

COMPUTATIONAL CHEMISTRY IN PARALLEL

by

Dylan Hennessey

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science.

Department of Chemistry
University of Alberta

© Dylan Hennessey, 2018

Abstract

put abstract here

Preface

put preface here

put dedication here

Acknowledgements

Put acknowledgements here

Contents

1	Introduction	1
1.1	Hartree-Fock SCF	1
1.1.1	Relativistic SCF	1
2	Basis Sets	3
2.1	Methods	3
2.1.1	Basis set size optimization	3
2.2	Discussion	4
3	CUDAProphet	7
3.1	GPU Architecture	7
3.1.1	Threads, Blocks, and Grids	7
3.1.2	Memory	8
3.1.3	Occupancy and Performance	12
3.1.4	Matrix Multiplication Example	12
3.2	Program flow	16
3.3	Alterations for CUDA	17
3.3.1	Two-electron Integrals	17
3.3.2	Density Matrix Formation	28
3.4	Discussion	37
3.4.1	Resource Usage	37
3.4.2	Two-Electron Integral performance	38
3.4.3	P and Q Matrix Formation Performance	39
3.4.4	Total Speedup	44
3.5	Input Description	45
	Bibliography	49

List of Tables

2.1	Basis sets optimized using rwtbs	4
3.1	Kernel Resource Usage	37
3.2	<i>eint2</i> Kernel Performance	38
3.3	<i>formpq_alg1</i> Kernel Profiling Results.	40
3.4	<i>formpq_alg2</i> Kernel Profiling Results.	40
3.5	Total Speedup of Alg1	44
3.6	Total Speedup of Alg2	44

List of Figures

3.1	Architecture of a GPU	11
3.2	Two-Electron Integral Layout: Naive Implementation	21
3.3	Two-Electron Integral Layout: 2D Blocks and Grid	22
3.4	Two-Electron Integral Layout: 1D Blocks and Grid	23
3.5	Two-electron Integral Matrix as a Matrix	25
3.6	Two-electron Integral Matrix as a Vector	26
3.7	Density Matrix as a Matrix	29
3.8	Density Matrix as a Vector	30
3.9	Profiling of Two-Electron Integral Calculations	39
3.10	Profiling of the <i>formpq</i> Kernels.	41
3.11	Integrals Needed for the First Element of \mathbf{P} : Matrix Representation	42
3.12	Integrals Needed for the First Element of \mathbf{P} : Vector Representation	43
3.13	Total Time need to complete a calculation.	45

Chapter 1

Introduction

1.1 Hartree-Fock SCF

1.1.1 Relativistic SCF

In the previous subsection we discussed the calculation of the two-electron integrals, now we will discuss how they are actually used.

SCF begins with the formation of the density matrices. In this program, each spinor symmetry λ has its own density matrix. There are also two types of density matrices, one for contributions from closed shell spinors (\mathbf{D}_C) and one for contributions from open shelled spinors (\mathbf{D}_O).

Using a procedure developed by Roothaan and Hall^{[citation needed](#)}, we first need to find the matrix \mathbf{C} which satisfies

$$\mathbf{F}\mathbf{C} = \epsilon\mathbf{S}\mathbf{C} \quad (1.1)$$

where \mathbf{F} is the Fock matrix, ϵ is the vector of eigenvalues of \mathbf{F} with respect to matrix \mathbf{S} , and \mathbf{S} is the overlap matrix. The non-relativistic \mathbf{F} for closed shells of symmetry λ is given by

$$\mathbf{F}_{p\lambda,q\lambda} = \langle p_\lambda | h(1) | q_\lambda \rangle + \sum_{\mu=1}^{nsym} \sum_{r=1}^K \sum_{s=1}^K \left(2 \sum_{a=1}^{occ} \mathbf{C}_{r\mu,a} \mathbf{C}_{s\mu,a}^* \right) \left[(p_\lambda q_\lambda | r_\mu s_\mu) - \frac{1}{2} (p_\lambda q_\lambda | s_\lambda r_\lambda) \right] \quad (1.2)$$

where p , q , r , and s are basis functions, K is the total number of basis functions for symmetry μ , and occ is the number of occupied closed shells in symmetry μ . Because p and q will always be of the same symmetry, and likewise with r and s , the λ and μ subscripts will be dropped from these indices and instead will be obtained from context. As we can see, \mathbf{F} is itself a function of \mathbf{C} and we left with a rather disappointing outcome. That is, in order for us to find \mathbf{C} , we first must know what it is. And so it would seem that all is lost and

all of this hard work was for nothing. And yet, there is a glimmer of hope. \mathbf{F} is made up of two terms, and as luck would have it, only the term corresponding to contributions from the two-electron integrals depends on \mathbf{C} , the term from one-electron integrals can be used independently of \mathbf{C} . Thus, we begin by assuming that \mathbf{F} depends solely on the one-electron term, effectively assuming that $\mathbf{C} = \mathbf{0}$. Solving Equation 1.1 with the H^{core} hamiltonian matrix only will give us the initial guess of \mathbf{C} and now we can begin our work in earnest.

Chapter 2

Basis Sets

2.1 Methods

2.1.1 Basis set size optimization

In keeping with the theme of “computers should work, people should think”, we decided to use a somewhat novel method of generating our basis sets. We begin by choosing an arbitrarily large basis set size, such as $s(1 : 40)p(1 : 40)d(1 : 40)f(1 : 40)$ (here $i(\# : \#)$ refers to the starting and ending index of ζ ’s used for symmetry i). We then find the optimal α , β , δ , and γ wtbs parameters for this basis set. We then use the following steps to find the optimal basis set size.

Step 1. Begin by finding the fewest number of f functions necessary. This can be done by generating .inps files that range in size from $s(1 : 40)p(1 : 40)d(1 : 40)f(1 : 1)$ to $s(1 : 40)p(1 : 40)d(1 : 40)f(1 : 40)$.

Step 2. Optimize the basis sets and select the smallest basis set that is still below some minimum accuracy threshold. We chose a relative error of no greater than 5.0×10^{-9} to numerical calculations where relative error is defined as $\frac{|(\text{optimized energy}) - (\text{numerical energy})|}{\text{numerical energy}}$. The size of this basis set is $s(1 : 40)p(1 : 40)d(1 : 40)f(1 : x_f)$.

Step 3. Replace the wtbs parameters with those from the newly optimized set and generate a list of .inp files that range in size from $s(1 : 40)p(1 : 40)d(1 : x_f)f(1 : x_f)$ to $s(1 : 40)p(1 : 40)d(1 : 40)f(1 : x_f)$.

Step 4. Optimize these new sets and select the smallest one that brings about an error that is still below the accuracy threshold. The size of this basis set is $s(1 : 40)p(1 : 40)d(1 : x_d)f(1 : x_f)$.

Step 5. Repeat steps 3 and 4 for the remaining symmetries. The size of the basis set at the end of this step will be of size $s(1 : x_s)p(1 : x_p)d(1 : x_d)f(1 : x_f)$.

Step 6. Replace the wtbs parameters with those from the newly optimized set and generate a list of input files that range in size from $s(1 : x_s)p(1 : x_p)d(1 : x_d)f(1 : x_f)$ to

$s(1 : x_s)p(1 : x_p)d(1 : x_d)f(x_f : x_f)$.

Step 7. Optimize and select from the basis sets. The new set will be of size $s(1 : x_s)p(1 : x_p)d(1 : x_d)f(y_f : x_f)$.

Step 8. Repeat steps 6 and 7 for the other symmetries except s. The final basis set will be of size $s(1 : x_s)p(y_p : x_p)d(y_d : x_d)f(y_f : x_f)$.

This process has a major drawback in that it requires a lot of computer power to run efficiently. But this is not really a problem if access to large computer clusters is available. **Additionally, considering that calculations for the chain 1-8 can be run in parallel for different atoms, this optimization process is trivially parallel.** The advantages of this are it finds a very small basis set that is still accurate, and it is also completely automatable. If there are no problems with individual calculations not converging, the output files never even need to be manually examined!

2.2 Discussion

The optimized wtbs parameters and basis set sizes for elements 2 to 86 are shown in Table 2.1.

Table 2.1: *Basis sets optimized using rwtps*

Element	Term Symbol	α	β	δ	γ	s	p	d	f
02 He	1S	8.140×10^{-02}	1.953	4.504	1.515	(1:18)			
03 Li	2S	1.596×10^{-02}	1.933	5.701	1.573	(1:22)			
04 Be	1S	2.647×10^{-02}	1.938	5.841	1.594	(1:22)			
05 B	2P	3.238×10^{-02}	1.948	5.573	1.523	(1:22)	(1:15)		
06 C	3P	4.613×10^{-02}	1.941	4.717	1.317	(1:23)	(1:15)		
07 N	4S	5.976×10^{-02}	1.923	5.183	1.442	(1:23)	(1:16)		
08 O	3P	6.967×10^{-02}	1.936	5.170	1.426	(1:23)	(1:16)		
09 F	2P	8.321×10^{-02}	1.942	5.084	1.408	(1:23)	(1:16)		
10 Ne	1S	9.943×10^{-02}	1.945	4.988	1.392	(1:23)	(1:16)		
11 Na	2S	1.696×10^{-02}	1.950	6.056	1.469	(1:26)	(3:19)		
12 Mg	1S	2.375×10^{-02}	1.932	5.686	1.430	(1:26)	(3:19)		
13 Al	2P	2.239×10^{-02}	1.898	5.555	1.413	(1:27)	(1:20)		
14 Si	3P	3.352×10^{-02}	1.921	5.419	1.410	(1:26)	(1:19)		
15 P	4S	4.410×10^{-02}	1.907	5.196	1.390	(1:26)	(1:19)		
16 S	3P	5.004×10^{-02}	1.896	4.962	1.362	(1:26)	(1:19)		
17 Cl	2P	5.831×10^{-02}	1.886	4.784	1.344	(1:26)	(1:19)		
18 Ar	1S	6.834×10^{-02}	1.878	4.654	1.331	(1:26)	(1:19)		
19 K	2S	1.392×10^{-02}	1.889	6.354	1.523	(1:28)	(3:22)		
20 Ca	1S	1.837×10^{-02}	1.874	6.109	1.504	(1:28)	(3:22)		
21 Sc	2D	2.040×10^{-02}	1.878	6.160	1.505	(1:28)	(3:22)	(2:16)	

Table 2.1: (*continued*)

Element	Term Symbol	α	β	δ	γ	s	p	d	f
22 Ti	3F	2.218×10^{-02}	1.886	6.247	1.508	(1:28)	(3:22)	(2:16)	
23 V	4F	2.361×10^{-02}	1.887	6.573	1.507	(1:29)	(3:23)	(2:16)	
24 Cr	7S	2.345×10^{-02}	1.865	5.348	1.407	(1:29)	(3:22)	(2:17)	
25 Mn	6S	2.572×10^{-02}	1.865	5.336	1.407	(1:29)	(3:22)	(2:17)	
26 Fe	5D	2.621×10^{-02}	1.869	5.301	1.395	(1:29)	(3:22)	(3:17)	
27 Co	4F	2.643×10^{-02}	1.871	5.905	1.413	(1:29)	(4:23)	(3:17)	
28 Ni	3F	2.882×10^{-02}	1.875	5.935	1.413	(1:29)	(3:23)	(3:17)	
29 Cu	2S	2.283×10^{-02}	1.857	5.485	1.413	(1:30)	(4:23)	(3:18)	
30 Zn	1S	3.297×10^{-02}	1.881	5.817	1.339	(1:30)	(3:24)	(2:17)	
31 Ga	2P	2.742×10^{-02}	1.863	5.703	1.462	(1:30)	(1:23)	(3:18)	
32 Ge	3P	3.143×10^{-02}	1.848	5.290	1.386	(1:30)	(1:23)	(4:18)	
33 As	4S	4.728×10^{-02}	1.869	5.325	1.340	(1:30)	(1:23)	(3:17)	
34 Se	3P	5.087×10^{-02}	1.847	4.949	1.533	(1:30)	(1:22)	(3:18)	
35 Br	2P	5.927×10^{-02}	1.861	5.192	1.333	(1:30)	(1:23)	(3:17)	
36 Kr	1S	6.804×10^{-02}	1.859	5.510	1.370	(1:29)	(1:23)	(3:17)	
37 Rb	2S	1.421×10^{-02}	1.856	7.181	1.626	(1:30)	(3:25)	(6:21)	
38 Sr	1S	1.808×10^{-02}	1.846	6.674	1.553	(1:30)	(3:25)	(6:20)	
39 Y	2D	1.997×10^{-02}	1.841	6.569	1.543	(1:30)	(3:25)	(2:20)	
40 Zr	5F	2.184×10^{-02}	1.837	6.485	1.535	(1:30)	(3:25)	(2:20)	
41 Nb	6D	2.340×10^{-02}	1.835	6.434	1.530	(1:30)	(3:25)	(2:20)	
42 Mo	7S	2.537×10^{-02}	1.832	6.372	1.524	(1:30)	(3:25)	(2:20)	
43 Tc	6S	2.597×10^{-02}	1.836	6.453	1.534	(1:30)	(3:25)	(2:20)	
44 Ru	5F	2.682×10^{-02}	1.833	6.788	1.606	(1:30)	(3:25)	(2:21)	
45 Rh	4F	2.751×10^{-02}	1.835	6.883	1.620	(1:30)	(3:25)	(2:21)	
46 Pd	1S	5.944×10^{-02}	1.827	5.980	1.504	(1:29)	(2:24)	(1:19)	
47 Ag	2S	2.438×10^{-02}	1.800	5.521	1.423	(1:31)	(4:25)	(3:21)	
48 Cd	1S	2.911×10^{-02}	1.786	5.334	1.408	(1:31)	(4:25)	(3:21)	
49 In	2P	2.714×10^{-02}	1.799	5.540	1.431	(1:31)	(1:25)	(3:21)	
50 Sn	3P	2.884×10^{-02}	1.794	5.422	1.415	(1:31)	(1:25)	(4:21)	
51 Sb	4S	4.481×10^{-02}	1.815	6.567	1.605	(1:30)	(1:25)	(3:21)	
52 Te	3P	4.846×10^{-02}	1.817	6.247	1.528	(1:30)	(1:26)	(3:20)	
53 I	2P	5.369×10^{-02}	1.810	6.016	1.505	(1:30)	(1:25)	(3:20)	
54 Xe	1S	5.981×10^{-02}	1.802	5.862	1.490	(1:30)	(1:25)	(3:20)	
55 Cs	2S	1.177×10^{-02}	1.792	6.485	1.510	(1:33)	(4:28)	(5:23)	
56 Ba	1S	1.560×10^{-02}	1.770	6.152	1.518	(1:32)	(3:28)	(6:23)	
57 La	2D	1.716×10^{-02}	1.764	6.067	1.513	(1:32)	(3:28)	(2:23)	(5:14)
missing									
59 Pr	4I	1.696×10^{-02}	1.776	6.297	1.533	(1:32)	(3:28)	(6:23)	(4:19)
60 Nd	5I	1.737×10^{-02}	1.778	6.563	1.579	(1:32)	(3:28)	(6:24)	(4:19)
61 Pm	6H	1.784×10^{-02}	1.780	6.615	1.585	(1:32)	(3:28)	(6:24)	(4:19)
62 Sm	7F	1.857×10^{-02}	1.778	5.842	1.464	(1:33)	(3:27)	(6:23)	(4:19)

Table 2.1: (*continued*)

Element	Term Symbol	α	β	δ	γ	s	p	d	f
63 Eu	8S	1.876×10^{-02}	1.784	6.718	1.596	(1:32)	(3:28)	(6:24)	(4:19)
missing									
missing									
66 Dy	5I	2.020×10^{-02}	1.788	6.543	1.540	(1:33)	(3:28)	(6:23)	(4:19)
missing									
68 Er	3H	2.106×10^{-02}	1.792	7.029	1.638	(1:32)	(3:28)	(6:24)	(4:20)
69 Tm	2F	2.152×10^{-02}	1.794	7.075	1.643	(1:32)	(3:28)	(6:24)	(4:20)
71 Lu	2D	2.373×10^{-02}	1.790	7.018	1.640	(1:32)	(3:28)	(2:24)	(4:20)
72 Hf	3F	2.511×10^{-02}	1.788	6.963	1.636	(1:32)	(3:28)	(2:24)	(5:20)
73 Ta	4F	2.684×10^{-02}	1.784	6.907	1.633	(1:32)	(2:28)	(2:24)	(5:20)
74 W	5D	2.814×10^{-02}	1.783	6.889	1.633	(1:32)	(3:28)	(2:24)	(5:20)
75 Re	6S	2.931×10^{-02}	1.782	6.886	1.634	(1:32)	(3:28)	(2:24)	(5:20)
76 Os	5D	3.075×10^{-02}	1.781	6.896	1.637	(1:32)	(2:29)	(2:24)	(4:20)
77 Ir	4F	3.195×10^{-02}	1.780	6.867	1.635	(1:32)	(3:28)	(2:24)	(5:20)
78 Pt	3D	3.173×10^{-02}	1.785	7.020	1.654	(1:32)	(3:28)	(2:24)	(5:20)
79 Au	2S	3.190×10^{-02}	1.791	6.702	1.569	(1:33)	(4:28)	(2:23)	(5:19)
80 Hg	1S	3.546×10^{-02}	1.781	6.691	1.604	(1:32)	(3:28)	(2:23)	(5:20)
81 Tl	2P	3.241×10^{-02}	1.794	6.826	1.561	(1:34)	(1:29)	(3:23)	(6:20)
82 Pb	3P	3.867×10^{-02}	1.778	6.948	1.656	(1:32)	(1:28)	(3:25)	(6:21)
83 Bi	4S	4.514×10^{-02}	1.766	6.158	1.545	(1:32)	(1:27)	(2:24)	(6:19)
84 Po	3P	4.783×10^{-02}	1.763	5.977	1.516	(1:32)	(1:27)	(3:23)	(6:19)
85 At	2P	5.210×10^{-02}	1.757	5.846	1.502	(1:32)	(1:27)	(3:23)	(6:19)
86 Rn	1S	5.716×10^{-02}	1.749	5.695	1.486	(1:32)	(1:27)	(3:23)	(6:19)

Chapter 3

CUDAProphet

3.1 GPU Architecture

While general purpose computing on graphics processing units (GPGPU) (GPUs) has been adopted by the high performance computing (HPC) community for quite some time, it can seem quite complex to the uninitiated. While most quantum chemists who decide to dip their toes into computational waters can get away with having little to no understanding of what is actually going on under the hood when programming for a central processing unit (CPU), the same can not at all be said for GPUs. Therefore, in this section I will explain what makes a GPU tick in order to help understand the terms and techniques used in the following chapter. As CUDAProphet was optimized for the Nvidia Tesla C2050, I will be referring to the Tesla C2050 specifications for examples when needed. I will begin with the most granularity possible, and zoom out so that by the end of this section, the reader should come away prepared for the rest of this chapter.

3.1.1 Threads, Blocks, and Grids

remember to go through label threads, blocks etc once a decision on how to label them has been made. The most granular element of computation on a GPU is the thread. When a CUDA function or subroutine (hereafter called a kernel) is called, it is a thread that actually executes the code. What makes GPGPU so powerful is that while a thread in a kernel is always executing the same code, the data a thread works with can be completely different between threads. This method of parallelism is called single instruction, multiple data (SIMD).

32 threads are organized into a structure called a warp. Within a warp, all threads execute code in lock-step. If a condition arises where some threads in warp must execute some code, and other threads in the same warp execute some other code (for instance in a **if else** statement), this leads to serialization of code execution, a process called warp

divergence. In a worst case scenario, this could cause all 32 threads to execute serially which could lead to a massive hit to performance. It can also cause warps to fall behind other warps and has the potential to cause race conditions to appear. If one warp needs data generated by another warp that is several steps behind, this can cause all kinds of confusing errors to appear. Therefore, it is generally advisable to avoid warp divergence whenever possible. But, for programs of any considerable usefulness, warp divergence will be inevitable. Thankfully, the good people of Nvidia have included the **syncthreads** command. This provides a barrier that all threads within a block must meet before any can continue.

The next highest structure is the thread block, or block for short. Simply put, a block is a collection of one or more threads. When a kernel is called, the block scheduler assigns each block to a streaming multiprocessor (SM). How many blocks a SM can run at once depends on the resources each block uses (more on this in section 3.1.2), but no more than 8 blocks per SM can ever be run at once on our C2050. Blocks also add an element of dimensionality to threads through the thread index (*threadIdx*) variable. This variable has three parts, *threadIdx.x*, *threadIdx.y*, and *threadIdx.z*. These variables make dealing with matrixes and arrays much simpler and can help with paralyzing large nested loops. The size of dimensions of the grid can be accessed through the *blockDim* variable. *blockDim* also has *x*, *y*, and *z* parts. These variables are all integer type, and the *threadIdx* variables range from 1 in Fortran (or 0 in C/++) to the relevant part of *blockDim* in Fortran (or *blockDim* -1 in C/++). The *threadIdx* set of variables are all reset at the boundary of each block.

Finally there is the grid. The grid is the entire collection of blocks that are launched for a kernel. They also give a dimensionality to blocks through the *blockIdx* variable which has similar parts to the *threadIdx* variable. It also adds in the *gridDim* variable which is analogous to *blockDim*.

3.1.2 Memory

To simplify the discussion of the kinds of memory available on a GPU we will limit ourselves to the three main types. They are the global memory, the shared memory, and the register memory. Because the execution time of a calculation is often not limited by the actual number of floating point operations to perform, but rather the reading and writing to memory, it is essential to be sure that the three main types of memory are managed correctly.

Global memory is both the most plentiful and the slowest available memory on a GPU. In a typical calculation, global memory will be allocated in code executed by the CPU (host). Data will be uploaded to the allocated memory and then used by the GPU (device). Global memory is distinct from the other kinds of device memory in that it can be read from and wrote to by all threads currently executing. This can allow threads of one block to pass

messages to threads in an other block, but great care must be taken to avoid race conditions. There is an additional caveat with global memory in that if threads are reading from non-contiguous memory locations, then memory bandwidth plummets. There is also a problem when multiple threads try to read from the same location. Instead of being one transaction, the read is serialized which leads to a massive performance penalty. It is therefore to the programmers advantage to write their kernels with as little use of the global memory as possible.

Shared memory, as its name implies, is shared and private to threads in a single block. It is able to be access much faster than global memory, but this comes at the cost of a much reduced volume. The amount of it available per SM varies with hardware generation, but the maximum available on our card is 48KB. This amount is shared among all blocks that are running on the SM, so the amount available per block will typically be even less. Also, there is no guarantee that memory loaded by one thread will be available to all other threads right away, only a call to **syncthreads** ensures proper loading. But, when used properly, shared memory is indispensable for speeding up calculations. It does not suffer from threads reading from non-contiguous memory locations like global memory does, and if all threads in a warp read from the same memory location a “broadcast” occurs. This means that instead of the read being serialized, all threads are able to read the data in one transaction. Shared memory is also distinct from global memory in that instead of being declared in the host code, it is declared in the device code. This is done by giving the variable the “shared” attribute in its declaration. One must be careful not to over use the shared memory however. As mentioned, there is only 48KB available per SM. If a block needs more than is available, the function will fail to execute. It can also affect the occupancy of the kernel, a concept that will be explained later on.

Lastly there is the register memory. This is the fastest memory available and is also private to an individual thread. **As this may have been a lot to take in, Figure 3.1 shows the hierarchy of the grid, blocks, and threads, as well as how each memory type is localized.** On the Tesla C2050, there are a total of 32768 32-bit registers available per SM. An individual thread in a block can use no more than 63 registers, and if the number of threads in a block times the number of registers needed per thread for a kernel is larger than the total registers available per SM then the execution of the kernel will fail. If a thread needs more memory than the registers will allow, then “register-spilling” takes place. This means that instead of a variable being stored into a register, it “spills over” into a fourth memory type called local memory. Local memory is really just a section of the global memory and comes with all the problems that global memory has, but the compiler is able to arrange it such that it can be read efficiently without the programmer having to worry about this. There is no way for a programmer to specify which variables will be stored in register or local memory,

the compiler handles all of this. There are compiler flags which can change the maximum amount of registers per thread allowed which will prompt the compiler to place more data into local memory. While this may sound like a bad idea, we will see in the next section how there are situations where this might be useful.

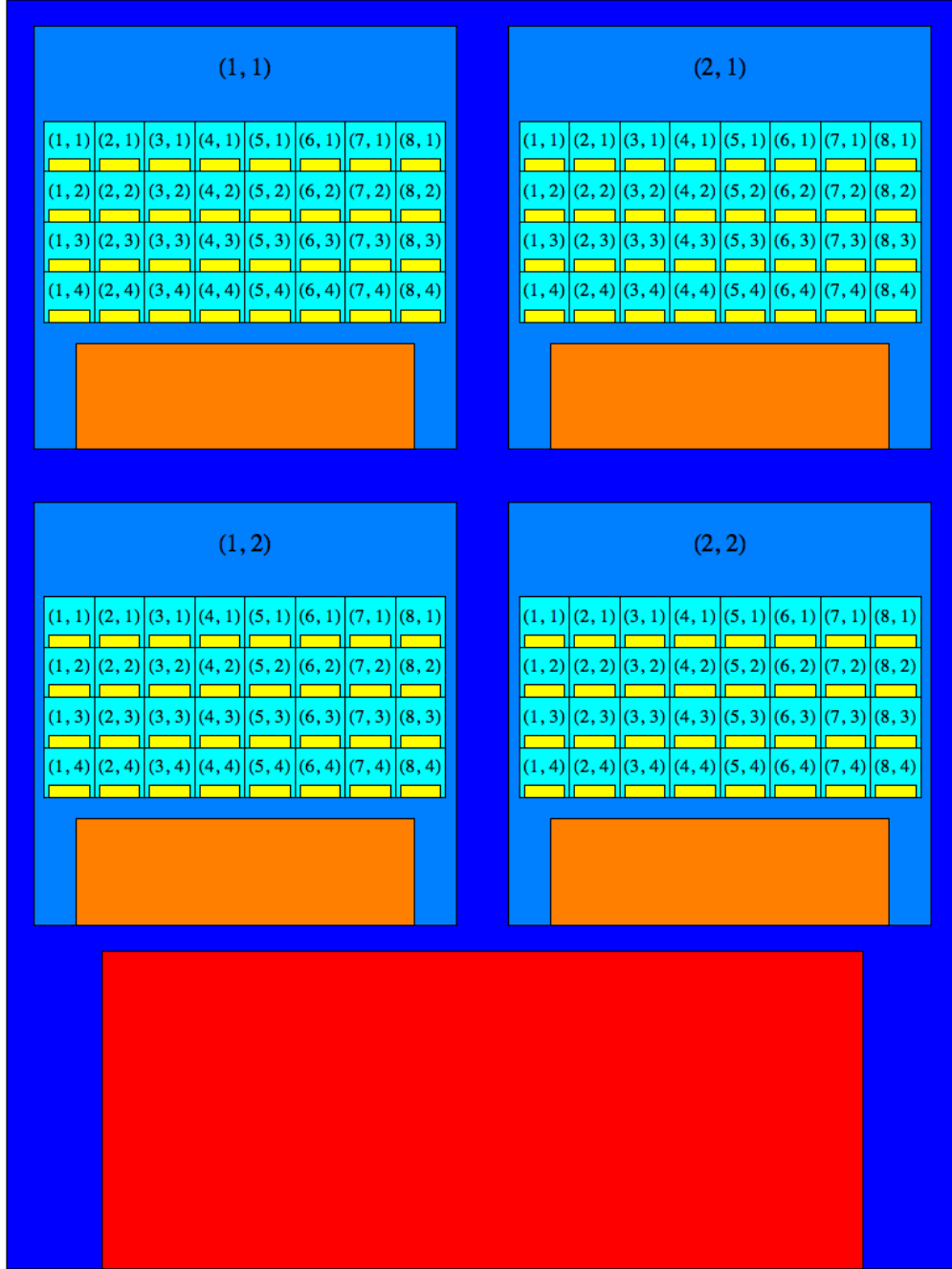


Figure 3.1: *Architecture of a GPU. This figure shows how threads and blocks are organized in the grid, as well as how the different memory types are localized within them. The grid is coloured dark blue, blocks are coloured light blue, and threads are coloured cyan. As for memory, the global memory is coloured red, shared memory is coloured orange, and register memory is coloured yellow. The numbers in parenthesis indicate the block or thread's `blockIdx` or `threadIdx` variables. In this particular case, $\text{blockDim} = (8, 4)$ and $\text{gridDim} = (2, 2)$. A thread is able to access memory that is inside its registers, but not the registers of other threads, and memory that is inside the shared memory of its block, but not the shared memory of other blocks. All threads can access all of the global memory.*

3.1.3 Occupancy and Performance

With the description of the various resources out of the way, now we can talk about how they can be used most effectively. The occupancy of a kernel is the number of warps that are actually able to run on a SM divided by the maximum possible warps a SM can run. Occupancy is important, as it determines the maximum number of blocks, and thus threads that are able to be run simultaneously. The occupancy of a kernel is determined by three things: the block size, the amount of shared memory per block, and the number of registers used per thread. While block size and shared memory are explicitly set by the programmer, it is next to impossible to guess how registers will be used before compile time. Thankfully, there are compiler flags that will display this information.

Armed with the resources our kernel will need, we can now determine its occupancy. The “CUDA Occupancy Calculator” is an Excel [cite the link to the sheet](#) spreadsheet provided by NVIDIA and is an indispensable tool. We can input the needed resources and the hardware generation of our card and then receive the occupancy of the current resource configuration, where the bottle neck of the current configuration is, as well as information on how the occupancy will change upon tweaking the configuration. In some sense, the best block size is determined by how the other resources are used. Therefore, in order to maximize the occupancy, we must first make sure that we are using the memory provided correctly.

Registers are the fastest memory available, but they are also the easiest to over use. Therefore, they are best suited for small variables that need to be referenced frequently, such as accumulators. Shared memory is best used for storing large sections of non-contiguous data stored in the global memory. Finally data that is able to be stored nicely, or is infrequently used might best be left in the global memory.

As with everything, these rules are only guide lines. The best resource configuration will be individual not only to every kernel, but the GPU it is to be run on as well. The best way to maximize the efficiency of a calculation is to make minor alterations to where data is stored and play around with the code. Finally, it might not always be the best idea to maximize the occupancy of a kernel. If the bottleneck is not the actual calculation, but how fast the data can be read, then it might be advantageous to fill up the registers and shared memory as much as possible and have a very low occupation. The only way to know is to try!

3.1.4 Matrix Multiplication Example

To complete the introduction to CUDA programming, we will go over a real example of CUDA Fortran code. For our example, we will examine code that performs matrix multiplication on matrices of arbitrary dimensions. This makes use of all the concepts we’ve covered

so far, so it will be a good exercise to examine. The code is given below.

```
1 module mat_mul
2   implicit none
3   integer, parameter :: dp = kind(0.d0)
4   integer, parameter :: block_dim = 16
5   integer, constant :: rowA, colA, rowB, colB
6   contains
7     attributes(global) subroutine mat_mul_kernel(inputA, inputB, output)
8       real(dp), dimension(:,:) :: inputA, inputB, output
9       real(dp) :: total
10      real(dp), shared, dimension(block_dim, block_dim) :: s_tileA, s_tileB
11      integer :: x, y, i, j
12
13      x = threadIdx%x + blockDim%x * (blockIdx%x - 1)
14      y = threadIdx%y + blockDim%y * (blockIdx%y - 1)
15      total = 0.0_dp
16
17      do i = 0, colA - 1, block_dim
18        call syncthreads()
19        if ((x <= rowA) .and. (threadIdx%y + i <= colA)) then
20          s_tileA(threadIdx%x, threadIdx%y) = inputA(x, threadIdx%y + i)
21        else
22          s_tileA(threadIdx%x, threadIdx%y) = 0.0_dp
23        endif
24
25        if ((threadIdx%x + i <= rowB) .and. (y <= colB)) then
26          s_tileB(threadIdx%x, threadIdx%y) = inputB(threadIdx%x + i, y)
27        else
28          s_tileB(threadIdx%x, threadIdx%y) = 0.0_dp
29        endif
30
31        call syncthreads()
32
33        do j = 1, block_dim
34          total = total + s_tileA(threadIdx%x, j) * s_tileB(j, threadIdx%y)
35        enddo
36
37      enddo
38
39      if((x <= rowA) .and. (y <= colB)) then
40        output(x, y) = total
41      endif
42    end subroutine mat_mul_kernel
43 end module mat_mul
44
45 program main
46   use cudafor
```

```

47 use mat_mul
48 implicit none
49 ! Host variables
50 real(dp), allocatable, dimension(:, :) :: inputA, inputB, output
51 ! Device variables
52 real(dp), device, allocatable, dimension(:, :) :: d_inputA, d_inputB,
    d_output
53 ! For generating random numbers
54 integer, allocatable, dimension(:) :: seed
55 integer :: n
56 ! block and grid dimensions
57 type(dim3) :: blockDim, gridDim
58
59 rowA = 1024
60 colA = 432
61 rowB = colA
62 colB = 1783
63
64 ! Set up random number generator
65 call random_seed(size = n)
66 allocate(seed(n))
67 ! Set seed to a constant so we always get the same matrices
68 seed = 150987
69 call random_seed(put = seed)
70 deallocate(seed)
71
72 ! Allocate host matrices and set up matrices
73 allocate(inputA(rowA, colA))
74 allocate(inputB(rowB, colB))
75 allocate(output(rowA, colB))
76
77 call random_number(inputA)
78 call random_number(inputB)
79
80 ! Allocate and upload device data
81 allocate(d_inputA(rowA, colA))
82 allocate(d_inputB(rowB, colB))
83 allocate(d_output(rowA, colB))
84 d_inputA = inputA
85 d_inputB = inputB
86
87 blockDim = dim3(block_dim, block_dim, 1)
88 gridDim = dim3((rowA - 1) / block_dim + 1, (colB - 1) / block_dim + 1, 1)
89
90 call mat_mul_kernel <<<gridDim, blockDim>>> (d_inputA, d_inputB, d_output)
91
92 ! Download device data

```

```

93  output = d_output
94
95  ! Deallocate variables
96  deallocate(d_inputA)
97  deallocate(d_inputB)
98  deallocate(d_output)
99  deallocate(inputA)
100 deallocate(inputB)
101 deallocate(output)
102 end program main

```

Listing 3.1: *Matrix Multiplication Kernel*

make sure line numbers are correct Lines 1 - 43 define a module which contains our matrix multiplication kernel. The kernel takes the matrices `inputA` and `inputB`, multiplies them together, and stores the result in `output`. The kernel has the `attributes(global)` attribute, which means that it can be called from either device or host code. `rowA`, `colA`, `rowB`, and `colB` are the rows and columns of `inputA` and `inputB`. Because they are small and don't change over the course of the calculation, they get stored in a special memory type called constant memory. This is a small cache of read only memory that is 64KB in size and allows for constant values to be read very quickly. There are two shared variables, `s_tileA` and `s_tileB`. Each of these are square matrices with dimensions of `block_dim` and these will be used to store sections or "tiles" of `inputA` and `inputB` respectively. We want each thread to do all the calculations needed to compute a simple element of `output`. Line 13 and 14 use the `threadIdx`, `blockDim`, and `blockIdx` variables so that each thread get a unique combination of `x` and `y` values. `x` corresponds to a row of the `output` matrix and `y` corresponds to a column, so each thread is then uniquely mapped to an element of `output`. We will see in the host code how these index values are determined, but for now it is sufficient to know that there will be at least as many threads as elements of `output`.

make sure line numbers are correct Lines 17 - 37 are where the actual calculation takes place. For each iteration of the loop, a different tile of `inputA` and `inputB` are fetched from the global memory and stored into the shared memory. The reason for using this tiling approach can be seen when we consider how the threads must read `inputB`. If we did not use shared memory, the loop would just be `total = total + inputA(x, i) * inputB(i, y)`. In this case, access to `inputA` is coalesced, but access to `inputB` is not. Therefore, the kernel will be stunted due this unoptimized memory usage. Instead, we load tiles in a coalesced manor into the shared memory, synchronize the threads, and then read through the shared memory in an uncoalesced fashion which carries no penalty. At the end of each loop, the tiles are moved in the `y` direction for `inputA` and in the `x` direction for `inputB`, and the process repeats. Notice how it is possible that a tile might fall on the boundary of an `input`

matrix. This is why there are checks to make sure the coordinates of the tile don't exceed the boundaries of a matrix so we don't start reading garbage.

make sure line numbers are correct In the host code on lines 45 - 102, we have two sets of variables, those for the host and those for the device. Notice how device variables have the **device** attribute. For clarity, they are given the same name as their host variables but with a **d_** prefix. Next we assign constant variables from line 5 to arbitrary values. Next we set up the random number generator, allocate the host matrices memory, and randomly initialize the input matrices. Then the device variables are allocated memory and the host data is then uploaded to the GPU. On lines 87 and 88 are where the block and grid dimensions are setup. **blockDim** and **gridDim** are variables of **type(dim3)** which is a tuple that is defined in the **cudafor** module. The block has two dimensions containing **block_dim²** threads. The grid also has two dimensions, and as we need one thread per element of output we perform a calculation which returns the minimum number of blocks that will cover it. Finally, we launch the kernel on line 90 using the **<<< >>>** operators to provide the grid and block dimensions, download the result from the GPU, and deallocate the memory.

3.2 Program flow

In this section, the control flow of a typical calculation is given.

Step 1. The input file is read by the **intin** subroutine. If needed, the basis set and open-shell configurations are calculated by **formbs** and **find_bin_configurations** respectively. The options set in the input file are rewritten to stdout.

Step 2. The calculation of small arrays and other constants is performed by **calc_parameters** and **bsnorm**. **calc_parameters** calls **bcoef** and **setvc**.

Step 3. The mapping of threads to the unrolled one and two-electron integral matrixes are calculated by **lmpqrsa** on the GPU. All other data from step 1 and 2 is then uploaded to the GPU.

Step 4. The one and two electron integrals are calculated on the GPU by **eint1gpu** and **eint2gpu** respectively.

Step 5. The initial guess of the density matrix is calculated by **guess** on the GPU. This is done by the diagonalization of the one-electron Hamiltonian matrix.

Step 6. **scfiter** then performs the SCF until convergence of the density matrix has been reached, or the maximum number of iterations has been reached. SCF is performed with cuSOLVER functions, as well as a few custom helper functions. The converged eigenvectors, values, and energies are downloaded from the GPU and then written to stdout.

Step 7 (optional). If **jobtyp='bsopt'**, then an optional basis set optimization is then carried out. This starts by assigning pointers to variables on the CPU and GPU through

hookup_cpu and *hookup_gpu*. Then, the four wtbs parameters are optimized by *newuoa*. *newuoa* calls *calc_energy* which essentially reconstructs the basis set from new parameters from *newuoa*, then repeats steps 2 - 6 and feeds the energy back into *newuoa*. This repeats until optimal wtbs parameters have been found.

3.3 Alterations for CUDA

While almost all of the code from the original DFRATOM was modified in some way, the most extreme changes were the integral evaluation and the formation of the P and Q matrixes. Therefore, we will discuss these changes in detail in this section.

3.3.1 Two-electron Integrals

The original DFRATOM calculates the two electron integrals with a long series of nested **for** loops. Working from the outside in, the indices of the loops are L , P , Q , M , R , and S where L and M are spinor symmetries, P and Q are basis functions of symmetry L , and R and S are basis functions of symmetry M . The pseudocode for these is shown in Algorithm 1. Where $nsym$ is the total number of symmetries, and $nbs(i)$ is the number of basis functions for symmetry species i . Thus, each set of J and K integrals is uniquely defined by its L , M , P , Q , R , and S values. In order to have an effective CUDA implementation of this algorithm, we need to find a way to map these six numbers to CUDA threads. The simplest approach would be to map the values of L , P , and Q onto the *threadIdx.x*, *threadIdx.y*, and *threadIdx.z* variables for each thread, launch the needed number of blocks, and then have each thread loop over the remaining indices. The pseudocode for this can be seen in Algorithm 2. A similar method is employed by many other programs, and for larger systems it works perfectly well. But because this program is for only single atoms, problems begin to appear.

The first problem arises due to warp divergence. Because the maximum values of M , R , and S depend on L , P , and Q , different threads will have a different number of loops to complete than others. Because a streaming multiprocessor (SM) must finish the block it is currently working on before it can grab another, there is the possibility that most of the threads in a block are idling while waiting for others in the same block to finish. The second problem is that with this method, there will always be several blocks that have threads that remain idle throughout the block's runtime, no matter what. **This can be seen more clearly in Figure 3.2.** The third problem with this is that with the maximum values for L , P , and Q available for a single atom, there might not even be enough combinations to completely fill the GPU. While all of these issues begin to disappear once the number of integrals to

Algorithm 1 The original

```
for  $L = 1$  to  $nsym$  do
  for  $P = 1$  to  $nbs(L)$  do
    for  $Q = 1$  to  $P$  do
      for  $M = 1$  to  $L$  do
        if  $L = M$  then
           $maxr = P$ 
        else
           $maxr = nbs(M)$ 
        end if
        for  $R = 1$  to  $maxr$  do
          if  $(L = M)$  and  $(P = R)$  then
             $maxs = Q$ 
          else
             $maxs = R$ 
          end if
          for  $S = 1$  to  $maxs$  do
            compute the J and K integrals of  $L, M, P, Q, R,$  and  $S$ 
          end for
        end for
      end for
    end for
  end for
end for
```

Algorithm 2 Easy Code

$$L = threadIdx.x + (blockIdx.x - 1) * blockDim.x$$
$$P = threadIdx.y + (blockIdx.y - 1) * blockDim.y$$
$$Q = threadIdx.z + (blockIdx.z - 1) * blockDim.z$$

if $(L \leq nsym)$ **and** $(P \leq nbs(L))$ **and** $(Q \leq P)$ **then**

for $M = 1$ to L **do**

if $L = M$ **then**

$maxr = P$

else

$maxr = nbs(M)$

end if

for $R = 1$ to $maxr$ **do**

if $(L = M)$ **and** $(P = R)$ **then**

$maxs = Q$

else

$maxs = R$

end if

for $S = 1$ to $maxs$ **do**

 compute the J and K integrals of L , M , P , Q , R , and S

end for

end for

end for

end if

evaluate becomes large enough, we are still very much in the range where they are in play. Therefore a smarter algorithm had to be used.

The second problem can be solved by opting restricting the dimensionality of the solution instead of the using the three dimensional one of Algorithm 2. We might consider using a two dimensional solution, but then we still will have blocks with idle threads which can be seen in Figure 3.3 By restricting ourselves to only using *threadIdx.x*, we ensure that only the last block to run will have the possibility of having threads remaining idle throughout the block's runtime as shown in Figure 3.4. The third problem can be solved by having each thread calculate one and only one set of J and K integrals. If we start with a valid combination of *L*, *M*, *P*, *Q*, *R*, and *S* values, we can very easily figure out which thread will calculate that set of integrals by using the following equations:

$$n'(j) = \frac{j^2 + j}{2} \quad (3.1)$$

$$y = n'(nbs(L - 1)) + n'(P - 1) + Q \quad (3.2)$$

$$x = n'(nbs(M - 1)) + n'(R - 1) + S \quad (3.3)$$

$$i = n'(x) + y \quad (3.4)$$

$$i_{max} = n'(\sum_{j=1}^{nsym} n'(nbs(j))) \quad (3.5)$$

where $nbs(0) = 0$ and $i = threadIdx.x + (blockIdx.x - 1) * blockDim.x$. Starting with a value of *i* and working our way back is a much more challenging task. It becomes easier if we reframe it in the following way.

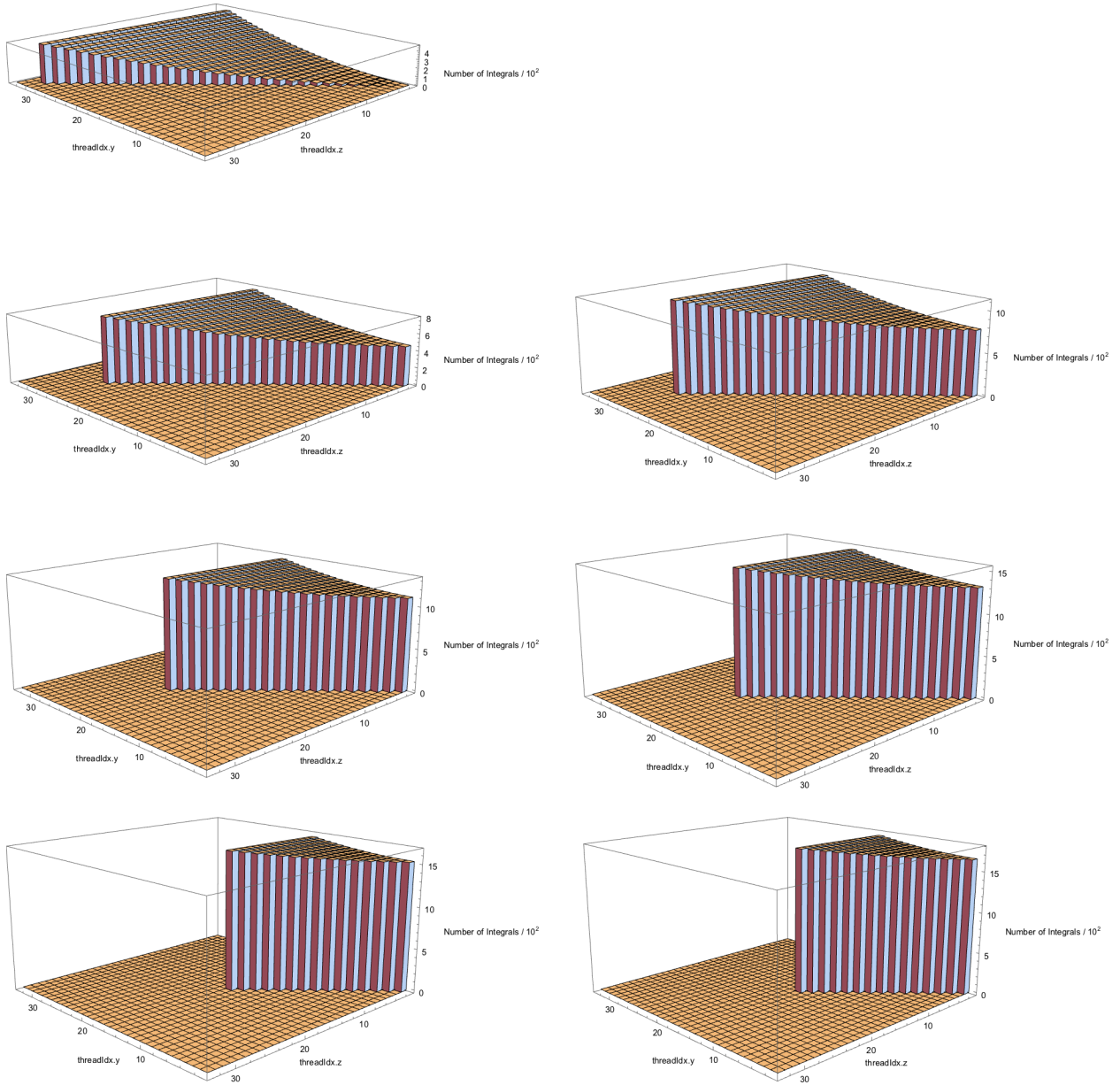


Figure 3.2: *Two-Electron Integral Layout: Naive Implementation.* This figure shows how integrals get mapped to threads using Algorithm 2. The basis set used had $n_{\text{sym}} = 7$ with $n_{\text{bs}}(i) = (30, 25, 25, 20, 20, 15, 15)$. The $\text{blockDim} = (1, 32, 32)$ and $\text{gridDim} = (7, 1, 1)$. Each histogram is one block of threads, where the horizontal axes are the threadIdx.y and threadIdx.z variables, and the vertical axis is the number of Two-Electron Integrals it must compute. Notice that in all blocks there are threads that calculate no integrals and therefore are idle throughout the lifetime of the block. Also note that the number of integrals each thread must calculate grows both from block to block, and from thread to thread within a block. Both of these factors make this algorithm an undesirable choice.

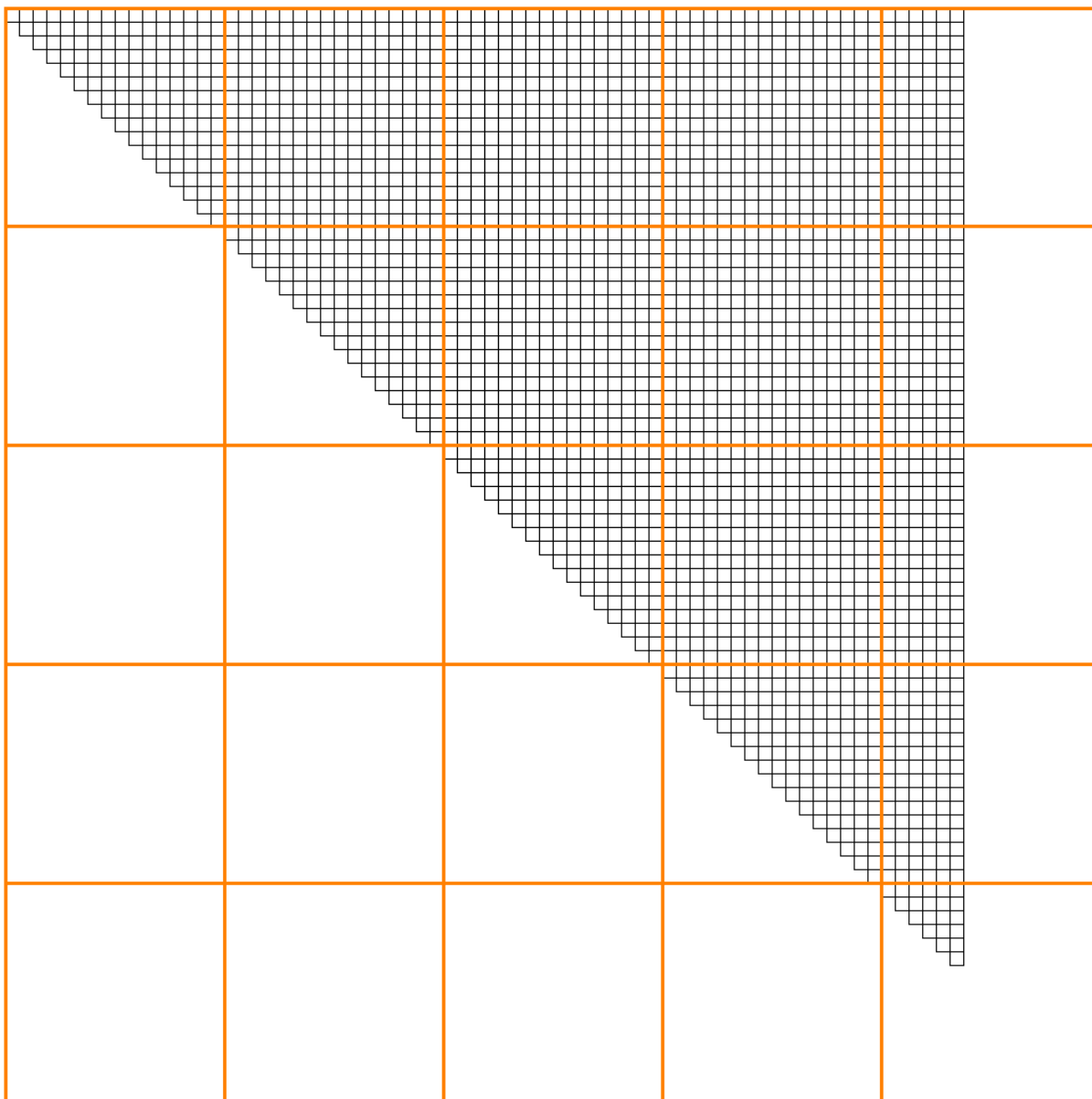


Figure 3.3: *Two-Electron Integral Layout: 2D Blocks and Grid. Orange lines indicate the border of blocks. Notice that many blocks have idle threads.*

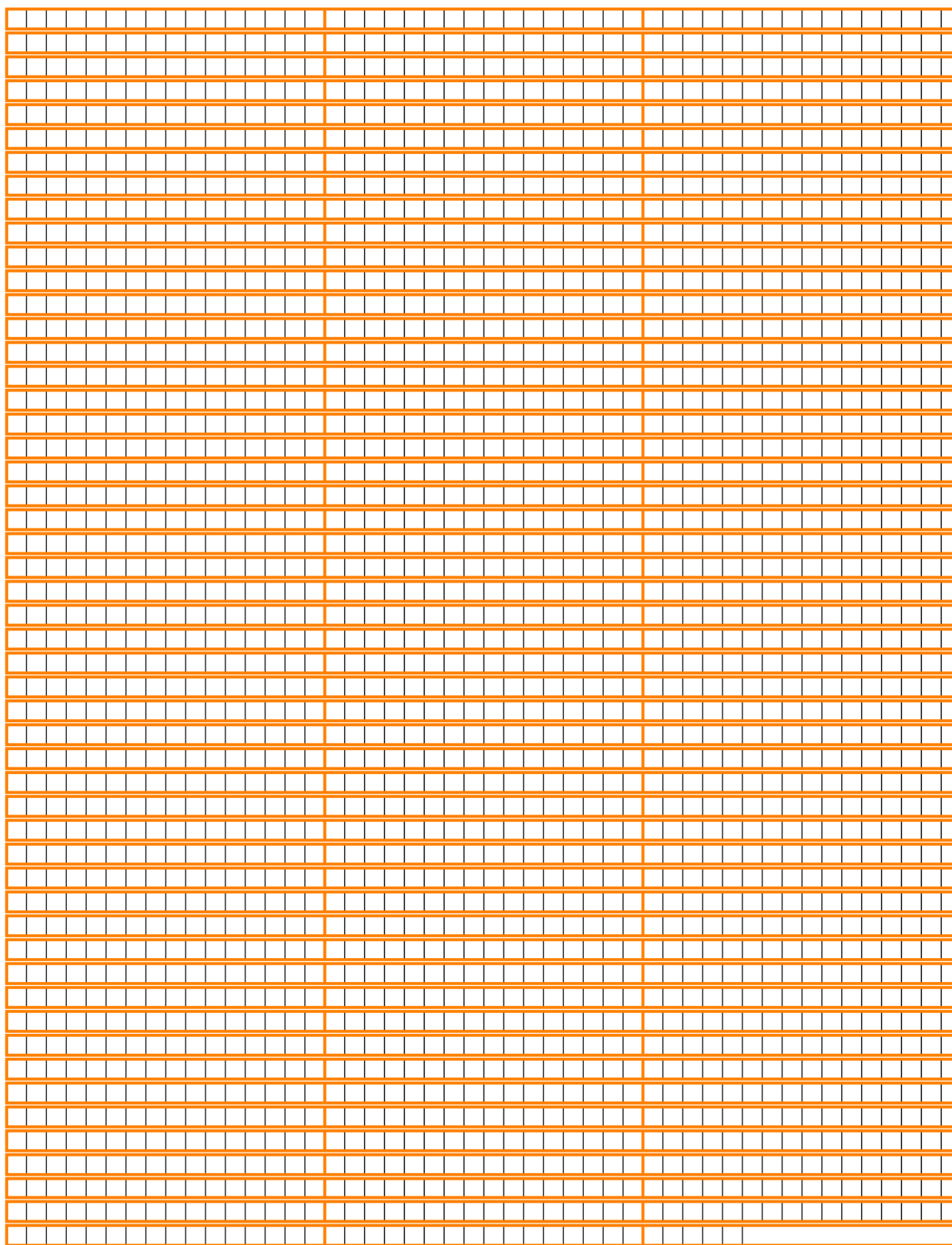


Figure 3.4: *Two-Electron Integral Layout: 1D Blocks and Grid. Orange lines indicate the border of blocks. Notice that only the last block contains idle threads.*

Consider Figures 3.5 and 3.6. Figure 3.5 shows the top half of the symmetric two-electron integral matrix for a problem with $nsym = 2$ and 3 basis function for symmetry one, and 2 for symmetry two. Figure 3.6 shows the same but mapped onto a vector representation. In each element of both figures, there is a set of seven numbers. The top two are L and M , then P and Q , then R and S , and the last number is the value of i for the thread calculating that integral. With this, it can be seen from the matrix representation that each element in the same row have the same M , R , and S values and the elements in the same column have the same L , P , and Q values. Therefore, finding out which column the element belongs to gives us the value for y , and finding the row give us the value of x . This can be done with the binary search algorithm shown in Algorithm 3.

Algorithm 3 Binary Search for x and y

```

if  $theadIdx.x \leq nsym$  then
     $s\_nsym = nsym$ 
     $s\_nbs(theadIdx.x) = nbs(theadIdx.x)$ 
     $s\_nprime(theadIdx.x) = n'(s\_nbs(theadIdx.x))$ 
end if
call syncthreads
 $i = threadIdx.x + (blockIdx.x - 1) * blockDim.x$ 
if  $i \leq i_{max}$  then
     $low = 1$ 
     $high = \text{sum}(s\_nprime(1 : s\_nsym))$ 
    while  $low \leq high$  do
         $mid = \frac{(low+high)}{2}$ 
         $lownum = \frac{(mid-1)(mid-2)}{2} + mid$ 
         $highnum = lownum - 1 + mid$ 
        if  $(i \leq highnum)$  and  $(i \geq lownum)$  then
             $y = mid$ 
            exit
        else if  $i > highnum$  then
             $low = mid + 1$ 
        else if  $i < lownum$  then
             $high = mid - 1$ 
        end if
    end while
     $x = i - lownum + 1$ 
end if

```

In this algorithm, variables with the $s_$ prefix refer to those in the shared memory, and $lownum$ and $highnum$ refer to the minimum and values the i could be for the current guess (mid) of y . From here, L can be found with Algorithm 4, and P and Q can be found with Algorithm 5. The same set of algorithms can then be used to get M , R , and S

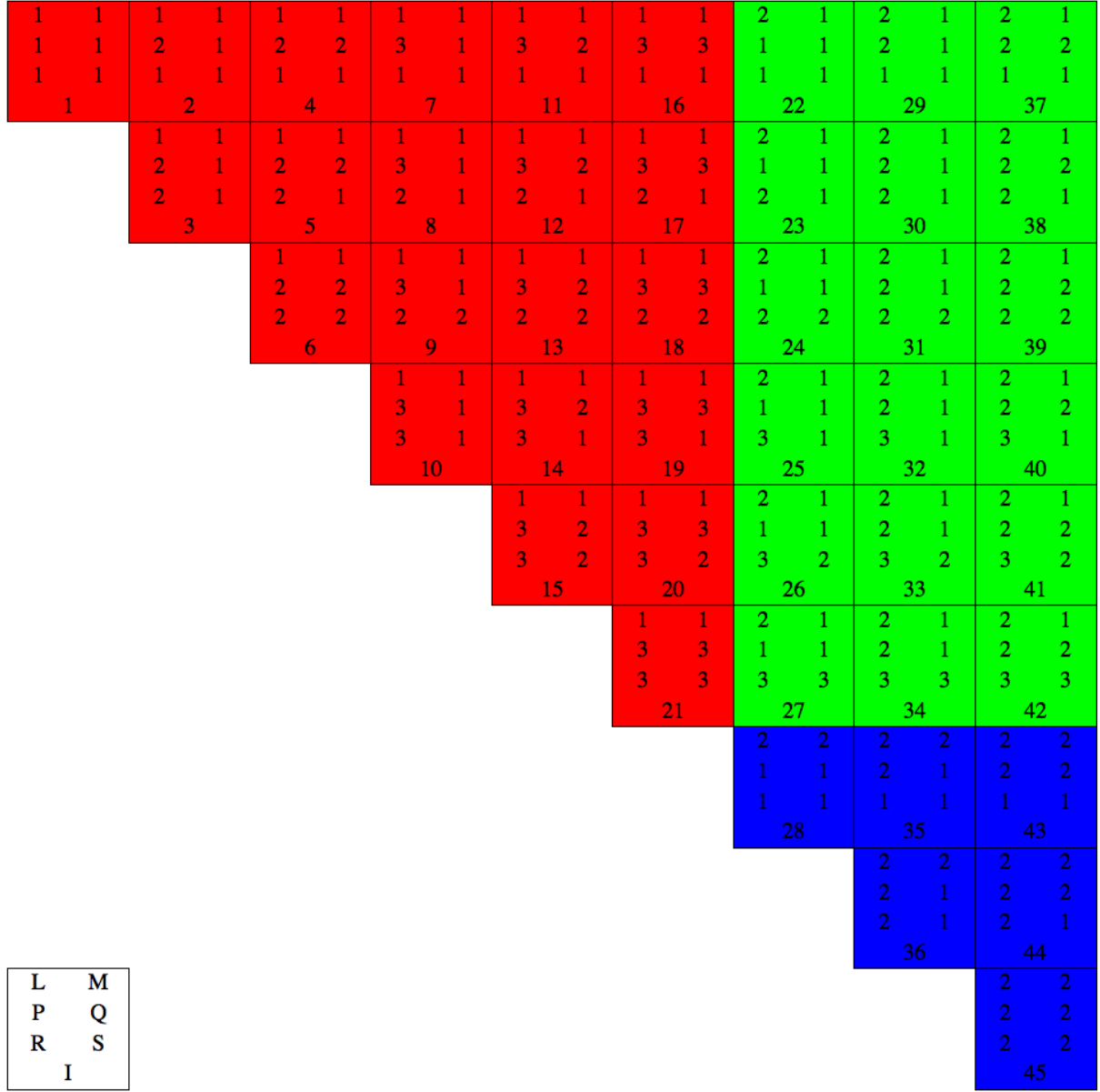


Figure 3.5: *Two-electron Integral Matrix as a Matrix.* This figure shows the top triangle of the two-electron symmetric matrix in a matrix representation. In this instance, the total number of symmetries is 2, with 3 basis functions for the first symmetry and 2 basis functions for the second. The key for translating the numbers in each box into the L, M, P, Q, R, and S values is given in the bottom left. Notice how all rows have the same M, R, and S values, while all columns have the same L, P, and Q values. For clarity, the boxes have been coloured based on the spinor symmetries that they contain.

by substituting the relevant variables. If there is sufficient global memory available, these values can be stored and referred to later as needed. Otherwise, they could be calculated on the fly as needed. Because binary search scales as $\mathcal{O}(n \log n)$, this should ensure that this

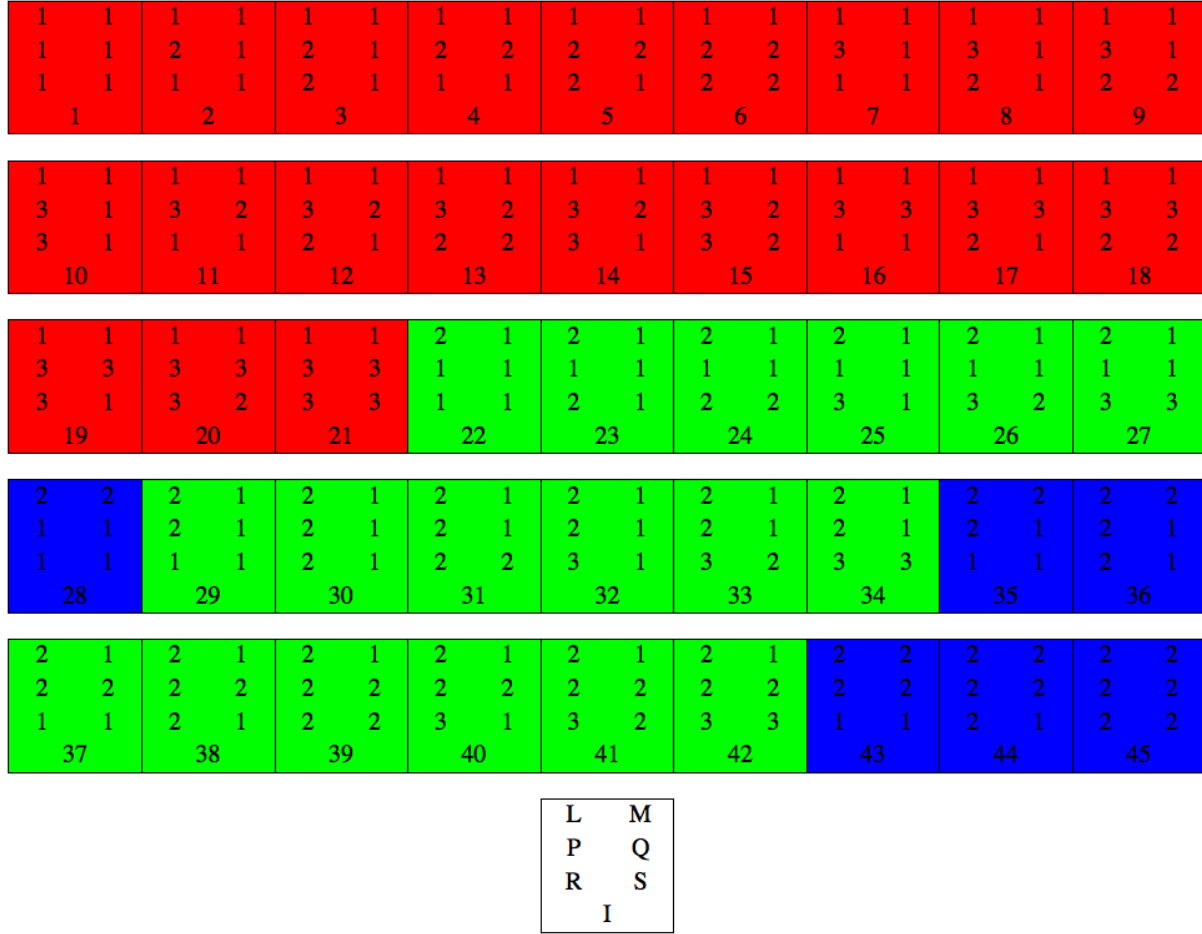


Figure 3.6: *Two-electron Integral Matrix as a Vector.* This figure shows the top triangle of the two-electron symmetric matrix in a vector representation. In this instance, the total number of symmetries is 2, with 3 basis functions for the first symmetry and 2 basis functions for the second. The key for translating the numbers in each box into the L , M , P , Q , R , and S values is given in the bottom center. Notice how the L , M , P , Q , R , and S values are much harder to predict than from the matrix representation. For clarity, the boxes have been coloured based on the spinor symmetries that they contain. Gaps between rows are used to reinforce that the elements are not connected like a matrix.

remains a fast method of mapping threads to integrals for large problems as well. With some alterations, this method could also apply to molecular symmetries and not only to a single atom. For instance in the C_1 symmetry, all possible combinations of four basis functions must be used (ignoring those that appear on the bottom triangle of the two electron integral matrix of course). We could simply remove the search for L and M , have the initial value of *high* in Algorithm 5 be the total number of basis functions, and remove the `sum(s_nprime(1 : L - 1))` term from *lownum*.

From here, the code for evaluating the integrals remains largely the same as the original

Algorithm 4 Binary Search for L

```
 $i = \text{threadIdx}.x + (\text{blockIdx}.x - 1) * \text{blockDim}.x$ 
if  $i \leq i_{\max}$  then
   $\text{low} = 1$ 
   $\text{high} = s_{\text{nsym}}$ 
  while  $\text{low} \leq \text{high}$  do
     $\text{mid} = \frac{(\text{low} + \text{high})}{2}$ 
     $\text{lownum} = 1 + \text{sum}(s_{\text{nprime}}(1 : \text{mid} - 1))$ 
     $\text{highnum} = \text{lownum} - 1 + s_{\text{nprime}}(\text{mid})$ 
    if  $(i \leq \text{highnum})$  and  $(i \geq \text{lownum})$  then
       $L = \text{mid}$ 
      exit
    else if  $y > \text{highnum}$  then
       $\text{low} = \text{mid} + 1$ 
    else if  $y < \text{lownum}$  then
       $\text{high} = \text{mid} - 1$ 
    end if
  end while
end if
```

Algorithm 5 Binary Search for P and Q

```
 $i = \text{threadIdx}.x + (\text{blockIdx}.x - 1) * \text{blockDim}.x$ 
if  $i \leq i_{\max}$  then
   $\text{low} = 1$ 
   $\text{high} = s_{\text{nbs}}(L)$ 
  while  $\text{low} \leq \text{high}$  do
     $\text{mid} = \frac{(\text{low} + \text{high})}{2}$ 
     $\text{lownum} = \frac{(\text{mid} - 1)(\text{mid} - 2)}{2} + \text{mid} + \text{sum}(s_{\text{nprime}}(1 : L - 1))$ 
     $\text{highnum} = \text{lownum} + \text{mid} - 1$ 
    if  $(i \leq \text{highnum})$  and  $(i \geq \text{lownum})$  then
       $P = \text{mid}$ 
      exit
    else if  $y > \text{highnum}$  then
       $\text{low} = \text{mid} + 1$ 
    else if  $y < \text{lownum}$  then
       $\text{high} = \text{mid} - 1$ 
    end if
  end while
   $Q = y - \text{lownum} + 1$ 
end if
```

code^{citation needed}, except for some minor changes to allow for more efficient global or shared memory access. We also use a process referred to as “grid-stride looping” where all these binary search algorithms have their **if** $i \leq i_{max}$ **then** removed, and then are placed within the following loop: **for** $i = threadIdx.x + (blockIdx.x - 1) * blockDim.x$ to i_{max} , $i += blockDim.x * gridDim.x$ **do**. If we know the occupancy of the algorithm on the GPU beforehand, we can launch exactly the number of blocks that will fill the GPU. This reduces the overhead of block swapping and lets us further eke out some performance.

3.3.2 Density Matrix Formation

The term in parenthesis in Equation 1.2 corresponds to the non-relativistic density matrix. But as we are interested in relativistic calculations, it will need to be altered. Firstly, because a relativistic wavefunction has both large and small components, so too does its density matrix. Therefore, it will have two dimensions that span from 1 to $2K$ where dimensions 1 to K are the large components and the rest are for small components. Secondly, the factor of 2 is replaced by the number of electrons that can occupy spinors of symmetry μ . Its matrix and vector representations are shown in Figures 3.7 and 3.8 respectively. Its mathematical form is given in Equation 3.6

$$\mathbf{D}_{C_{r,s}} = N_{\mu} \sum_{a=1}^{occ} \mathbf{C}_{r,a} \mathbf{C}_{s,a}^* \quad (3.6)$$

where N_i is the number of electrons that occupy a closed shell of symmetry i . $\mathbf{D}_{\mathbf{O}}$ is similar to this and is shown below.

$$\mathbf{D}_{\mathbf{O}_{r,s}} = N_i \mathbf{C}_{r,(occ+1)} \mathbf{C}_{s,(occ+1)}^* \quad (3.7)$$

where N_i is equal to the number of electrons in the open shell. Note that in this program, only the most energetic shell of any symmetry can be open.

The total density matrix is $\mathbf{D}_{\mathbf{T}} = \mathbf{D}_{\mathbf{C}} + \mathbf{D}_{\mathbf{O}}$

3.3.2.1 P Q Super Matrices

With the density matrices in hand we can begin to combine the two-electron integrals from the previous section. These integrals have the following form.

$1^L \quad 1^L$ 1	$2^L \quad 1^L$ 2	$3^L \quad 1^L$ 4	$4^L \quad 1^L$ 7	$5^L \quad 1^L$ 11	$1^S \quad 1^L$ 16	$2^S \quad 1^L$ 22	$3^S \quad 1^L$ 29	$4^S \quad 1^L$ 37	$5^S \quad 1^L$ 46
	$2^L \quad 2^L$ 3	$3^L \quad 2^L$ 5	$4^L \quad 2^L$ 8	$5^L \quad 2^L$ 12	$1^S \quad 2^L$ 17	$2^S \quad 2^L$ 23	$3^S \quad 2^L$ 30	$4^S \quad 2^L$ 38	$5^S \quad 2^L$ 47
		$3^L \quad 3^L$ 6	$4^L \quad 3^L$ 9	$5^L \quad 3^L$ 13	$1^S \quad 3^L$ 18	$2^S \quad 3^L$ 24	$3^S \quad 3^L$ 31	$4^S \quad 3^L$ 39	$5^S \quad 3^L$ 48
			$4^L \quad 4^L$ 10	$5^L \quad 4^L$ 14	$1^S \quad 4^L$ 19	$2^S \quad 4^L$ 25	$3^S \quad 4^L$ 32	$4^S \quad 4^L$ 40	$5^S \quad 4^L$ 49
				$5^L \quad 5^L$ 15	$1^S \quad 5^L$ 20	$2^S \quad 5^L$ 26	$3^S \quad 5^L$ 33	$4^S \quad 5^L$ 41	$5^S \quad 5^L$ 50
					$1^S \quad 1^S$ 21	$2^S \quad 1^S$ 27	$3^S \quad 1^S$ 34	$4^S \quad 1^S$ 42	$5^S \quad 1^S$ 51
						$2^S \quad 2^S$ 28	$3^S \quad 2^S$ 35	$4^S \quad 2^S$ 43	$5^S \quad 2^S$ 52
							$3^S \quad 3^S$ 36	$4^S \quad 3^S$ 44	$5^S \quad 3^S$ 53
								$4^S \quad 4^S$ 45	$5^S \quad 4^S$ 54
									$5^S \quad 5^S$ 55

$$\begin{matrix} R^x & S^x \\ I \end{matrix}$$

Figure 3.7: *Density Matrix as a Matrix.* This figure shows the top triangle of the symmetric density matrix in a matrix representation. The key for translating the numbers in each box into the R and S values is given in the bottom right. All active symmetries would have a similar, separate matrix. For clarity, the elements have been colorized based on the combination of large and small components they contain.

1^L 1^L 1	2^L 1^L 2	2^L 2^L 3	3^L 1^L 4	3^L 2^L 5	3^L 3^L 6	4^L 1^L 7	4^L 2^L 8	4^L 3^L 9	4^L 4^L 10	
5^L 1^L 11	5^L 2^L 12	5^L 3^L 13	5^L 4^L 14	5^L 5^L 15	1^S 1^L 16	1^S 2^L 17	1^S 3^L 18	1^S 4^L 19	1^S 5^L 20	
1^S 1^S 21	2^S 1^L 22	2^S 2^L 23	2^S 3^L 24	2^S 4^L 25	2^S 5^L 26	2^S 1^S 27	2^S 2^S 28	3^S 1^L 29	3^S 2^L 30	
3^S 3^L 31	3^S 4^L 32	3^S 5^L 33	3^S 1^S 34	3^S 2^S 35	3^S 3^S 36	4^S 1^L 37	4^S 2^L 38	4^S 3^L 39	4^S 4^L 40	
4^S 5^L 41	4^S 1^S 42	4^S 2^S 43	4^S 3^S 44	4^S 4^S 45	5^S 1^L 46	5^S 2^L 47	5^S 3^L 48	5^S 4^L 49	5^S 5^L 50	
5^S 1^S 51	5^S 2^S 52	5^S 3^S 53	5^S 4^S 54	5^S 5^S 55						P^x Q^x I

Figure 3.8: *Density Matrix as a Vector.* This figure shows the top triangle of the symmetric density matrix in a vector representation. The key for translating the numbers in each box into the R and S values is given in the bottom left. All active symmetries would have a similar, similar vector that continue one after the other. For clarity, the elements have been colorized based on the combination of large and small components they contain.

$$\begin{aligned}
\mathbf{X}_{I(pqrs)}^1 &= (p_L q_L | r_L s_L) - \frac{1}{2} [(p_L r_L | q_L s_L) + (p_L s_L | q_L r_L)] \\
\mathbf{X}_{I(pqrs)}^2 &= (p_L q_L | r_S s_S) \\
\mathbf{X}_{I(pqrs)}^3 &= (p_S q_S | r_L s_L) \\
\mathbf{X}_{I(pqrs)}^4 &= -(p_L r_S | q_S s_L) \\
\mathbf{X}_{I(pqrs)}^5 &= -(p_L s_S | q_S r_L) \\
\mathbf{X}_{I(pqrs)}^6 &= -(p_S r_L | q_L s_S) \\
\mathbf{X}_{I(pqrs)}^7 &= -(p_S s_L | q_L r_S) \\
\mathbf{X}_{I(pqrs)}^8 &= (p_S q_S | r_S s_S) - \frac{1}{2} [(p_S r_S | q_S s_S) + (p_S s_S | q_S r_S)]
\end{aligned} \tag{3.8}$$

where the subscripts L and S refer to the large and small components respectively. I is a function that returns the values of x and y from Equations 3.2 and 3.3 where $L = \lambda$ and $M = \mu$. From here, all subscripts of \mathbf{X} will be implied to go through this I function.

The \mathbf{P} and \mathbf{Q} super matrices have a similar layout to the density matrices. The elements of \mathbf{P} have the following form

$$\begin{aligned}
\mathbf{P}_{pq}^{LL} &= \sum_{\mu=1}^{nsym} \sum_{r=1} \sum_{s=1} \left(\mathbf{D}_{\mathbf{T}rs}^{LL} \mathbf{X}_{pqrs}^1 + \mathbf{D}_{\mathbf{T}rs}^{SS} \mathbf{X}_{pqrs}^{J(pqrs)} \right) \\
\mathbf{P}_{pq}^{SS} &= \sum_{\mu=1}^{nsym} \sum_{r=1} \sum_{s=1} \left(\mathbf{D}_{\mathbf{T}rs}^{SS} \mathbf{X}_{pqrs}^8 + \mathbf{D}_{\mathbf{T}rs}^{LL} \mathbf{X}_{pqrs}^{J(pqrs)} \right) \\
\mathbf{P}_{pq}^{LS} &= \sum_{\mu=1}^{nsym} \sum_{r=1} \sum_{s=1} \left(\mathbf{D}_{\mathbf{T}rs}^{LS} \mathbf{X}_{pqrs}^5 + \mathbf{D}_{\mathbf{T}rs}^{SL} \mathbf{X}_{pqrs}^{J(pqrs)} \right) \\
\mathbf{P}_{pq}^{SL} &= \sum_{\mu=1}^{nsym} \sum_{r=1} \sum_{s=1} \left(\mathbf{D}_{\mathbf{T}rs}^{SL} \mathbf{X}_{pqrs}^7 + \mathbf{D}_{\mathbf{T}rs}^{LS} \mathbf{X}_{pqrs}^{J(pqrs)} \right)
\end{aligned} \tag{3.9}$$

where $J(pqrs)$ is an integer function that selects specific definitions of the two-electron integrals from Equation 3.8. How exactly this works is shown in the following sections.

The form of the \mathbf{Q} super matrix is almost the same as \mathbf{P} , but there will be an additional multiplication of the vector coupling coefficient between the open shells for each summation.

3.3.2.2 FORMPQ Algorithm

Forming the \mathbf{P} and \mathbf{Q} matrices proved to be the most difficult part of this program to parallelize. Its difficulty was due to the second term in the summations where each \mathbf{X} matrix used depends on how far the summation has progressed. Further, each element of

\mathbf{P} will have the \mathbf{X} matrix change at different times. These two factors result in many **if** statements which GPUs have difficulty handling in parallel. There is also the issue where each element in the LL and SS sections need to loop over every element of the same sections of the density matrix, as do the LS and SL sections. This means that there will be lots of global memory reads unless handled properly. All together, this makes for a very ugly algorithm to try and parallelize.

Two different attempts were made to try and accelerate this algorithm. Each tried to exploit a different property of GPUs. One attempted to combat the amount of floating point operations to perform by distributing the amount of work to as many threads as possible. This will be referred to as the Multiple Threads Single Element (MTSE) algorithm. The second one tried to deal with the amount of global memory reads by making the most of each read. This will be referred to as Single Thread Single Element (STSE) algorithm. Each will be described in the sections below.

3.3.2.3 MTSE Algorithm

The acronyms for these algorithms don't really describe what each does, think of something better

As previously stated, the purpose of the MTSE algorithm is to distribute the amount of work needed to be done to as many threads as possible. This was achieved by having multiple threads each computing two of the multiplications needed for a single element of two matrices in Equation 3.9, and then combining the products of these multiplications in parallel. These two tasks were performed by two separate kernels and the pseudocode for each is shown in Algorithms 6 and 7 respectively. The pseudocode shows only the calculation for the **the final algorithm has changed from 2 to 1 kernel, update the code \mathbf{P} matrices, and Algorithm 7 shows only the calculation of the \mathbf{P}^{LL} portion.**

inttype refers to one or more if the \mathbf{P} or \mathbf{Q} matrices. In Algorithm 6, *inttype* can have a value of 0 or 1, where 0 refers to the LL and SS \mathbf{P} or \mathbf{Q} matrices, and 1 is the SL and LS matrices. In Algorithm 7, *inttype* can have a values ranging from 0 to 7 where 0 to 3 are the \mathbf{P} LL , SS , SL , LS matrices, and the rest are the \mathbf{Q} matrices in the same order. d is \mathbf{D} in its vector representation and pos are the locations of the p and q elements within it. $locmx$ is an array with 7 elements, and each element is equal to the number of unique LL , SS , SL , LS elements of symmetry $l - 1$. $fctr$ accounts for when an element is single or double counted in the summation. $x(\chi, :)$ is the vector representation of \mathbf{X}^χ , loc is the location of needed pqr s element in the vector, $total$ is the length of one of the dimensions of \mathbf{X}^χ , and $k1$ and $k2$ are the outputs of the J function from Equation 3.9. Finally, pXY variables are matrices dimension $total \times total$ and store the sum of the two multiplications of the d and x vectors, pmx if the vector representation of the \mathbf{P} matrices, and $stile$ is a vector in the

Algorithm 6 Kernel 1 for MTSE

```
i = threadIdx.x + (blockIdx.x - 1) * blockDim.x
j = blockIdx.y
inttype = blockIdx.z - 1
if i ≤ total then
  l = lpq_array(i, 1)
  p = lpq_array(i, 2)
  q = lpq_array(i, 3)
  fctr = 1.0
  if (p ≠ q) and (inttype == 0) then
    fctr = 2.0
  else if (p == q) and (inttype == 1) then
    fctr = 0.5
  end if
  loc = max(i, j) + ((max(i, j) - 1) * (max(i, j) - 2)/2) + min(i, j) - 1
  if (inttype == 0) then
    if i > j then
      k1 = 3
      k2 = 2
    else
      k1 = 2
      k2 = 3
    end if
    pos1 = locmx(l) + ((p - 1)2 + p - 1)/2 + q
    pos2 = locmx(l) + ((p - 1 + nbs(l))2 + p + nbs(l) - 1)/2 + q + nbs(l)
    pLL(i, j) = d(pos1) * fctr * x(1, loc) + d(pos2) * fctr * x(k1, loc)
    pSS(i, j) = d(pos1) * fctr * x(k2, loc) + d(pos2) * fctr * x(k8, loc)
  else
    if i > j then
      k1 = 4
      k2 = 6
    else
      k1 = 6
      k2 = 4
    end if
    pos1 = locmx(l) + ((p - 1 + nbs(l))2 + p - 1 + nbs(l))/2 + q
    pos2 = locmx(l) + ((q - 1 + nbs(l))2 + q - 1 + nbs(l))/2 + p
    pSL(i, j) = d(pos1) * fctr * x(7, loc) + d(pos2) * fctr * x(k1, loc)
    pLS(i, j) = d(pos1) * fctr * x(k2, loc) + d(pos2) * fctr * x(5, loc)
  end if
end if
```

Algorithm 7 Kernel 2 for MTSE

```
i = threadIdx.x + (blockIdx.x - 1) * blockDim.x * 2
inttype = blockIdx.z - 1
if inttype == 0 then
  if i ≤ total then
    s_tile(threadIdx.x) = pLL(i, blockIdx.y)
  else
    s_tile(threadIdx.x) = 0.0
  end if
  if i + blockDim.x ≤ total then
    s_tile(threadIdx.x + blockDim.x) = pLL(i + blockDim.x, blockIdx.y)
  else
    s_tile(threadIdx.x + blockDim.x) = 0.0
  end if
else
  Perform a similar instruction on the other pXY or qXY matrices depending on inttype
end if
for j =  $\log_2(\text{blockDim.x})$ , j ≥ 0, j = j - 1 do
  call syncthreads
  if threadIdx.x ≤  $2^j$  then
    s_tile(threadIdx.x) = s_tile(threadIdx.x) + s_tile(threadIdx.x +  $2^j$ )
  end if
end for
if threadIdx.x == 1 then
  l = lpq_array(blockIdx.y, 1)
  p = lpq_array(blockIdx.y, 2)
  q = lpq_array(blockIdx.y, 3)
  if (inttype == 0) then
    pos = locmx(l) + ((p - 1)2 + p - 1)/2 + q
    istat = atomicadd(pmx(pos), s_tile(threadIdx.x))
  else
    Perform a similar instruction on the other pXY or qXY matrices depending on inttype
  end if
end if
```

shared memory for summing up the elements of each row of the pXY matrices.

The grid will need to have $\frac{total-1}{blockDim.x} \times \frac{total-1}{blockDim.y} \times 2$ dimensions for Algorithm 6 and $\frac{total-1}{blockDim.x} \times \frac{total-1}{blockDim.y} \times 8$ for Algorithm 7. Algorithm 6 is fairly straight forward, but Algorithm 7 could use a little more explanation. It is a “parallel reduction” algorithm. At the start, each thread loads an element of the relevant pXY array into the shared variable s_tile and also loads the element that is $blockDim.x$ elements ahead of the original element. Then, in a loop starting from $j = \log_2(blockDim.x)$ and descending to $j = 0$ threads with a $threadIdx.x$ value of less than 2^j add the value of $s_tile(threadIdx.x + 2^j)$ to $s_tile(threadIdx.x)$. The key feature is that the amount of elements that need to be added up is reduced by half every loop. Because it is possible that the total number of elements of pXY needed to sum up for the relevant pmx element could be larger than one block can handle, atomic operations are used to force the final addition to occur serially.

3.3.2.4 STSE Algorithm

The STSE algorithm tries a different approach to this problem than the MTSE algorithm. Instead of trying to spread out the amount of work to do as much as possible, it tries to reduce the amount of global memory reads by making each thread do as much work as it can. This results in fewer blocks, but each block needs to run longer. Giving only the calculations for pmx and when $inttype = 0$, Algorithm 8 shows how this is done.

All variables are the same as in Algorithms 6 and 7, but $inttype$ can only be 0 or 1 (the meanings are the same as in Algorithm 6). As for the new variables, s_d is a chunk of d that has been loaded into the shared memory. min_k and max_k are used to calculate the sections of x that current chunk of d in shared memory applies to and $loop$ is used to adjust the value of loc if the needed two-electron integral is on the bottom triangle of a \mathbf{X} matrix. $pm1$ and $pm2$ are accumulator variables that track the sum of an element of pmx as d is looped through.

Algorithm 8 has two nested loops. The outer one loops over elements of d , loading chunks of it into the shared memory as it goes. The inner one loops over elements of x and performs the multiplications of d and x elements, keeping a running tally of the sum of these multiplications. Something that is not shown in the pseudocode is checking if l and m have open shells. If both of them do, then $qm1$ and $qm2$ will be calculated as well. If only one or neither of them do, then this calculation is skipped. Because most of the shells will be closed, this allows for many calculations to be skipped. Once the inner loop has finished, the next section of d is loaded into the shared memory and then the multiplication of x and s_d continues. This process goes on until all elements of a row of \mathbf{X} have been used.

Algorithm 8 STSE Algorithm

```
i = threadIdx.x + (blockIdx.x - 1) * blockDim.x
inttype = blockIdx.z - 1
if i ≤ total then
  l = lpq_array(i, 1)
  p = lpq_array(i, 2)
  q = lpq_array(i, 3)
  fctr = 1.0
  loop = 0
  for j = 1, j ≤ (total - 1)/blockDim.x + 1, j = j + 1 do
    mink = 1 + (j - 1) * blockDim.x
    maxk = min(j * blockDim.x, total)
    if threadIdx.x + (j - 1) * blockDim.x ≤ total then
      m = lpq_array(threadIdx.x + (j - 1) * blockDim.x, 1)
      r = lpq_array(threadIdx.x + (j - 1) * blockDim.x, 2)
      s = lpq_array(threadIdx.x + (j - 1) * blockDim.x, 3)
      if (r ≠ s) and (inttype == 0) then
        fctr = 2.0
      end if
      s_d(threadIdx.x, 1) = fctr * d(locmx(m) + ((r - 1)2 + r - 1)/2 + s - 1)
      s_d(threadIdx.x, 2) = fctr * d(locmx(m) + ((r - 1 + nbs(m))2 + r + nbs(m) - 1)/2 +
        s + nbs(m))
    end if
    fctr = 1.0
    call syncthreads
    for k = mink, k ≤ maxk, k = k + 1 do
      if (i ≥ k) and (inttype == 0) then
        loc = ((i - 1)2 + i - 1)/2 + k
        k1 = 3
        k2 = 2
      else if (i < k) and (inttype == 0) then
        loop = loop + k - 1
        loc = (i2 + i)/2 + loop
        k1 = 2
        k2 = 3
      end if
      pm1 = pm1 + s_d(1 + k - mink, 1) * x(1, loc) + s_d(1 + k - mink, 2) * x(k1, loc)
      pm2 = pm2 + s_d(1 + k - mink, 1) * x(k2, loc) + s_d(1 + k - mink, 2) * x(k8, loc)
    end for
    call syncthreads
  end for
  pmx(locmx(l) + ((p - 1)2 + p - 1)/2 + q) = pm1
  pmx(locmx(l) + ((p - 1 + nbs(l))2 + p + nbs(l) - 1)/2 + q + nbs(l)) = pm2
end if
```

3.4 Discussion

In this section, the results of GPU resource usage and profiling the various algorithms discussed previously will be presented. For profiling, the radon atom was chosen. Radon was selected because it contains many spinor orbitals and should be a good test of the ***formpq*** and ***eint2*** kernels on large atomic systems. Here, we only care about how long a calculation will take, not how accurate it is. Therefore, basis sets use the arbitrarily chosen wtbs parameters of $\alpha = 6.87 \times 10^{-2}$, $\beta = 1.83$, $\delta = 3.97$, and $\gamma = 0.78$. Additionally, due to the spherical symmetry of an atom exploited by CUDAPropheT, the growth of a calculation differs depending on how the basis functions are distributed among the spinor symmetries. Therefore, results are reported in terms of the number of two-electron integrals instead of number of basis functions as is typically done.

3.4.1 Resource Usage

In the case of the work in this Thesis, the resource bottleneck for my kernels was almost always the number of registers used per thread. 63 32-bit registers provide only 252 bytes to work with, which when using double precision 8-byte numbers is hardly anything at all. A summary of the block sizes and resource usage of all kernels is given in Table 3.1.

Table 3.1: *Kernel Resource Usage*

Kernel	Block Size	Registers	Shared Memory (bytes)
<i>eint1</i>	(512, 1, 1)	63	45056
<i>eint2</i>	(512, 1, 1)	63	45088
<i>vec2matrix</i>	(512, 1, 1)	17	0
<i>formd</i>	(512, 1, 1)	33	28672
<i>binary_search</i>	(512, 1, 1)	30	60
<i>convd</i>	(512, 1, 1)	25	0
<i>formpq_alg1</i>	(512, 1, 1)	46	32768
<i>formpq_alg2</i>	(512, 1, 1)	56	39388
<i>forme</i>	(512, 1, 1)	39	4152
<i>formf</i>	(512, 1, 1)	18	0
<i>xtrpf</i>	(512, 1, 1)	12	0
<i>formg</i>	(32, 16, 1)	55	12288
<i>swapcol</i>	(32, 16, 1)	10	0

replace all instances of ***formpq_alg1*** with whatever acronym you decide on.

Of these kernels, ***eint2*** and the ***formpq_alg*** kernels take up the bulk of the time needed in a typical calculation. This is in part due to the exponential growth in the number of two-electron integrals, but it is also caused by the amount of register spilling that each kernel will have. As stated in section 3.1.2 when the amount of registers a thread needs exceeds

the amount that is available, it will begin to store the data in the global memory. As global memory is the slowest available memory type, this is very undesirable. To counteract this, many variables that would otherwise be better stored in register memory were moved into the shared memory. But this is also undesirable because it means that there will be less shared memory available for storing data that needs to be read frequently from global memory. This in turn leads to more frequent reads from the global memory. Solving this balancing act required many iterations of testing, moving variables around, compiling, and retesting. Eventually, I arrived at the configurations shown above. Note how each kernel uses a block containing 512 threads. This number was chosen because it is the largest possible block size that can run when the required number of register per thread exceeds 32. 512 is also a good block size because in instances where a kernel is not limited by register or shared memory usage, it allows for the maximum number of threads per SM (1536) to be run.

3.4.2 Two-Electron Integral performance

Table 3.2 and Figure 3.9 show the results of profiling the *eint2* kernel against its CPU counterpart.

Table 3.2: *eint2* Kernel Performance. Times are in ms.

Two-Electron Integrals ^a	CPU Time	<i>eint2</i>	<i>binary_search</i>	Total GPU time	Speedup ^b	Speedup ^c
74305	59	4.68	0.18	4.86	12.81	12.32
353220	320	18.05	0.71	18.77	17.72	17.04
1081185	980	52.51	2.21	54.72	18.66	17.90
2588950	2360	118.20	5.36	123.56	19.96	19.09
5299140	4910	237.96	11.07	249.03	20.63	19.71
9726255	8880	437.88	20.81	458.69	20.27	19.35
16476670	15140	730.24	35.96	766.20	20.73	19.75
26248635	24180	1159.98	57.99	1217.97	20.84	19.85

^a For only the upper triangle of one of the 8 **X** matrices.

^b Speedup comparing just the time of *eint2* to the CPU time.

^c Speedup comparing the time of *eint2* and *binary_search* to the CPU time.

[include the hardware info](#)

As can be seen, the GPU calculations outperform the CPU calculations with an average speedup factor of over 19. For calculations with fewer number of integrals to compute the speedup is not as dramatic. This is most likely due to the overhead involved in just starting a GPU kernel. Table 3.2 shows two different speedups: one for just calculation of the integrals, and one that shows the speedup if the time needed for *binary_search* is included. The time needed to evaluate indices does not have much of an impact on the total performance on

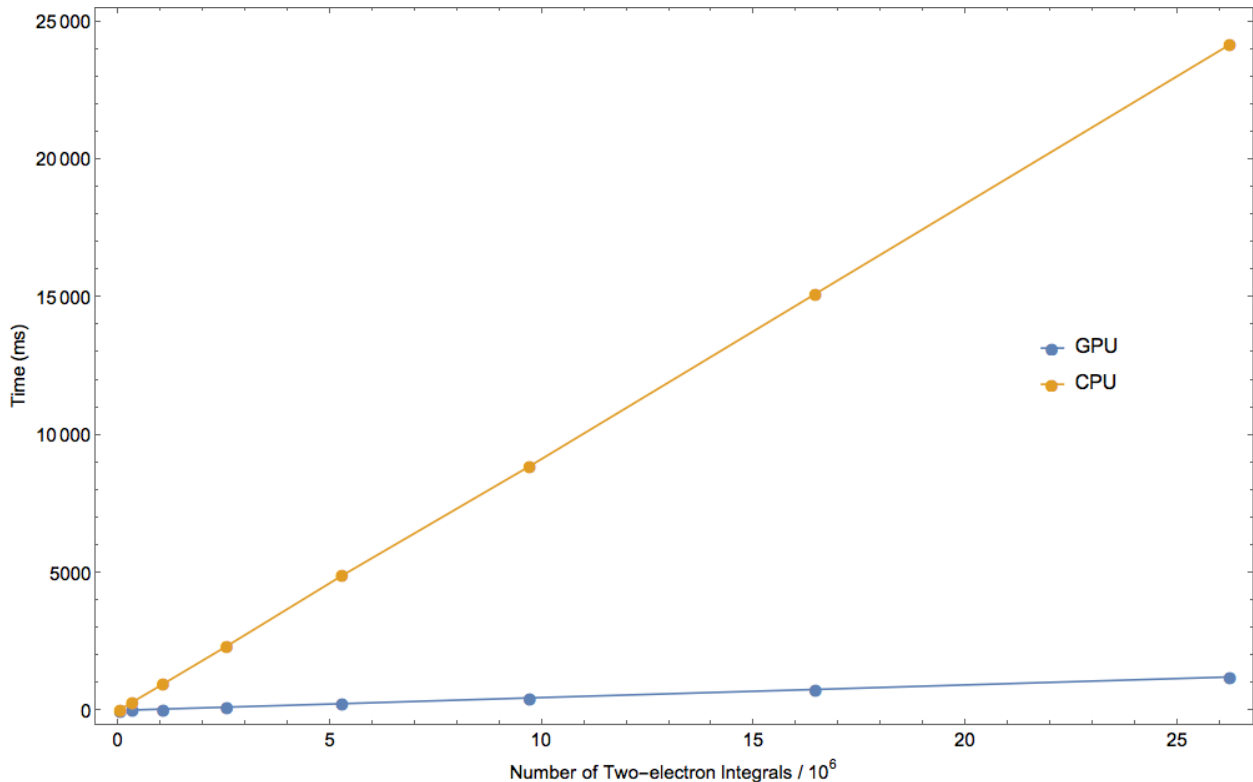


Figure 3.9: *Profiling of Two-Electron Integral Calculations.* This figure shows the results of profiling the calculation of growing amounts of two-electron integrals. The blue line shows the time needed for GPU calculations and the yellow line shows time needed for CPU calculations. The time for the GPU calculations is the combined time of the **eint2** and **binary_search** kernels.

the total calculation time which proves this was a good way to generate the *lmpqrs* indices. Additionally, because CUDAPropheT is a basis set optimizer, **binary_search** will need to be called only once, whereas **eint2** may need to be called many hundreds of times. Therefore, for an optimization calculation (which the original program is incapable of doing) the total speedup will limit to the just the integral calculations.

3.4.3 P and Q Matrix Formation Performance

As explained in section 3.3.2.1, there were two different attempts to speedup the formation of **P** and **Q**. The results of these attempts are shown in Tables 3.3 and 3.4 and also in Figure 3.10.

As can be seen, the attempt at **formpq_alg1** was a complete failure and it isn't even faster than the CPU calculations. **formpq_alg2** was much better, but only just. It achieves little more than a two-fold speedup at best.

There are several reasons this could be the case. The first due to the poor global memory

Table 3.3: *formpq_alg1* Kernel Profiling Results. Times are in ms.

Two-Electron Integrals	Time per Iteration CPU	Time per Iteration GPU	Speedup
74305	1.90	6.06	0.31
353220	5.78	14.25	0.40
1081185	18.42	27.81	0.66
2588950	39.47	53.95	0.73
5299140	81.05	96.13	0.84
9726255	152.50	173.64	0.87
16476670	266.53	291.79	0.91
26248635	426.25	459.38	0.92

include the hardware info

Table 3.4: *formpq_alg2* Kernel Profiling Results. Times are in ms.

Two-Electron Integrals	Time per Iteration CPU	Time per Iteration GPU	Speedup
74305	1.90	2.94	0.64
353220	5.78	7.56	0.76
1081185	18.42	15.34	1.20
2588950	39.47	26.32	1.49
5299140	81.05	39.45	2.05
9726255	152.50	88.78	1.71
16476670	266.53	135.64	1.96
26248635	426.25	193.99	2.19

include the hardware info

access that reading x has. x is stored as a vector. This helps greatly with memory cost and integral calculation, but makes it difficult to retrieve it as if it were a matrix. There is the cost of calculating where the necessary integral is, and then the cost of the very uncoalesced reads. Calculating the integrals as a vector, and then copying them into a matrix afterwards could help with this, but the large amount of integrals needed for large calculations makes this impractical on current hardware. How much of an effect inefficient access of global memory has on a calculation does have a limit however which is most likely why the speedup sees an improvement as the calculations get larger. A second reason that the speedup is limited is caused by the occupancy that the kernels can achieve on the GPU. A block size of 512 means that there is a maximum number of 3 blocks running per SM at a time. But these kernels are both limited by the register usage to just running one block per SM at a time. This is also most likely why *formpq_alg1* performs so poorly compared *formpq_alg2*. *formpq_alg1* was designed to spread out the amount of work to as many threads as possible. This results in a lot of blocks to be run. But because the amount of blocks that can run at a time is so limited there is no chance to see an improvement. But in the case of *formpq_alg2*, there

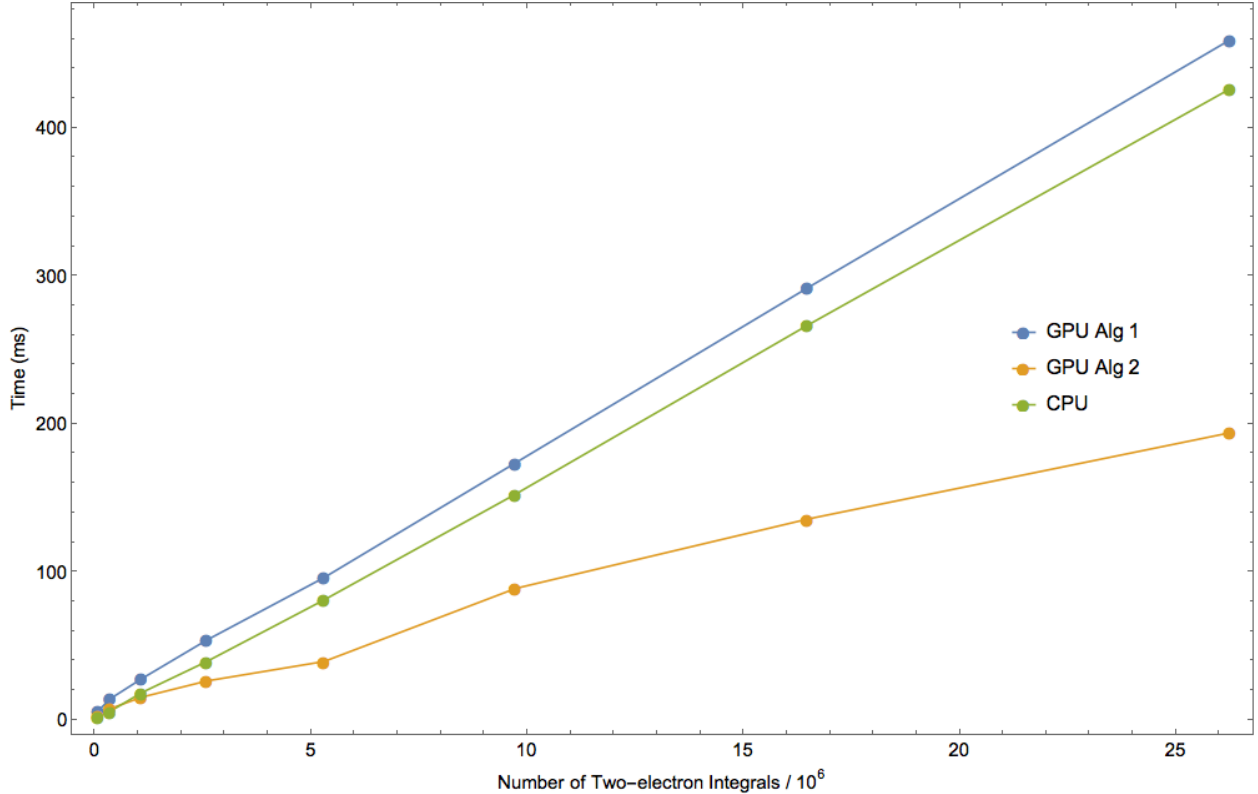


Figure 3.10: Profiling of the **formpq** Kernels. This figure shows the results of profiling the formation of the **P** and **Q** matrices. The blue line shows the time needed for GPU calculations using the **formpq_alg1** algorithm, the yellow line shows time needed for GPU calculations using **formpq_alg2**, and the green line shows the time needed for the CPU. Because the GPU calculations converge in fewer SCF iterations than the CPU calculations, only the average time for one iteration is shown.

are a fewer number of blocks needing be be run, although each one needs to run for longer than those of **formpq_alg1**. Overall **formpq_alg1** comes out on top.

How long will this be true though? At the moment, the limitations of these algorithms are caused by the hardware they run on, not the efficiency of the algorithms themselves. If we imagine removing these limitations, we could consider which one will be better. I propose that **formpq_alg1** will eventually surpass **formpq_alg2** for very large systems for the following reasons: the time it takes for a single block of **formpq_alg1** to run is mostly governed by how well it can read the x vector. The effect of spreading out the elements has on memory bandwidth has a limit, once the elements are spread out enough, spreading them out further has no impact. Therefore there is a maximum amount of time it takes for a single block to run. The time for a single block of **formpq_alg1** on the other hand is controlled by both the efficiency of reading global memory *and* the amount of two-electron integrals it must read though. So there is no limit to how long a block can take. Therefore,

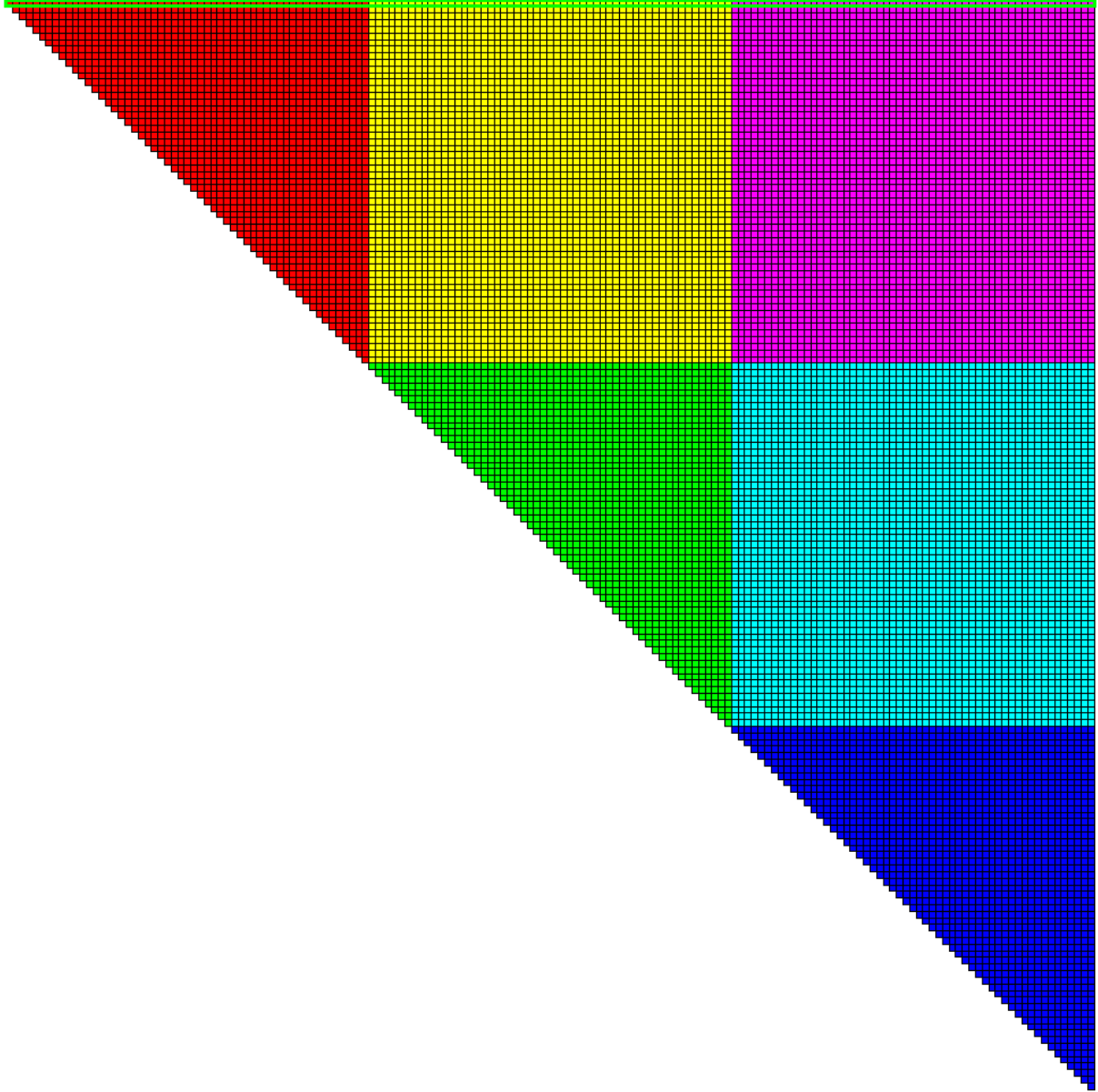


Figure 3.11: *Integrals Needed for the First Element of \mathbf{P} : Matrix Representation.* This figure shows the top triangle of the symmetric two-electron integral matrix in a matrix representation. The required integrals have been highlighted with a green boarder. The basis set used had $nsym = 7$ with $nbs(i) = (10, 10, 10)$. For clarity, the boxes have been coloured based on the spinor symmetries that they contain. *talk about this in text*

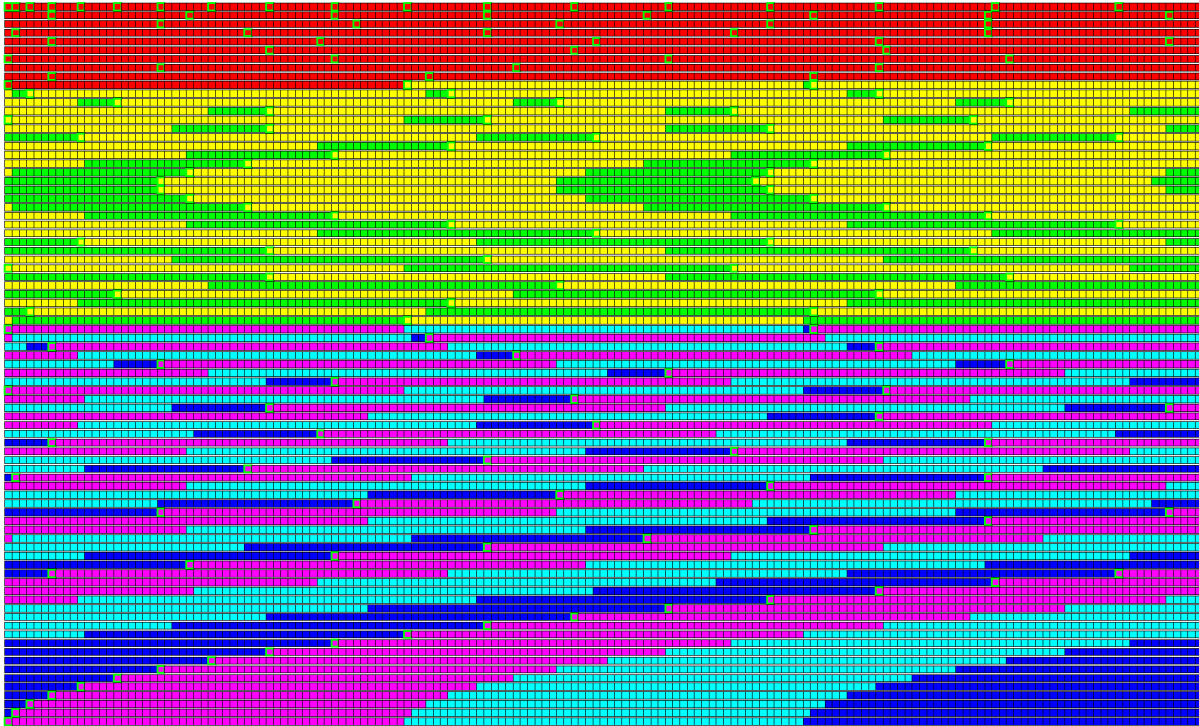


Figure 3.12: *Integrals Needed for the First Element of \mathbf{P} : Vector Representation.* This figure shows the top triangle of the symmetric two-electron integral matrix in a vector representation. The required integrals have been highlighted with a green boarder. The spreading out of the elements means that reads to them can not be done effectively. The basis set used had $nsym = 7$ with $nbs(i) = (10, 10, 10)$. For clarity, the boxes have been coloured in the same way as Figure 3.12. *talk about this in text*

I predict that *formpq_alg1* should be faster as the number of integrals becomes very large.

3.4.4 Total Speedup

There was some difficulty in determining the total speedup of a calculation. The reason why can be seen in Tables 3.5 and 3.6.

Table 3.5: *Total Speedup of Alg1. The results for a total calculation are shown. All times are in s.*

Two-Electron Integrals	SCF Iterations CPU	Total CPU Time	SCF Iteractions GPU	Total GPU Time	Speedup
74305	21	0.12	21	0.84	0.14
353220	19	0.49	19	1.19	0.41
1081185	19	1.44	19	1.75	0.82
2588950	19	3.38	19	2.58	1.30
5299140	19	6.78	19	3.80	1.78
9726255	24	13.29	19	5.77	2.30
16476670	26	23.23	19	8.60	2.69
26248635	24	35.63	19	12.5	2.82

Table 3.6: *Total Speedup of Alg2. The results for a total calculation are shown. All times are in s.*

Two-Electron Integrals	SCF Iterations CPU	Total CPU Time	SCF Iteractions GPU	Total GPU Time	Speedup
74305	21	0.12	21	0.78	0.15
353220	19	0.49	19	1.07	0.46
1081185	19	1.44	19	1.51	0.95
2588950	19	3.38	19	2.08	1.62
5299140	19	6.78	19	2.72	2.49
9726255	24	13.29	19	4.14	3.20
16476670	26	23.23	19	5.57	4.16
26248635	24	35.63	19	7.47	4.76

As can be seen, the number of SCF iterations differs when comparing the GPU calculations to the CPU even though the convergence criteria was exactly the same. This is somewhat puzzling but I offer the following explanation. First, when writing the SCF routines, it became apparent that rewriting the linear algebra into CUDA would be rather pointless when I could just use the cuSOLVER and cuBLAS libraries and achieve the same result. As such, the SCF is completed using entirely different algorithms. Second, it is important to realize that a real number can only be *approximated* using double precision variables. This introduces numerical instability. For instance, assume that we have a computer that

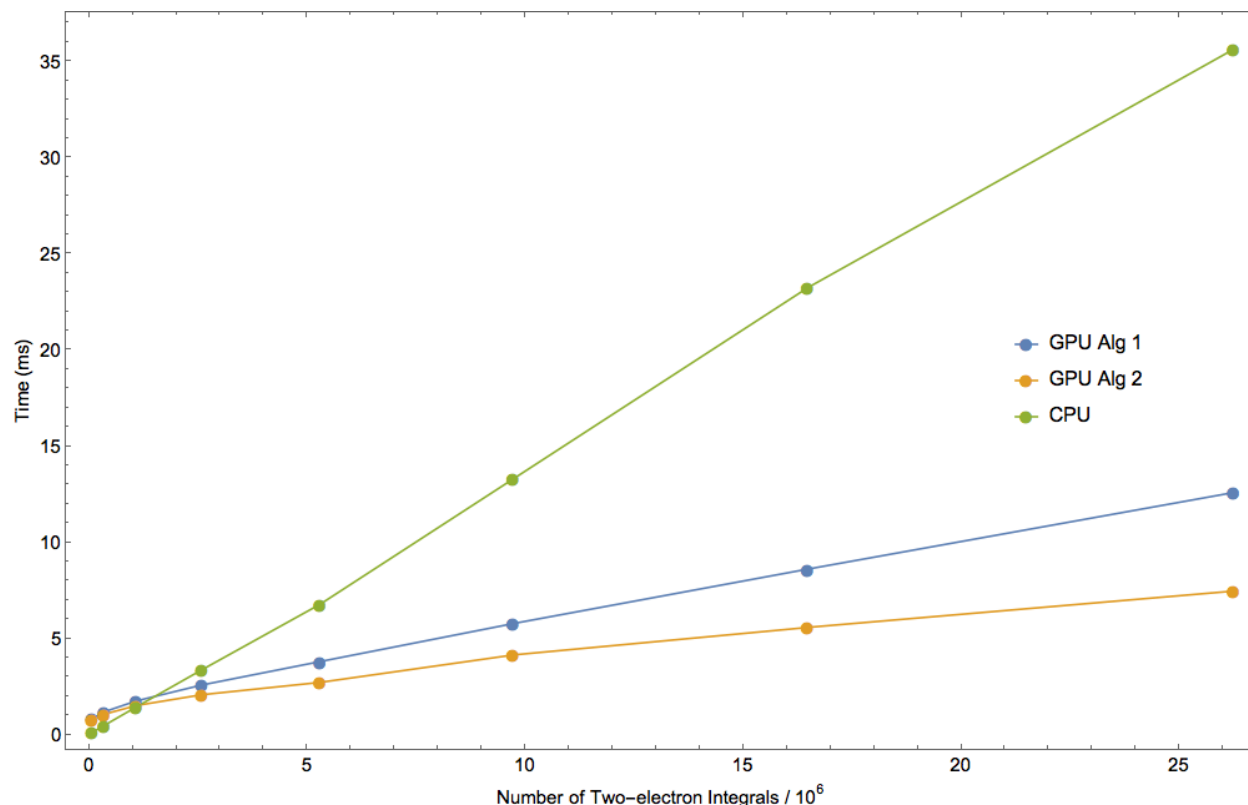


Figure 3.13: *Time need to complete a calculation.*

could only hold the first four digits of a number (leading zeros don't count). Then we take the number 1.000 and we want to add 0.0001000 to it 10000 times. The result of the first addition should be 1.0001, but because our computer can only hold four digits, the number is truncated to 1.000. The final result of this computation would not be 2.000 as expected but 1.000. During an SCF calculation, there are millions of numbers being multiplied together and added up, many of which may be close to, but not quite 0. This allows for lots of error to creep into a calculation just because of the limitations of the hardware. The fact that two different algorithms are being used on the GPU and CPU, and that they might not be handling numerical stability in the same way is what is most likely causing this to occur. It should also be pointed out that the final energy was not exactly the same from GPU to CPU either. Typically, the last 4 of 16 significant digits would be different. Because the rest are the same however, we still have chemical accuracy and I argue that CUDAProphets results are therefore valid.

3.5 Input Description

The program can be executed on Unix-like systems in the following way

```
1 | $ path_to_executable input_file > output_file
```

The input file must end in “.inp” or an error will be given. Redirection of stdout to an output file is optional, but is recommended to save the results of a calculation. The input file is read using the namelist functionality of Fortran. A description of what must appear on each line of the input file is given below. Sample input files are also given in [Appendix whatever](#).

1. A title of no more than 200 characters.

2. \$contrl

jobtype		The type of calculation to be performed.
=	'energy'	Will do a single point energy calculation.
=	'bsopt'	Will optimize the basis set. Can only be used if bastype equals 'wtbs'.
c		The speed of light. If not given, the default is set to 137.03599976 au.

3. \$nuc

znuc		The charge of the nucleus.
nucmdl		The nuclear model to use.
=	1	Point nucleus (default).
=	2	Finite sphere (not yet supported).
=	3	Gaussian.
rnuc		The radius of the finite sphere nucleus.
alpha		The exponent for the gaussian nucleus.
		Defaults are given in litdata.f90.

4. \$bas

nsym		The number of symmetries to be used.
bastype		The type of basis set given.
=	'wtbs'	Use a wtbs.
=	'rdin'	Read in the basis set from the input file.
ngroup		The number of different groups to use in the wtbs scheme (default 1).

The next line will depend on the value of bastype. If bastype equals 'rdin' then the following lines must be the number of functions for the S+ symmetry, followed by the exponents to use, each on a new line. The pattern repeats for each new symmetry. See the sample input files for further clarification. Otherwise, if bastype equals 'wtbs' the \$wtbs group is read next.

5. \$wtbs has to be given if bastype='wtbs'.

wtbspara	The α , β , δ , and γ wtbs parameters. If there is more than one group, the order would be $\alpha_1, \beta_1, \delta_1, \gamma_1, \alpha_2, \beta_2, \delta_2, \gamma_2$ and so on.
nbs	The number of functions used in each symmetry.
start	Where in the ζ pool each symmetry starts taking exponents from (default=1,1,1,1,1,1).
groups	What group each symmetry belongs to (default=1,1,1,1,1,1).

The next line depends on what the jobtype was set to. If jobtype equals 'energy' \$newuoa is skipped and \$seconfig will be read next. If jobtype equals 'bsopt', then the \$newuoa group will be needed.

6. \$newuoa has to be given if jobtype='bsopt'. Refer to the newuoa [citation needed](#) documentation for detailed information if needed.

rhobeg	The initial value of the trust region used by newuoa (default=0.1).
rhoend	The final value of the trust region used by newuoa. Must be smaller than rhobeg (default= 1.0×10^{-4}).
iprint	The print level for newuoa.
= 0	No printing from newuoa (default).
= 1	Print only when newuoa has finished.
= 2	Print only when the trust region has decreased by an order of magnitude.
= 3	Print every iteration of newuoa.
maxfun	The maximum number of calls to calfun newuoa will make before terminating (default=500).

7. \$seconfig

nclose	The number of closed spinors for each symmetry.
nopen	The number of open spinors for each symmetry. There is a limit to one open orbital per symmetry.
freeel	The number of electrons available in the open spinors.
autogen	Automatically generates all possible combinations of spinor occupancies (default=.false.).
nconf	The number of configurations to be read in (needed if autogen is false).

If autogen is false, then the next nconf lines will be the spinor occupancies. They will be given as real numbers with one configuration per line.

8. \$scf			
maxitr			The maximum number of SCF iterations (default=50).
ixtrp			The method of extrapolation.
	=	0	No extrapolation (default).
	=	1	Extrapolate the Fock matrix.
dfctr			Damping factor for Fock maxtrix (default=0.3).
thdll			Convergence limit for the large-large components of the density matrix (default= 1.0×10^{-5})
thdsl			Convergence limit for the small-large components of the density matrix (default= 1.0×10^{-7})
thdss			Convergence limit for the small-small components of the density matrix (default= 1.0×10^{-9})

Bibliography