number of neutrons and N_{max} we could construct a single-particle basis with a generous energy bound of $N_{\text{single,max}} = N_{\text{tot}}$, so that the truncation of the single-particle basis will be no constraining factor when constructing the many-body basis. However, that would generally make the single-particle basis unnecessarily large. In the following example we describe a somewhat better approach.

Example. To calculate N_0, we simply sum up the energies of the A most low-energetic single-particle states. For $A = 3$ (neutrons only) the lowest possible energy will be $N_0 = 0 + 0 + 1 = 1$, using the energies of the first three states in Table 3.1, illustrated by the harmonic oscillator potential well in Figure 3.1. Thus, by choosing $N_{\text{max}} = 1$ we will have $N_{\text{tot}} = N_0 + N_{\text{max}} = 1 + 1 = 2$. To choose $N_{\text{single,max}}$, we assign the most low-energetic single-particle states to the $A - 1$ first particles, and then investigate how much energy that might be assigned to the last particle while still fulfilling the energy constraint (3.8). In this case, since the first two single-particle states have energy 0, we will have $N_{\text{single, max}} = N_{\text{tot}} - 0 - 0 = 2$, but with $A > 3$ (neutrons only) there will be at least one energy quantum in the first $A - 1$ states, and thus $N_{\text{single,max}} < N_{\text{tot}}$. ■

Given the single-particle basis, we construct the many-body basis as all possible combinations of A single-particle states, obeying the Pauli exclusion principle and the energy constraint (3.8). However, not all of these combinations are of interest. Since we are working within the M-scheme, with parity commuting operators, we will construct basis states:

- with the same parity

$$\Pi = \prod_{i=1}^{A} \pi_i = (-1)^{\sum_{i=1}^{A} l_i} = (-1)^{\sum_{i=1}^{A} N_i}, \tag{3.9}$$

  positive or negative; and

- with the same sum of individual m_js, i.e. having the same total angular momentum projection,

$$M = \sum_{i=1}^{A} m_{ji}. \tag{3.10}$$

In Appendix A we show that the many-body basis constructed in Chapter 2 by single-particles states with good jm_j actually has good M. For even A, we will usually choose $M = 0$, and for odd A usually $M = \pm 1/2$.

The dimensions of the many-body bases are shown in Figure 3.2 for positive parity and $M = 0$ or $1/2$, depending on whether the number of neutrons is even or odd. In order to reach convergence for expectation values, an N_{max} of at least 8 is usually required, but sometimes possibly 10 or even more [5]. Apparently, the dimensions explode already for fairly small nuclei. One should recall that these bases take account of only one kind of particle. When handling both protons and neutrons, the dimensions are even higher for fixed N_{max} and A, and the dimensions may exceed the dimensions of our bases by several orders of magnitude [6].

Example. To create the many-body basis out of the single-particle basis in Table 3.1, for $A = 3$ (neutrons only) and $N_{\text{max}} = 1$, we could start by forming all possible combinations satisfying (3.8) and the Pauli exclusion principle,

$$[1,2,3], \quad [1,2,4], \quad [1,2,5], \dots \quad [2,7,8] \tag{3.11}$$

It turns out that there are 48 such states. However most of those do not satisfy either the parity or the M constraint. For example, if we choose the parity to be positive, the state [1,2,3] is invalid since $(-1)^{0+0+1} = -1$. In this case, the parity disqualifies six states. Further, if we choose $M = 1/2$, the state [1,2,4] is invalid $(-1/2 + 1/2 - 1/2 = -1/2)$ but [1,2,5] passes the test $(-1/2+1/2+1/2 = 1/2)$. The M constraint alone disqualifies all but 13 states for $M = 1/2$. We end up with 11 valid states fulfilling all conditions;

$$\begin{array}{llllll} [2,4,5], & [2,3,6], & [1,4,6], & [2,5,7], & [1,6,7], & [2,4,8], \\ [1,5,8], & [2,7,8], & [1,2,10], & [1,2,14], & [1,2,19]. & \end{array} \tag{3.12}$$

■

In our example, the M constraint was more restrictive than the parity. This is by no means a surprise since all states will have either positive or negative parity, while the sum of m_j:s varies more extensively. In Figure 3.3 we see how the constraints affect the size of the basis, for $A = 6$, positive parity and $M = 0$, and different N_{max}. We should keep in mind that even though we have only plotted the constraints in the particular case of $M = 0$, another choice of M would be even more restricting since it is a well known fact from combinatorics.

3.2.3 Identifying connections using a one-body operator

Given the many-body basis, the next step of the *ab initio* computation would be to construct the matrix representation of the Hamiltonian. Since we are exclusively interested

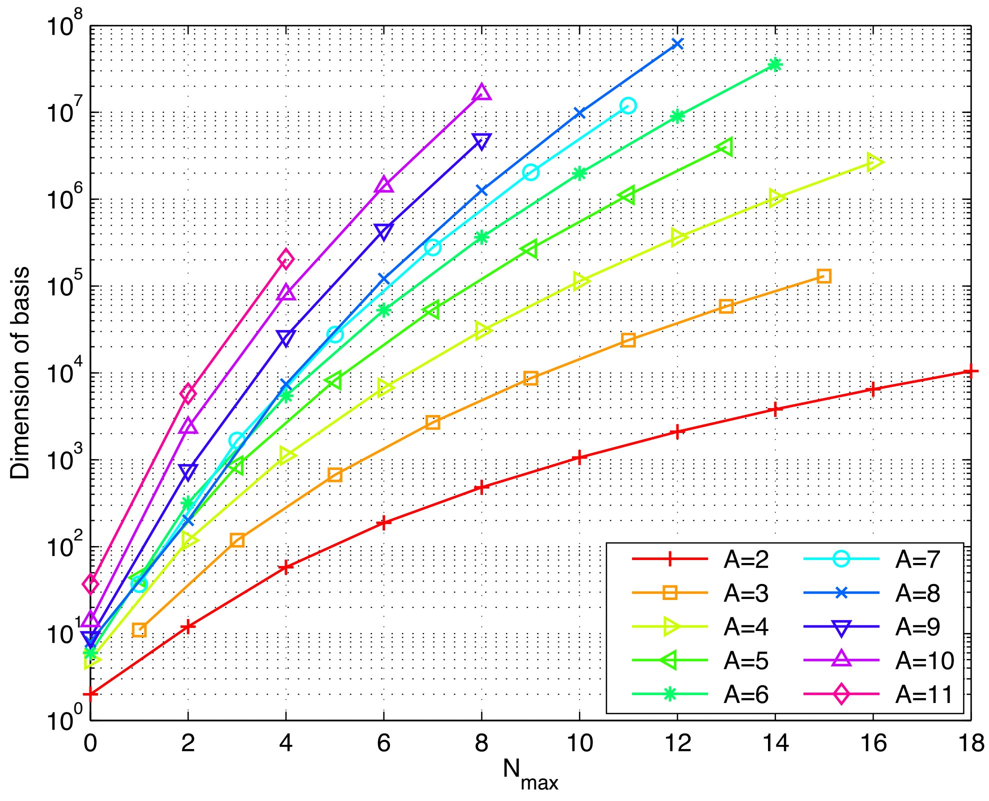


Figure 3.2: Size of the many-body basis for different numbers of neutrons and positive parity, as a function of N_{max}. The sum of m_js was set to $M = 0$ and $1/2$ for even and odd numbers of neutrons, respectively. Due to the positive parity, an odd N_{tot} will result in the same basis as for an N_{tot} with one energy quantum less. Hence, the bases were constructed for even N_{tot} only. Note that the x-axis is for $N_{\text{max}} = N_{\text{tot}} - N_0$. The plots for 3,5 and 7 neutrons all have odd N_{max}, because they correspond to odd N_0.

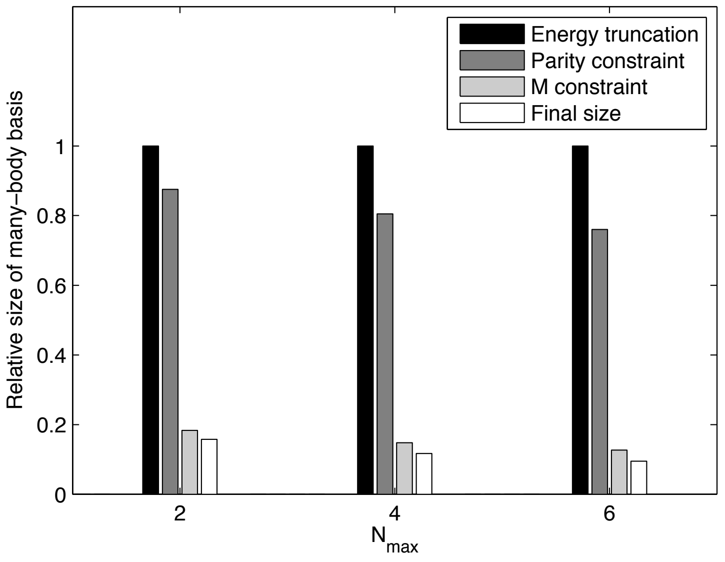


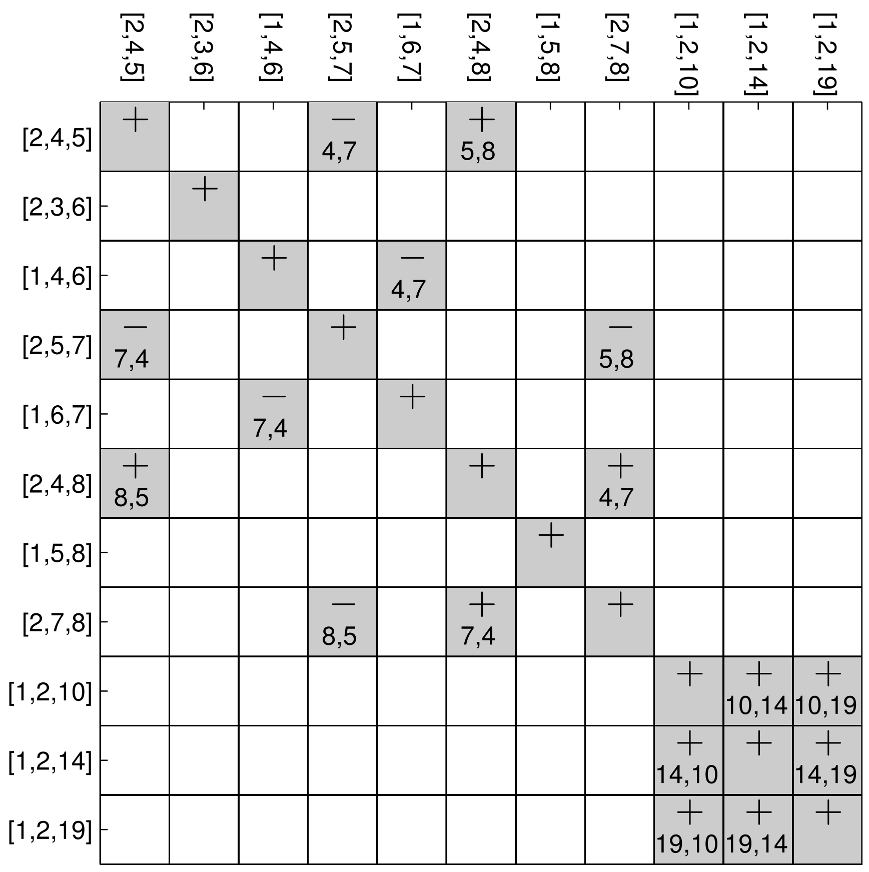
Figure 3.3: Reduction of many-body states due to the parity and M constraint, for $A = 6$, neutrons only. Here we chose positive parity and $M = 0$. The parity and M constraint were applied independently to the energy truncated basis, so that the bars show the size of the basis after applying the constraint, relative to the energy truncated basis. In this case, the M constraint was by far the most restrictive.

in the calculation of transition densities, we assume that this step has already been accomplished, and therefore proceed to step 5, i.e. the computation of transition density matrix elements. To do this computation, the method is to find all connections between many-body states for a one-body operator. A *connection* between the many-body states $|\nu_j\rangle$ and $|\nu_k\rangle$ is defined by that the element

$$\langle \nu_j | a_\alpha^\dagger a_\beta | \nu_k \rangle \tag{3.13}$$

is non-zero. This is what is known as a transition rule, i.e. a condition for (3.13) to be non-zero, according to which the many-body states $|\nu_j\rangle$ and $|\nu_k\rangle$ can differ by at most one single-particle state.

In Figure 3.4 we show how the many-body states of our example connect. The grey squares represent a connection. The numbers in the respective squares indicate what single-particle states have been annihilated and created in order to find the connection, using the indices of the single-particle basis. The diagonal connections are accomplished by replacing any of the included single-particle states with itself. Further, when annihilating one single-particle state, and creating a new, the resulting many-body state representation may need to be sorted to maintain the ascending order of single-particle states. As described in Chapter 2, this permutation may introduce a negative sign. In practice, the sign is determined by how many steps we have to move the inserted state. If we insert the new state at place i, and have to move it to place j, the sign will be $(-1)^{j-i}$. In the grey squares, it is indicated whether each connection comes with positive or negative sign.



	[2,4,5]	[2,3,6]	[1,4,6]	[2,5,7]	[1,6,7]	[2,4,8]	[1,5,8]	[2,7,8]	[1,2,10]	[1,2,14]	[1,2,19]
[2,4,5]	+			− 4,7		+ 5,8					
[2,3,6]		+									
[1,4,6]			+		− 4,7						
[2,5,7]	− 7,4			+				− 5,8			
[1,6,7]			− 7,4		+						
[2,4,8]	+ 8,5					+		+ 4,7			
[1,5,8]							+				
[2,7,8]				− 8,5		+ 7,4		+			
[1,2,10]									+	+ 10,14	+ 10,19
[1,2,14]									+ 14,10	+	+ 14,19
[1,2,19]									+ 19,10	+ 19,14	+

Figure 3.4: Connections in the many-body basis of our example, for a one-body operator. Grey square represents a connection, and the numbers in the square indicate what single-particle states have been changed, referring to the indices of Table 3.1. On the diagonal, the connections are made by switching any of the included single-particle states for itself. It is also indicated whether the connection comes with positive or negative sign.

Example. Let us find all connections involving [2,4,5] in our example basis. To begin with, a many-body state always connects to itself, with a positive sign. This is accomplished by annihilating any of the present single-particle states, and creating it again. Further, [2,4,5] does not connect to either [2,3,6] or [1,4,6] since more than one single-particle state differs, and we are dealing with a one-body operator. On the other hand, [2,4,5] does connect to [2,5,7]. If we annihilate 4 and create 7, we get the state [2,7,5]. Since [2,7,5] is not in ascending order, it needs to be sorted. We need to move 7 from place 2 to place 3, introducing the sign $(-1)^{3-2} = -1$. The new state is therefore $-$ [2,5,7] and we have found a connection with a negative sign. Between [2,4,5] and [1,6,7], all states differ and we have no connection. Finally the last connection for [2,4,5] is to [2,4,8], if we annihilate 5 and create 8. The new state does not need to be sorted, and we have found a connection with a positive sign. ■

The brute-force approach used in the above example, of manual comparison of all many-body states with one another, will be horribly lengthy in a real-world computation. For a many-body basis of dimension 10^9 we would have to do 10^{18} comparisons, most of which would be unsuccessful. A better approach would be to use an algorithm that starts with a many-body state, and constructs new many-body states fulfilling the transition rule (i.e. states differing with no more than one single-particle state). If the constructed many-body states are valid (i.e. fulfills the Pauli exclusion principle, energy truncation, has the correct momentum projection as well as parity), we know they reside somewhere in our basis, and represent valid connections. This is the very heart of what should be accomplished on the FPGA, and detailed algorithms will be developed in Chapter 6.

In Figure 3.4, the matrix may seem not so sparse, but in a real-world computation, involving a many-body basis of millions or billions of elements, the matrix will be much more sparse. The connections will therefore typically not be stored in matrix-form with a lot of zeroes, but rather in some kind of list of all connections. The list should contain the following information about each connection:

- The product of the probability amplitudes of the connecting many-body states, i.e. $c^f_{\nu_j} c^i_{\nu_k}$.
- The annihilated and created single-particle states β and α, used to accomplish the connection.
- The sign of the connection.

Having this information, the next step is to compute the transition density matrix elements.

3.2.4 Calculate transition densities for a one-body operator

Recalling (3.3), a one-body transition density matrix element is given as

$$\langle\psi_f|a^\dagger_\alpha a_\beta|\psi_i\rangle = \sum_{\nu_j\nu_k} c_{\nu_j} c_{\nu_k} \langle\nu_j|a^\dagger_\alpha a_\beta|\nu_k\rangle. \tag{3.14}$$

This means that for each pair of single-particle states α and β, we must sum coefficients of the type $c_{\nu_j} c_{\nu_k}$, referred to as probability amplitudes. Some will come with a plus and some with a minus, according to the sign of $\langle\nu_j|a^\dagger_\alpha a_\beta|\nu_k\rangle$.

When having all transition density matrix elements computed, the observable can be calculated by doing the full sum in (3.3). However, the sum is not trivial because there might be thousands of transition density matrix elements in a real-world example. Therefore, the number of elements should be reduced, if possible, and be ordered in a carefully prepared way. This topic is addressed in the following chapter.

Chapter 4

Reduction of matrix elements and transition densities

As the dimension of the many-body basis explodes, not only does the computations become a headache, but also the form in which we present the results. Although the results of the computation, i.e. the transition density matrix elements

$$\langle \psi_f | a_\alpha^\dagger a_\beta | \psi_i \rangle, \tag{4.1}$$

contain all the information we might be interested in, there will be a huge amount of them. A real computation might include a couple of hundred single-particle states, so that we would have tens of thousands of densities of type (4.1), and most of them would not contribute to the computations of a given observable. However, thanks to certain symmetries in the physics of the system, there are ways to reduce the amount of transition density matrix elements, and group them into physically relevant categories.

This chapter briefly describes the physics of angular momenta and spherical tensors, leading to the concept of reduced matrix elements and reduced transition densities, which we may use to present the results in a more useful way. We conclude with an example of how this presentation might be arranged.

4.1 Clebsch-Gordan coefficients

Let $\mathbf{J}_1$ and $\mathbf{J}_2$ be two commuting angular momentum operators and let $|j_1 m_1\rangle$ and $|j_2 m_2\rangle$ be eigenstates of $\mathbf{J}_1^2$, $J_{1,z}$ and $\mathbf{J}_2^2$, $J_{2,z}$, respectively (J_z is the projection of the angular momentum on the z-axis). We know that the product states

$$|j_1 m_1 j_2 m_2\rangle = |j_1 m_1\rangle |j_2 m_2\rangle \tag{4.2}$$

are simultaneous eigenstates to the complete operator set

$$\{\mathbf{J}_1^2, J_{1,z}, \mathbf{J}_2^2, J_{2,z}\}. \tag{4.3}$$

However, states of the type (4.2) do not correspond to well-defined angular momenta in the coupled system, i.e. the states (4.2) are not eigenstates to the total angular momentum operator

$$\mathbf{J}^2 = (\mathbf{J}_1 + \mathbf{J}_2)^2 \,. \tag{4.4}$$

The complete set of states $\{|j_1 m_1 j_2 m_2\rangle\}$ is thus called the uncoupled basis.

Although the operator set (4.3) is indeed complete, the complete operator set including the total angular momentum,

$$\{\mathbf{J}_1^2, \mathbf{J}_2^2, \mathbf{J}^2, J_z\}, \tag{4.5}$$

exhibits more charming properties. The eigenstates to this operator set are called the coupled basis states, and may be constructed as linear combinations of the uncoupled basis states,

$$|j_1 j_2 j m\rangle = \sum_{m_1,m_2} |j_1 m_1 j_2 m_2\rangle\langle j_1 m_1 j_2 m_2|j_1 j_2 j m\rangle \equiv \sum_{m_1,m_2} (j_1 m_1 j_2 m_2|jm)\,|j_1 m_1 j_2 m_2\rangle, \tag{4.6}$$

where we have defined the Clebsch-Gordan coefficients,

$$\langle j_1 m_1 j_2 m_2|j_1 j_2 j m\rangle \equiv (j_1 m_1 j_2 m_2|jm) \equiv C^{jm}_{j_1 m_1 j_2 m_2}. \tag{4.7}$$

The Clebsch-Gordan coefficients vanish unless

$$|j_1 - j_2| \leq j \leq j_1 + j_2 \quad \text{and} \quad m = m_1 + m_2 \tag{4.8}$$

as is expected from the vector addition in (4.4). The values of the coefficients are easiest found in tables, but if one wishes to actually compute them, the standard methods involve the following recursion relation [1] (which we will need later on),

$$\begin{aligned} &\sqrt{(j \pm m)(j \mp m + 1)}\,(j_1 m_1 j_2 m_2|jm \mp 1) = \\ &= \sqrt{(j_2 \mp m_2)(j_2 \pm m_2 + 1)}\,(j_1 m_1 j_2 m_2 \pm 1|jm) + \\ &+ \sqrt{(j_1 \mp m_1)(j_1 \pm m_1 + 1)}\,(j_1 m_1 \pm 1, j_2 m_2|jm)\,. \end{aligned} \tag{4.9}$$

4.2 Spherical tensors

When dealing with tensors, we usually represent them in terms of Cartesian coordinates. Although the Cartesian representation might be the conceptually most straight-forward, it has some disadvantages. For example, when rotating a tensor, it is often important to keep track of symmetries, but in the Cartesian representation such characteristics are generally hidden. However, a Cartesian tensor may be reduced to spherical tensor components, exhibiting different rotational behaviour so that the symmetries of the tensor are more explicit. Spherical tensors will be a central concept in order to reduce the transition density matrix elements and present them in purposive groups.

In order to define spherical tensors it is instructive to start with ordinary tensors. From classical physics we recall that a vector is what transforms as a vector under rotation, by definition $V_i' = \sum_j R_{ij} V_j$. We can generalize this to an arbitrary number of indices, called tensors, which transform as $T'_{ijk...} = \sum_{i'j'k'...} R_{ii'} R_{jj'} R_{kk'} ... T_{i'j'k'...}$ under rotation. Here R are orthogonal matrices producing the rotation.

We now turn to quantum mechanics in which the unitary rotational operator U describes how states transform under rotation, $|\alpha'\rangle = U|\alpha\rangle$. For an operator O, we define the rotated operator O' by requiring that the expectation value of the rotated operator acting on a rotated state is the same as without rotation, i.e.

$$\langle\alpha|O|\alpha\rangle = \langle\alpha'|O'|\alpha'\rangle. \tag{4.10}$$

But for the rotated states we have

$$\langle\alpha'|O'|\alpha'\rangle = \langle\alpha|U^\dagger O' U|\alpha\rangle \tag{4.11}$$

so that $O = U^\dagger O' U$. Thus, since U is unitary, an operator transforms as

$$O' = UOU^\dagger. \tag{4.12}$$

Now, to define spherical tensor operators, we proceed in the same way as for Cartesian tensors – we define them by how they transform. We want our spherical tensors to transform in the same way as eigenstates $|jm\rangle$ of angular momentum, i.e.

$$U|jm\rangle = \sum_{m'=-j}^{j} |jm'\rangle\langle jm'|U|jm\rangle = \sum_{m'=-j}^{j} D^j_{m'm}(R)|jm'\rangle \tag{4.13}$$

where $D^j_{m'm}(R)$ is a so-called Wigner D-function [7]. We are now ready to define the spherical tensors. By definition, a spherical tensor T_{JM} transforms as [1]

$$UT_{JM}U^\dagger = \sum_{M'=-J}^{J} T_{JM} D^J_{M'M}(R). \tag{4.14}$$

Here $\mathbf{T}_J$ is a spherical tensor of rank J, that is, it has $2J+1$ components T_{JM} with $M = -J, -J+1,..., J$, i.e. projections on the z-axis. We see from the definition that the components T_{JM} couples within themselves under rotation just as the eigenstates of angular momentum.

An equivalent definition of spherical tensors can be made using the commutators with the angular momentum operators[1]

$$[J_z, T_{JM}] = MT_{JM} \tag{4.15}$$

$$[J_\pm, T_{JM}] = \sqrt{(J \mp M)(J \pm M + 1)}T_{J,M\pm1} \tag{4.16}$$

where $J_\pm = J_x \pm iJ_y$ are the so-called ladder operators [1].

The perhaps simplest example of spherical tensors is the coordinate representation of $|jm\rangle$, the spherical harmonics $Y_{lm}(\mathbf{n})$, with $\mathbf{n}$ replaced by a vector operator V_i, i.e. $T_{JM} = Y_{l=J,m=M}(V_i)$. An important property of $Y_{lm}(\mathbf{n})$ is that it is irreducible. This holds for T_{JM} as well [1]. By irreducible we mean that the tensor operator spans the smallest possible space of operators that is closed under rotation [8]. These spaces have the dimension $2J+1$.

The irreducibility is an advantage over Cartesian operators which in general are reducible. The Cartesian operators can however be reduced into spherical tensors. For example, a Cartesian tensor formed by the multiplication of the components of two vectors, $T_{ij} = U_iV_j$ may be reduced to three spherical tensors of rank J = 0,1 and 2. We see that the dimensions add up $3 \times 3 = 1 + 3 + 5 = 9$ [1]. The Cartesian tensor equals the sum of three spherical tensors of different rank, whose individual behaviour under rotation is known from (4.14).

[1]From now on we set $\hbar = 1$.

Example. Gamma transition of a nucleus is mediated by multipole components of the radiation field. We categorize them as electric Q_{LM} and magnetic M_{LM} multipole transitions. The electric multipole transition is written as [9]

$$Q_{LM} = i^L \sum_{j=1}^{A} q_j r_j^L Y_{LM}(\mathbf{n}_j) \tag{4.17}$$

where the sums run over all nucleons and q_j is the charge. The rank of the operator answers for how much angular momentum the emitted photon carries with it. Here $J = 0$ is not allowed since the photon must carry at least one angular momentum quantum. ■

When dealing with spherical tensors one must know how they can be combined into new spherical tensors. With two spherical tensors we may construct a new spherical tensor by the use of the following theorem (a proof can be found in e.g. [1]):

Theorem 4.2.1. *Let $X_{J_1M_1}$ and $Y_{J_2M_2}$ be two spherical tensors. Then the product*

$$[\mathbf{X}_{J_1}\mathbf{Y}_{J_2}]_{JM} \equiv \sum_{M_1}\sum_{M_2} (J_1M_1J_2M_2|JM)\, X_{J_1M_1}Y_{J_2M_2} \tag{4.18}$$

is also a spherical tensor.

We have seen that spherical tensors transform like eigenstates of the total angular momentum, $|jm\rangle$. Just as is the case of $|jm\rangle$, spherical tensors are closely related to Clebsch-Gordan coefficients. In the next section, we will see how.

4.3 The Wigner-Eckart theorem

We are now ready to introduce the Wigner-Eckart theorem, which states that the matrix element of a spherical tensor can be factorized into one part containing the geometry of the system, expressed by Glebsch-Gordan coefficients, and one rotationally invariant part.

Theorem 4.3.1. *Let T_{JM} be a spherical tensor and let $|\xi jm\rangle$ be a state with good angular momentum j and projection m (ξ denotes other quantum numbers needed for completeness). Then, the matrix element $\langle \xi_1 j_1 m_1 | T_{JM} | \xi_2 j_2 m_2\rangle$ can be factorized as*

$$\langle \xi_1 j_1 m_1 | T_{JM} | \xi_2 j_2 m_2\rangle = \frac{(j_2m_2JM|j_1m_1)}{\hat{j}_1} \langle \xi_1 j_1 || \boldsymbol{T}_J || \xi_2 j_2\rangle. \tag{4.19}$$

where $\langle \xi_1 j_1 || \boldsymbol{T}_J || \xi_2 j_2\rangle$ is independent of m_1, m_2 and M, and where the notationally convenient hat factor, $\hat{j}_1 = \sqrt{2j_1+1}$, has been introduced.

The Wigner-Eckart theorem is an existence theorem stating that the matrix element of a spherical tensor may be divided into two parts. The factor $\langle \xi_1 j_1 || \mathbf{T}_J || \xi_2 j_2\rangle$ makes no reference to the orientation of the system. The geometrical dependency is instead absorbed in the Clebsch-Gordan coefficent and we can reproduce $\langle \xi_1 j_1 m_1 | T_{JM} | \xi_2 j_2 m_2\rangle$ by knowing that it is proportional to $\langle \xi_1 j_1 || \mathbf{T}_J || \xi_2 j_2\rangle$ times the proper Clebsch-Gordan coefficient. It should be clear that we can evaluate $\langle \xi_1 j_1 || \mathbf{T}_J || \xi_2 j_2\rangle$ for arbitrary m_1, m_2 and M, as it is independent of them.

Proof. By the use of (4.16), we have

$$\langle \xi_1 j_1 m_1 | \, [J_\pm, T_{JM}] \, | \xi_2 j_2 m_2 \rangle = \sqrt{(J \mp M)(J \pm M + 1)} \langle \xi_1 j_1 m_1 | T_{JM\pm 1} | \xi_2 j_2 m_2 \rangle. \quad (4.20)$$

Further, it can be shown that the action of the ladder operators on a state $|\xi jm\rangle$ is to raise/lower the projection number [1],

$$J_\pm |\xi jm\rangle = \sqrt{(j \mp m)(j \pm m + 1)} |\xi jm \pm 1\rangle \quad (4.21)$$

so that (4.20) can be rewritten as

$$\begin{aligned} \sqrt{(j_1 \pm m_1)(j_1 \mp m_1 + 1)} \langle \xi_1 j_1 m_1 \mp 1 | T_{JM} | \xi_2 j_2 m_2 \rangle = \\ = \sqrt{(j_2 \mp m_2)(j_2 \pm m_2 + 1)} \langle \xi_1 j_1 m_1 | T_{JM} | \xi_2 j_2 m_2 \pm 1 \rangle + \\ + \sqrt{(J \mp M)(J \pm M + 1)} \langle \xi_1 j_1 m_1 | T_{JM\pm 1} | \xi_2 j_2 m_2 \rangle. \end{aligned} \quad (4.22)$$

This recursion relation is exactly the same as the one for Clebsch-Gordan coefficients (4.9), if we substitute $j \to j_1$, $m \to m_1$, $j_1 \to J$ and $m_1 \to M$. Since both the matrix elements and the Clebsch-Gordan coefficients follow the same recursion relation, the ratios between them must be the same for corresponding sets of m_1, m_2 and M. That is, they differ only by a constant indpedent of m_1, m_2 and M. By identifying for example the last term in (4.22) with the last term in (4.9), and substituting $M - 1 \to M$, we may write

$$\langle \xi_1 j_1 m_1 | T_{JM} | \xi_2 j_2 m_2 \rangle = (j_2 m_2 JM | j_1 m_1) \times \text{const. indep. of } m_1,\, m_2 \text{ and } M, \quad (4.23)$$

which proves the theorem. □

4.4 Reduced one-body transition densities

We will now show how the Wigner-Eckart theorem can be used in second quantization formalism to express matrix elements for a one-body operator in a more practical form. We will represent our quantum numbers with $|\alpha\rangle = |\xi jm\rangle$. We have seen that we can expand an arbitrary one-body operator as

$$\hat{O} = \sum_{\xi_1 j_1 m_1 \xi_2 j_2 m_2} \langle \xi_1 j_1 m_1 | O | \xi_2 j_2 m_2 \rangle a^\dagger_{\xi_1 j_1 m_1} a_{\xi_2 j_2 m_2} \quad (4.24)$$

Now, consider a spherical tensor $\hat{T}_{JM}$. We may express the operator in the same way, and then apply the Wigner-Eckart theorem,

$$\begin{aligned} \hat{T}_{JM} &= \sum_{\xi_1 j_1 m_1 \xi_2 j_2 m_2} \langle \xi_1 j_1 m_1 | T_{JM} | \xi_2 j_2 m_2 \rangle a^\dagger_{\xi_1 j_1 m_1} a_{\xi_2 j_2 m_2} \\ &= \sum_{\xi_1 j_1 m_1 \xi_2 j_2 m_2} \hat{j}_1^{-1} \, (j_2 m_2 JM | j_1 m_1) \, \langle \xi_1 j_1 || \mathbf{T}_J || \xi_2 j_2 \rangle a^\dagger_{\xi_1 j_1 m_1} a_{\xi_2 j_2 m_2} \\ &= \sum_{\xi_1 j_1 \xi_2 j_2} \hat{j}_1^{-1} \langle \xi_1 j_1 || \mathbf{T}_J || \xi_2 j_2 \rangle \sum_{m_1 m_2} (j_2 m_2 JM | j_1 m_1) \, a^\dagger_{\xi_1 j_1 m_1} a_{\xi_2 j_2 m_2}. \end{aligned} \quad (4.25)$$

The operator is now factorized in one m-independent part, and one part which is independent of the operator's action on single particle states. We will now see that we can rearrange some terms in the last sum to construct a new spherical tensor. By the use of (4.16) and $J_z, J_\pm$ expressed in second quantization formalism one can show that $a^\dagger_{\xi jm}$ and $\tilde{a}_{\xi jm} = (-1)^{j+m} a_{\xi j,-m}$ are spherical tensors, but $a_{\xi jm}$ is not [7]. We will therefore express the last sum of (4.25) in terms of $a^\dagger_{\xi jm}$ and $\tilde{a}_{\xi jm}$,

$$\begin{aligned}
&\sum_{m_1 m_2} (j_2 m_2 J M | j_1 m_1)\, a^\dagger_{\xi_1 j_1 m_1} a_{\xi_2 j_2 m_2} \\
&= \sum_{m_1 m_2} \frac{1}{(-1)^{j_2, -m_2}} (j_2 m_2 J M | j_1 m_1)\, a^\dagger_{\xi_1 j_1 m_1} \tilde{a}_{\xi_2 j_2, -m_2} \\
&= \sum_{m_1 m_2} \frac{1}{(-1)^{j_2 + m_2}} (j_2, -m_2 J M | j_1 m_1)\, a^\dagger_{\xi j_1 m_1} \tilde{a}_{\xi j_2 m_2}.
\end{aligned} \tag{4.26}$$

We now use the following identity for Clebsch-Gordan coefficients [7]:

$$(j_2 m_2 J M | j_1 m_1) = (-1)^{j_2, -m_2} \frac{\hat{j}_1}{\hat{J}} (j_1 m_1 j_2, -m_2 | J M) \tag{4.27}$$

which gives

$$\begin{aligned}
&\sum_{m_1 m_2} \frac{1}{(-1)^{j_2 + m_2}} (j_2, -m_2 J M | j_1 m_1)\, a^\dagger_{\xi_1 j_1 m_1} \tilde{a}_{\xi_2 j_2 m_2} \\
&= \sum_{m_1 m_2} \frac{1}{(-1)^{j_2 + m_2}} (-1)^{j_2 + m_2} \frac{\hat{j}_1}{\hat{J}} (j_1 m_1 j_2 m_2 | J M)\, a^\dagger_{\xi_1 j_1 m_1} \tilde{a}_{\xi_2 j_2 m_2} \\
&= \sum_{m_1 m_2} \frac{\hat{j}_1}{\hat{J}} (j_1 m_1 j_2 m_2 | J M)\, a^\dagger_{\xi_1 j_1 m_1} \tilde{a}_{\xi_2 j_2 m_2}.
\end{aligned} \tag{4.28}$$

By comparison with Theorem 4.2.1 we see that this constitutes a spherical tensor, that is

$$\begin{aligned}
&\sum_{m_1 m_2} (j_2 m_2 J M | j_1 m_1)\, a^\dagger_{\xi_1 j_1 m_1} a_{\xi_2 j_2 m_2} \\
&= \frac{\hat{j}_1}{\hat{J}} \sum_{m_1 m_2} (j_1 m_1 j_2 m_2 | J M)\, a^\dagger_{\xi_1 j_1 m_1} \tilde{a}_{\xi_2 j_2 m_2} \\
&= \frac{\hat{j}_1}{\hat{J}} \left[a^\dagger_{\xi_1 j_1} \tilde{a}_{\xi_2 j_2} \right]_{JM}
\end{aligned} \tag{4.29}$$

where we have used the "bracket notation" of Theorem 4.2.1. Insertion into (4.25) gives

$$\hat{T}_{JM} = \frac{1}{\hat{J}} \sum_{\xi_1 j_1 \xi_2 j_2} \langle \xi_1 j_1 || \mathbf{T}_J || \xi_2 j_2 \rangle \left[a^\dagger_{\xi j_1} \tilde{a}_{\xi j_2} \right]_{JM} \tag{4.30}$$

which is a spherical tensor-equivalent of (4.24) with the advantage of less references to the geometry.

We now want to find a matrix element of $\hat{T}_{JM}$. Let us say we have a many-body state $|\lambda J M\rangle$ which is eigenstate to J^2 and J_z (here λ denotes other quantum numbers for many-body states, equivalent to ξ for single particle states). Then, by the use of (4.30)

$$\begin{aligned}\langle\lambda_f J_f M_f|\hat{T}_{JM}|\lambda_i J_i M_i\rangle &= \\ &= \frac{1}{\hat{J}}\sum_{\xi_1 j_1 \xi_2 j_2}\langle\xi_1 j_1||\mathbf{T}_J||\xi_2 j_2\rangle\langle\lambda_f J_f M_f|\left[a^{\dagger}_{\xi_1 j_1}\tilde{a}_{\xi_2 j_2}\right]_{JM}|\lambda_i J_i M_i\rangle.\end{aligned} \tag{4.31}$$

Now, since $\left[a^{\dagger}_{\xi_1 j_1}\tilde{a}_{\xi_2 j_2}\right]_{JM}$ is a spherical tensor, we are allowed to use the Wigner-Eckart theorem for the last factor,

$$\begin{aligned}\langle\lambda_f J_f M_f|\left[a^{\dagger}_{\xi_1 j_1}\tilde{a}_{\xi_2 j_2}\right]_{JM}|\lambda_i J_i M_i\rangle &= \\ &= \frac{1}{\hat{J}_f}\left(J_i M_i J M|J_f M_f\right)\langle\lambda_f J_f||\left[a^{\dagger}_{\xi_1 j_1}\tilde{a}_{\xi_2 j_2}\right]_{J}||\lambda_i J_i\rangle.\end{aligned} \tag{4.32}$$

If we insert (4.32) into (4.31) we can express the matrix element as

$$\begin{aligned}\langle\lambda_f J_f M_f|\hat{T}_{JM}|\lambda_i J_i M_i\rangle &= \\ &= \frac{(J_i M_i J M|J_f M_f)}{\hat{J}_f\hat{J}}\sum_{\xi_1 j_1 \xi_2 j_2}\langle\xi_1 j_1||\mathbf{T}_J||\xi_2 j_2\rangle\langle\lambda_f J_f||\left[a^{\dagger}_{\xi_1 j_1}\tilde{a}_{\xi_2 j_2}\right]_{J}||\lambda_i J_i\rangle\end{aligned} \tag{4.33}$$

or since we can use Wigner-Eckart on the left side as well,

$$\begin{aligned}\langle\lambda_f J_f||\mathbf{T}_J||\lambda_i J_i\rangle &= \\ &= \frac{1}{\hat{J}}\sum_{\xi_1 j_1 \xi_2 j_2}\langle\xi_1 j_1||\mathbf{T}_J||\xi_2 j_2\rangle\langle\lambda_f J_f||\left[a^{\dagger}_{\xi_1 j_1}\tilde{a}_{\xi_2 j_2}\right]_{J}||\lambda_i J_i\rangle.\end{aligned} \tag{4.34}$$

The factor $\langle\lambda_f J_f||\left[a^{\dagger}_{\xi_1 j_1}\tilde{a}_{\xi_2 j_2}\right]_{J}||\lambda_i J_i\rangle$ is called a matrix element for the *reduced one-body transition density* and is independent of both the geometry and the operator $\mathbf{T}_J$ (apart from its rank) [7]. The factor $\langle\lambda_f J_f||\mathbf{T}_J||\lambda_i J_i\rangle$ is called the *reduced matrix element* and is independent of the geometry.

Example (the number operator). The number operator summed up in m,

$$N_{\xi j} = \sum_m a^{\dagger}_{\xi j m}a_{\xi j m}, \tag{4.35}$$

can be expressed in the form of a spherical tensor as

$$N_{\xi j} = \hat{j}\left[a^{\dagger}_{\xi j}\tilde{a}_{\xi j}\right]_{00}, \tag{4.36}$$

i.e. a spherical tensor of rank 0. Inserted into (4.33) with $J = M = 0$ and $i = f$ we get

$$\begin{aligned}&\langle\lambda_i J_i M_i|\hat{N}_{\xi j}|\lambda_i J_i M_i\rangle \\ &= \frac{(J_i M_i 0 0|J_i M_i)}{\hat{J}_i\hat{0}}\sum_{\xi_1 j_1 \xi_2 j_2}\langle\xi_1 j_1||\mathbf{N}_0||\xi_2 j_2\rangle\langle\lambda_i J_i||\left[a^{\dagger}_{\xi_1 j_1}\tilde{a}_{\xi_2 j_2}\right]_{0}||\lambda_i J_i\rangle \\ &= \frac{1}{\hat{J}_i}\langle\xi j||\mathbf{N}_0||\xi j\rangle\langle\lambda_i J_i||\left[a^{\dagger}_{\xi j}\tilde{a}_{\xi j}\right]_{0}||\lambda_i J_i\rangle\end{aligned} \tag{4.37}$$

The factor $\langle \xi j || \mathbf{N}_0 || \xi j \rangle$ can be calculated using the Wigner-Eckart theorem backwards,

$$\begin{aligned}
&\langle \xi j || \mathbf{N}_0 || \xi j \rangle \\
&= \frac{\hat{j}}{(jm_1 00|jm_2)} \langle \xi j m_1 | N_{\xi j} | \xi j m_2 \rangle \\
&= \frac{\hat{j}}{(jm_1 00|jm_2)} \langle \xi j m_1 | \sum_m a^\dagger_{\xi j m} a_{\xi j m} | \xi j m_2 \rangle \\
&= \frac{\hat{j}}{(jm00|jm)} = \hat{j}
\end{aligned} \tag{4.38}$$

so that, inserted into (4.37), we get

$$\langle \lambda_i J_i M_i | \hat{N}_{\xi j} | \lambda_i J_i M_i \rangle = \frac{\hat{j}}{\hat{J}_i} \langle \lambda_i J_i || \left[a^\dagger_{\xi j} \tilde{a}_{\xi j} \right]_0 || \lambda_i J_i \rangle. \tag{4.39}$$

We recognize the factor $\langle \lambda_i J_i || \left[a^\dagger_{\xi j} \tilde{a}_{\xi j} \right]_0 || \lambda_i J_i \rangle$ as a reduced one-body transition density matrix element for a rank 0-operator, so if we manage to calculate such transition density matrix elements, the occupation numbers are given by multiplication by a simple constant. ■

To calculate the reduced one-body transition densities, we use the Wigner-Eckart theorem backwards and split up the operator $\left[a^\dagger_{\xi_1 j_1} \tilde{a}_{\xi_2 j_2} \right]_{JM}$, and the result turns out to be

$$\begin{aligned}
&\langle \lambda_f J_f || \left[a^\dagger_{\xi_1 j_1} \tilde{a}_{\xi_2 j_2} \right]_J || \lambda_i J_i \rangle \\
&= \frac{\hat{J}_f \hat{J}}{(J_i M_i J M | J_f M_f)\, \hat{j}_1} \sum_{m_1 m_2} (j_2 m_2 J M | j_1 m_1) \langle \lambda_f J_f M_f | a^\dagger_{\xi_1 j_1 m_1} a_{\xi_2 j_2 m_2} | \lambda_i J_i M_i \rangle.
\end{aligned} \tag{4.40}$$

A proper approach could therefore involve computing the standard one-body transitions densities, and then use them in all relevant sums to form the reduced one-body transition densities, arranged in proper groups. The concept will be explained in the following section.

4.5 Structure of presentation of reduced transition densities

The objective of our computations will be to calculate matrix elements for the reduced one-body transition density, i.e. elements

$$\langle \lambda_f J_f || \left[a^\dagger_{\xi_1 j_1} \tilde{a}_{\xi_2 j_2} \right]_J || \lambda_i J_i \rangle, \tag{4.41}$$

and group them in proper lists. However, from (4.40) we deduce that they will be zero unless

$$|J_i - J| \leq J_f \leq J_i + J. \tag{4.42}$$

We can also see, by the use of the property

$$(J_i M_i J M | J_f M_f) = (-1)^{J_i - M_i} \frac{\hat{J}_f}{\hat{J}} (J_f M_f J_i, -M_i | J M) \tag{4.43}$$

that

$$|J_f - J_i| \leq J \leq J_f + J_i \tag{4.44}$$

must hold as well. Thus, a spherical tensor can only connect states with a difference in angular momenta less than or equal to the tensor's rank.

Because of the Clebsch-Gordan coefficients in Theorem 4.2.1, the same condition holds also for the creation and annihilation operators, so that a given pair of creation and annihilation operators with angular momenta j_1 and j_2 are only relevant for an operator with rank J satisfying

$$|j_1 - j_2| \leq J \leq j_1 + j_2 \tag{4.45}$$

provided that (4.44) holds. When calculating the reduced one-body transition densities, we do not need to evaluate the sum in (4.40) if the triangle condition in (4.45) is not fulfilled.

This makes it appropriate to list reduced transition density matrix elements between two eigenstates for different ranks of the operator. The amount of reduced transition density matrix elements can then be reduced, since only the creation and annihilation operators for which (4.45) is fulfilled are relevant. When the specific operator is known, e.g. an electromagnetic moment, we can choose the appropriate list and calculate (4.34) based on the transition density matrix elements in this list.

In Table 4.1 we show the structure of such a list. For the first pair of eigenstates we have chosen $J_i = J_f = 3/2$. By (4.44) we get four possible spherical tensors that can connect these state, $J = 0,1,2,3$. For every one of these operators we list the reduced one-body transition density matrix elements, for all pairs of annihilation and creation operators allowed by (4.45). The second pair of eigenstates has $J_i = 3/2$ and $J_f = 1/2$, from which we get $J = 1,2$, and finally $J_i = J_f = 1/2$ makes $J = 0,1$.

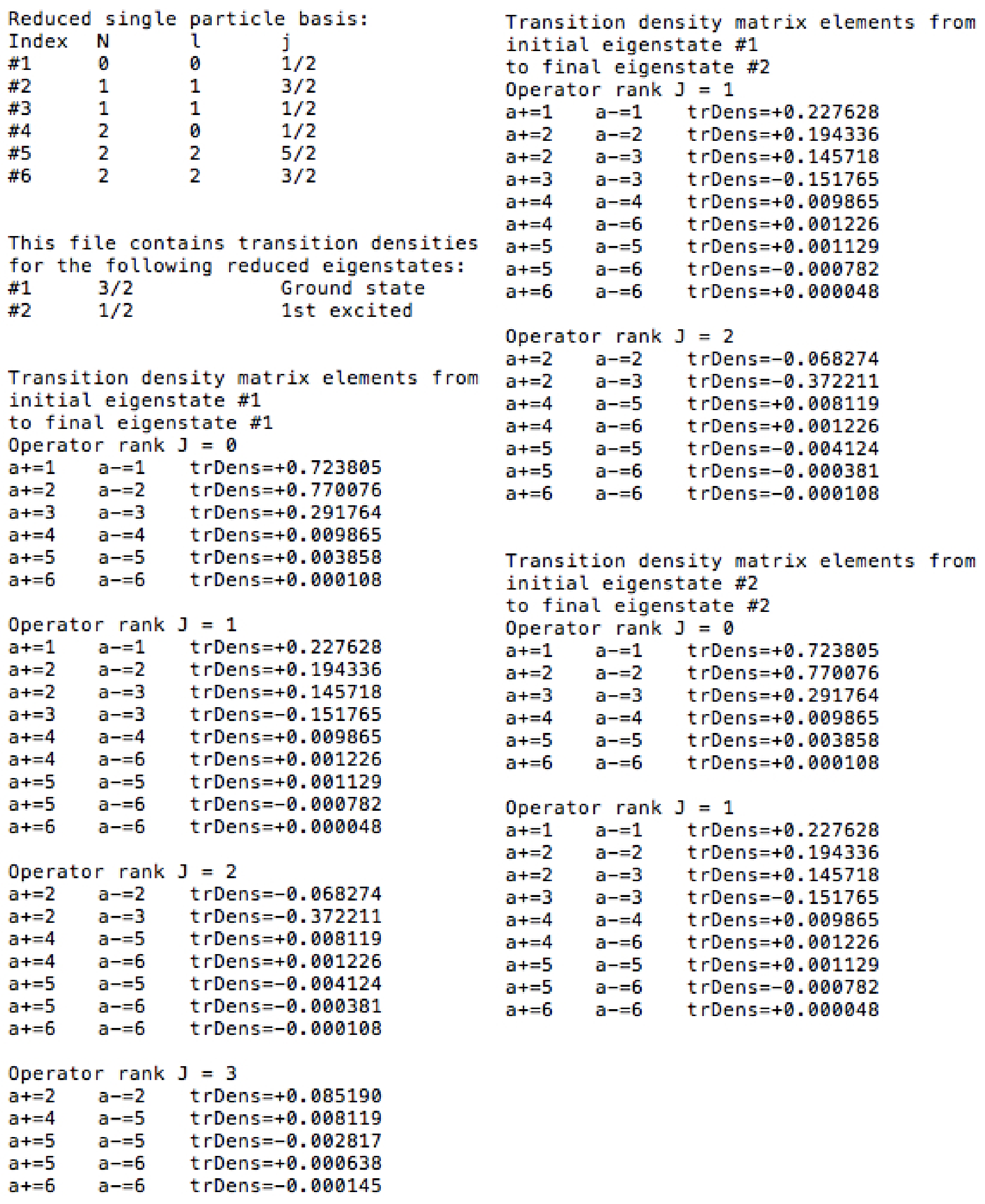
In Figure 4.1 we show a more explicit example produced by a full implementation in MATLAB, using the basis in the example of Chapter 3. The values have no physical significance since the eigenstates were randomly generated. We have used the same arbitrary eigenstate Js as in Table 4.1. `a+` and `a-` are the creation and annihilation operators characterized by reduced single particle states defined in the top of Figure 4.1. We can see how (4.45) plays its part: for operators of rank $J = 0$, `a+` and `a-` cannot have different j. This is reflected by the fact that all transition densities have the same creation as annihilation operator for all operators of rank $J = 0$. However, based solely on this fact, we would have expected e.g. single particle state no 1 to couple to single particle state no 3, but these states have different parities, hence it cannot happen. Further, for $J = 1$, we see only pairs of `a+` and `a-` whose difference in j is 0 or 1, which is what we would have expected from (4.45).

Table 4.1: Possible arrangement of the output. The reduced one-body transition densities in the rightmost column are ordered by: 1) the initial and final eigenstates, 2) the rank J of the operator and 3) the created and annihilated single particle states in the transition. For a given pair of eigenstates, the operator ranks of interest are given by (4.44), and for a given operator rank, the annihilation and creation operator pairs are given by (4.45), provided they give rise to any non-zero connection. The eigenstates have been chosen to have $J = 3/2$ and $J = 1/2$ which is an arbitrary choice, even though it is not impossible as ground state and first excited state for a real physical system.

Transitions densities for initial state $\vert\lambda_i J_i = 3/2\rangle$ to final state $\vert\lambda_f J_f = 3/2\rangle$		
Operator of rank $J = 0$		
$a^\dagger_{\xi_1 j_1}$	$\tilde{a}_{\xi_2 j_2}$	$\langle\lambda_f J_f = 3/2\vert\vert \left[a^\dagger_{\xi_1 j_1}\tilde{a}_{\xi_2 j_2}\right]_{J=0} \vert\vert\lambda_i J_i = 3/2\rangle$
$\vdots$	$\vdots$	$\vdots$
Operator of rank $J = 1$		
$a^\dagger_{\xi_1 j_1}$	$\tilde{a}_{\xi_2 j_2}$	$\langle\lambda_f J_f = 3/2\vert\vert \left[a^\dagger_{\xi_1 j_1}\tilde{a}_{\xi_2 j_2}\right]_{J=1} \vert\vert\lambda_i J_i = 3/2\rangle$
$\vdots$	$\vdots$	$\vdots$
Operator of rank $J = 2$		
$a^\dagger_{\xi_1 j_1}$	$\tilde{a}_{\xi_2 j_2}$	$\langle\lambda_f J_f = 3/2\vert\vert \left[a^\dagger_{\xi_1 j_1}\tilde{a}_{\xi_2 j_2}\right]_{J=2} \vert\vert\lambda_i J_i = 3/2\rangle$
$\vdots$	$\vdots$	$\vdots$
Operator of rank $J = 3$		
$a^\dagger_{\xi_1 j_1}$	$\tilde{a}_{\xi_2 j_2}$	$\langle\lambda_f J_f = 3/2\vert\vert \left[a^\dagger_{\xi_1 j_1}\tilde{a}_{\xi_2 j_2}\right]_{J=3} \vert\vert\lambda_i J_i = 3/2\rangle$
$\vdots$	$\vdots$	$\vdots$

Transitions densities for initial state $\vert\lambda_i J_i = 3/2\rangle$ to final state $\vert\lambda_f J_f = 1/2\rangle$		
Operator of rank $J = 1$		
$a^\dagger_{\xi_1 j_1}$	$\tilde{a}_{\xi_2 j_2}$	$\langle\lambda_f J_f = 1/2\vert\vert \left[a^\dagger_{\xi_1 j_1}\tilde{a}_{\xi_2 j_2}\right]_{J=1} \vert\vert\lambda_i J_i = 3/2\rangle$
$\vdots$	$\vdots$	$\vdots$
Operator of rank $J = 2$		
$a^\dagger_{\xi_1 j_1}$	$\tilde{a}_{\xi_2 j_2}$	$\langle\lambda_f J_f = 1/2\vert\vert \left[a^\dagger_{\xi_1 j_1}\tilde{a}_{\xi_2 j_2}\right]_{J=2} \vert\vert\lambda_i J_i = 3/2\rangle$
$\vdots$	$\vdots$	$\vdots$

Transitions densities for initial state $\vert\lambda_i J_i = 1/2\rangle$ to final state $\vert\lambda_f J_f = 1/2\rangle$		
Operator of rank $J = 0$		
$a^\dagger_{\xi_1 j_1}$	$\tilde{a}_{\xi_2 j_2}$	$\langle\lambda_f J_f = 1/2\vert\vert \left[a^\dagger_{\xi_1 j_1}\tilde{a}_{\xi_2 j_2}\right]_{J=0} \vert\vert\lambda_i J_i = 3/2\rangle$
$\vdots$	$\vdots$	$\vdots$
Operator of rank $J = 1$		
$a^\dagger_{\xi_1 j_1}$	$\tilde{a}_{\xi_2 j_2}$	$\langle\lambda_f J_f = 1/2\vert\vert \left[a^\dagger_{\xi_1 j_1}\tilde{a}_{\xi_2 j_2}\right]_{J=1} \vert\vert\lambda_i J_i = 3/2\rangle$
$\vdots$	$\vdots$	$\vdots$



```
Reduced single particle basis:
Index   N       l       j
#1      0       0       1/2
#2      1       1       3/2
#3      1       1       1/2
#4      2       0       1/2
#5      2       2       5/2
#6      2       2       3/2


This file contains transition densities
for the following reduced eigenstates:
#1      3/2             Ground state
#2      1/2             1st excited


Transition density matrix elements from
initial eigenstate #1
to final eigenstate #1
Operator rank J = 0
a+=1    a-=1    trDens=+0.723805
a+=2    a-=2    trDens=+0.770076
a+=3    a-=3    trDens=+0.291764
a+=4    a-=4    trDens=+0.009865
a+=5    a-=5    trDens=+0.003858
a+=6    a-=6    trDens=+0.000108

Operator rank J = 1
a+=1    a-=1    trDens=+0.227628
a+=2    a-=2    trDens=+0.194336
a+=2    a-=3    trDens=+0.145718
a+=3    a-=3    trDens=-0.151765
a+=4    a-=4    trDens=+0.009865
a+=4    a-=6    trDens=+0.001226
a+=5    a-=5    trDens=+0.001129
a+=5    a-=6    trDens=-0.000782
a+=6    a-=6    trDens=+0.000048

Operator rank J = 2
a+=2    a-=2    trDens=-0.068274
a+=2    a-=3    trDens=-0.372211
a+=4    a-=5    trDens=+0.008119
a+=4    a-=6    trDens=+0.001226
a+=5    a-=5    trDens=-0.004124
a+=5    a-=6    trDens=-0.000381
a+=6    a-=6    trDens=-0.000108

Operator rank J = 3
a+=2    a-=2    trDens=+0.085190
a+=4    a-=5    trDens=+0.008119
a+=5    a-=5    trDens=-0.002817
a+=5    a-=6    trDens=+0.000638
a+=6    a-=6    trDens=-0.000145
```

```
Transition density matrix elements from
initial eigenstate #1
to final eigenstate #2
Operator rank J = 1
a+=1    a-=1    trDens=+0.227628
a+=2    a-=2    trDens=+0.194336
a+=2    a-=3    trDens=+0.145718
a+=3    a-=3    trDens=-0.151765
a+=4    a-=4    trDens=+0.009865
a+=4    a-=6    trDens=+0.001226
a+=5    a-=5    trDens=+0.001129
a+=5    a-=6    trDens=-0.000782
a+=6    a-=6    trDens=+0.000048

Operator rank J = 2
a+=2    a-=2    trDens=-0.068274
a+=2    a-=3    trDens=-0.372211
a+=4    a-=5    trDens=+0.008119
a+=4    a-=6    trDens=+0.001226
a+=5    a-=5    trDens=-0.004124
a+=5    a-=6    trDens=-0.000381
a+=6    a-=6    trDens=-0.000108


Transition density matrix elements from
initial eigenstate #2
to final eigenstate #2
Operator rank J = 0
a+=1    a-=1    trDens=+0.723805
a+=2    a-=2    trDens=+0.770076
a+=3    a-=3    trDens=+0.291764
a+=4    a-=4    trDens=+0.009865
a+=5    a-=5    trDens=+0.003858
a+=6    a-=6    trDens=+0.000108

Operator rank J = 1
a+=1    a-=1    trDens=+0.227628
a+=2    a-=2    trDens=+0.194336
a+=2    a-=3    trDens=+0.145718
a+=3    a-=3    trDens=-0.151765
a+=4    a-=4    trDens=+0.009865
a+=4    a-=6    trDens=+0.001226
a+=5    a-=5    trDens=+0.001129
a+=5    a-=6    trDens=-0.000782
a+=6    a-=6    trDens=+0.000048
```

Figure 4.1: Output from our MATLAB-implementation in reduced form. The many-body basis was the same as in the example of Chapter 3. Here, the probability amplitudes of the eigenstates were generated randomly. The values for `a-` and `a+` (annihilated and created states) refer to the indices in the list of reduced single particle states. For a given pair of eigenstates, the operator ranks of interest are given by (4.44), and for a given operator rank, the annihilation and creation operator pairs are given by (4.45), provided that they give rise to a non-zero connection.

Chapter 5

Properties of FPGAs

This project concerns the FPGA-implementation of the calculation of transition densities. Hence the properties and use of an FPGA are cornerstones in motivating and explaining developed algorithms and methods, in later chapters. Therefrom, we obtain the main objective of the current chapter, i.e. to explain such concepts. This will be made in two parts, where we in a first section treat general properties of FPGAs, properties built into the very hardware, while the latter part regards the programming of the specific FPGA, that will be used. More exactly, it concerns the Maxeler programming interface and their MaxelerOS, adjoined with the MAX3 card.

5.1 Technical limitations and possibilities of an FPGA

We shall now give a brief description of what an FPGA is and introduce important concepts such as pipelining, dataflow programming and kernels.

An FPGA is essentially a reconfigurable electronic circuit. It is built out of many small logic blocks with programmable interconnections, which makes it possible to reconfigure the circuit by only changing the code describing it. This code – called *Hardware Description Language*, or HDL – describes how the logic blocks, which represent different operations, are configured and interconnected. With this reconfigurability comes the ability to employ all the on-chip resources of the FPGA into the specific requirements of a computation. In a conventional CPU, this is typically not the case [10].

The different components of which the FPGA is composed of, are:

- LUTs: *Look Up Table.* Main component of the FPGA that implements logic functions by modelling them as truth tables.
- FFs: *Flip-Flops.* Used to keep track of values as counters and pipeline stages.
- BRAM: On-chip memory, *Block RAM*, that might be accessed very quickly.
- DSP slices: *Digital Signal Processing elements*, good for multiplications and other arithmetic operations.

The limitation of the FPGA is that, once configured, the layout cannot be changed

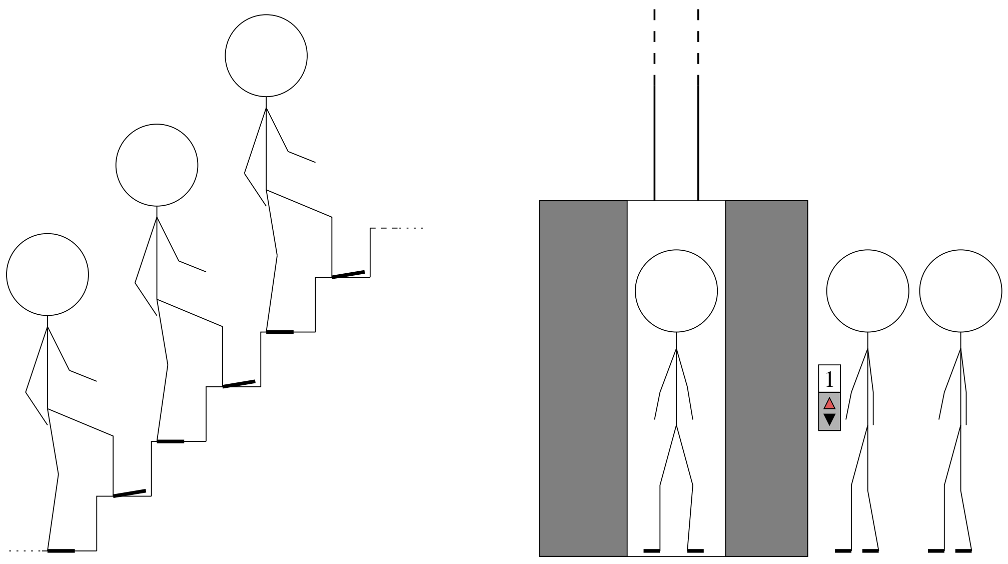


Figure 5.1: An anology comparing an FPGA to an CPU. The FPGA can, like the stairs, process many inputs at the same time but it takes a while for each input to pass. The CPU, on the other hand, can like a small elevator only process one input at any given time, but does so more quickly than the FPGA.

as long as it is performing calculations[1]. Thus, it must be decided beforehand exactly what calculations are to be done, since all operations have to be connected statically in such a way that an input constantly flows from one operation to the next.

With the dataflow-style of the FPGA, on the other hand, comes the ability to process several inputs in parallel. Let us use an analogy to illustrate the difference between a CPU and an FPGA; using the CPU corresponds to taking the flight of stairs and the FPGA to using the elevator, respectively, as seen in Figure 5.1. In the elevator one person enters, goes up and sends it down again before the operation is completed. This limits us to process one person at any given time while the rest have to wait at the bottom. Using the stairs, several persons could climb them at once; there is enough space for people to go side by side and there is room for new persons to enter as soon as the previous ones have advanced to the second step. If the number of people desiring to go up is small, it will be considerably faster to use the elevator. But if there are a couple of thousand persons that for some reasons want to climb to the top, the elevator will not suffice and a queue will form. The stairs, on the other hand, will allow many of them to climb at the same time and thus increase the flow.

The FPGA is, as the flight of stairs, not as fast at each individual step as the CPU – it has a much lower clock frequency – but has the capacity to process numerous inputs at the same time. Since each step is a separate block of hardware in the kernel, the FPGA has the ability to continuously take in new inputs and therefore process one input each

[1]This is not entirely true: it is possible to reconfigure parts of the FPGA during run-time. The catch is that this takes time and would significantly slow down the application to such an extent that it in most cases makes the FPGA much slower than a CPU. Thus we will in this report consider the FPGA as static after compilation.

clock cycle. Connecting several operations in such a way that an output from one stage is directly connected to the input of the next stage, without intermediate storage in memory, is commonly called *pipelining* and greatly increases the speed at which the FPGA can process inputs.

Furthermore, the complete set of instructions, which together form a *kernel*, can be duplicated inside the hardware if the resources allow it. This would in our analogy be represented with the set of stairs being wide enough to allow multiple persons to go side by side. One could thus build a design in which two identical, but independent, kernels exists. Such a design would double the rate at which inputs are processed and further enhance the performance of the FPGA. These two traits, the pipelining and the parallelism, are the two greatest strengths of an FPGA and will be used extensively in our implementations.

As a final example, note that the FPGA on the MAX3 card has a clock frequency of 75 MHz in its standard configuration. Compared to the 3.5 GHz of many modern CPUs this might seem slow, but if each input requires for example 200 separate operations then the process rate of the CPU reduces roughly to $\frac{3.5\,\text{GHz}}{200} = 17.5\,\text{MHz}$ of inputs, while the FPGA still accepts new inputs at a rate of 75 MHz. This demonstrates the ability of the FPGA to process highly regular calculations at an extremely high speed.

5.1.1 Dataflow programming

The programming style used for programming an FPGA is called *dataflow programming.* The name comes from the need to keep track of how data flow from one operation to another in the pipelining structure.

A consequence of having to know how the data will flow through the hardware is that loops that execute an unspecified number of times are impossible to implement. Implementing such a loop would require preparing for the worst-case scenario: an infinite loop. However, because of the pipelining, every iteration of the loop must be implemented as a separate set of hardware elements and thus an infinite loop would require an infinite amount of hardware. Furthermore, as each input has to pass through all steps of this infinite pipeline, processing a single input would take infinite time.

This leads to the concept called *pipeline depth.* This would in the analogy in the previous section be the number of steps in the stairs, and is essentially the number of clock cycles one input requires to pass through the kernel. In applications with few inputs but many operations this will be a limiting factor but as the the number of inputs grows beyond the pipeline depth its significance will diminish.

5.1.2 Architecture of the MAX3 card

The MAX3 card that is used in this project, contains a *Xilinx Virtex 6* FPGA and is built into a CPU-based computer system. The structure of the card and how the different parts connect to each other can be seen in Figure 5.2. The card communicates with the CPU using a PCIe 2x8 interface which has the maximal bandwidth 4 GB/s in both directions. The MAX3 card also contains an external DRAM[2] of size 48 GB, close

[2]DRAM stands for Dynamic random-access memory and is a volatile memory, i.e. its information is quickly lost once power is lost.

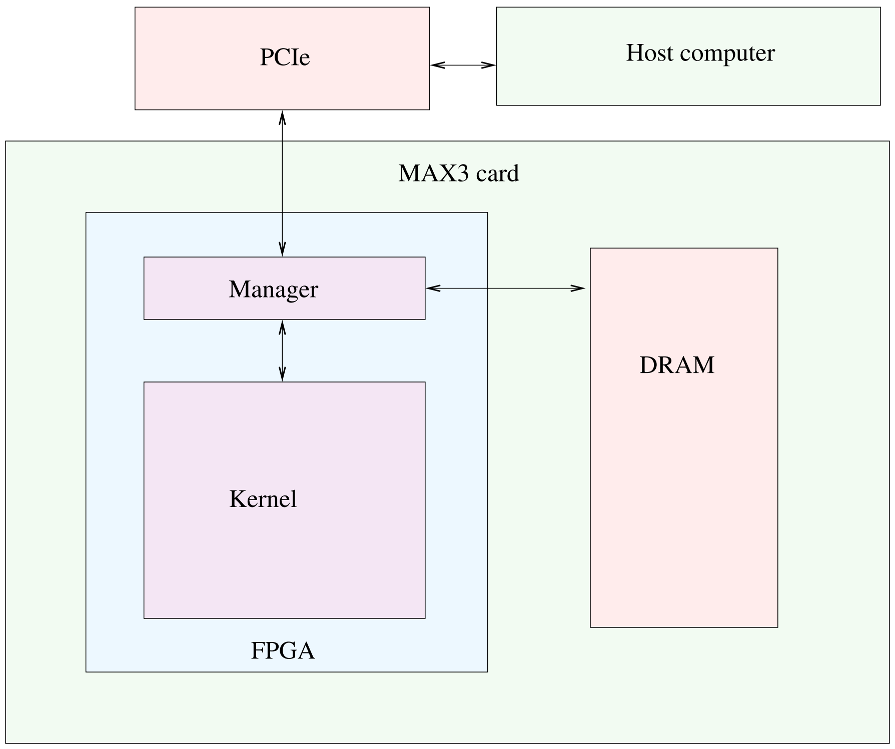


Figure 5.2: Schematic view of the MAX3 card and how the different parts are connected to each other. Only parts linked with arrows can directly communicate with each other.

the FPGA. The connection between the DRAM and the FPGA operates at a bandwidth of up to 38 GB/s.

5.2 Maxeler programming interface

This section serves to illustrate the Maxeler programming interface, used to implement algorithms described in later chapters. The interface consists of a suite of programming languages, compilers and libraries designed to make the implementation of an application on the FPGA easier, as opposed to using a HDL, since the interface allows one to program in a high-level language [11]. It also includes tools for simulating and debugging implementations to verify correct function and/or finding errors in designs. Important concepts that will be explained are: kernels, managers, input/output, counters, on-board memory, DRAM and scalar inputs.

The main elements needed to program an FPGA-implementation are kernels and a manager. Kernels are the programming constructs specifying the datapaths within the hardware implementing the basic arithmetic and logical computations required for a specific algorithm. A manager is a dataflow controller containing the logic for directing streams of data between the different components within a Maxeler system. The separation of the two is purposeful because this way, kernels can have high degrees of dataflow computations while the programmer can disregard synchronization considerations which otherwise have to be taken into account when programming on multiple control-flow based cores.

5.2.1 Kernels

The kernel is as previously mentioned the part of the FPGA-implementation that performs calculations. HDL is normally used to specify kernels. However, for the Maxeler machine, kernels are specified in a high-level language, the Java programming language. By compiling this high level code, the MaxelerOS converts it to HDL and thereafter optimizes it to maximize its performance when implemented on the FPGA hardware.

The optimization MaxelerOS performs can be seen in Figure 5.3 which contains graphs rendered with the tool *maxRenderGraphs* provided by Maxeler for debugging. The graphs show the pipelined stages of an implementation of a simple kernel, before and after optimization, respectively. To optimize the dataflow, three buffers have been inserted which allows the input to more conveniently flow from one stage to the next.

Input and output

The Maxeler programming interface implements data transfers to/from the FPGA, using streams. This streaming of data can be thought of as arrays that are incrementally passed to/from the FPGA. It is possible to use a boolean condition to determine whether or not for a given clock cycle, a data transfer should occur. However note that the type of data being transferred in a stream is always the same. Streams can be defined between the FPGA and CPU, the FPGA and DRAM and the CPU and DRAM.

Counters

A counter represents an integer value, which in steps can be incremented. For example, it is possible to use a counter to keep track of the number of clock cycles that a kernel has been run. To implement this, a counter is instantiated that increments its value each clock cycle. However, the base class for counters, also allows more flexible behaviours. More specifically, a counter can be defined with:

- A maximum value, i.e. a highest value that it can reach. Also, there are possibilities to specify what happens when this value is reached. For example, it could wrap and restart counting from zero (its initial value).
- An enable-condition, i.e. a condition that controls whether the counter should increment its value, in a given clock cycle.
- A reset-condition, i.e. a condition that if true resets the value of the counter to zero.

Besides the use of counters to keep track on the number of clock cycles that a kernel has been run, they are important control structures that can be used to govern the behavior of the kernel. More specifically, the counter can be used to make a kernel execute a for-loop, similar to those in conventional programming languages. In such implementations,

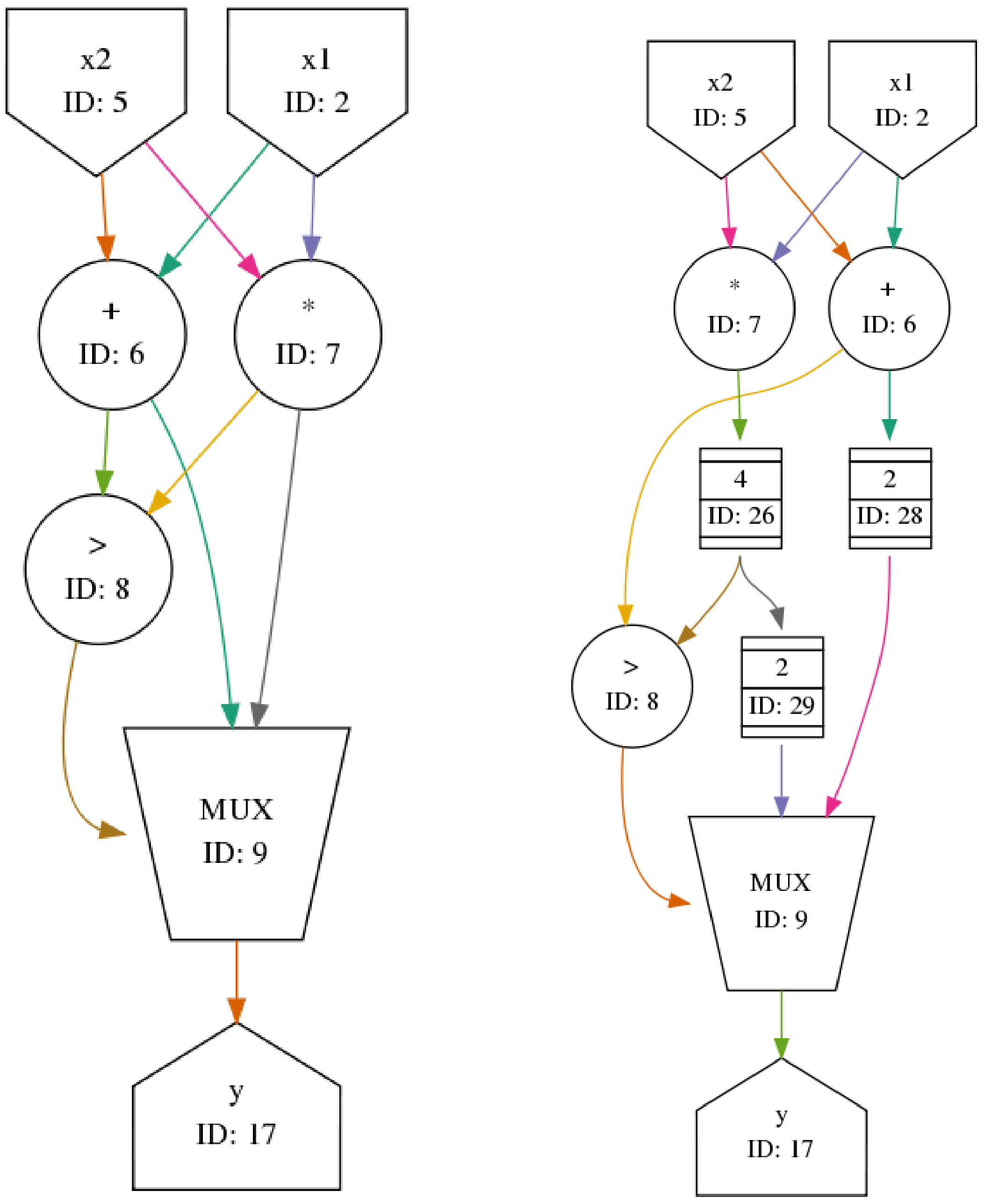


Figure 5.3: A kernel generated with Maxelers RenderGraph tool, with its layout before (left) and after (right) optimization. The figure shows how different operations connect to each other within the kernel. This kernel takes two inputs, x1 and x2, and outputs the largest of the sum or the product of these as y. In the final version three elements have been added. These are buffers to align the inputs and are a kind of null operation. This allows the input to flow forward at each cycle instead of waiting in the plus and multiplication operators until the comparison is complete.

the value of the counter is equivalent to the index-variable of the for-loop. Using enable-conditions, it is also possible to connect two or more counters, that is to chain them, so that their values replicate the behavior of the index-variables in a nested for-loop.

On-chip memory and DRAM

Besides the earlier mentioned external DRAM, the FPGA also has on-chip memory referred to as ROM. It can be noted that the amount of ROM is very small and therefore only can store limited amounts of data. On the other hand, it has very high bandwidth, meaning that read/write operations can be performed very fast. When it comes to the DRAM, it is much larger, but read/write operations are not as fast as for ROM. In conclusion, only data that must be accessed closely to the application, should be stored in ROM, while other forms of data are kept in DRAM.

It is important to note that the addressing of a DRAM is done in units called bursts. The size of a burst unit is given by the burst-size, which is 384 byte, for the DRAM of the MAX3 card. This burst-size is the smallest amount of data that can be read/written to DRAM. For applications where the size of a data item is much smaller, it leads to that the bandwidth of the DRAM cannot be fully utilized.

Kernel example

The code in listing 5.1 gives an example of the syntax used when specifying a kernel. More specifically, we show the syntax for arrays (KArray), scalar inputs, counters and input/output. Note the similarities to the programming language Java.

```java
public class ExampleKernel extends Kernel {

    /* Declaration of an arraytype.  */
    public KArrayType<HWVar> arrayType = new KArrayType<HWVar>(hwInt(32),
        2);

    public ExampleKernel(KernelParameters parameters) {

    super(parameters);

    /* Scalar input, set from the CPU.*/
    HWVar scalar = io.scalarInput(''scalar'', hwInt(32));

    /* Declaration of counter parameters and instantiation of counter. */
    Count.Params params = control.count.makeParams(32);
    Count.Counter counter = control.count.makeCounter(params);

    /* Input stream that accepts arrays of the type type, previously
        defined. */
    KArray<HWVar> inputArray = io.input(''inputArray'', arrayType);

    /* Instantiation of output array.*/
    KArray<HWVar> outputArray = arrayType.newInstance(this);

    /*Performs a calculation and assigns the values to the output array.*/
    outputArray[0]<==inputArray[0] * counter.getCount();
    outputArray[1]<==inputArray[1] * scalar;
    /* Sends the output array to the CPU, by means of an output stream. */
    io.output(''outputArray'', outputArray,arrayType);
  }
}
```

Listing 5.1: Example of the syntax when specifying a kernel. This kernel has inputs of two kinds, one being a scalar input (named scalar) and one being an array of two elements (named inputArray). The kernel also has a counter, that increments its value each clock cycle. The function of the kernel is to in each clock cycle accept a new input array, which is transferred in a stream from the CPU. It then produces a new array (named outputArray) by multiplying the elements of input by the value of the scalar input and counter respectively. The code then specifies that the array with calculated elements should be transferred back to the CPU, by means of a stream.

Concerning the example code shown in listing 5.1, it represents a kernel that takes two inputs. The first input is a scalar input, which is set from a CPU, before running any calculations. This way it is possible to pass a constant to a kernel, without the need to recompile the design. Besides this input, there is an input stream that in each clock cycle delivers an array, consisting of two elements. Additionally, the kernel implements a counter, which keeps track on the number of clock cycles that have been executed. An output is produced in the form of a new array, also consisting of two elements. In this example, these are calculated by multiplying the respective elements of the input array, by the value of the scalar input and counter, respectively. The output array is finally transferred to the CPU, using an output stream.

5.2.2 Manager

The manager in a design is the element responsible for controlling the dataflow between kernels and the various components in a Maxeler system. It contains all the logic related to streams. This logic specifies which components data is streamed to/from, and over which medium. For example, a stream may be declared to pass values between the CPU and a kernel over the PCIe bus. Also, there are possibilities to incorporate blocks of HDL code, alongside with kernels written in the described high-level language. This makes it possible to use HDL, which in many instances can be more flexible than the high-level programming interface [12].

Chapter 6

Algorithm design for one-body operators

In this chapter, we discuss a number of possible algorithm designs that make use of an FPGA to calculate transition densities for a one-body operator. Its non-zero matrix elements are characterized by that connecting many-body states differ with no more than one single-particle state, as described in Chapter 3. By studying these, we can present the major challenges of an FPGA-implementation as well as computational structures that can also be used to check transition rules for two- and three-body operators.

6.0.1 Computational complexity of the calculation of transition densities

The first important fact about the calculation of transition densities is that it is very computationally intense. The calculation is often structured by sequentially processing many-body states, using an inner loop to find out what other many-body states that initial states connect to. If computational resources were not a concern, one might suggest to let the inner-loop form all possible pairs of many-body states and check if they connect. Doing so would however result in the computational complexity

$$\mathcal{O}(\#\texttt{mbState}^2), \tag{6.1}$$

which, since `#mbState` often is of order 10^9, becomes very large.[1] Actually it is so huge that the calculations become unfeasible [13], even for the biggest computing clusters.

Here, we will instead use a different approach. This approach is based on the transition rule for a one-body operator, which makes the corresponding matrix very sparse. Specifically, we can limit the load of the inner-loop by only considering connections to many-body states that are formed by interchanging one single-particle state. Doing this, we obtain the computational complexity

$$\mathcal{O}(A \cdot \#\texttt{mbState} \cdot \#\texttt{spState}), \tag{6.2}$$

which generally is smaller than for the former approach, as `#spState` and A are much

[1] `#mbState` is the dimension of the many-body basis.

more modest numbers than `#mbState`.[2] Since the process of interchanging single-particle states exhibits a regular structure, we deem it appropriate for FPGA-implementation.

6.0.2 FPGA- and CPU-based parts of the calculation

The general structure for calculating transition densities, by performing single-particle state interchanges, is given by:

```
InputStream: mbState;
for(i in {1,...,$A$})
    for(spState $\in S_{\texttt{mbState},i}$)
        Use spState to replace the state at position $i$
        in mbState;
        if(Modified mbState is valid)
            Add the contribution of the product of probability amplitudes of
                the found connection, to the reduced transition density;
        end
    end
end
```

Here, the `mbStates` are provided to the FPGA as an input-stream. Their connections are then found by replacing individual single-particle states, using new single-particle states belonging to an appropriately designed set $S_{\texttt{mbState},i}$[3]. When a connection is found, we form the product of probability amplitudes for the connecting states, and add its contribution to the reduced transition density, as discussed in Chapter 4. This latter calculation is unfortunately rather complicated to perform on an FPGA. In this project we therefore chose to perform it off the FPGA, in a CPU-based calculation. The task for the FPGA will be to present the CPU-based calculation with a list of all connections, with all information needed to calculate the reduced transition density. A flow-chart for the chosen sub-division is shown in Figure 6.1. Also, we chose to focus on optimizing the FPGA-part of the calculation. The list provided to the CPU, by the FPGA, will only consist of valid connections, and we might hence suspect that it only needs limited computing efforts to be converted to a reduced transition density[4].

6.0.3 Specific parts of the calculation of transition densities

In this chapter we discuss both the FPGA- and CPU-based computations that are used to calculate reduced transition densities. Since the FPGA-part of the implementation is less straightforward than the CPU-part, it will be our main focus. A selection of choices and computations that must be explained is:

- How many-body states should be represented in the FPGA computation.
- How many-body states should be supplied as input to the FPGA.

[2]`#spState` is the dimension of the single-particle basis.

[3]When we optimize the calculation, we aim at minimizing the size of $S_{\texttt{mbState},i}$, so that we can calculate the non-zero matrix element with as limited amount of work as possible.

[4]Or otherwise one might have to reconsider to also implement this part of the calculation on the FPGA, but which will have to be a topic of a further study. This matter will be discussed in Chapter 8.

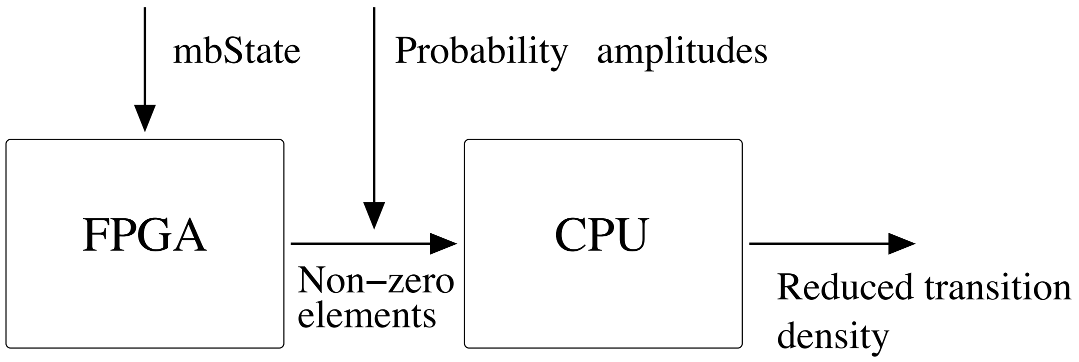


Figure 6.1: Flow-chart of the main features of the computation. We supply the FPGA with an input-stream of many-body states. The FPGA identifies the valid connections and outputs them to a CPU, where the calculation of the reduced transition density is performed. At some point, either on the FPGA or CPU, we retrieve probability amplitudes which are associated with the found connections.

- Where probability amplitudes should be kept, in order to be accessed when needed.
- How single-particle state interchanges should be performed in order to generate valid connections.
- The calculation of signs for found connections.
- The creation of representations for many-body states obtained by single-particle state interchanges.
- Implementation of a hash-function, to access probability amplitudes.
- The form and method of delivery of output, from the FPGA.

The following presentation will begin by describing representations of many-body states and choices regarding I/O in Section 6.1. This will be followed by a description, in Section 6.2, of how the FPGA-kernel should be designed, incorporating the more computational steps. Only as a final part, we will describe the off-FPGA calculations. This is done in Section 6.3.

6.1 Data representation and FPGA-related I/O-design

This section describes a number of representations for many-body states which later will be used, within FPGA-implementations. We also discuss how they can and should be supplied to the FPGA as well as how probability amplitudes can be stored and accessed. Finally we specify the desired output from the FPGA, that is needed for the calculation of the reduced matrix-representation.

6.1.1 Representation of many-body states

To obtain a size-efficient representation, we describe the many-body states by a list of occupied single-particle states. Within the FPGA, such a list can be represented as an

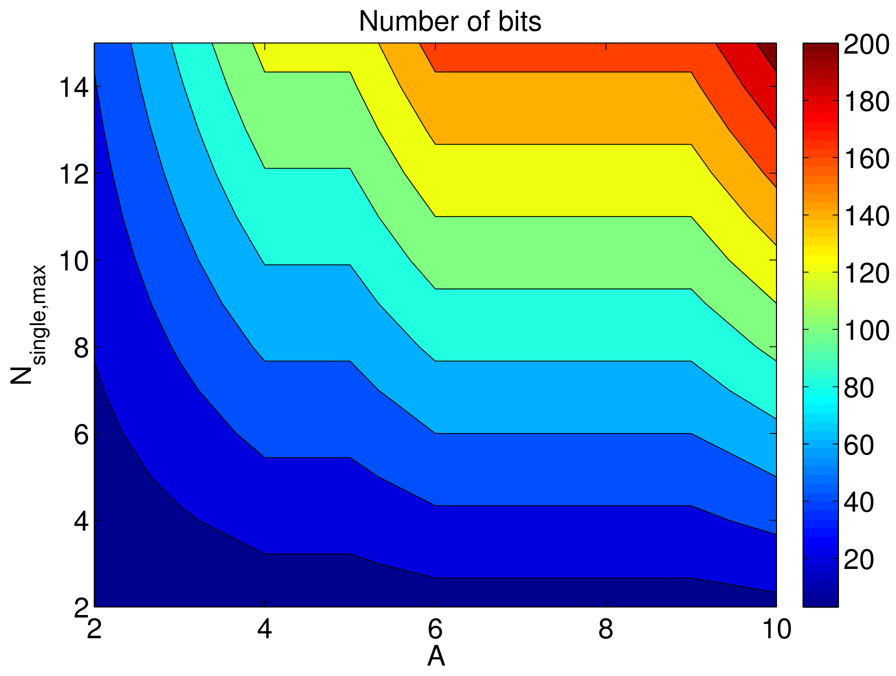


Figure 6.2: Minimal number of bits needed to represent many-body states, using a list-representation, for the cases $2 \leq N_{\text{single,max}} \leq 15$ and $2 \leq A \leq 10$.

array of numbers, identifying the single-particle states. These numbers can for example be the indices of the single-particle states, in an enumeration. Note however, that such representation does not directly provide any information about the physical properties of the single-particle states. One must instead store such information in on-chip ROMs or provide it as additional input. The latter option will be further explored in one of our kernel-designs.

Besides the above mentioned kernel-design, we will also propose designs that do not need any additional input, except the many-body states, whose connections shall be found. In these contexts, we will denote our single-particle states by their quantum numbers (i.e. n, l, j and m_j). The benefit of this representation is that we directly can evaluate all possible properties about the described many-body state.

Note that the bit-sizes of the above representations affect the number of many-body states that can be transferred to the FPGA, due to that input bandwidth is limited. If using minimal types, both representations however are of similar size, given by

$$|\texttt{mbState}| = 3A \cdot \log_2(N_{\text{single,max}}) + \mathcal{O}(A) \tag{6.3}$$

bits. Based on this expression, we give numerical values for the sizes of our representations of many-body states, which are found in Figure 6.2. We see that the sizes of the representations range over about one order of magnitude, from 20 to 200 bits, for systems with up to 10 particles and 15 units of energy, with 100 bits, being a somewhat typical value.

Finally, concerning both of the proposed representations, they are only unique up to permutation. This is a slight problem as we need unique representations in order to construct well-defined hash-addresses, i.e. memory addresses used to retrieve probability amplitudes. To solve this, we will sort the single-particle states in ascending order, according to some appropriate ordering. However, when a single-particle state is replaced, it cannot a priori be known where in the old representation that the new state should be inserted, so that the resulting representation becomes properly sorted. This highlights that the construction of new representations of many-body states, needs an FPGA-based insertion algorithm. This will be further discussed when we outline kernel-designs in Section 6.2.

6.1.2 Implementation of minimal data types

The calculations of bit-sizes for representations, in the previous section, assumed that we could use minimal data types. Out of the hardware perspective, this is straightforward to implement. However, inputs (as well as outputs) should also be compatible with the types of the CPU-interface for the FPGA, which is written in `C++`. Implementing minimal types would therefore demand the use of bit-array structures. As this is a feasibility study, we will limit ourselves to use integer types which is more straightforward, but at the cost of that the representations become larger than necessary. On the other hand, we will not try to increase parallelism by instantiating many kernels, meaning that limits of input-bandwidth will not be that much of a critical issue.

6.1.3 Input-source for many-body states and bandwidth aspects

In this project, we will use the PCIe to transfer input, i.e. many-body states, to the FPGA. The PCIe has the bandwidth[5] $B_{\text{PCI}} = 2\,\text{GB/s}$ and assuming that the FPGA operates at its standard frequency $f = 75\,\text{MHz}$, we can transfer

$$\frac{B}{f \cdot |\texttt{mbState}|} \tag{6.4}$$

many-body states per clock cycle. For example, in the earlier described case $|\texttt{mbState}| = 100\,\text{bits}$, we could theoretically transfer 2 many-body states per clock cycle. An important question is whether this will be a bounding factor for our implementations. However, as we later will see, our kernels will individually not be able to process many-body states at this rate. In order for input-bandwidth to become a bottleneck, we would have to implement many parallel kernels. In that situation, we would on the other hand most certainly have problems with bounds related to output-bandwidth, as the amount of output will be larger than the amount of input.

6.1.4 Storage of probability amplitudes

In addition to the many-body states, vectors of probability amplitudes are another form of input, which need to be accessed when a valid connection is found. Which probability

[5]There were some trouble in finding proper documentation of the what edition of PCIe-express, that the MAX3 card used. Some documentation indicated the bandwidth 4 GB/s, but 2 GB/s was the value that experimentally could be verified, as will be seen in Chapter 7.

amplitudes that become needed in a specific step of the calculation, can however not be predicted. As a result, the access of probability amplitudes must be from memory, with run-time calculation of addresses. Since our FPGA-implementations only aim at identifying the non-zero matrix elements on the FPGA, we may perform the lookups, either from FPGA-based or CPU-based DRAM.

A first important matter concerning the storage of probability amplitudes is that the respective DRAMs must be of sufficient size. To evaluate this, we note that the probability amplitudes either are single or double precision floats, i.e. numbers of sizes $S = 32\,\text{bit}$ or $64\,\text{bits}$. As there are one probability amplitude for each state in the many-body basis, we estimate that they will need

$$S \cdot \#\texttt{mbState} \tag{6.5}$$

of memory, if stored in an array. For the many-body bases illustrated in Chapter 3, this is equivalent to up to 1 GB of memory. Some additional amount of memory must however also be added to store keys and over-size the addressing space, as the probability amplitudes are stored by hashing. On the other hand, the DRAM of the FPGA has a volume of 48 GB and the DRAM of the CPU might be even larger, leading to the conclusion that size is not a limit, at least not for the many-body bases available in this project.

Another important point concerning lookups is that consecutive such typically correspond to uncorrelated reads of memory. This means that we will not be able to efficiently use the bandwidth of the storage-resource, unless the burst-size is small (similar to either 32 or 64 bits, with the addition of bits corresponding to the storage of a hash-key). Compared to this, the burst-size of the FPGA-DRAM is quite big, $W = 3072\,\text{bits}$, resulting in a maximum lookup rate of $B_{\text{DRAM}}/f \cdot W = 1.32\,\text{lookups/clock-cycle}$. This number is smaller than the rate at which input can be accepted and since each many-body state will have several connections, it might become a bottleneck. As there also is more flexibility in storing the probability amplitudes in the DRAM of the CPU, we chose to do so, in the current project.

6.1.5 Output from FPGA to CPU

As described in Chapter 3, an FPGA-implementation that does not calculate the reduced transition density on the FPGA, will need to produce the following information about the valid connections:

- What single-particle states that have been destroyed and created.
- The signed product of probability amplitudes corresponding to the connecting many-body states.

As we, in this project, perform probability amplitude lookups off the FPGA, we must however also know from/to what many-body state the connections go. To keep track on the many-body state from which a connection goes, we can use a counter. Concerning the many-body state that the connection goes to, it can be identified with an FPGA-calculated hash-key. There could be some benefits in transferring all mentioned outputs in individual output-streams. However, as the Maxeler programming interface limits the number of I/O-streams, we will transfer them in array-format (in one stream). The transfer will be performed by use of the PCIe, as indicated in the initial flow-chart of this chapter, found in Figure 6.1.

6.1.6 Conclusion regarding FPGA-related I/O

Based on the discussion in the previous sections, we make the following choices regarding I/O:

- The many-body basis will be represented as an array of occupied single-particle states. This representation will furthermore be streamed to the FPGA, using the PCIe. This will typically not be a bottleneck, compared to other forms of I/O.
- Probability amplitudes can either be stored in FPGA- or CPU-based DRAM, depending on whether lookups are performed on or off the FPGA. As the burst-size of the FPGA is large and there is larger flexibility in using the DRAM of the CPU, it will be chosen for this project.
- Each found connection will be transferred to the CPU, in array-format, using the PCIe. The array will contain the information described in the previous section.

6.2 Design of the computational kernel

This section describes FPGA-kernels that perform the FPGA-part of the calculation of transition densities, described in Section 6.0.2. An important part of this is the algorithm that is used to identify the valid connections. We will describe three different algorithms, one that uses no pre-calculated control inputs (besides the many-body basis) and two that does. In spite of differences in the detailed design, the overall structure of the computation is the same, illustrated in Figure 6.3. The Figure shows how our kernels will accept many-body states as input:

$$\texttt{mbState} = [\alpha_1,...,\alpha_A] \tag{6.6}$$

and in parallel interchange their single-particle states α_i. These processes are implemented in hardware with different output-streams and can hence produce up to A valid connections, simultaneously in a clock cycle. Therefore, we can increase the throughput of an individual kernel, compared to a design that only performs one single-particle state interchange at a time. However, the real benefit is that we can customize the implementation of the interchanges of single-particle states, depending on the position of α_i in `mbState`. Specifically, this makes the creation of new representations for many-body states as well as calculation of the sign of a connection straightforward, as such matters will depend on the position of the replaced states, which must be hard-coded in an FPGA-implementation.

This section will also present an evaluation of the algorithms with respect to their performance, in terms of the number of clock cycles they need to identify the connections of the many-body states in the many-body basis[6].

[6]As the FPGA operates on a constant frequency, the number of clock cycles is proportional to the

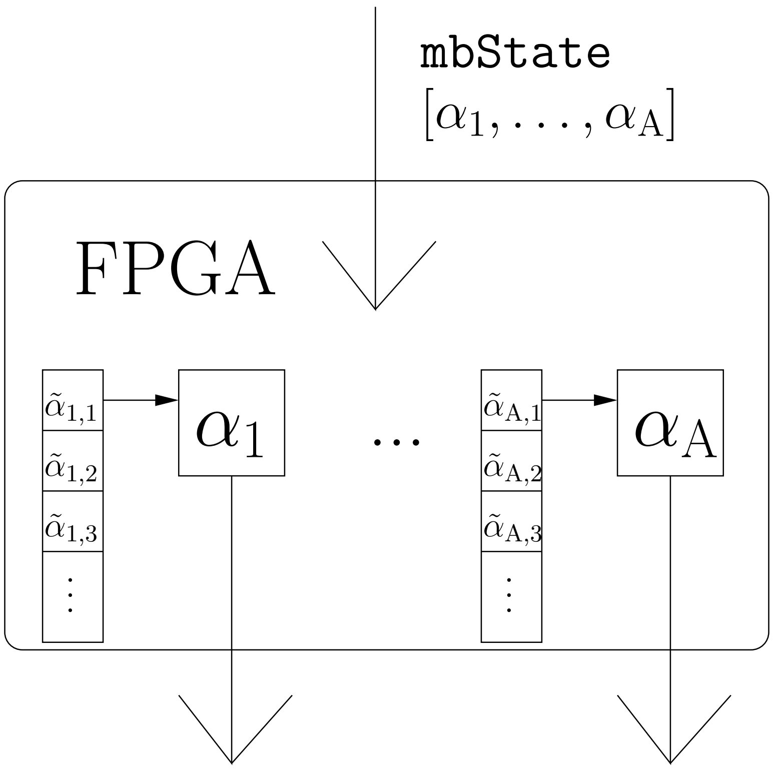


Figure 6.3: Structure of the kernels used to identify non-zero matrix elements of one-body operators. The many-body state, consisting of single particle states α_i, is sent as input. For each single-particle state, one block of hardware is assigned which performs the interchanges of the single-particle state α_i to a set of new states $\tilde{\alpha}_{i,j}$. These processes generate connections in parallel, with individual output-streams to the CPU.

Besides describing the algorithms that identify valid connections, this section also describes how surrounding calculations should be FPGA-based. This concerns:

- How the list representations for many-body states should be sorted.
- How new properly sorted representations for many-body states should be calculated, after performing a single-particle state interchange.
- How to calculate the sign of a connection.
- How to calculate hash-keys used to identify the many-body states to which connections go (later used for lookups).

6.2.1 Sorting of many-body states

The representation of many-body states by enumerations of constituent single-particle states, results in the need of sorting, in order to create unique representations. If the single-particle states are denoted using an enumeration, this sorting is to order them in ascending order.

When representing single-particle states by their quantum numbers, we however need a more complicated rule to sort them. One way of constructing such rule is to compare quantum numbers and let $(n_1,l_1,j_1,m_{j1}) \leq (n_2,l_2,j_2,m_{j2})$ be equivalent to

$$\{n_1 < n_2\}|\{(n_1 == n_2) \wedge (l_1 < l_2)\}|\{(n_1 == n_2) \wedge (l_1 == l_2) \wedge (j_1 < j_2)\}...$$
$$...|\{(n_1 == n_2) \wedge (l_1 == l_2) \wedge (j_1 == j_2) \wedge (m_{j1} \leq m_{j2})\}.$$

Another option is to find an injective mapping of the quantum numbers and order the single-particle states in ascending order, according to their values under this mapping. For example, we could use the following theorem [14]:

Theorem 6.2.1. *Let p_i be distinct primes and $P_i = \prod_{i \neq j} p_j$. It then holds true that*

$$(x_1,...,x_n) \in \bigoplus_i \mathbb{Z}_{p_i} \to \sum_i x_i P_i \tag{6.7}$$

is an injective mapping.

The theorem is associated with the construction of a solution to congruence equations in the Chinese remainder theorem. If p_i are chosen appropriately, so that the quadruples (n,l,j,m_j) are distinct elements of $\bigoplus_i \mathbb{Z}_{p_i}$, we can use the mapping (6.7) to order them.

When working with algorithms that represent single-particle states by their quantum numbers, we have chosen to use the mapping (6.7). The reason for this is that it lets us calculate comparison values for the single-particle states in the input many-body state, once in the code. Ordering single-particle states by comparing their quantum numbers (i.e. the first rule that was suggested to order them) leads to more complicated expressions that must be repeated at each place in the code, where we need to determine

execution time for the computation. The use of this performance measure is motivated by that the pipelined computational architecture makes it straightforward to calculate the number of clock cycles, needed by an algorithm.

the ordering of two single-particle states. This choice is however not obvious as the multiplications in the mapping, creates a need of larger types for the quantum numbers, than what is beneficial out of an I/O-perspective. Also, type-casts are expensive to implement in hardware. However, as we have chosen to work with integer data types, this will not be that much of a concern.

6.2.2 Insertion of new single-particle states

The input many-body states to our kernels will be sorted arrays of single-particle states. Since it is these sorted representations that are used to construct hash-addresses we need a way to remove old single-particle states and replace them by new ones, without violating the sorting. To show how this is done, consider a many-body state

$$\texttt{mbState} = [\alpha_1,...,\alpha_A], \tag{6.8}$$

where α_i are single-particle states, such that $\alpha_i < \alpha_{i+1}$. Also assume that the state α_i has been removed, and replaced by a new state α_{new}. We can then construct a sorted representation for the many-body state obtained by replacing α_i with α_{new}:

$$\texttt{mbState}_{\text{new}} = [\tilde{\alpha}_1,...,\tilde{\alpha}_A] \tag{6.9}$$

by using the ternary-if operator[7] and write

$$\begin{aligned}
&\tilde{\alpha}_1 = (\alpha_{\text{new}} > \alpha_1)?\alpha_1 : \alpha_{\text{new}} \\
&\tilde{\alpha}_k = (\alpha_{\text{new}} > \alpha_k)?\alpha_k : ((\alpha_{\text{new}} > \alpha_{k-1})?\alpha_{\text{new}} : \alpha_{k-1}) \qquad 1 < k < i \\
&\tilde{\alpha}_i = (\alpha_{\text{new}} > \alpha_{i+1})?\alpha_{i+1} : ((\alpha_{\text{new}} > \alpha_{i-1})?\alpha_{\text{new}} : \alpha_{i-1}) \\
&\tilde{\alpha}_k = (\alpha_{\text{new}} > \alpha_{k+1})?\alpha_{k+1} : ((\alpha_{\text{new}} > \alpha_k)?\alpha_{\text{new}} : \alpha_k) \qquad i < k < A \\
&\tilde{\alpha}_A = (\alpha_{\text{new}} > \alpha_A)?\alpha_{\text{new}} : \alpha_A
\end{aligned} \tag{6.10}$$

with slight modifications when $i = 1$ or $i = A$. Note that this algorithm for insertion is dependent on the position of the removed single-particle state. However, as interchanges for different α_i are associated with different hardware implementations, there are no problems related to hard-coding the special design, for each such interchange process.

6.2.3 Calculating the sign of a connection

In Chapter 3, the product of probability-amplitudes, for a connection, was seen to be associated with a sign, occurring in the calculation of its contribution to the reduced transition density. More specifically, the sign of a connection is given by

$$(-1)^{i-j}, \tag{6.11}$$

where i denotes the index[8] of the single-particle state that is being replaced and j is the index of the single-particle that is being inserted.

Now, let $\pm_i$ denote the sign for a connection which has resulted by removing a single-particle state at position i in a many-body state. We can then calculate $\pm_i$ by using the ternary-if operator.

[7]By definition, we have that $B?a_1 : a_2 = a_1$ if B is true and $B?a_1 : a_2 = a_2$ if B is false.

[8]Here, index refers to the position the single-particle state has in the list-representation of the many-body state, to which it belongs.

Example. In the case of $A = 3$ particles, we can use the following expressions to calculate signs:

$$\begin{aligned}
\pm_1 &= (\alpha_{\text{new}} < \alpha_2)?1 : ((\alpha_{\text{new}} < \alpha_3)? -1 : 1), \\
\pm_2 &= (\alpha_{\text{new}} < \alpha_1)? -1 : ((\alpha_{\text{new}} < \alpha_3)?1 : -1), \\
\pm_3 &= (\alpha_{\text{new}} < \alpha_1)?1 : ((\alpha_{\text{new}} < \alpha_2)? -1 : 1).
\end{aligned} \tag{6.12}$$

Note that the expressions differ, dependent on the position of the removed single-particle state. ■

The generalization to arbitrary A essentially involves differing the number of nested ternary-if:s.

6.2.4 Algorithm 1: Algorithm only using many-body basis as input (non-dynamic energy-bound strategy)

In this section, we describe our first algorithm to find matrix connections. In this algorithm we denote single-particle states by their quantum numbers. Furthermore, the algorithm will use no inputs besides the many-body states, whose connections shall be found. As a starting point in deriving the algorithm, we note that a replacement of a single-particle state only results in a valid connection if the following conditions are fulfilled:

- The total energy of the new many-body state, obtained by the single-particle state replacement, does not exceed the maximum energy N_{tot}.
- The new state has the correct parity.
- The total angular momentum projection has the correct value.
- The newly inserted single-particle state does not coincide with an already occupied state.

What we seek is a way to describe, i.e. parametrize, new single-particle states to replace old ones in the many-body states, so that a subset or all of these conditions automatically are fulfilled[9]. To formulate a way of doing this, we note that the interchange of $\alpha_i = (n_i,l_i,j_i,m_{ji})$ by $\alpha_{\text{new,i}} = (n'_i,l'_i,j'_i,m'_{ji})$, only can be valid provided that:

- $l_i + \sum_{k\neq i} l_k \equiv l'_i + \sum_{k\neq i} l_k \bmod 2$, according to the parity condition. Hence it follows that $l'_i = 2\tilde{l} + l_i \bmod 2$, where $\tilde{l}$ is a non-negative integer
- $m_{ji} + \sum_{k\neq i} m_{jk} = m'_{ji} + \sum_{k\neq i} m_{jk} \Rightarrow m_{ji} = m'_{ji}$, according to the condition for the total magnetic moment projection.

From this, it can be seen that we can use the variables $(\tilde{n},\tilde{l},\tilde{j})$ and limit our efforts to consider single-particle states $\alpha_{\text{new,i}}$, such that:

$$\begin{aligned}
n'_i &= \tilde{n}, \\
l'_i &= 2\tilde{l} + l_i \bmod 2, \\
j'_i &= \tilde{j}, \\
m'_{ji} &= m_{ji}.
\end{aligned} \tag{6.13}$$

[9]Hence we can reduce the number of single-particle state interchanges that must be considered, which is beneficial for performance.

Even though the variables $(\tilde{n},\tilde{l},\tilde{j})$ were introduced when interchanging a single-particle state at a specific position i in a many-body state, it is possible to use the same variables to parametrize interchanging single-particle states for all of the parallel processes in Figure 6.3, simultaneously. All that needs to be done, is to find a control structure that varies $(\tilde{n},\tilde{l},\tilde{j})$ appropriately. The meaning of appropriately, is defined by that all relevant $\alpha_{\mathrm{new,i}}$ should be considered.

To determine the relevant range of $(\tilde{n},\tilde{l},\tilde{j})$, we begin by noting that the maximum total energy condition lets us limit our considerations to triplets for which there exists an i, with:

$$\left(\sum_{k\neq i}(2n_k + l_k) + 2\tilde{n} + 2\tilde{l} + l_i \bmod 2\right) \leq N_{\mathrm{tot}} \tag{6.14}$$

Differently written, we must consider all $(\tilde{n},\tilde{l},\tilde{j})$ such that

$$\tilde{n} + \tilde{l} \leq \frac{1}{2}\left(N_{\mathrm{tot}} - \min_i \left\{\sum_{k\neq i}(2n_k + l_k) + l_i \bmod 2\right\}\right) = \tilde{N}. \tag{6.15}$$

Note that without any pre-calculated control inputs, the only way to incorporate that $\tilde{N}$ depends on the currently processed many-body state, is by using a register. Such implementation will however be associated with delays, due to the pipe-lining structure of the FPGA, and is hence a poor choice. The remaining option is to use the rather brute-force non-dynamic bound

$$\tilde{N} = N_{\mathrm{tot}}/2. \tag{6.16}$$

If we use chained counters, as described in Chapter 5, to parametrize $(\tilde{n},\tilde{l},\tilde{j})$, we must however make the further simplification to allow

$$\max\{\tilde{n},\tilde{l}\} \leq \tilde{N}. \tag{6.17}$$

instead of $\tilde{n} + \tilde{l} \leq \tilde{N}$. Doing so, the total number of clock cycles, to identify the valid connections of a many-body state will be

$$2 \cdot (\tilde{N} + 1)^2. \tag{6.18}$$

As the dimension of the single-particle basis has a cubic dependence on energy, this strategy results in an improvement by a factor $A \cdot N_{\mathrm{tot}}$, compared to if all possible single-particle state interchanges were considered and performed sequentially, for one α_i at a time.

Algorithm implementation with counter-based control structure

The details concerning the implementation of the described algorithm are shown in code in Listing 6.1. The code includes three counters c_1, c_2 and c_3 as control structure, chained to each other and hence forming a three-fold nested loop. Their individual ranges are given by c_1, $c_2 = 0,...,\tilde{N}$ and $c_3 = 0,1$, respectively[10]. In the context of the previous section, we set

$$(\tilde{n},\tilde{l},\tilde{j}) = (c_1,c_2,c_3). \tag{6.19}$$

[10] 0,1 corresponding to $j = |l \pm 1/2|$, respectively.

Based on $(\tilde{n},\tilde{l},\tilde{j})$, the next step in the code is to calculate the quantum numbers of new single-particle states $\alpha_{\text{new,i}} = (n'_i,l'_i,j'_i,m'_{ji})$. This is followed by three checks, to ensure that replacing α_{i} by $\alpha_{\text{new,i}}$, results in a valid many-body state, and hence a connection. These are:

- To check that the constructed single-particle state, actually is a valid single-particle state. In our case, this is equivalent to that
$$V_S = 2(l'_i + j'_i) > |m'_{ji}|, \tag{6.20}$$
where j'_i is represented by 0 or 1 and m'_{ji} is twice its physical value (to let us work with integer types).

- To check that the total energy of the many-body state, after performing the single-particle state interchange, does not exceed its maximum value. This is checked by the condition
$$V_E = N_{\text{tot}} - N + 2n_i + l_i - 2n'_i - l'_i \geq 0, \tag{6.21}$$
where N is the total energy of the input many-body state, calculated in a previous step of the code.

- Finally, we check that the Pauli exclusion principle is fulfilled, i.e. that the newly inserted single-particle state does not coincide with an already occupied state. This means that a new single-particle state can coincide with the state it is replacing, but not any other of the occupied single-particle states. Expressed formally, it shall hold that
$$P = \neg(\alpha_{\text{new,i}} \in \{\alpha_1,...,\alpha_A\} \setminus \{\alpha_i\}). \tag{6.22}$$
Note that this expression will result in that the diagonal of matrix for the one-body operator is produced A times, corresponding to that each of the single-particle states in $[\alpha_1,...,\alpha_A]$ replaces itself.

After all checks have been performed, the newly constructed many-body state is sorted according to Section 6.2.2, resulting in a representation that is used to calculate a hash-key. This is encapsulated by the line of code:
$$\texttt{key} = \text{hash}(\tilde{\alpha}_1,...,\tilde{\alpha}_A), \tag{6.23}$$
with further explanation in Section 6.2.7. We also calculate the sign $\pm_i$, of the performed exchange of single-particle states.

In a final step of the algorithm, it is verified whether all the booleans V_S, V_E and P are true, which would mean that a valid matrix connection has been found. If so, we output the relevant information described in Section 6.1.5, to the CPU.

```
Control structure: Three counters $c_1$, $c_2$ and $c_3$, where the two former takes
    the values 0, 1,..., $\tilde{N}$ and the latter takes the values 0 and 1. The
    counters are chained, meaning that $c_1$, $c_2$ and $c_3$ can be thought of as the
     index variables of a three-fold nested loop.

InputStream: accepts a  many-body state mbState = $[\alpha_1,...,\alpha_A]$, where
    $\alpha_i = (n_i,l_i,j_i,m_{ji})$, if $c_1 = c_2 = c_3 = 0$;
```

```
/*Calculation of total energy of the input many-body state.*/
$N = 2n_1 + ...2n_i + ... + 2n_A + l_1 + ...l_i + ... + l_A$;

/*Parallel processes interchanging single-particle states.*/
for in parallel(i in {1,...,A})

    /*New single-particle state.*/
    $\alpha_{\text{new,i}} = (n'_i, l'_i, j'_i, m'_{ji})$;

    $n'_i = c_1$;
    $l'_i = 2c_2 + l_i \bmod 2$;
    $j'_i = c_3$;
    $m'_{ji} = m_{ji}$;

     /*Check if the single-particle state is valid.*/
    Boolean: $V_S = 2(l'_i + j'_i) > |m'_{ij}|$;

    /*Check  if the energy of the new many-body state is smaller than
     maximum.*/
    Boolean: $V_E = N_{\text{tot}} - N + 2n_i + l_i - 2n'_i - l'_i \geq 0$;

    /*Check whether the new single-particle state already is occupied.*/
        Boolean: $P = \neg(\alpha_{\text{new,i}} \in \{\alpha_1,...,\alpha_A\} \setminus \{\alpha_i\})$;

    /*Calculate the hash-key for the new many-body state, as well as the
        sign of the connection.*/
    key =hash($\tilde{\alpha}_1$,...,$\tilde{\alpha}_A$);

    Calculate $\pm_i$;

    /*If all conditions are met, output the resulting connection.*/
    if($V_S \wedge V_E \wedge P$)
        OutputStream: Array of relevant information about the identified
            connection;
    end

end
```

Listing 6.1: Algorithm based on three-fold chained counters, for calculating valid connections, only using many-body states as input.

Possibilities of using ROMs for complex parametrization patterns

In the previous algorithm outline, we were unable to exactly parametrize the triangular region, defined by the non-negative integer solutions of

$$\tilde{n} + \tilde{l} \leq \tilde{N}. \tag{6.24}$$

Instead, we made the simplification to parametrize the quadratic region, defined by the non-negative integer solutions of

$$\max\{\tilde{n}, \tilde{l}\} \leq \tilde{N}, \tag{6.25}$$

which resulted in a doubling of the total number of clock cycles used for calculation. To obtain a better parametrization, we note that counters primarily are structures that can

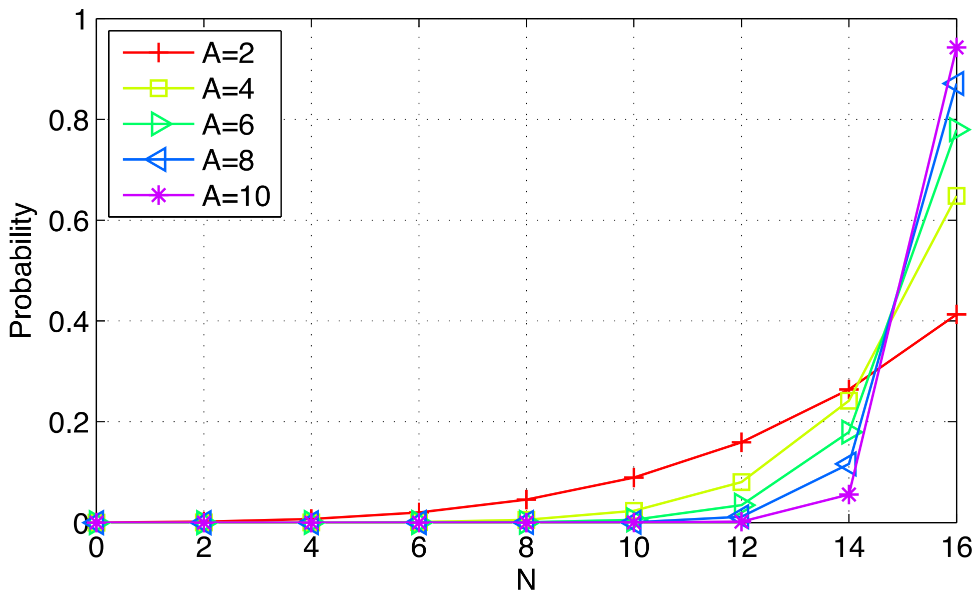


Figure 6.4: Proportion of states in the many-body basis having a specific energy N. This is shown for a number of different A, but with all bases having $N_{\text{tot}} = 16$. Note that most of the many-body states have the highest possible energy, i.e. $N = 16$.

be used to internally govern the state of the FPGA, while memory is more useful for describing meaningful properties. Specifically we could enumerate all valid pairs $(\tilde{n},\tilde{l},\tilde{j})$, defined by (6.24), and store them in on-chip ROM. A counter could then be used to generate addresses to access them. This also extends to more complex parametrization patterns, where the option of using chained counters, is not viable.

6.2.5 Algorithm 2: Algorithm with dynamically set upper energy-bound

In our first algorithm, we were forced to use a non-dynamic energy-bound, i.e. we could not make use of that

$$\tilde{n} + \tilde{l} \leq \frac{1}{2}\left(N_{\text{tot}} - \sum_{k\neq i}(2n_k + l_k) - l_i \bmod 2\right), \tag{6.26}$$

both has a dependency on the many-body state, being processed, as well as the replaced single-particle state, within that many-body state. The right hand side in (6.26) can however be pre-calculated off the FPGA and supplied as a control input to limit the number of single-particle state interchanges that the FPGA considers. Doing this, will be our second suggestion for an algorithm that identifies connections. In the course of this, we will make the slight change of notation to denote the right hand side of (6.26) by $\tilde{N}_{\text{dynamic}}$. Our reasons to believe that dynamically setting the energy-bound $\tilde{N}_{\text{dynamic}}$ is beneficial, stems from that:

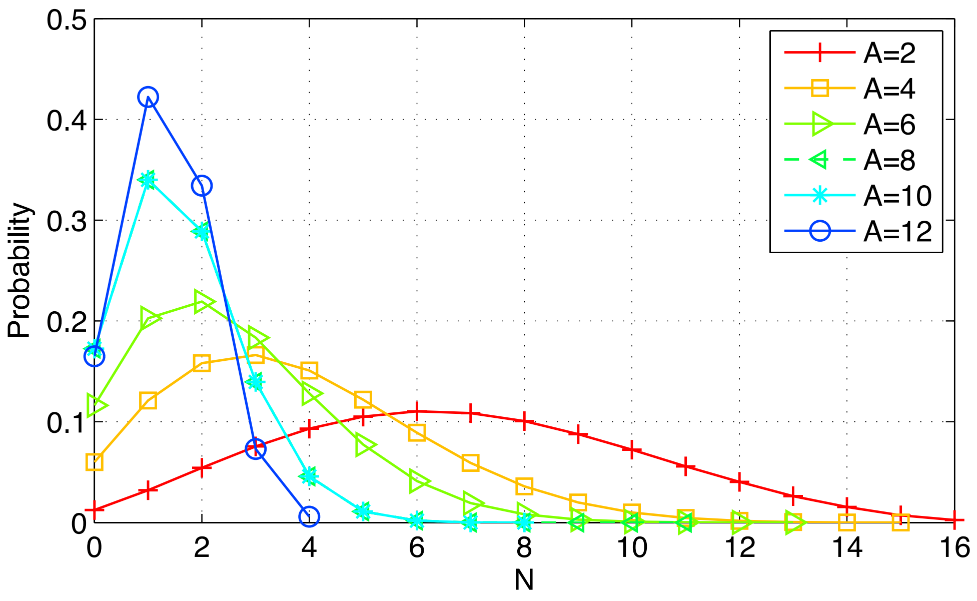


Figure 6.5: Energy distributions of single-particle states within the many-body states, for a number of different many-body bases, with $N_{\text{tot}} = 16$. The distributions are shifted to lower energies as the number of particles is increased.

- A large proportion of the many-body states have an energy close to the maximally allowed total energy. This is indicated in Figure 6.4, which shows how the total energy is distributed, among the states in the many-body basis, for bases with $N_{\text{tot}} = 16$, but different number of particles.

- Single-particle states of low energies are predominant in the many-body states, compared to their occurrence in the single-particle basis. This can be motivated by that the average energy of the single-particle states in a many-body state always must be smaller than or equal to

$$N_{\text{tot}}/A. \tag{6.27}$$

  The effects of this are numerically illustrated in Figure 6.5, where we have calculated the energy-distribution of single-particle states, that occur in many-body states of various many-body bases.

The conclusion is that the total energy of a many-body state, often is close to N_{tot}, but that the removal of one single-particle state, does not have a significant effect on the energy of the state. This exactly means that using the worst case bound $\tilde{N} = N_{\text{tot}}/2$, as was done in the first algorithm, leads to severely redundant case testing, compared to if we would use $\tilde{N}_{\text{dynamic}}$ to limit the number of single-particle state interchanges that are considered.

To implement a dynamic energy-bound, there are two options. One of these is to use $\tilde{N}_{\text{dynamic}}$ as given by (6.26). The interchanges of single-particle states at different

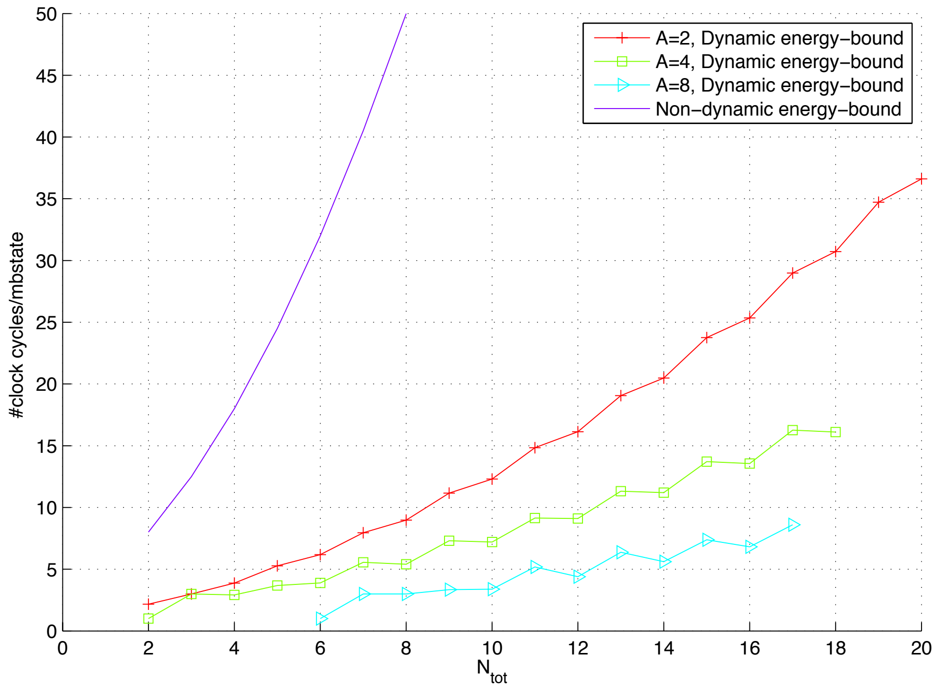


Figure 6.6: Average number of clock cycles per many-body state, needed to find all valid connections, as a function of N_{tot}, when using the dynamic energy-bound strategy. Also included is the corresponding average number of clock cycles for the algorithm with non-dynamic energy-bound.

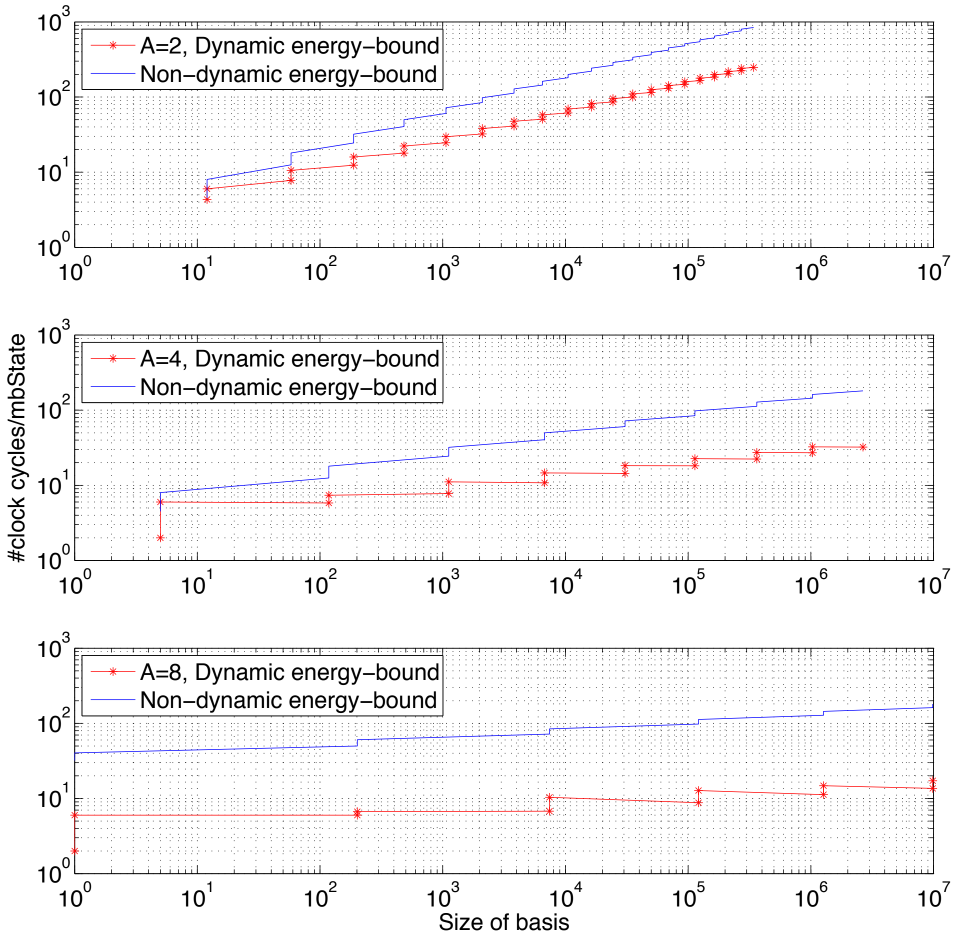


Figure 6.7: The number of clock cycles, needed to find all valid connections, per many-body state, when using the dynamic energy-bound strategy, as a function of dimension of the many-body basis, using a logarithmic scale. Also included is the corresponding number of clock cycles, for the non-dynamic energy-bound.

positions in the list-representation for many-body states, would then need individual control structures, provided that we insist on using the kernel-design in Figure 6.3. This is however not a particularly well-suited solution to implement on the FPGA hardware, but as the kernel-design in Figure 6.3 has several (earlier described) other benefits, we chose to make the slight alteration of the algorithm, to use a uniform (but still dynamically set) energy-bound, defined by

$$\tilde{N}_{\text{dynamic,uniform}} = \frac{1}{2}\left(N_{\text{tot}} - \min_i \left\{ \sum_{k\neq i}(2n_k + l_k) + l_i \bmod 2 \right\}\right). \tag{6.28}$$

This energy-bound is more straightforward to supply and incorporate in the control structure of the kernel.

Implementation of kernel using dynamic energy-bound

The kernel that is used to implement the dynamic energy-bound $\tilde{N}_{\text{dynamic,uniform}}$ is similar to that for our first algorithm. The only difference is that we use control inputs to adjust the control structure to consider a more limited set of values for $(\tilde{n},\tilde{l},\tilde{j})$. How this is done depends on whether the control structure is formed by chaining counters, as was illustrated for the first algorithm, or is a single counter that parametrizes memory addresses, storing the values of $(\tilde{n},\tilde{l},\tilde{j})$.

If the alternative involving chained counters is used, we need two control inputs to set the maximums of the counters which set the values of $\tilde{n}$ and $\tilde{l}$. Using these control inputs we are actually able to exactly parametrize the non-negative integer solutions of $\tilde{n} + \tilde{l} \leq \tilde{N}_{\text{dynamic,uniform}}$, without resorting to simplifications such as $\max\{\tilde{n},\tilde{l}\} \leq \tilde{N}_{\text{dynamic,uniform}}$, which we were forced to do when we did not use control inputs.

In explaining how the ROMs can be used to implement the dynamic energy-bound, we chose for simplicity to forget about $\tilde{j}$ and only consider $(\tilde{n},\tilde{l})$. These pairs are then stored in on-chip ROM, with memory addresses ordered according to increasing $\tilde{n} + \tilde{l}$.

Example. If we use ROMs to parametrize $(\tilde{n},\tilde{l})$, and we allow energies such that $\tilde{n} + \tilde{l} \leq 2$, we can according to the above description store $(\tilde{n},\tilde{l})$ with the following addressing:

Address	1	2	3	4	5	6
$\tilde{n}$	0	0	1	2	1	0
$\tilde{l}$	0	1	0	0	1	2
$\tilde{n}+\tilde{l}$	0	1	1	2	2	2
Index	1		3			6

■

In the above table, we have indicated the highest memory address for each $\tilde{n} + \tilde{l}$, with an index. Now, if we process a many-body state which has the dynamic energy-bound $\tilde{N}_{\text{dynamic,uniform}}$, we should let the counter used as control structure range from zero up to the index corresponding to $\tilde{N}_{\text{dynamic,uniform}} = \tilde{n} + \tilde{l}$. If the values for $\tilde{n}$ and $\tilde{l}$, stored at the corresponding memory addresses, are used to perform single-particle state interchanges, we will consider all necessary possibilities. To reset the counter, so that

Table 6.1: Values for the constants C_A and E_A, that describe the quotient of the number of clock cycles, $M(d)$, for the non-dynamic energy-bound and dynamic energy-bound strategy, needed to find matrix connections.

A	C_A	E_A	$M(10^9)$	$M(10^{12})$
2	1.1990	0.0891	7.6	14.1
4	1.5411	0.0902	10.0	18.6
8	7.2905	0.0157	10.1	11.3

the kernel can start processing a new many-body state, once it has reached the index of $\tilde{N}_{\text{dynamic,uniform}}$, we use a reset-signal. This is much more I/O-efficient, than the above use of two control inputs to set maximum values for counters, since a reset-signal only needs to transfer one bit per clock cycle, to the FPGA.

Numerical evaluation of uniformly setting the upper energy-bound

To determine the performance when using $\tilde{N}_{\text{dynamic,uniform}}$ to parametrize the interchanges of single-particle states, we have calculated the average number of clock cycles needed to process a many-body state, as a function of N_{tot}. This is shown in Figure 6.6. We have also included the corresponding number of clock cycles, for our first algorithm with a non-dynamic energy-bound. Note that the gain compared to our first algorithm is largest for systems of many particles.

Even though Figure 6.6 indicates that the benefits of setting a dynamic energy-bound increases with energy, it is less obvious how to quantify this trend. To easier do this, we have calculated the average number of clock cycles as a function of the dimension of the many-body basis, shown in Figure 6.7. The Figure indicates that the average number of clock cycles has a linear dependence on dimension, both for the non-dynamic and dynamic energy-bound, with respect to a logarithmic scale. The quotient of the number of clock cycles needed when using the non-dynamic energy-bound and the corresponding number for the dynamic energy-bound, can hence be written as

$$M(d) = C_A d^{E_A}, \tag{6.29}$$

where d is the dimension of the many-body basis and C_A, E_A are constants, depending on A. The constants can be determined by linear curve-fitting, with values found in Table 6.1. The table also shows extrapolated values, for systems of higher dimension, than those considered in this study. Note that $M(d)$ represents the performance increase from using the dynamic energy-bound compared to the non-dynamic energy-bound.

6.2.6 Algorithm 3: Algorithm using m_j-parity groups

We now turn to our third algorithm for finding connections of a one-body-operator. In this algorithm we no longer denote single-particle states by their quantum-numbers, but rather use an index in an enumeration. Specifically we group the single-particle states in such a way that interchanging particles belonging to the same group results in that the m_j- as well as parity-condition is fulfilled. The strategy is hence called the m_j-parity group strategy and it will be shown that the groups can be parametrized so that we only perform interchanges of single-particle states that fulfill the maximum total energy condition. Hence we can restrict the interchanges of single-particle states in a similar way as when using the dynamic energy-bound strategy, but with the additional advantages:

- By using an index to parametrize single-particle states, we can restrict our attention to consider interchanges for which a removed single-particle state is replaced by one with a higher index. Since the relation representing connecting many-body states is symmetric, this induces no loss of generality, but ideally lets us limit the number of tested interchanges by a factor two.
- We only attempt to replace single-particle states with entities that indeed are valid single-particle states. Using the strategies that work directly with the quantum numbers of single-particle states, this could not with certainty be assured.

Construction of m_j-parity groups

The m_j-parity groups refer to the following sorting of the single-particle basis, which we may assume contains all single-particle states, with energies not larger than $N_{\text{single,max}}$:

- Single-particle states with the same value of m_j and parity are said to form an m_j-parity group. The states within each such m_j-parity group are listed according to decreasing energy.
- The ordered lists corresponding to different m_j-parity groups are concatenated to form a single list containing all single-particle states in the basis. The single-particle states are then assigned the index they obtain in this list.

An example of this, corresponding to $N_{\text{single,max}} = 2$, is shown Table 6.2. In this table, we have also assigned the different m_j-parity groups with a group index, which can be used to identify them.

Algorithm outline

Based on the presentation in the previous section, we are now ready to present how the m_j-parity groups shall be used to find connections for a one-body-operator. To do this, we supply the FPGA, with two arrays

$$\texttt{mbState} = [\alpha_1,...,\alpha_A] \quad \text{and} \quad \texttt{maxState} = [\alpha_{1,\text{max}},...,\alpha_{A,\text{max}}],$$

where the first array represents a many-body state, where α_i are the indices of its single-particle states, sorted in ascending order. An element $\alpha_{i,\text{max}}$ of `maxState` is called a maximum state, and is given by the highest index for which there is a single-particle state belonging to the same m_j-parity group as α_i. Hence, the maximum states are properties of the m_j-parity groups. These are exemplified in Table 6.3, for the case of a single-particle basis with $N_{\text{single,max}} = 2$.

Table 6.2: Table showing how the single-particle basis can be sorted, according to the description of m_j-parity groups, in the case of $N_{\text{single,max}} = 2$.

$N = 2n + l$	l	j	$2m_j$	Parity	Group index	Index
2	2	1	-5	1	1	1
2	2	1	-3	1	2	2
2	2	0	-3	1	2	3
2	0	1	-1	1	3	4
2	2	1	-1	1	3	5
2	2	0	-1	1	3	6
0	0	1	-1	1	3	7
2	0	1	1	1	4	8
2	2	1	1	1	4	9
2	2	0	1	1	4	10
0	0	1	1	1	4	11
2	2	1	3	1	5	12
2	2	0	3	1	5	13
2	2	1	5	1	6	14
1	1	1	-3	-1	7	15
1	1	1	-1	-1	8	16
1	1	0	-1	-1	8	17
1	1	1	1	-1	9	18
1	1	0	1	-1	9	19
1	1	1	3	-1	10	20

Table 6.3: Table showing the maximum states corresponding to m_j-parity groups, for the m_j-parity groups formed in Table 6.2.

Group index	1	2	3	4	5	6	7	8	9	10
Maximum state	1	3	7	11	13	14	15	17	19	20

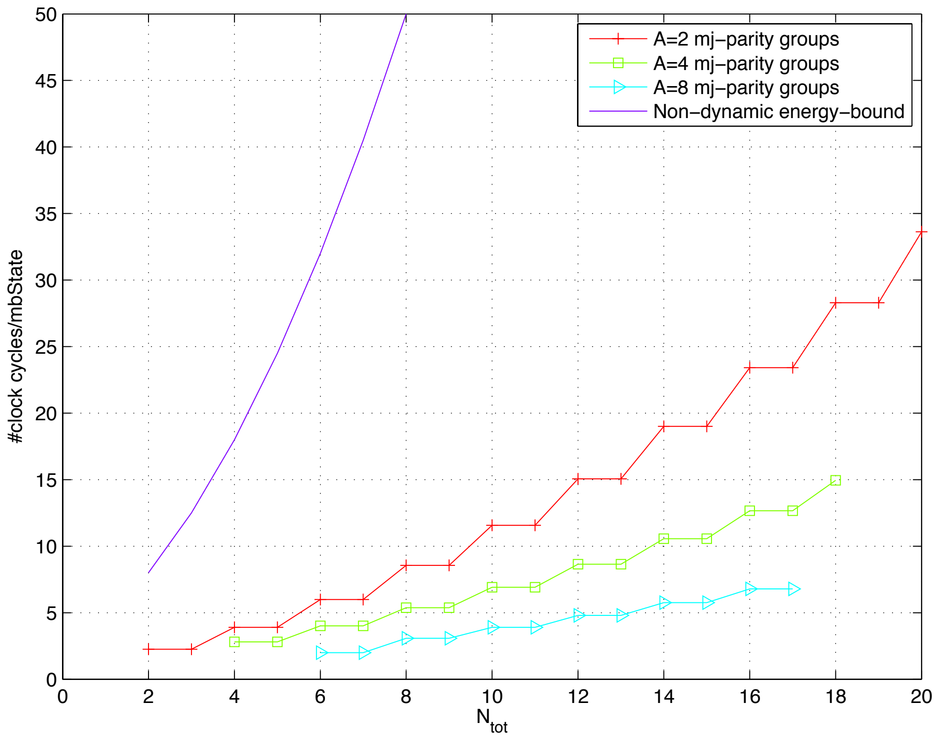


Figure 6.8: Average number of clock cycles per many-body state, needed to find all valid matrix connections, as a function of N_{tot}, when using the m_j-parity group strategy. Also included is the corresponding numbers of clock cycles for the non-dynamic energy-bound strategy.

Using the two inputs, we can employ the kernel-design shown in Figure 6.3. Its parallel processes are used to generate connections by increasing the indices of the individual single-particle states in `mbState`. To generate all connections, this should be performed for as long as the new indices are smaller than or equal to their corresponding maximum states. The resulting single-particle state interchanges not only preserve the total angular momentum projection and parity, but also fulfill the total energy-constraint, since the energy of the single-particle states form a decreasing function of index (within each given m_j-parity group). As control structure to perform the single-particle state interchanges, we use a counter. To perform the interchanges, we simply add the value of this counter to the respective indices of the single-particle states in `mbState`. To perform all necessary single-particle state interchanges, we should let the counter range up to

$$\Delta = \max\{\texttt{maxState} - \texttt{mbState}\}. \tag{6.30}$$

Note that Δ is dependent on the many-body state that is being processed and needs to be supplied to the FPGA, as a pre-calculated reset-signal `reset`. This reset-signal should be used to reset the counter whenever it reaches its maximum value Δ, followed by that the kernel accepts a new many-body state, to be processed.

Example Consider a system with four particles and $N_{\text{single,max}} = 2$. The many-body state

$$\texttt{mbState} = [1,4,8,12] \tag{6.31}$$

is then, according to the Tables 6.2 and 6.3, seen to correspond to the maximum list

$$\texttt{maxState} = [1,7,11,13]. \tag{6.32}$$

We also have that

$$\texttt{maxState} - \texttt{mbState} = [0,3,3,1] \tag{6.33}$$

and hence that $\Delta = 3$. The counter that governs the single-particle state interchanges should therefore be supplied with the `reset`-signal[11]

$$\texttt{reset} = 0001, \tag{6.34}$$

which resets its value, once it reaches 3. ■

Now, lets finally have a look on the checks that must be performed in order to ensure that an A-tuple, constructed according the above method, is a valid many-body state and hence results in a valid connection. In total, these are:

- Verifying that the Pauli exclusion principle holds, i.e. that the inserted single-particle state does not coincide with some already occupied state.
- Checking that the inserted state does not exceed its maximum state, i.e. belongs to the same m_j-parity group as the removed single-particle state.

[11]The zeros in `reset` correspond to that the counter does not reset its value, i.e. continues to increment its value by one, each clock cycle. Since we want the counter to count from 0 to 3, we begin by supplying it with three zeros. However, in the fourth clock cycle where it has the value 3, we want it to reset to zero, so that the kernel can start processing a new many-body state. This is done by supplying the number one at the fourth position of `reset`.

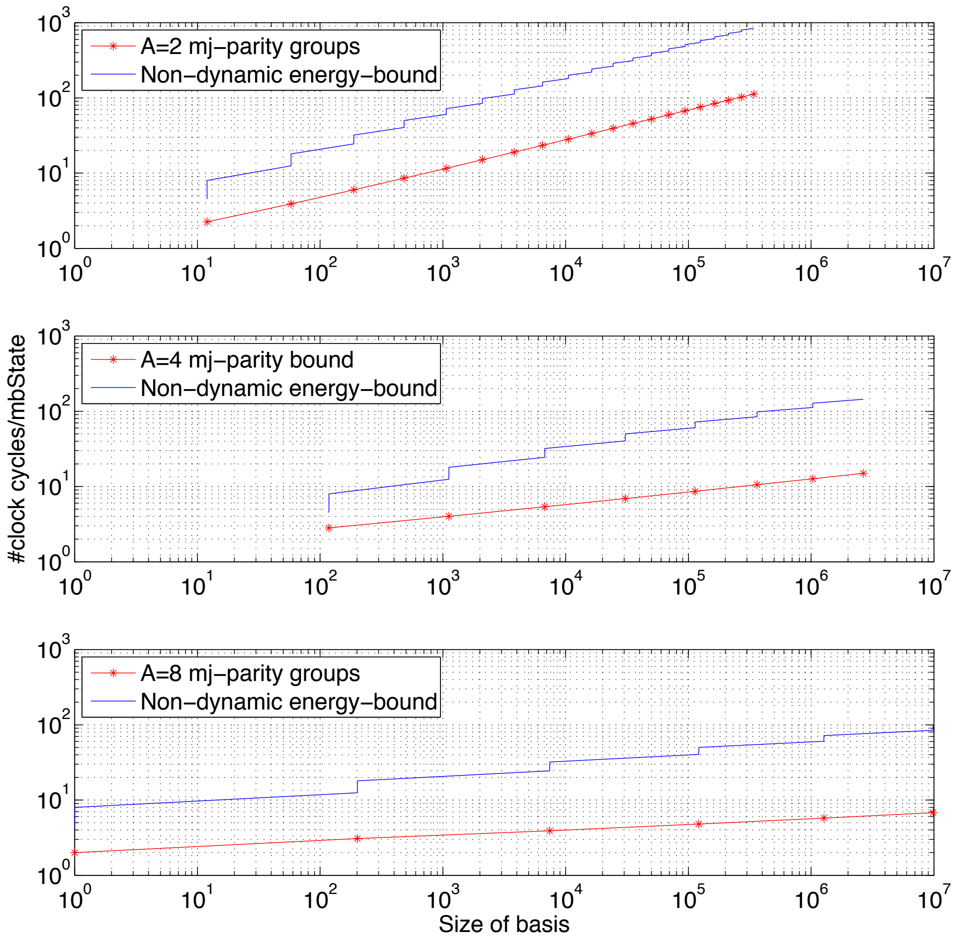


Figure 6.9: The number of clock cycles, needed to find all valid connections, per many-body state as a function of the dimension of the many-body basis, when using the m_j-parity group strategy. Also included are the corresponding numbers of clock cycles, for the non-dynamic energy bound strategy.

The former of these checks is implemented equivalently to what is done in Section 6.2.4, while the latter simply is the evaluation of an inequality. The last check does in some sense capture the simplicity of the method, i.e. that all three checks concerning that a many-body state has the correct total angular momentum projection, parity and energy can be summarized in one inequality.

Numerical evaluation of algorithm performance

To evaluate performance, we have calculated the average number of clock cycles needed to process a many-body state, as a function of N_{tot}, for a number of different A, when

Table 6.4: Values for the constants C_A and E_A, that describe the factor by which the total number of clock cycles is reduced, $M(d)$, when using the m_j-parity groups instead of a non-dynamic energy-bound strategy to find matrix connections.

A	C_A	E_A	$M(10^9)$	$M(10^{12})$
2	2.8199	0.0865	16.9	33.1
4	3.4069	0.0945	24.1	46.3
8	12.9565	0.0328	24.8	32.1

using the m_j-parity group strategy. This is shown in Figure 6.8, where we also have included the average number of clock cycles needed when using the non-dynamic energy-bound strategy. Compared to this benchmark we see that the m_j-parity groups achieve a significant reduction of the number of necessary clock cycles. Compared to the corresponding numbers for the dynamic energy-bound strategy, there is also an improvement by a factor approximately two, which is expected as the m_j-parity group strategy only considers connections for which single-particle states are interchanged by new states with higher indices.

Finally, to predict how the performance of the m_j-parity group strategy develops compared to the performance of the non-dynamic energy-bound strategy, for larger systems, we have illustrated the average number of clock cycles used to process a many-body state, for both of these strategies, as a function of the dimension of the many-body basis. This is shown in Figure 6.9. The Figure indicates that the average number of clock cycles has a linear dependence on dimension, both for the m_j-parity groups and the no-dynamic energy-bound strategy, with respect to a logarithmic scale. The quotient of the number of clock cycles needed when using the non-dynamic energy-bound and the corresponding number for the m_j-parity groups, can hence be written as

$$M(d) = C_A d^{E_A}, \tag{6.35}$$

where d is the dimension of the many-body basis and C_A, E_A are constants, depending on A. Their values values are found in Table 6.4, determined by linear curve-fitting. The table also shows extrapolated values of $M(d)$, for a selection of systems of higher dimension. These can be compared to equivalent values for the dynamic energy-bound strategy, found in the previously introduced Table 6.1.

6.2.7 Implementation of hash map

In our FPGA-implementations, we need to use a hash-function to retrieve probability amplitudes corresponding to many-body states. For a discussion of the general principles of hashing, the reader is referred to appendix B. Here, we will present an XOR-shift hash-algorithm [15], which operates on binary words. To begin with, we must therefore convert $[\alpha_1,...,\alpha_A]$ into a binary word. If α_i are represented by indices, this can be done by concatenating the bits of these numbers. When using the quantum number representation for α_i, we can concatenate the corresponding images under the mapping (6.7). The binary word w, formed according to this method, is unique and can hence be used as a key. To this key, we should then apply XOR-shift operations:

$$\begin{aligned} w &= w^\wedge(w >> p_1), \\ w &= w^\wedge(w << p_2), \\ w &= w^\wedge(w >> p_3), \end{aligned} \tag{6.36}$$

where p_1, p_2 and p_3 are appropriately chosen primes, to ensure sufficiently long period [15]. To convert the word into a memory address of appropriate size (number of bits), lets say l, we slice the word into parts of length l, followed by applying the XOR-operation to combine the resulting parts. Collisions are dealt with, by storing the key together with its probability amplitude.

To evaluate the performance of the described hashing algorithm, we have made calculations based on a many-body basis, corresponding to $A = 6$ and $N_{\mathrm{tot}} = 10$. This many-body basis has the dimension $d = 364728$. A compact, non-hashing approach, for storing the probability amplitudes would then demand

$$\log_2(d) = 18.48 \tag{6.37}$$

bits for addressing. If we use hashing, we must however oversize the addressing space by some appropriate factor. The effect of this, is shown in Figure 6.10, where we have calculated the number of elements occupying each memory address, for a number of different sizes of the addressing space. The Figure indicates that if we use an 19 bit addressing space, we will not have more than ten collisions for any memory address. Additional over-sizing does not have a significant effect on the worst case number of collisions.

Besides the worst case, number of collisions, an important feature is the distribution of the number of collisions. This is shown in Figure 6.11, from which we conclude that 19 bits for addressing results in 70 % chance of no collisions, with almost zero probability of more than two collisions.

6.3 CPU-based computations

In this section, we present the calculation of the reduced transition density, which is based on output produced by the FPGA, as illustrated in Figure 6.1. We also make a comment about the off-FPGA pre-calculation costs for the algorithms using control inputs.

6.3.1 Pre-calculation costs

First and most importantly, the calculation cost for control inputs is linear in terms of the dimension of the many-body basis. Also, the control inputs are determined by properties such as the energy of many-body states and their calculation could hence be incorporated in the generation of the many-body basis, where such quantities already are calculated. In conclusion, the calculation of control inputs can be performed with limited computational costs.

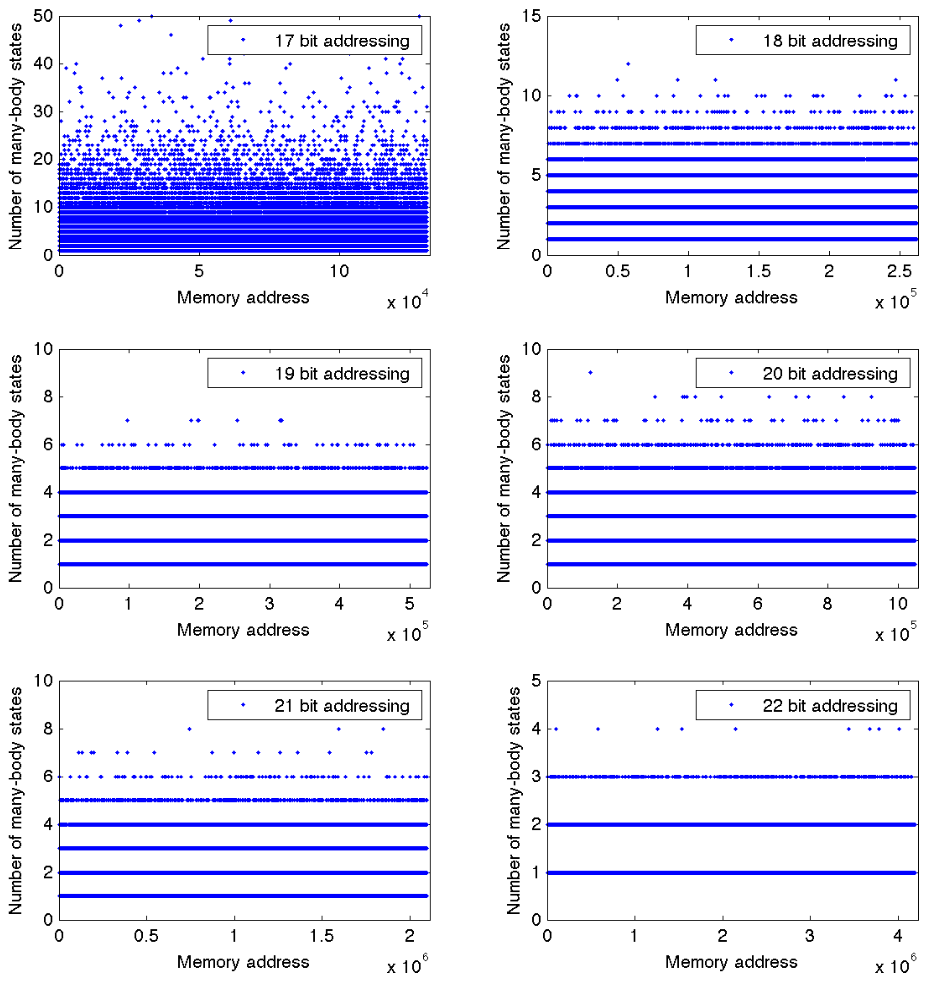


Figure 6.10: Number of many-body states per memory address, when using the XOR-shift hash-function, for different sizes of the addressing space. Note that 18.48 is the minimal number of bits that are needed to assign each item an individual address.

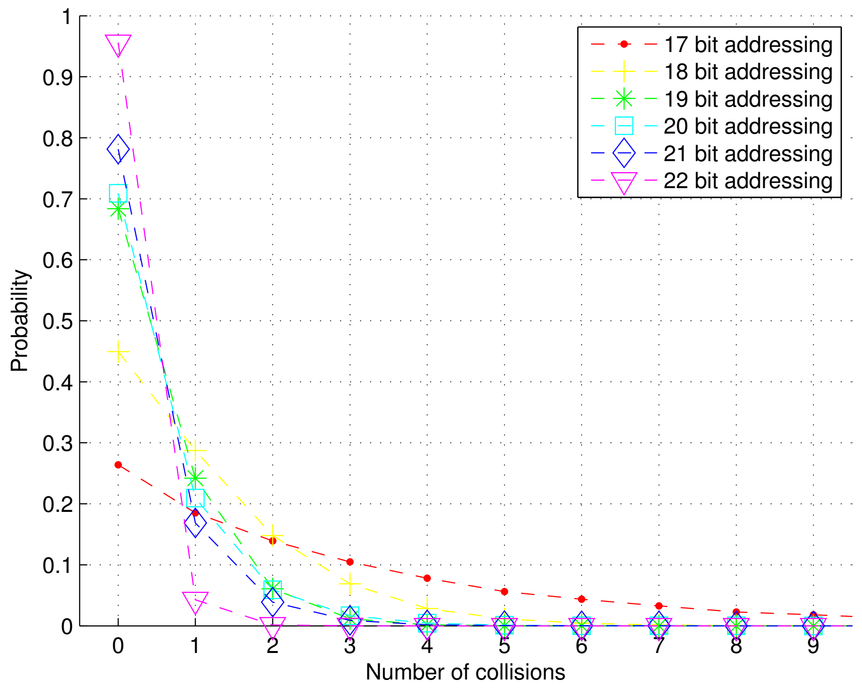


Figure 6.11: Distribution of collisions when using the XOR-shift hash-function, applied to a many-body basis of size 18.48 bits.

6.3.2 Calculation of reduced matrix representation

Once the FPGA-based computation has found all valid connections, we use a CPU to compute the reduced matrix elements of the transition density. This section presents an algorithm for this computation, which can be implemented in `C++`, MATLAB or similar programming languages. The approach is quite straightforward, but with some less obvious steps, concerning which this text aims at bringing clarity.

Since the reduced matrix-representation is defined with respect to a single-particle basis with quantum numbers (n,l,j) instead of (n,l,j,m_j), we begin by creating a mapping from the latter to the former states. To explain this mapping, we define

- A two-dimensional array `spBasis`, whose constituent rows contains the quantum numbers of single-particle states, for the original basis using (n,l,j,m_j).
- A two-dimensional array `reducedSpBasis` whose constituent rows contains the quantum numbers of single-particle states, for the reduced basis using (n,l,j).

The mapping is then constructed as a vector `spBasisMapping`, of the same length as the `spBasis`. It is constructed according to the rule:

`spBasisMapping`$[i]$ = index of the single-particle state in `reducedSpBasis` which has the same first three quantum numbers as `spBasis`$[i] = (n,l,j,m_j)$.

Now, we have established all necessary tools to use FPGA-output, to calculate the reduced transition density matrix elements. The code for this is shown in Listing 6.2. It uses the following inputs:

- The arrays `spBasis`, `reducedSpBasis` and `spBasisMapping`, constructed according to the previous description.
- Lists of created `a+` and destroyed `a-` single-particle states, with respect to their indices in `spBasis`, for the found connections. These are output from the FPGA.
- Signed products of probability amplitudes, denoted `amplitudes`, for the found connections. These are retrieved through lookups, using FPGA-output.
- Total angular momenta of the initial and final eigenstates in order to choose the correct ranks of the spherical tensor, described in Chapter 4.

The method of calculation, is to sequentially consider the found connections and add their contributions to the reduced matrix elements of the transition density. The results are finally written to a file. The detailed motivation of the steps is treated in Chapter 4.

```
Input spBasis /* Single-particle basis.*/
Input reducedSpBasis /*Reduced single-particle basis. */
Input spBasisMapping /* Mapping from spBasis to reducedSpBasis */
Input amplitudes /* Array of products of amplitudes for all connections,
    including the sign of the connection. */
Input a+ and a- /* Arrays of inserted/destroyed single-particle state in
    all valid connections. */
Input $J_i$ and $J_f$ /* Angular momenta of initial and final eigenstates. */
```

```
Array OpRanks = $\{|J_i - J_f|, |J_i - J_f| + 1, ..., J_i + J_f\}$ /* Only spherical tensors of these ranks will couple the initial and final state. */
for(J in OpRanks)
    Allocate transitionDensity$_J$ = 2D-array of size (length of reducedSpBasis)$^2$; /* In these 2D arrays, we will save reduced transition density matrix elements. */
end

$M_i$ = $M_f$ = minimum of $J_i$ and $J_f$; /* Arbitrary choices for momentum projection, will be used to find Clebsch-Gordan coeffs. */

/* For all connections, find what transition density elements the connection contributes to and add it there. */
for (i ∈ all connections) /* i.e. the length of amplitudes or a+/a-. */
    $M = 0$; /* Arbitrary choice for tensor component, Wigner-Eckart guarantees independence. */
    $m$ = spBasis(a-($i$,4)); /* m-value of destruction operator. */
    $j_1$ = spBasis(a-($i$,3)); /* j-value of destruction operator. */
    $j_2$ = spBasis(a+($i$,3)); /* j-value of creation operator .*/

    /* Get the indices for the reduced representations of the current operators. */
    a-,red = spBasisMapping(a-);
    a+,red = spBasisMapping(a+);

    Jlist = intersection of opRanks and $\{|j_1 - j_2|, |j_1 - j_2| + 1, ..., j_1 + j_2\}$; /* The only spherical tensor ranks which the current operators may contribute to .*/

    /* Add the connection where it contributes. */
    for(J in Jlist)
        transitionDensity$_J$(a+,red,a-,red) = transitionDensity$_J$(a+,red,a-,red)
         $+(-1)^{j_2 - m}$ * ClebschGordan($j_1$,$j_2$,$J$,$m$,$-m$,$M$) * amplitudes;
    end
end

/* Multiply all 2D-transition densities with common factor independent of a-,red and a+,red */
for(J in OpRanks)
    transitionDensity$_J$ =
        $\sqrt{2J+1}$ * transitionDensity$_J$/(($(-1)^{J_i - M_i}$ClebschGordan($J_f$,$J_i$,$J$,$M_f$,$-M_i$,$M$));
    Write transitionDensity$_J$ to file;
end
```

Listing 6.2: Algorithm for computation of a reduced transition density, using FPGA output, following (4.40).

Chapter 7

Results

In this chapter we present results that will lead to a conclusion regarding the feasibility of implementing the calculation of transition densities on an FPGA. These results will be presented as an analysis in two parts, where the first part concerns the performance of FPGA-kernels while the second part treats calculations made on a CPU, to convert FPGA output to reduced transition densities, as described in Chapter 6.

In the first part, we will specifically verify that the implemented kernels produce the correct results, study their hardware costs, and make timing measurements to determine their efficiency. The main focus will however not be optimization, but rather to give reasonable estimates for the bottle-necks of the calculation.

The second part of the analysis will consist of timing measurements on the conversion of FPGA-output to a reduced transition density. In addition to this, we will compare the performance of the FPGA-implementations with the performance of TRDENS [18], a CPU-based code currently used by the nuclear theory group at the Department of Fundamental Physics at Chalmers, which calculates transition densities in reduced form.

7.1 Implemented algorithms

To evaluate the possibility of using FPGAs to calculate transition densities, we have made efforts to implement the computational algorithms, described in Chapter 6, to a maximum extent. However, due to time limits, the following restrictions have been made:

- Timing measurements and hardware-cost evaluations have been performed on full implementations of kernels, as described in Chapter 6. These are designed such that memory lookups should be performed outside of the FPGA using hash-keys, which are provided as output.

- The off-FPGA calculation of transition densities has been implemented, except the part that makes lookups by the use of hashing. Instead, we have used an intermediate step that finds the addresses of probability amplitudes by making searches in an array of many-body states. This constitutes an unnecessarily time-consuming step, but it is a step which vanishes whenever a hash implementation is available for the CPU code. Also, the implementation is in MATLAB, which typically will not be optimal for performance.

Concerning the FPGA-implementations, we have also been forced to make a non-optimal design choice, since the high-level part of the Maxeler programming interface does not easily allow an unknown amount of output to be read from the FPGA. This is however needed, since we cannot a priori know how many non-zero matrix elements that will be found. As a temporary work-around, we have programmed the kernels to create output every clock-cycle, with a dumb value if no connection is found. With this solution, the implementations suffer higher risks of hitting I/O-bounds. Note, however, that this non-optimal design choice is not an indication that FPGAs would be less suited for the current calculations, but rather that the implementation demands more advanced programming in a HDL.

Finally, the specific FPGA-kernels, that were implemented, are

- Kernels using counters as control structure and the non-dynamic energy-bound, for systems of sizes $A = 2$, 4 and 8.
- Kernels using counters as control structure and the dynamic energy-bound, for systems of sizes $A = 2$, 4 and 8.
- Kernels using m_j-parity groups, for systems of sizes $A = 2$, 4 and 8.

References to source-code are found in Appendix C.

7.2 Verification of correctness

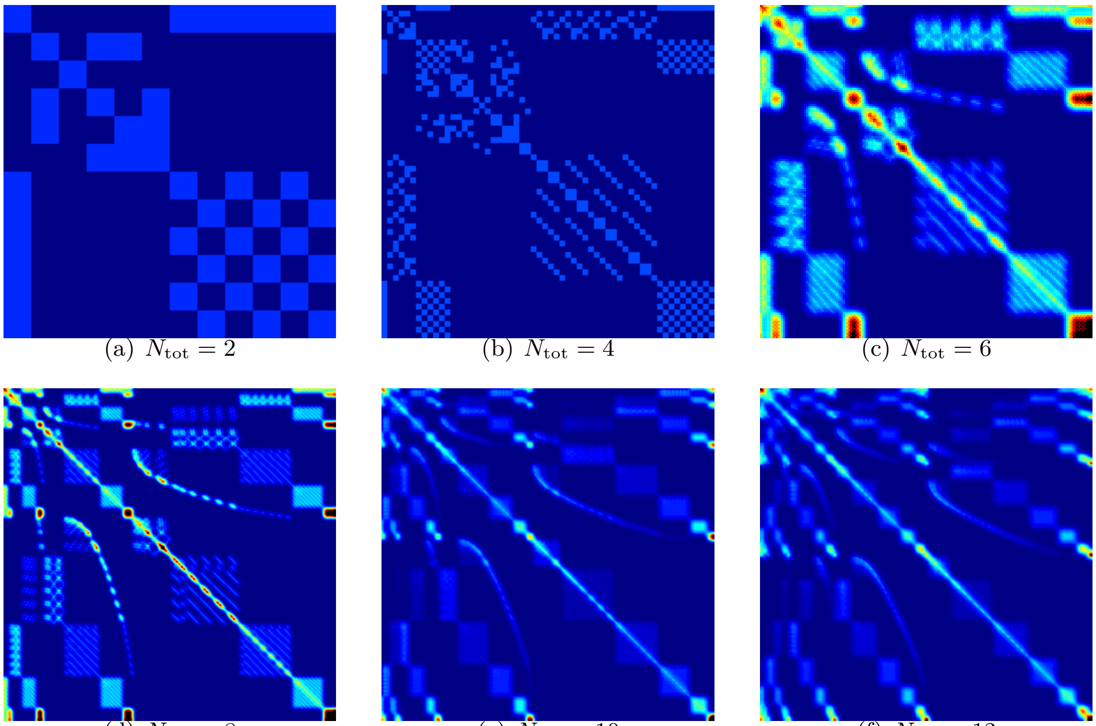
To verify that the implemented kernels produce the correct results, their output has been compared to the corresponding output from a MATLAB script (see Appendix C). This script uses a reliable, though not very efficient, brute-force approach.

As a result, we have only been able to test systems of dimension up to 10^4. For larger systems we have counted the number of non-zero matrix elements, and confirmed that this number is the same between the different kernel-designs.

In addition to testing the correctness of the kernel-calculations, we have also verified the results of the MATLAB-implementation which converts FPGA-output to a reduced matrix-representation. This has been done by comparing the output with TRDENS, for systems where input to the latter code has been available.

7.3 Calculated one-body matrices and kernel load

By using the implemented kernels, we have calculated the non-zero matrix elements of one-body operators, for a number of different cutoff energies, which is exemplified in the Figures 7.1, 7.2 and 7.3. The densities in these matrices might appear blurred; this is an intentional choice to increase the visibility of the non-zero elements. Furthermore, the scale is individual for each matrix in an attempt to make each figure as informative as possible.


(a) $N_{\text{tot}} = 2$ (b) $N_{\text{tot}} = 4$ (c) $N_{\text{tot}} = 6$

(d) $N_{\text{tot}} = 8$ (e) $N_{\text{tot}} = 10$ (f) $N_{\text{tot}} = 12$

Figure 7.1: Density of non-zero matrix elements for one-body operators, corresponding to $A = 2$ and $N_{\text{tot}} = 2$, 4, 6, 8, 10 and 12.

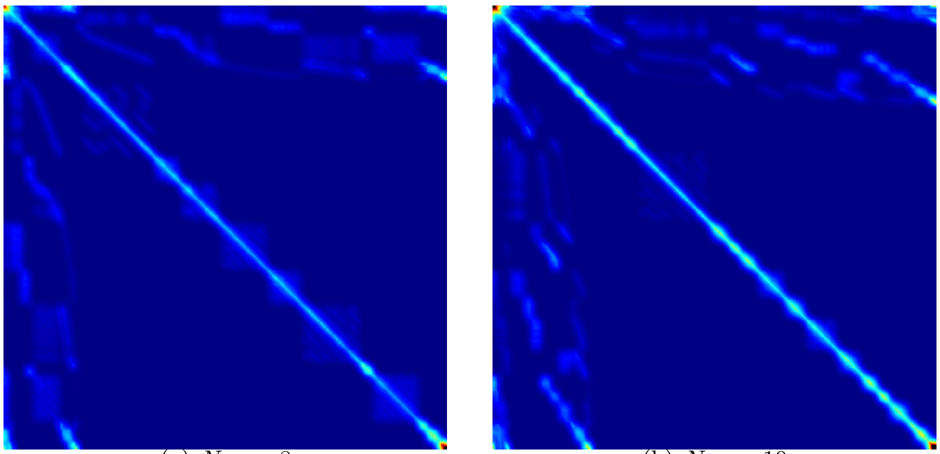

(a) $N_{\text{tot}} = 8$ (b) $N_{\text{tot}} = 10$

Figure 7.2: Density of non-zero matrix elements for one-body operators, corresponding to $A = 4$ and $N_{\text{tot}} = 8$, 10.

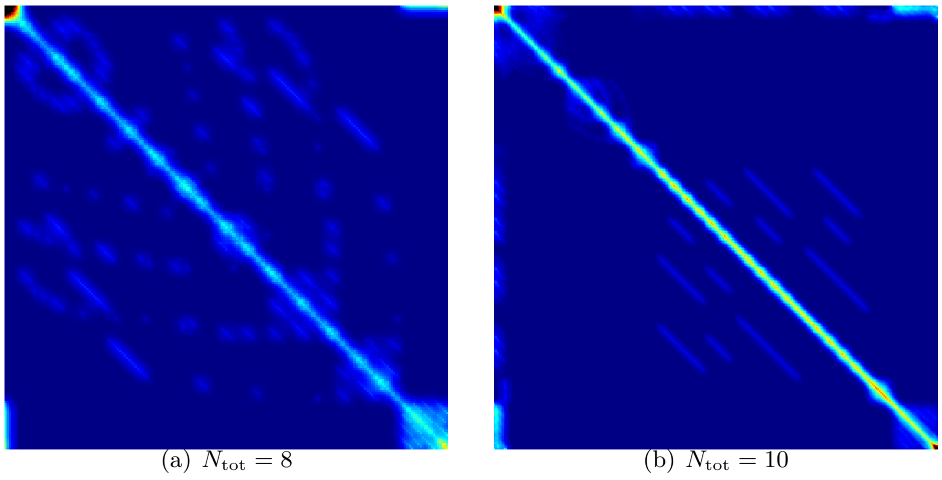

(a) $N_{\text{tot}} = 8$ (b) $N_{\text{tot}} = 10$

Figure 7.3: Density of non-zero of matrix elements of one-body operators, corresponding to $A = 8$ and $N_{\text{tot}} = 8$, 10.

Note that the locations of non-zero elements in these matrix representations exhibit a self similar, almost fractal structure. This is dependent on the ordering of the many-body basis, and could be altered if the many-body basis is sorted differently. Such considerations could be important in some applications where load-balancing is of essence, for example in parallel CPU-environments, but is of less importance within the current context.

Additionally, we have calculated the average number of non-zero matrix elements associated with each many-body state, i.e. how many other states each many-body state couples to on average. This is shown in Figure 7.4 together with the average number of clock cycles needed to process each many-body state for the two different kernel-designs using control inputs. The corresponding numbers for the non-dynamic energy bound kernels are much larger and have hence been excluded to improve visuality. The figure indicates that both strategies use an average number of clock cycles that is lower than the average number of non-zero matrix elements. This is possible since each kernel can identify up to A non-zero matrix elements in parallel. Our conclusion is that we on average identify at least one non-zero matrix element per clock-cycle. This should not be too far from optimal considering that the number of clock-cycles needed to process one many-body state is set uniformly for all the parallel processes that perform single-particle state interchanges.

7.4 Hardware costs

The hardware costs of a kernel refers to the amount of hardware-components, i.e. lookup tables (LUTs), flip-flops (FFs), digital signal processing units (DSPs) and block RAMs (BRAMs), that are needed to implement the kernel. In Table 7.1 these costs have been summarized for the different kernel-designs. Note that none of the designs utilize more than 10 % of the FPGA because of their relative low amount of calculations. For the designs that use a non-dynamic or dynamic energy-bound algorithm the LUTs

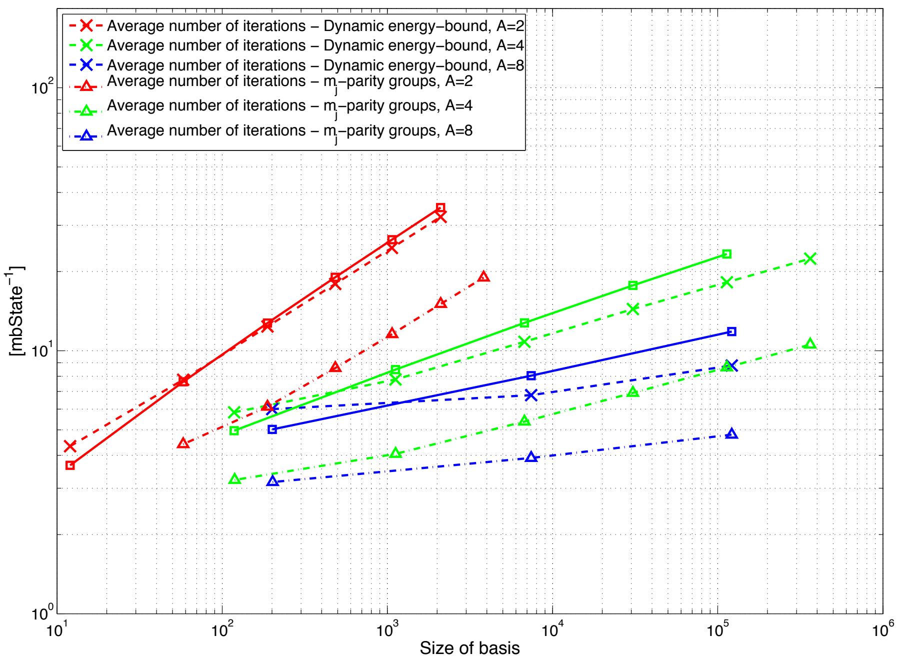


Figure 7.4: The solid lines represent the average number of non-zero matrix elements per many-body state, calculated with the implemented kernels, for $A = 2$, 4 and 8. These can be compared with the dashed lines with Xs and triangles, which represent the average number of clock-cycles needed to process the many-body states for kernels using a dynamic energy-bound and m_j-parity groups, respectively.

Table 7.1: Hardware resources needed to build one kernel of the specified type. It can be seen that the resource usage grows linearly or possibly slightly slower, as the particle number is increased. Also note that none of the hardware-resources are utilized to more than 10 %, which leaves a large part of the FPGA unused.

Algorithm type	LUT	[%]	FF	[%]	DSP	[%]	BRAM	[%]
MAX3 (FPGA) total resources	297600	100.	595200	100.	1064	100.	2016	100.
Non-dynamic energy-bound, A=2	9480	3.19	12112	2.03	23	2.16	48	2.38
Non-dynamic Energy-bound, A=4	12960	4.35	18910	3.18	23	2.16	96	4.76
Non-dynamic energy-bound, A=8	28161	9.46	41191	6.92	47	4.42	192	9.52
Dynamic energy-bound, A=2	9501	3.19	12757	2.14	28	2.63	48	2.38
Dynamic energy-bound, A=4	14148	4.75	19548	3.28	28	2.63	96	4.76
Dynamic energy-bound, A=8	28717	9.52	42249	7.10	52	4.89	192	9.52
m_j-parity groups, A=2	7419	2.49	9222	1.55	0	0	12	1.79
m_j-parity groups, A=4	10390	3.49	13852	2.33	0	0	21	1.97
m_j-parity groups, A=8	21528	7.23	31270	5.25	0	0	37	3.48

and BRAMs are the most limiting hardware-resources. The kernels that use m_j-parity groups generally use fewer components, especially DSPs and BRAMs. This is because these kernels perform less arithmetic operations on the FPGA.

Another important issue is the dependence between hardware-cost and particle number, A. We see that the use of hardware-resources grows linearly, or slightly slower, as A increases. The explanation for this is that when we increase the number of particles, we essentially add an equivalent number of identical computational units which perform single-particle state interchanges.

Based on such estimates, it could be expected that the current FPGA has sufficient hardware-resources to implement kernels with tens of particles or designs in which several kernels work in parallel effectively doubling the process speed. However, this does not mean that it actually would be possible to implement such system. For example the I/O-interface is actually a more limiting factor as we shall see from following sections.

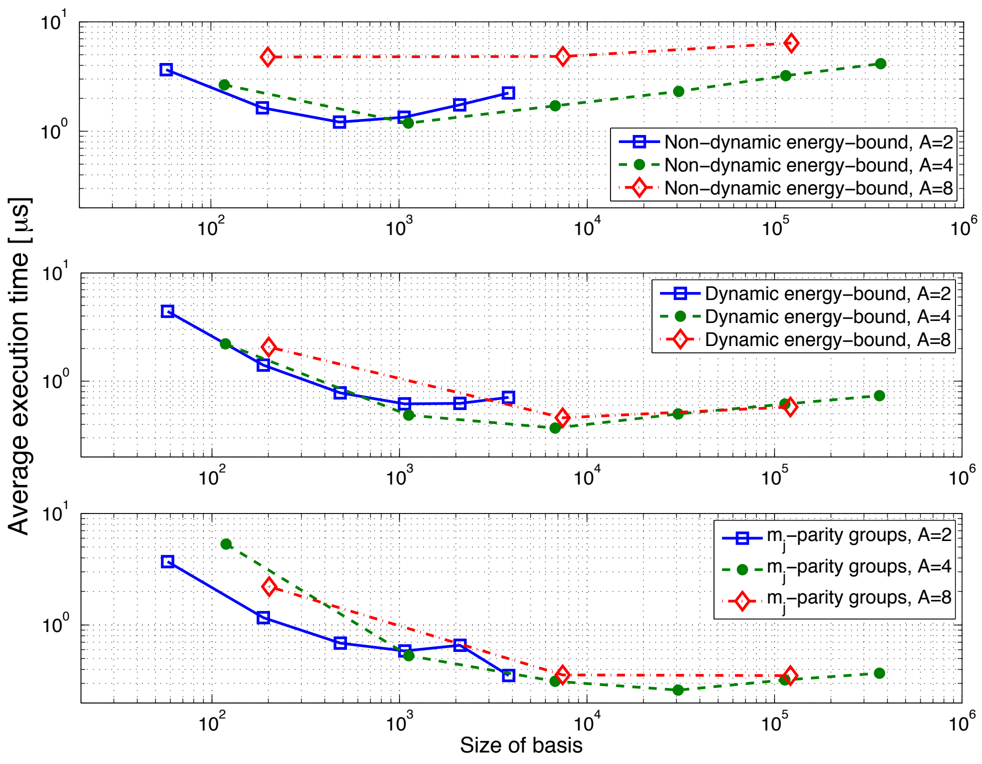


Figure 7.5: Average execution time per many-body state, for different kernel-designs and dimensions of the many-body basis. There are two competing factors that affect performance. The first is pipeline length, which is significant for small systems, while the second is that larger bases correspond to higher cutoff energies which results in that more single-particle state interchanges must be considered for each many-body state.

7.5 Performance of FPGA-implementations

To evaluate kernel-performance, we have measured the execution time to process the many-body basis for different cutoff energies. Since some of the bases are quite small we made measurements on multiple runs of the FPGA and could hence increase the stability of the measurement. As shown in Figure 7.5, we have then calculated the average execution time per many-body state as a function of the dimension of the many-body basis which increases with greater cutoff energies.

From these measurement results we see that the kernels can process the many-body states at a speed of 0.5-8 μs/many-body states, depending on what cutoff energy and kernel that is used. We see that the largest differences in performance for kernels with the same particle number A occur between kernels that use and do not use control inputs. The corresponding relative acceleration of the calculation when these kernels are compared is shown in Figure 7.6, and is up to a factor greater than one magnitude for large systems. The performance difference between the two designs, which both use

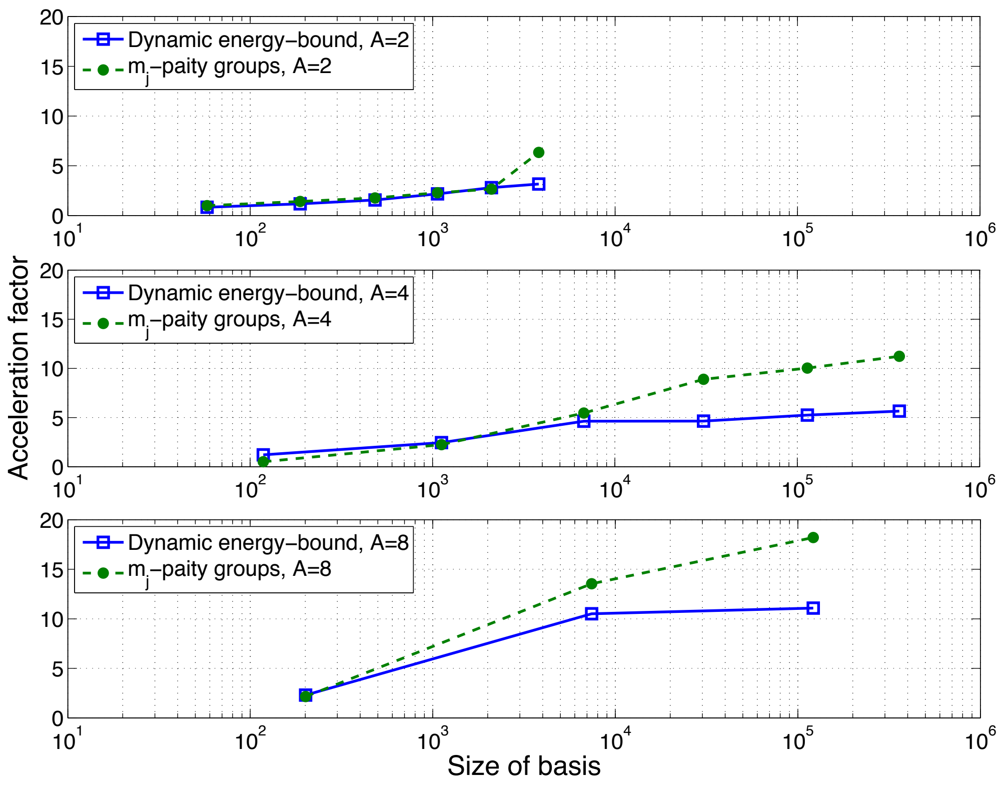


Figure 7.6: Acceleration factor for the kernels using control inputs, compared to those using a non-dynamic energy bound.

control inputs, is however less, typically a factor two in the limit of large systems.

As seen in Figure 7.5, the average time per many-body state decreases as a function of the size of the many-body basis, for the smallest bases. This decline is because of the pipeline depth of the kernels. In these cases the number of input states is less than the pipeline depth and thus we essentially measure the time it takes for the inputs to pass the pipeline. Therefore the total run time is about the same for all of these small bases but since we increase the number of states the average time decreases. The slight increase for larger bases originates in that an increased number of single-particle state interchanges must be considered, as the increased size of the basis also implies higher cutoff energy. In our design this is accounted for by making the kernel use the same input several times and thus decreases how often it accepts new inputs. Therefore, the time per many-body state increases.

From studying the effective frequency of the FPGA, as seen in Figure 7.7, we can identify some informative patterns. The effective frequency is calculated by dividing the number of clock cycles needed to perform the calculations, without counting the pipeline length, with the execution time. We see that in all cases the frequency starts out low whereafter it increases asymptotically to some value. This frequency is lower than the frequency at which the FPGA is expected to operate, 75 MHz, and only depends on

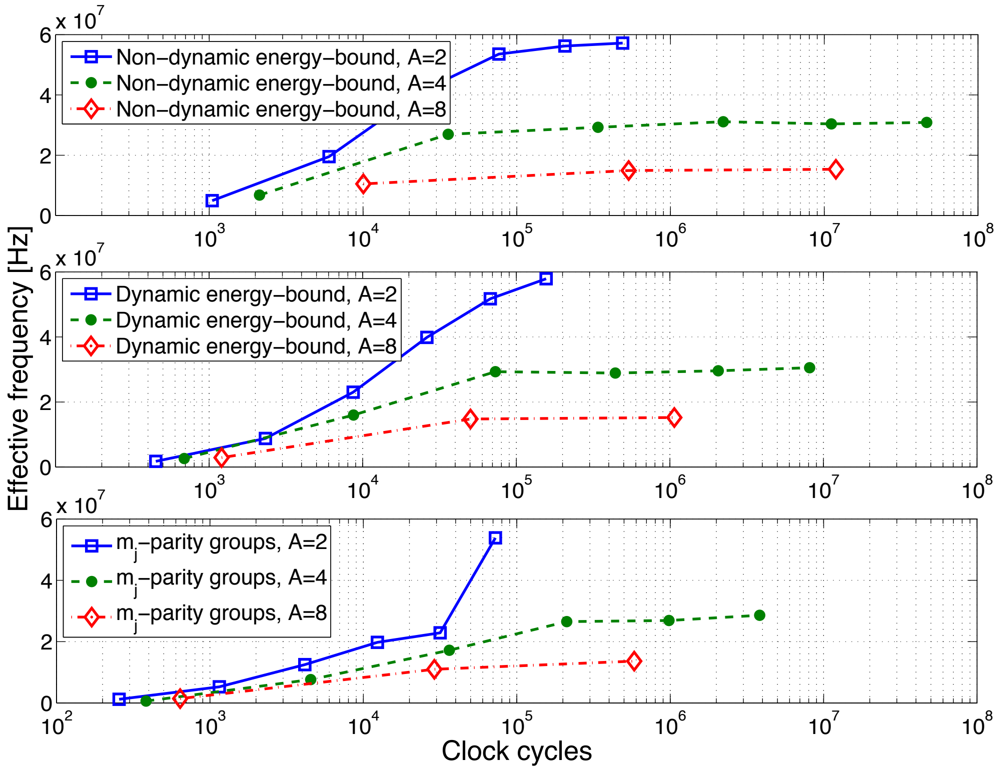


Figure 7.7: Effective frequency of the FPGA, i.e. the number of clock-cycles needed to process the many-body basis, without counting the pipeline length, divided by the execution time. Note that this frequency always is lower than 75 MHz, which is the frequency at which the FPGA is expected to run. The asymptotic value for the effective frequency for different kernels is seen to scale inversely to A. As the amount of output is proportional to A, this indicates that we have an I/O-bound.

the number of particles, A, but not on the kernel-design. We can also note that this assumptotic value decreases as $1/A$ at the same time as the output generated by the kernels increases linearly with A, which indicates an I/O-bound. If we account for the size of the output that each clock-cycle produces we can see that all final frequencies correspond to an output rate of 2 GB/s. This is the PCIe-bandwidth which has been measured by others using the same machine from Maxeler, for example [16] and [17], which additionally proves the assumption of an I/O-bound.

7.6 Performance of reduced matrix calculation

In order to evaluate the performance of the MATLAB-implementation, we have measured the execution times for a number of different systems and cutoff energies. The calculations and time measurements were performed using an *Intel i5* 3.3 GHz processor, with results shown in Figure 7.8. From the figure, we see that the average execution

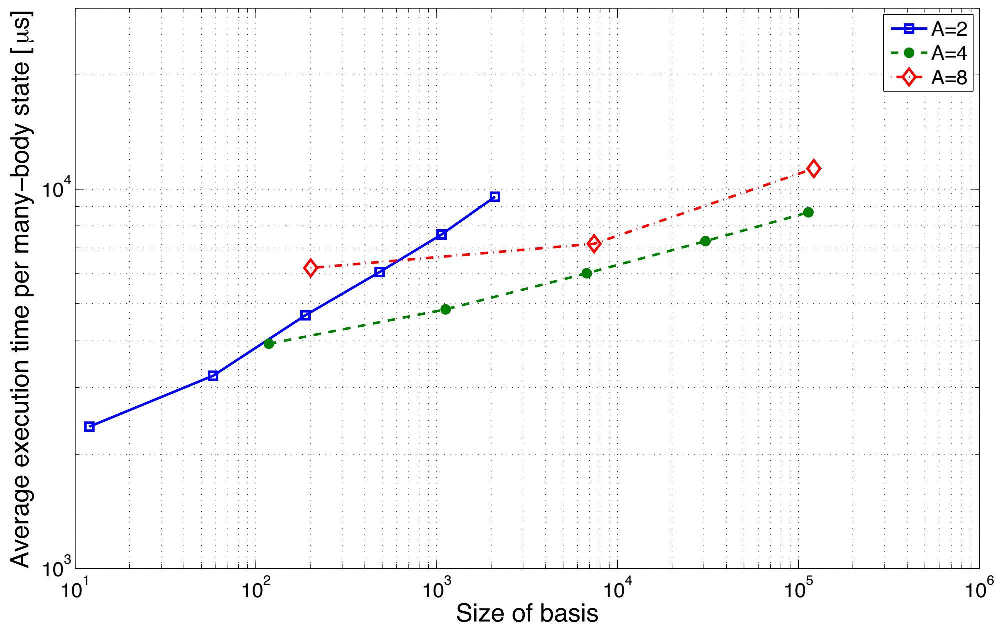


Figure 7.8: Execution time per many-body state of the MATLAB-implementation, for a number of different A and sizes of the many-body basis. This essentially shows how long time it took for MATLAB to process all non-zero elements originating from one many-body state.

time per many-body state grows with the dimension of the many-body basis, which is due to that the number of non-zero matrix elements per many-body state increases with the size of the basis. For the bases that were studied, we see that the average execution times correspond to that the connections for between $10^2 - 10^3$ many-body states being processed per second.

When comparing the performance for the FPGA-based calculation and the MATLAB-implementation, we see that the latter is several orders of magnitude less efficient and will hence be the bottleneck of the overall implementation. Note however, that this statement is both dependent on the hardware used for computation and the degree of optimization of the code. With larger off-FPGA computational resources and implementation in a more efficient language, such as `C++`, the calculation would have been much more efficient.

7.7 Performance compared to Trdens

To benchmark the FPGA-based calculations, we have measured execution times for the CPU-based Trdens-code, which calculates reduced transition densities without the use of FPGAs. These measurements have been performed using a *Dual Core AMD Opteron* 1.8 GHz processor, with results that can be found in Figure 7.9. If the results in this figure are compared to those of the FPGA-kernels and the MATLAB implementation, as done in Figure 7.10, we see that the FPGA-calculations are around two orders of

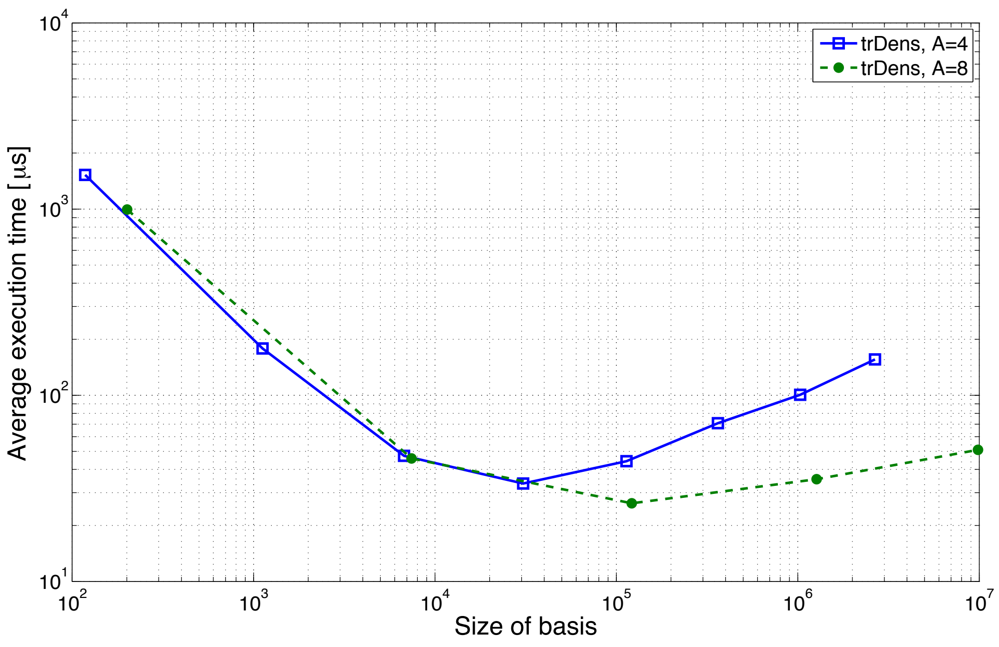


Figure 7.9: Execution time per many-body state of TRDENS, for systems with a number of different A and sizes of the many-body basis.

magnitude faster than TRDENS while the MATLAB implementation for computing the reduced transition densities, is slower by a similar factor. Hence, the off-FPGA calculation makes the overall performance less efficient than that of TRDENS.

Now it would be optimal to be able to compare the FPGA to the part of TRDENS that finds the connections. This would be needed to make clear conclusions about whether or not the FPGA is faster for finding connections. As of this project, this has not been possible because of the nature of the TRDENS code, but using the results of Figure 7.10 one could come to some conclusions. The figure indicates that the, unoptimized, reduction of the matrix in MATLAB grows as $\sim \mathcal{O}(n^{1/6})$, where n is the size of the basis, whereas the time for TRDENS grows as $\sim \mathcal{O}(n^{1/2})$ for large systems. With this comparison, one possible conclusion is that the time TRDENS uses, is not bounded by the reduction of of the matrix , and therefore, finding the non-zero elements of the matrix should be the dominant part. If so, there is much to gain from using the FPGA for these calculations. This implies that the main issue in making FPGAs a competitive choice for the calculation of transition densities is whether the conversion to a reduced matrix representation can be made more efficient.

It might be noted that if all calculations could be performed on the FPGA it might not significantly increase the execution time of the FPGA, as, possibly, it only would increase the pipeline length. The possibility of this, together with other optimization related aspects will be further discussed in Chapter 8.

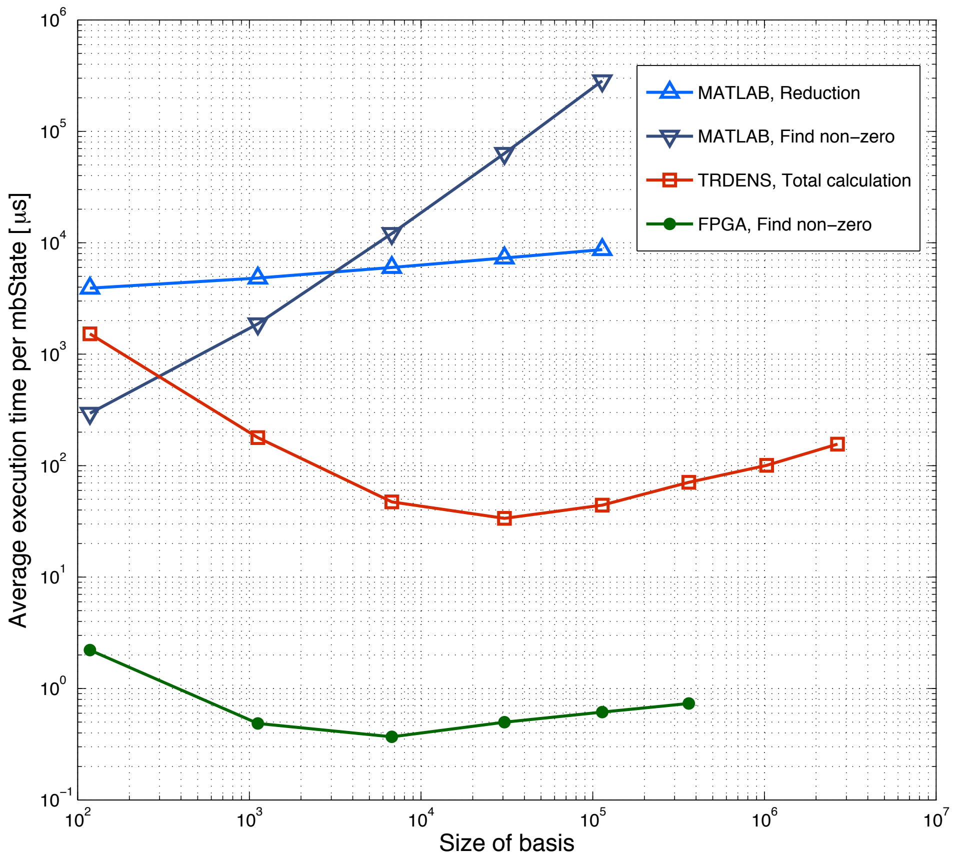


Figure 7.10: Execution time per many-body state for $A = 4$ using different programs. The MATLAB implementation is split into two parts where one finds the non-zero elements and the other uses the output of the first MATLAB script or the FPGA to calculate the transition densities.

Chapter 8

Discussion

In Chapter 7, we presented performance results from the use of FPGAs to calculate transition densities, according to methods described in Chapter 6. Here, we aim at identifying how these results imply whether FPGAs are feasible for the calculation. We also point out how the devised methods can be improved and extended to more general contexts.

8.1 Feasibility of using FPGAs for the calculation of transition densities

In the first part of this section, we will discuss the feasibility of implementing the calculation of transition densities on an FPGA. However, as seen from the results this will not in principle be a question about the efficiency of the on-FPGA calculations, but rather a discussion if the necessary off-FPGA calculations can be made more efficient. Following this, we analyze which of the currently devised algorithms that is the most effective. Then, in a final part, we address how the algorithms should be improved and give recommendations for the future design of FPGA-kernels. This will also include a suggestion of how to move a larger part of the computation to the FPGA and hence reduce the amount of off-FPGA calculation.

8.1.1 Conclusion regarding the feasibility

According to the obtained results, it was seen that the time needed for the FPGA-based calculation of non-zero elements was only a fraction of the time needed by TRDENS, when the latter was run on a single CPU-kernel. Of course, this is not a reflection of the overall efficiency of the FPGA-implementation, but indicates that the implementation can be competitive with TRDENS, when TRDENS is run with MPI[1], on several CPU-kernels, provided that the off-FPGA calculations do not become too time-consuming. Unfortunately this was exactly what happened in the MATLAB-implementation that converted FPGA-output to a reduced matrix representation. Hence, the conclusion

[1]Message Passing Interface; a protocol that makes it possible to perform computations in parallel, on many CPU-kernels.

about the feasibility of an FPGA-implementation becomes a conditional statement, dependent on whether this CPU-computation can be made more efficient or implemented on the FPGA. We begin with discussing the first of these options.

Since the feasibility of an FPGA-implementation was seen to depend on the possibility to improve the conversion of FPGA-output to a reduced matrix-representation, we should make an analysis of whether such improvements should be deemed possible. In this matter, it has been noted that a first step towards a more efficient implementation is to use a more well-suited programming language such as `C++`. However, the reason that the MATLAB-implementation is so slow, has a rather specific origin. It is related to the calculation of Clebsch-Gordan coefficients. In spite of that the total number of different Clebsch-Gordan coefficients is rather modest, they occur in many parts of the calculation and are re-calculated, whenever they are needed. Our observation is that this may be the bottleneck as, for example, the coefficient

$$(3/2,3/2,3/2,-1/2|1,1) \tag{8.1}$$

needs 0.379 ms to be calculated,[2] using the same algorithm as in the implementation. As the execution times of the MATLAB-implementation, were between 2 ms and 10 ms (on average per many-body state), it had not been possible to calculate more than between 5 and 26 Clebsch-Gordan coefficients for each many-body state. However, for each many-body state several Clebsch-Gordan coefficients must be calculated and it is therefore probable that a major part of the execution time was consumed by their calculation. If the Clebsch-Gordan coefficients instead had been pre-calculated and accessed from memory, the calculation would have been more than two orders of magnitude faster, as indicated by [19]. This would have made the overall execution time comparable to that of TRDENS, which would make FPGAs competitive.

Finally, we note that whether an FPGA-implementation should be recommended is not an unconditional question about whether it can perform better than TRDENS. The decisive factor is its cost-effectiveness. Specifically one has to consider the hardware and maintenance costs compared to CPU-clusters. Note that, what other uses one might have of the hardware, except calculating transition densities, also needs to be considered. However, this is a question beyond the scope of this study but that might be interesting to investigate.

8.1.2 Optimal choice of algorithm

The most important issue, when determining which algorithm is most efficient, is its performance in the limit of large many-body bases since real problems could have dimensions exceeding 10^9. Note, however, that the three strategies that have been developed within this project have only been tested for bases of small to moderate sizes, which means that we cannot make conclusions based on straightforward measurement results. Instead, we must interpret these results in such a way, that we can simply extrapolate the performance to calculations with larger bases.

For the implementations in this report, which create output each clock cycle, this is however rather straightforward since the FPGA approaches a stable effective frequency

[2]Averaged over 10^5 calculations.

due to the I/O-bound. This means that the execution time is proportional to the average number of clock cycles per many-body state, used to find connections. This quantity was analyzed in Chapter 6, see Figure 6.6 and 6.8. Based on this, we can generally expect growing benefits from using the strategies with control inputs as the dimension of the many-body basis, i.e. cutoff energy, increases. Also, the benefits of these strategies increase as the number of particles increase, as interchanges to single-particle states of high energies are generally uncommon in the many-body states.

In choosing between using the dynamic energy-bound and m_j-parity group strategies, the main trade-off is increased amount of input in favor for a performance increase by a factor of two, when choosing the latter of these strategies. Since input has never been an issue compared to the bounds due to output for any of our implementations, we conclude that the m_j-parity group strategy is the optimal choice[3] for performance in the limit of large systems.

Note that the statement in the previous paragraph, about which algorithm is optimal in performance, is highly dependent on that the kernel-load does not affect the effective frequency of the FPGA[4]. If the issue regarding output generation, mentioned in Section 7.1, is solved, so that the kernels only send output when a connection is found, different considerations must be made. Specifically, it is only meaningful to optimize how many non-zero elements are found each cycle, up to the point where the bandwidth of the output channel is saturated. If it is possible to saturate the bandwidth using a strategy that does not need pre-calculated control inputs, then such a strategy should of course be used, as it results in the same FPGA-performance without any pre-calculation costs.

Finally, our conclusion is that the optimal choice of algorithm in some sense depends on what FPGA that is used. For example, the current FPGA has large hardware-resources compared to the dimensions of its I/O-interface, and it would hence in principle be best to instantiate many parallel kernels with a non-dynamic energy-bound, instead of using pre-calculated control inputs[5]. On the other hand, if the hardware-resources of the FPGA are more limited compared to the I/O-interface, it is better to use a strategy with control inputs. This raises the question of what the most optimal ratio is between I/O capabilities and hardware-resources that leads to the most cost-effective solution. However, this is beyond the scope of this study.

8.1.3 Improvements of the implemented algorithms and kernels

This section singles out a number of areas where further work can be done on the FPGA-implementations, in order to improve their performance. Also, we point out that the CPU-based conversion of FPGA-output to a reduced transition density, could be incorporated into the on-FPGA calculations. We note that this could be a way to make the FPGA-implementation of the calculation competitive with Trdens.

[3]Also partly motivated by that the pre-calculation cost of control inputs is similar to that for the dynamic energy-bound strategy.

[4]Since the same amount of output is generated each clock cycle, resulting in a constant I/O-bound and hence no fluctuations in the effective frequency.

[5]This is not entirely true when using the Maxeler programming interface, since multiple kernels would demand a greater number of I/O-streams, a number that is not allowed to exceed eight.

Use of HDL

In a final implementation it would be desirable to have kernels that only generate output when a non-zero matrix element is found. As a part of this, one must incorporate more advanced programming than was used within this project. For example, one could create an output-buffer where output accumulates, with the ability to signal to the CPU to read data whenever the buffer becomes full. This could either be incorporated in the currently used high-level code, as an HDL block, or one could consider to make the full implementation in an HDL. The latter will be a more demanding solution, but allows more degrees of freedom. For example, it might also be possible to use the eight I/O-streams more flexibly, so that their limited number does not become an obstacle for the possibility of instantiating more kernels.

Use of on-chip ROMs

The possibilities to use ROMs to store values, for example parametrizations of single-particle states, has previously been discussed, but was never utilized in the implementations. However, if ROMs were used, they would increase the performance of the non-dynamic energy-bound strategy by a factor two. It would also be possible to use less control inputs for the dynamic energy-bound and m_j-parity group strategies. Specifically, in the latter case, we could use the ROMs to store the maximum states on the FPGA and hence avoid to transfer them with each many-body state. In conclusion, there are many benefits of ROMs, and we therefore encourage that a future kernel implementation does not leave them unused.

Suggestions to perform a larger amount of the calculations on the FPGA

A final area of further study is to investigate whether a larger part of the calculation of transition densities, can be implemented on the FPGA. For example, we chose to perform lookups to find probability amplitudes off the FPGA, due to the large burst-size of the DRAM. But, after implementing the kernels and making measurements, we found that the identification of non-zero matrix elements was not much faster than the DRAM. As a conclusion, we find it appropriate for a future implementation to perform lookups on the FPGA, with probability amplitudes stored in DRAM (which for the MAX3 card has the size 48 GB).

Beside the lookups, it might be considered that calculations related to the reduced matrix-representation also could be performed on the FPGA. For example, one idea is to use FPGA-based memory to accumulate the values of the reduced matrix elements. As the on-chip ROM is rather small, this accumulation must be performed using DRAM. Doing this is however not very simple and it must be resolved how the contributions to the reduced matrix elements should be updated in a way that is compatible with the pipelined architecture of the FPGA. Also there might be problems regarding the possibilities of parallelization and issues related to burst-size of the DRAM.

If possible, though, such an implementation has potential to be considerably faster than TRDENS since these last steps would not increase the calculation time on the FPGA considerably. It would add new operations at the end of the pipeline and thus increase the pipeline depth but not the number of clock cycles needed to begin processing all inputs. Therefore it would not significantly increase the calculation time for large systems; it might actually reduce the total time since the output would be significantly decreased. Thus FPGAs have the capacity to vastly accelerate the calculation of reduced transition densities if these last issues can be solved.

8.2 Algorithm generalizations

Up to this point, we have only considered systems with one kind of particle, such as neutron-droplets. However, a real, stable nucleus usually consists of approximately equal amounts of protons and neutrons. Also, besides one-body operators, two- and three-body operators are often of interest in the study of many physical properties. In this section, we aim at explaining how the FPGA-based implementations can be extended to include these possibilities.

8.2.1 Nuclei incorporating both protons and neutrons

A mathematical formalism to describe systems with both protons and neutrons, is that of isospin, where all particles are considered as states of a generic particle, the nucleon. The isospin appears as an additional quantum-number, $t = 1/2$, with two projections $m_t = \pm 1/2$. It is the value of this projection, which characterizes whether a nucleon is a proton or neutron.

For the algorithms that work directly with quantum-numbers, i.e. algorithm 1 and 2, the implementation of isospin is straightforward and can achieved done by adding a fifth quantum number that is transferred to the kernel. If the Chinese remainder theorem is used to order the single-particle states, the difference is that one must use five, instead of four primes. However, a choice that must be made, is whether the particles are allowed to change type. This would be the case, e.g. when considering weak processes such as beta-decay transitions. If this is the case, one can use an additional chained counter that extends the already existing control structure, to vary m_t. Otherwise, we can leave m_t unchanged, exactly the same way as we have done with m_j. Concerning the m_j-parity group strategy, the implementation of a fifth quantum-number affects the grouping of single-particle states. The kernels, on the other hand, do not need to be modified.

8.2.2 Generalization of algorithms 1 and 2 to two- and three-body operators

This section outlines how the FPGA-based strategies that work directly with the quantum numbers of single-particle states, can be generalized to find non-zero matrix elements of two-body operators. Additionally, some features of the generalization to three-body operators, will be mentioned.

The difference between one- and two-body operators is that the latter allows interchanges of two single-particle states, instead on just one. To find the corresponding non-zero matrix elements, we can use a similar approach to that for one-body operators, and sequentially, for each many-body state

$$\texttt{mbState} = [\alpha_1,...,\alpha_A], \tag{8.2}$$

find what other states it connects to. However, we need to consider interchanges of pairs of single-particle states $(\alpha_i; \alpha_r) = (n_i, l_i, j_i, m_{ji}; n_r, l_r, j_r, m_{jr})$. These can be performed in parallel, with individual hardware implementations that remove and interchange states at specific positions i and r in `mbstate`. All which must be done is to find an efficient way to parametrize what pairs of new single-particle states $(\alpha'_i; \alpha'_r) = (n'_i, l'_i, j'_i, m'_{ji}; n'_r, l'_r, j'_r, m'_{jr})$, that we should try to replace the old ones by. To accomplish this, we begin by noting that alongside the maximum total energy condition and Pauli exclusion principle, the interchange of single-particle state pairs must fulfill:

- $l'_i + l'_r \equiv l_i + l_r \mod 2$, according to the parity constraint.
- $m'_{ji} + m'_{jr} = m_{ji} + m_{jr}$, according to the projection of magnetic moment constraint.

Compared to the equivalent conditions for a one-body operator, we now have additional degrees of freedom, which implies that we need more variables to parametrize the interchanges, than in the case for one-body operators. A closer analysis, indicates that it is sufficient to use eight variables, denoted by $\tilde{n}_1$, $\tilde{l}_1$, $\tilde{j}_1$, $\tilde{n}_2$, $\tilde{l}_2$, $\tilde{j}_2$, $\tilde{P}$ and $\tilde{m}$. These are then used to set:

$$\begin{array}{l|l}
\text{New single particle state } \alpha'_i & \text{New single particle state } \alpha'_r \\
\hline
n'_i = \tilde{n}_1 & n'_r = \tilde{n}_2 \\
l'_i = 2\tilde{l}_1 + \tilde{P} & l'_r = 2\tilde{l}_2 + (l_i + l_r - \tilde{P}) \bmod 2 \\
j'_i = \tilde{j}_1 & j'_r = \tilde{j}_2 \\
m'_{ji} = \tilde{m} & m'_{jr} = m_{ji} + m_{jr} - \tilde{m}.
\end{array} \tag{8.3}$$

To perform all necessary interchanges of pairs of single-particle states, we should consider all appropriate combinations of the introduced variables. This means that we should consider combinations where $\tilde{j}_1$, $\tilde{j}_2$ and $\tilde{P}$ take the values 0 and 1 (note that for $\tilde{j}_1$, $\tilde{j}_2$, we let 1 represent $j = l + \frac{1}{2}$ while 0 represents $j = l - \frac{1}{2}$). This should also be combined with $\tilde{n}_1$, $\tilde{l}_1$, $\tilde{n}_2$ and $\tilde{l}_2$ ranging over the non-negative integer solutions of

$$\tilde{n}_1 + \tilde{n}_2 + \tilde{l}_1 + \tilde{l}_2 \leq \frac{1}{2}\left(N_{\text{tot}} - \sum_{k \neq i,r} 2n_k + l_k - (l_i + l_r) \bmod 2\right) = \tilde{N}, \tag{8.4}$$

together with $\tilde{m}$ taking allowed values for the projection of magnetic moment for a single-particle state.

Use of on-chip ROMs and strategies for optimization

One way to implement the above parametrization, is by using several chained counters, imitating the behavior of a for-loop. Our experience however tells us that such implementation would be inefficient, since the chaining of counters is not very flexible. On the other hand, none of these drawbacks would be present if storing the parametrization in on-chip ROM and using a counter to generate addresses, which therefore is recommended.

Another point, worth exploring, is how control inputs can be used to increase the kernel performance. The same way as for one-body operators, we could dynamically set $\tilde{N}$, dependent on what many-body state that is being processed. To evaluate the benefits of this, we let $S_{\tilde{N}}$ denote the number of non-negative integer solutions to equation (8.4). It can then be shown that

$$\Delta S_{\tilde{N}} = \binom{\tilde{N}+3}{3} = \frac{1}{6}(\tilde{N}+3)(\tilde{N}+2)(\tilde{N}+1), \tag{8.5}$$

from which it follows that $S_{\tilde{N}} \sim \tilde{N}^4$. Dynamically setting $\tilde{N}$ is especially important for systems with many particles, for which the removal of two single-particle states has a negligible effect on the total energy of the many-body states.

Comment regarding kernel-design

As previously noted in this section, when dealing with two-body operators we suggest that each pair of indices i and r, corresponding to different positions in `mbState`, should be assigned with a hardware implementation and output stream of its own. This is similar to what was done for one-body operators, with the motivation that calculations related to connections often have dependencies on i and r, which otherwise can be difficult to incorporate in the hard-coded design of the kernel. However, the number of needed output streams grows fast, given by

$$\binom{A}{2} = \frac{1}{2}A(A-1), \tag{8.6}$$

where A is the number of particles. This means that the maximum of eight I/O-streams, allowed by the Maxeler programming interface, will be even more restricting than for one-body operators since A often is of order 10 or higher. As a conclusion, implementations for two-body operators are exclusively recommended to be performed in an HDL.

Three-body operators

The possibility to make the further generalization to three-body operators essentially involves further degrees of freedom in the conditions for parity and projection of angular momentum. The same way as for two-body operators this can be incorporated by using a suitable number of variables to parametrize interchanges of (in this case triplets of) single-particle states. It turns out that the benefits of dynamically setting $\tilde{N}$, will be even larger than for two-body operators. The maximum number of allowed I/O-streams, will however also be even more limiting than in the former case, meaning that implementation in an HDL is absolutely necessary.

8.2.3 Generalizing the m_j-parity group strategy

The possibility to generalize the m_j-parity group strategy to two- and three-body operators has been investigated. In the case of a two-body operator, this generalization would be performed by forming a two-body basis, in which the elements are grouped according to parity and angular momentum projection. For a three-body operator, the only difference would be that we use a three-body basis instead of a two-body basis.

Similarly to the case with a one-body operator, the main benefit of the m_j-parity group strategy is that it can be used to with a very high probability find a connection in each clock cycle. It is possible to perform interchanges of groups of single-particle states so that the only condition that can be violated is the Pauli exclusion principle. However, in spite of this it turns out that the generalization to two- and three-body operators is less suited for FPGA-implementation.

The reason for this, is that it demands too large amounts of on-chip ROM. This occurs as a many-body state is specified in terms of its occupied single-particle states. It is this representation that is used to create memory addresses for lookups. However, if an interchange of two single-particle states is performed using an index for a pair of single-particle states, this index must at some point be converted into information about what individual single-particle states that have been created. This needs to be stored as a conversion table in on-chip ROM. To see that this is not possible, we consider a kernel with $A = 8$ and maximum allowed single-particle energy $N_{\text{single,max}} = 10$. In this case, there are `#spState`=572 states in the single-particle basis, which results in roughly

$$\binom{\texttt{\#spState}}{2} = 163306 \tag{8.7}$$

two-body states. Using the same kernel-design as previously described, there are

$$\frac{1}{2}A(A-1) = 28 \tag{8.8}$$

parallel process that need access to the conversion table. As the ROMs only have two ports this means that we need 14 copies of the conversion table. Also, each index in the two-body basis corresponds to two indices of single-particle states, which we assume are single-precision numbers. Based on this, we identify a need of

$$2 \cdot 14 \cdot 163306 \cdot 4\,\text{B} = 18.29\,\text{MB} \tag{8.9}$$

ROM. This is more than five times as much ROM as is available using the FPGA in this project [20]. Also, the need of memory scales badly with A, which leads to the conclusion that generalizations of the m_j-parity group strategy should not be implemented on an FPGA. However, it is believed that the strategy will be efficient in a CPU-environment. Specifically, the method will benefit from the better control structures of a CPU, compared to those of an FPGA.

Chapter 9

Conclusions and recommendations

Here we present a condensed list of conclusions regarding the feasibility of FPGA-based computations of transition densities, as well as recommendations about how such computations should be performed:

- In this study, transition densities were calculated using a subdivision of tasks with an FPGA-part that identifies valid connections, and a CPU-part that converts the connections into reduced transition densities. Using MATLAB to implement the CPU-based part, the overall performance was found inferior to the CPU-based code TRDENS, which performs the full calculation.
- A closer look at the implementation shows that the FPGA-based part of the computations only consumes a small fraction of the total time used by TRDENS, when ran on a single CPU-kernel. This indicates that the FPGA-implementation would become feasible if the off-FPGA calculations could be optimized.
- To optimize the off-FPGA calculations, we recommend that the computation should be performed in `C++`. Focus should be put on the calculation of Clebsch-Gordan coefficients and the possibility to pre-calculate them.
- The full calculation of transition densities could also in principle be implemented on the FPGA. The execution time of the FPGA would not be affected by this, as it would correspond to additional stages being added to the pipeline. Hence, we recommended further investigation of this possibility as it would make FPGA-based computations competitive. There are however issues regarding how computational results should be accumulated in DRAM.
- I/O will be a bounding factor for FPGA-implementations using the same subdivision of calculations as in this project. The I/O-bound stems from that information about a large number of connections must be transferred to the CPU. This is an even larger problem for two- and three-body operators that have much larger amounts of connections.

- In this project we have used a kernel design with separate blocks of hardware for interchanges of single-particle states at different positions in the many-body states. This design was chosen because the calculations differ depending on the position of the removed single-particle state. It may also be a good design choice for future FPGA-implementations.

- The performance of an FPGA-implementation was seen to be boosted by using pre-calculated control inputs. The calculation cost of these was linear with respect to the size of the many-body basis. Using control inputs we have been able to identify more than one connection per clock-cycle. Of the designs implemented during this project, the algorithm denoted m_j-parity groups was the most efficient.

- For two- and three-body operators, we recommend generalizing the dynamic energy-bound strategy. We believe that such an implementation could have the same benefits as for one-body operators. Generalizations of the m_j-parity group strategy, were on the other hand found less applicable for FPGA-implementation, due to limited amount of ROM.

- We recommend that a future implementation should use HDL to a greater extent. This would allow implementing structures unavailable in Maxeler's programming interface. The benefits would include a more flexible I/O-interface.

- The project has identified some desirable properties to seek in an FPGA for future implementations. These are: smaller burst-size than 384 bytes, more on-chip ROM, and, if possible, support for more streams than the current FPGA.

References

[1] Sakurai, J.J., Napolitano, Jim J. (2010) *Modern Quantum Mechanics.* Edition 2. New Jersey: Pearson.

[2] Dickhoff, W. H. and Van Neck, D. (2005) *Many-Body Theory Exposed!* Singapore: World Scientific.

[3] Barrett, B. R., Navrátil, P. and Vary, J. P. (2013) *Ab initio* no core shell model. *Progress in Particle and Nuclear Physics,* vol. 69, pp. 131-181.

[4] Cockrell, C., Vary, J. P. and Maris, P. (2012) Lithium isotopes within the ab initio no-core full configuration approach. *Physical Review C,* vol. 86, p. 034325.

[5] Maris, P. et al. (2012) Large-scale ab initio configuration interaction calculations for light nuclei. *Journal of Physics: Conference Series,* vol. 40.

[6] Maris, P., Vary J. P. and Shirokov A. M. (2009) Ab initio no-core full configuration calculations of light nuclei. *Physical Review C,* vol. 79, p. 014308.

[7] Suhonen, J. (2007) *From Nucleons to Nucleus.* Berlin, Heidelberg: Springer-Verlag.

[8] Deutsch, I. (1996) *Irreducible Tensor Operators and the Wigner-Eckart Theorem.* The University of New Mexico. `http://info.phys.unm.edu/~ideutsch/classes/phys522s03/lecturenotes/tensoroperators.pdf` (April 7, 2013).

[9] Brown, B.A. (2005) *Lecture Notes in Nuclear Structure Physics.* Michigan State University. `http://www.nscl.msu.edu/~brown/Jina-workshop/BAB-lecture-notes.pdf` (April 27, 2013).

[10] Sourdis, I., Gaydadjiev, G. N. (2011) HiPEAC: Upcoming Challenges. In *Reconfigurable Computing: From FPGAs to Hardware/Software Codesign*, red. M. P. João, M. Hubner, pp. 35-52. New York: Springer.

[11] Maxeler Technologies Inc (2012) Dataflow Programming with MaxCompiler, Maxeler Technologies Inc, Palo Alto.

[12] Maxeler Technologies Inc (2012) MaxCompiler – Manager Compiler Tutorial, Maxeler Technologies Inc, Palo Alto.

[13] Sternberg, P. et al. (2008) Accelerating Configuration Interaction Calculations for Nuclear Structure. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*; November 15-21, 2008, Austin, Texas, USA.

[14] Hardy, G. H., Wright, E.M. (2008), *An Introduction to the Theory of Numbers.* Oxford: Oxford University Press.

[15] Marsaglia, July 2002, Xorshift RNGs, Journal Statistical Software, Vol. 7, Issue 3

[16] Chow, G.C.T. (2012) A mixed precision Monte Carlo methodology for reconfigurable accelerator systems. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*; February 22-24, 2013, Monterey, California. p. 7.

[17] Tkachov, T. (2012) *Accelerating Unstructured Mesh Computations using Custom Streaming Architectures.* London: Imperial College. (Exam thesis under the Department of Computing.) p. 23.

[18] Navrátil, P. (2004) Translationally invariant density. *Physical Review C*, vol. 70, p. 014317.

[19] Rasch, J. and Yu, A. C. H. (2003) Efficient storage scheme for precalculated Wigner 3J, 6J and Gaunt coefficients. *SIAM Journal on Scientific Computing*, vol. 25, issue 4, pp. 1416-1428.

[20] Stojanovic, S. (2011) *An Overview of Selected Hybrid and Reconfigurable Architectures* University of Belgrade. `http://home.etf.rs/~vm/os/vlsi/predavanja/Survey-ADV_v11.pdf`

[21] Mizusaki, T. (2005) *Shell model calculation.* University of Tokyo. `http://www.cns.s.u-tokyo.ac.jp/summerschool/ciss05/lecturenotes/Mizusaki.pdf` (May 9, 2013)

[22] Goodrich, M. T., Tamassia, R. (2011) *Data Structures and Algorithms in Java*, John Wiley & Sons.

Appendix A

M-scheme and J-scheme

We have seen how a many-body basis can be constructed to obey the Pauli exclusion principle by the use of second quantization. However, there are some other properties that we may want to consider as well. In finding eigenvectors of the Hamiltonian we want to employ the smallest basis possible. The fact that the Hamiltonian is spherically symmetric allows us to diagonalize the matrix representation in $\hat{J}_z$ and $\hat{J}^2$, see Chapter 2.

In the choice of basis there are two conventionally used techniques, M-scheme and J-scheme. In the M-scheme we choose a basis whose states are eigenfunctions of $\hat{J}_z$. This is the scheme used in this project. In the J-scheme, the basis states are eigenfunctions of both $\hat{J}^2$ and $\hat{J}_z$. The J-scheme basis is smaller but harder to construct than the M-scheme basis. As an M-scheme basis we can use the basis discussed in Chapter 2 and 3, as we will show in the next section.

A.1 Angular momentum operators

In this section we will define the total angular momentum operator for our many-body states and show that the basis discussed in this report has M, but not J as a good quantum number, i.e. constitutes a M-scheme basis.

We will use the notation $\alpha = \xi jm$ to label our single-particle states. We let ξ absorb all other quantum numbers besides jm. We will use α when a more dense notation is needed. For the total angular momentum operator for a single particle we have

$$\vec{j}^2|\xi jm\rangle = j(j+1)|\xi jm\rangle \tag{A.1}$$

$$j_z|\xi jm\rangle = m|\xi jm\rangle. \tag{A.2}$$

$\vec{j}^2$ can be written in terms of the lowering and raising operators given by $j_\pm = j_x \pm ij_y$ as

$$\vec{j}^2 = j_- j_+ + j_z^2 + j_z. \tag{A.3}$$

The total angular momentum operator is given by

$$\vec{J} = \sum_{k=1}^{A} \vec{j}_k. \tag{A.4}$$

We are interested in constructing eigenstates of the two operators $\hat{J}^2$ and $\hat{J}_z$. In second quantisation, $\hat{J}_z$ can be written as a one-body operator

$$\begin{aligned} \hat{J}_z &= \sum_{\alpha_1\alpha_2} \langle \xi_1 j_1 m_1 | J_z | \xi_2 j_2 m_2 \rangle a^\dagger_{\alpha_1} a_{\alpha_2} = \sum_{\alpha_1\alpha_2} m_2 \langle \xi_1 j_1 m_1 | \xi_2 j_2 m_2 \rangle a^\dagger_{\alpha_1} a_{\alpha_2} = \\ &= \sum_{\xi jm} m a^\dagger_{\xi jm} a_{\xi jm}. \end{aligned} \tag{A.5}$$

However, we see from (A.4) that $\hat{J}^2$ will mix indices, hence it is a two-body operator. $\hat{J}^2$ is simplest written as [9]

$$\hat{J}^2 = \hat{J}_- \hat{J}_+ + \hat{J}_z^2 + \hat{J}_z \tag{A.6}$$

where

$$\hat{J}_\pm = \sum_{\xi jm} \sqrt{j(j+1) - m(m \pm 1)} a^\dagger_{\xi jm\pm 1} a_{\xi jm}. \tag{A.7}$$

We let $\hat{J}_z$ operate on an anti-symmetric state $|\alpha_1\alpha_2...\alpha_A\rangle$,

$$\hat{J}_z |\alpha_1\alpha_2...\alpha_A\rangle = \sum_{k=1}^{A} m_k a^\dagger_{\alpha_k} a_{\alpha_k} |\alpha_1\alpha_2...\alpha_A\rangle = \sum_{k=1}^{A} m_k |\alpha_1\alpha_2...\alpha_A\rangle = M |\alpha_1\alpha_2...\alpha_A\rangle. \tag{A.8}$$

Here we used (A.5) and the fact that the only non-zero contributions to the sum comes from summing over the single-particle states in $|\alpha_1\alpha_2...\alpha_A\rangle$. Further, since we annihilate the same state as we create, we will not get any change of sign, see Chapter 2. We see that $|\alpha_1\alpha_2...\alpha_A\rangle$ are eigenstates of $\hat{J}_z$ with $M = \sum_{k=1}^{A} m_k$ as eigenvalues.

However, $|\alpha_1\alpha_2...\alpha_A\rangle$ will not be an eigenstate of $\hat{J}^2$. We know that $|\alpha_1\alpha_2...\alpha_A\rangle$ is an eigenstate of $\hat{J}_z$, hence also of $\hat{J}_z^2 + \hat{J}_z$. Since

$$\hat{J}_+ |\alpha_1\alpha_2...\alpha_A\rangle = \sum_{k=1}^{A} \sqrt{j_1(j_1+1) - m_1(m_1+1)} a^\dagger_{\xi_1 j_1 m_1+1} a_{\xi_1 j_1 m_1} |\alpha_1\alpha_2...\alpha_A\rangle \tag{A.9}$$

we understand that $|\alpha_1\alpha_2...\alpha_A\rangle$ will not be an eigenstate of $\hat{J}_-\hat{J}_+$ in general. We see the from the fact that after applying $\hat{J}_-$, the indices of raised and lowered particles will be mixed. Recalling (A.6), we conclude that $|\alpha_1\alpha_2...\alpha_A\rangle$ cannot be an eigenstate of $\hat{J}^2$ either.

Our constructed basis will, as we have seen, have M as a good quantum number but not J in general. We denote these elements as $|\phi M\rangle$ and the final eigenstate of our Hamiltonian as $|\lambda JM\rangle$. λ is the enumeration of states with same J, since eigenstates of the Hamiltonian in general will be degenerate in J. We know from Chapter 2 that $|\phi M\rangle$ will be a sufficient basis since $\hat{H}$ will not couple states with different M. We can however construct eigenstates of $\hat{J}^2$ from $|\phi M\rangle$ as we will see in the following section.

A.2 Projection operator

Even though $|\phi M\rangle$ are no eigenstates of $\hat{J}^2$, we know that the Hamiltonian commutes with $\hat{J}^2$. This means that eigenstates of $\hat{H}$ will simultaneously be eigenstates of $\hat{J}^2$.

Such eigenstates are constructed as linear combination of $|\phi M\rangle$,

$$|\lambda_i J_i M_i\rangle = \sum_{j}^{D_M(M_i)} c_j |\phi_j M_i\rangle, \tag{A.10}$$

where $D_M(M_i)$ is the number of states $|\phi M\rangle$ with $M = M_i$. Every $|\phi M\rangle$ will be a part of the linear combinations constructing $|\lambda JM\rangle$ with $J \geq M$. The system (A.10) can be solved for $|\phi M\rangle$ and we get

$$|\phi M\rangle = \sum_{J\geq M}^{J_{\max}} \sum_{\lambda=1}^{D_J(J)} d_{\lambda,J} |\lambda JM\rangle \tag{A.11}$$

where $D_J(J)$ is the number of $|\lambda JM\rangle$ with equal J. This is the set which λ runs over. It is possible to construct eigenstates of $\hat{J}^2$ with the use of a projection operator on $|\lambda JM\rangle$. It can be shown [9] that the projection operator

$$P_J = \prod_{J_i=M, J_i\neq J}^{J_{\max}} \frac{\hat{J}^2 - J_i(J_i+1)}{J(J+1) - J_i(J_i+1)} \tag{A.12}$$

will only leave the angular momentum J in the original state

$$P_J|\phi M\rangle = \sum_{\lambda=1}^{D_J(J)} d_{\lambda,J}|\lambda JM\rangle = \sum_{k=1}^{D_M(M)} e_k|\phi_k M\rangle. \tag{A.13}$$

This system can be solved for e_k and we can explicitly form linear combinations of $|\phi_k M\rangle$ that will be eigenstates of $\hat{J}^2$. This can be used to further diagonalize our matrix[1].

A.3 M-scheme

When we construct our basis with a specific choice of M, we have to take into consideration what J these states can be projections of i.e. what eigenstates of $\hat{J}^2$ we can represent. With the choice of M we can construct eigenstates with good quantum number $J \geq M$. $M = 0$ can originate from all J. If we instead are interested in finding just one particular state with quantum number J, it would be wise to choose $M = J$, since this would minimize the basis. It is instructive to walk through a number of examples on how to create an M-scheme basis. These examples will show how properties of the many-body states can be deduced for a number of single-particle configurations.

Our single-particle states will consist of quantum numbers (n,l,j,m). We are only interested in the quantum numbers jm. We will refer to k numbers of particles in a specific orbital with e.g. $(j = 5/2)^k$, leaving out nl, with the only possibility that the k particles differ in m, because of the Pauli exclusion principle. If we instead write $(j = 5/2)(j = 5/2)$, these two parenthesis refer to a different set of nl, in this case both particles can have the same m. The crucial point is whether or not the Pauli exclusive principle has to be taken into account. Hence the notation $(j = 5/2)(j = 3/2)$ could refer to the same set of nl as well as different ones.

[1]These state might not be normalized. Further, if $D_M(M) > D_J(J)$ this will be an over-complete system of equation. There are however ways to deal with this, see [9].

Example 1. Now we consider an example of how to construct an M-scheme basis. Let us look at the case $(j = 5/2)(j = 3/2)(j = 1/2)$ which is shown in Table A.1. The table shows the occupied state of every single-particle state and to what quantum number J it contributes. An equivalent way of seeing this is to form all different J a specific M could originate from. Here all particles are in different orbitals and we get values of J ranging from the maximum $J_{\max} = 5/2 + 3/2 + 1/2 = 9/2$, to the minimal value $J_{\min} = 1/2$.

Since we are dealing with vectors we see that some Js arise more than once. The first $M = 9/2$ can only originate from $J = 9/2$,which is when all vectors point the same way. From $J = 9/2$ comes $2M = 9,7,5,3,1,-1,-3,-5,-7,-9$. However, we get three possible arrangements for $M = 7/2$. One is accounted for by $J = 9/2$, and the other two come from two different ways two couple $J = 7/2$. The same goes for all other J with the exceptions $J = 9/2$ and $J = 1/2$, which can only be constructed in one way. ■

If we denote the M-scheme dimensions with $D_M(M)$ (e.g. $D_M(7) = 3$ in Table A.1). We can get the dimension of J, $D_J(J)$, which is the total number of states with J as

$$D_J(J) = D_M(J) - D_M(J + 1). \tag{A.14}$$

The quantum number λ, introduced before, ranges over this set of same J. Our basis will, after diagonalised in $\hat{J}^2$, be degenerate in $\hat{J}^2$.

Example 2. A more interesting physical case is when all particles occupy the same orbital. The configuration $(j = 5/2)^2$ is shown in Table A.2. We see that no J occur more than once, because there is only one way to form each vector. More interesting is that neither $J = 1$ nor $J = 3$ occurs. This is entirely due to the Pauli exclusion principle. We start in the highest state $J_{\max}$, with all particles occupying the highest available sub-orbits. There is only one way to decrease M one step and this is done by lowering the lowest occupied sub-orbit, otherwise we would violate the Pauli exclusion principle. The dimensions of $J_{\max} - 1$ is given by $D_J(J_{\max} - 1) = D_M(J_{\max} - 1) - D_M(J_{\max}) = 0$. Hence $J_{\max} - 1$ is not possible. The case is similar for $J = 1$. The fact that $J_{\max} - 1$ is not possible to construct holds true for anyl number of particles occupying the same orbital. ■

Example 3. In a concluding example we look at $(j = 5/2)^2(j = 1/2)$ which is shown in Table A.3. Since we now couple two different orbitals we can deduce the table by coupling the first system $(j = 5/2)^2$, which we know has $J_1 = 0,2,4$, with $J_2 = 1/2$ from a second system. We use the relation $|J_1 - J_2| \leq J \leq J_1 + J_2$ which for $J_1 = 0$ gives $J = 1/2$, $J_1 = 2$ gives $J = 3/2, 5/2$ and $J_1 = 4$ gives $J = 7/2, 9/2$. This is what we see in Table A.3. Here we do not have to consider the Pauli exclusion principle since we coupled two different orbits. ■

Table A.1: Here we show the case $(j = 5/2)(j = 3/2)(j = 1/2)$. The list is arrange in decreasing M and we have left out $M < 0$ since the list is completely symmetrical. The rightmost column shows the possible J for each configuration of M.

2j	5	5	5	5	5	5	3	3	3	3	1	1		
2m	5	3	1	-1	-3	- 5	3	1	-1	-3	1	-1	2M	2J
	x						x				x		9	9
	x						x					x	7	9,7,7
	x							x			x		7	
		x					x				x		7	
	x							x				x	5	9,7,7,5,5
	x								x		x		5	
		x					x					x	5	
		x						x			x		5	
			x				x				x		5	
	x								x			x	3	9,7,7,5,5,3,3
	x									x	x		3	
		x						x				x	3	
		x							x		x		3	
			x				x					x	3	
			x					x			x		3	
				x			x				x		3	
	x									x		x	1	9,7,7,5,5,3,3,1
		x							x			x	1	
		x								x	x		1	
			x					x				x	1	
			x						x		x		1	
				x			x					x	1	
				x				x			x		1	
					x		x				x		1	

Table A.2: Possible configuration of single-particle states for $(j = 5/2)^2$. Only one x can occur for each m, because of the Pauli exclusion principle. The list is arrange in decreasing M and we have left out $M < 0$ since the list is completely symmetrical for negative values. The rightmost column shows the possible J for each configuration of M.

2j	5	5	5	5	5	5		
2m	5	3	1	-1	-3	- 5	2M	2J
	x	x					8	8
	x		x				6	
		x	x				4	8,4
	x			x			4	
		x		x			2	
	x				x		2	
	x					x	0	8,4,0
		x			x		0	
			x	x			0	

Table A.3: Possible configuration of single-particle states for $(j = 5/2)^2(j = 1/2)$. The table shows the occupied state of every single-particle denoted by an x. The list is arrange in decreasing M and we have left out $M < 0$ since the list is completely symmetrical. The rightmost column shows the possible J for each configuration of M.

2j	5	5	5	5	5	5	1	1		
2m	5	3	1	-1	-3	- 5	1	-1	2M	2J
	x	x					x		9	9
	x	x						x	7	9, 7
	x		x				x		7	
	x		x					x	5	9,7,5
		x	x				x		5	
	x			x			x		5	
	x			x				x	3	9,7,5,3
		x		x			x		3	
	x				x		x		3	
		x		x			x		3	
		x		x				x	1	9,7,5,3,1
		x			x		x		1	
			x	x			x		1	
	x					x	x		1	
	x				x			x	1	

A.4 J-scheme

Here we will give a brief description on how to form a J-scheme basis. As stated before, this basis will be smaller, but we will encounter some computational difficulties. This usually makes the M-scheme basis more preferable and is the reason why this project has used the M-scheme instead of J-scheme.

In order to construct a J-scheme we have to turn to Glebsch-Gordan coefficients to couple the spin of different states. In the case with two particles with quantum numbers $\alpha_i = \xi_i j_i m_i$, the procedure is known

$$|\xi_1 j_1 \xi_2 j_2; JM) = \sum_{m_1 m_2} (j_1 m_1 j_2 m_2|JM)\, |\alpha_1 \alpha_2). \tag{A.15}$$

We are interested in the states that obey the Pauli exclusion principle i.e. we use the anti-symmetrizing operator $\mathcal{A}$. By use of some Clebsch-Gordan identities, see Chapter 4 we can deduce that [9]

$$|\xi_1 j_1 \xi_2 j_2; JM\rangle = N_{12} \sum_{m_1 m_2} (j_1 m_1 j_2 m_2|JM)\, |\alpha_1 \alpha_2\rangle \quad \text{where} \quad N_{12} = \frac{1}{\sqrt{1+\delta_{\xi_1 \xi_2}}}. \tag{A.16}$$

With the use of second quantisation we can write this as

$$|\xi_1 j_1 \xi_2 j_2; JM\rangle = N_{12} \sum_{m_1 m_2} (j_1 m_1 j_2 m_2|JM)\, a^\dagger_{\alpha_1} a^\dagger_{\alpha_2} |0\rangle. \tag{A.17}$$

We can interpret the expression in front of $|0\rangle$ as an operator that operates on the vacuum state producing a many-body state with good quantum number J and M. The operator is a linear combination of creation-operators. We can generalise this to higher numbers of particles. The procedure is tedious but straight-forward, see [9] for more details.

We can continue to couple (A.17) with a third state by the use of a new set of Clebsch-Gordan coefficients, and continue in this manner. If all states have the same ξ we write it as

$$|\xi^A \lambda JM\rangle = Z^+(\xi^A \lambda JM)|0\rangle \tag{A.18}$$

where Z^+ is the operator constructed by linear combination of creation operators. Here λ is the quantum number that enumerates the different states with same J. A degeneracy will arise since we can couple states in different order.

For states occupying different ξ we couple the Z^+ operators instead. As for example with two different ξ,

$$|A\lambda JM\rangle = \sum_{M_1 M_2} (J_1 M_1 J_2 M_2|JM)\, Z^+(\xi_1^{n_1}\lambda_1 J_1 M_1) Z^+(\xi_2^{n_2}\lambda_2 J_2 M_2)|0\rangle \quad n_1 + n_2 = A. \tag{A.19}$$

From this we see that the computations needed to be done increase rapidly for higher number of particles. The J-scheme basis will be inferior to M-scheme basis in terms of computer resources for large systems [21]. For smaller system J-scheme can be an alternative.

Appendix B

Hash tables

A hash table is a data structure which uses a hash function to map a set of keys to a set of values [22]. It is usually implemented as an associative array, where the hash function maps the keys to a particular index in this array. If each key maps to a unique index, the hash function is said to be perfect. Given a specific key, a value can be retrieved very quickly, i.e. lookups are very efficient. This is one of the key advantages of a hash table, although it is important to note that two different keys will normally map to the same index, i.e. a collision will happen. This appendix presents some of the concepts and terminology related to hash tables, which are necessary for the developments in Chapter 6.

B.1 Main features of hash tables

As pointed out in the introductory paragraph the main advantage of a hash table is that lookups can be performed very quickly, essentially in constant time, i.e. in $\mathcal{O}(1)$ average time complexity. Delete and insert operations can be performed equally quickly. Finally, hash tables are storage efficient since storage space scales proportionally to the number of keys, that is, its space complexity is $\mathcal{O}(n)$.

B.2 Collisions

Ideally, each key should map to an unique index. However this rarely occurs in practice. As was alluded to in the introductory paragraph, when two or more keys map to the same index, a collision occurs. Collisions are unwanted and reducing the number of collisions is an important part of implementing an efficient hash table. A large part of this consists of designing or choosing an appropriate hash function. Also one has to oversize the array by some appropriate factor compared to the number of keys. This way, the number of collisions can be made few, but they will still be present.

There are different ways to handle collisions for a hash table. One option is to use a chained hash table, where each index is a chained list. When a lookup is performed, it is necessary to search through this list and in order to be able to distinguish the values from each other, the values are stored with their corresponding keys. Note, however, that the performance of the hash table will decrease as the depth of the chained lists increase.

A second way to handle collisions is to use open addressing, where all key and value pairs are stored in a single array. For an insertion, an initial index is calculated and if the index is unoccupied, the pair is stored there. If a collision occurs, the pair is stored at the first unoccupied index, starting with the next index after the hashed one, and continuing according to a specific probe scheme. An example of such a probe scheme is linear probing, where the number of indices between each probe is fixed, usually 1. When retrieving a value, the process is similar. Starting from the hashed index, a search is performed until a value is found using the aforementioned probe scheme. If a particular value is not found, the value does not exist in the hash table.

Appendix C

Link to source-code

Table C.1 provides a description of code that has been used within this project, which also can be found on `http://fy.chalmers.se/subatom/kand/2013/FPGA`. The code is grouped according to its use, in packages.

Table C.1: Table containing a description of code, used within the project.

Package	**Description**
Many-body basis	Functions to generate the many-body basis, for particles of one kind and different values of A, N_{tot}, M as well as parity. There are two such functions, one using MATLAB and one using `C++`. The basis is expressed in terms of an enumeration of occupied single-particle states, denoted by indices in the enumeration.
Non-dynamic energy-bound	Kernel and CPU-code to FPGA-implement the algorithm that uses a non-dynamic energy-bound. When using this kernel, the many-body states should be sorted by use of the Chinese remainder theorem and the package contains code for that.
Dynamic energy-bound	Kernel and CPU-code to FPGA-implement the algorithm that uses a dynamic energy-bound. The package also includes code to generate control inputs.
m_j-parity groups	Kernel and CPU-code to FPGA-implement the algorithm that uses m_j-parity groups. The package also includes code to generate maximum states as well as control inputs.
Verification of results	MATLAB script to calculate the non-zero matrix elements of a one-body operator, using a brute-force approach.
MATLAB implementation of TRDENS	Transforms the non-reduced transition density, calculated by the FPGA-kernels, into a reduced matrix-representation.