Chapter 1

Basis Sets

1.1 Methods

1.1.1 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). 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 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 .inp 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. 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!

1.2 Discussion

The optimized wtbs parameters and basis set sizes for elements 2 to 86 are shown in Table 1.1.

Table 1.1: Basis sets optimized using rwtbs

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 1.1: (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)	

Table 1.1: (continued)

Element	Term Symbol	α	β	δ	γ	s	p	d	f
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)
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 2

CUDAProphet

2.1 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 uint (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 (and is currently hardcoded for :( ) the Tesla C2050, I will be referring to its specs for examples when needed. I will begin will 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.

2.1.1 Threads, Blocks, and Grids

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 need 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 1 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 **??**), but no more than 8 block 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*.

2.1.2 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 FLOPs to compute, 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. On our card, 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 complier handles all of this. There are complier flags which can change the maximum amount of registers per thread allowed which will prompt the complier 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.

2.1.3 Occupancy and Performance

With the description of the various resources out of the way, now we can talk about how the 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 complier 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 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!

This is better put in the discussion section of this chapter.... 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 numbers is hardly anything at all.

2.2 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.

2.3 Alterations for CUDA

While almost all of the code from the original DFRATOM was modified in someway, 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.

2.3.1 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 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.

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
```

The first is the problem of warp divergence. Because the maximum values

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

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 is it 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 whatever. 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 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 for a one dimensional solution instead of the three dimensional one of Algorithm 2. 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. 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.

$$n'(j) = \frac{j^2 + j}{2} \tag{2.1}$$

$$y = n'(nbs(L-1)) + n'(P-1) + Q \tag{2.2}$$

$$x = n'(nbs(M-1)) + n'(R-1) + S \tag{2.3}$$

$$i = n'(x) + y \tag{2.4}$$

$$i_{max} = n'(\sum_{j=1}^{nsym} n'(nbs(j))) \tag{2.5}$$

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

Consider Figure whatever. It 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 two for symmetry two. In each element of this matrix, 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 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 following binary search algorithm.

Where 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 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 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 other than a single atom. For instance in C1, 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 $\mathbf{sum}(s\_nprime(1 : L - 1)$ term from $lownum$.

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

Algorithm 3 Binary Search for x and y

if $theadIdx.x \leq nsym$ **then**
  $s\_nsym = nsym$
  $s\_nbs(threadIdx.x) = nbs(threadIdx.x)$
  $s\_nprime(threadIdx.x) = n'(s\_nbs(threadIdx.x))$
end if
call **syncthreads**
$i = threadIdx.x + (blockIdx.x - 1) * blockDim.x$
if $i \leq i_{max}$ **then**
  $low = 1$
  $high = \mathbf{sum}(s\_nprime(1 : s\_nsym))$
  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

Algorithm 4 Binary Search for L

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

Algorithm 5 Binary Search for P and Q

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

2.3.2 P Q Matrix Formation and SCF

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

2.4 Input Description

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

```
$ 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 at the end of this document.

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 what bastype was set to. 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 α_1, β_1, δ_1, γ_1, α_2, β_2, δ_2, γ_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=1,1,1,1,1,1,1).
groups	What group each symmetry belongs to (default=1,1,1,1,1,1,1).

The next line depends on what the jobtype was set to. If jobtype equals 'energy' $newuoa is skipped and $econfig 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 documentation for more 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. $econfig

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})