Optimising the DBCSR GPU Implementation

Jay Chetty

August 18, 2011

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2011
Abstract

The DBCSR library solves the sparse matrix multiplication required to perform atomistic simulations using the CP2K software. The GPU implementation of DBCSR was targeted for optimisation, and having its scope increased to allow it to function with larger block sizes.

It was found that the main kernel could be sped up by 16% by augmenting the algorithm so multiple elements were assigned to each thread. By assigning each thread block its own local C matrix, the need for locks on the C matrix was removed. The cost of the required reduction step, however, outweighed the benefit of the lock removal. The Cublas *dgemm* function showed that it is a suitable candidate to handle block sizes too large for the original method to process.
# Contents

1 Introduction ................................................................. 1

2 Background ................................................................. 2
    2.1 Atomistic and Molecular Simulation .................................. 2
        2.1.1 Density Functional Theory .................................... 2
    2.2 DBCSR library .......................................................... 3
    2.3 Graphics Processing Units (GPUs) .................................... 4
        2.3.1 General Purpose Computing on GPU (GPGPU) .................. 5
        2.3.2 CUDA .............................................................. 6
        2.3.3 NVIDIA Tesla C2050 .............................................. 8
    2.4 Overview of Initial GPU Implementation ............................... 9
        2.4.1 Kernel: Reading in the Data .................................. 10
        2.4.2 Kernel: Computing C value and writing to global C Matrix .... 11
        2.4.3 Performance of Initial GPU implementation .................. 13

3 Assigning Multiple Elements to Each Thread ............................ 15
    3.1 Implementation ........................................................ 16
        3.1.1 Reading in the A and B Matrices ............................... 16
        3.1.2 Computing Local C Values ..................................... 17
        3.1.3 Writing C to global memory ................................... 20
    3.2 Performance of Multiple Element Implementations .................. 21

4 Multiple C Matrices ..................................................... 22
    4.1 Multiple C Matrices Implementation ................................ 23
    4.2 Multiple C Matrices Performance .................................... 27

5 Using the Cublas library .................................................. 29
    5.1 Implementing Dgemm ................................................ 30
    5.2 Dgemm Performance .................................................. 31

6 Further Work .................................................................... 32

7 Conclusion ....................................................................... 33

A Test Input File and Kernel Timing Output ................................. 34
List of Tables

2.1 NVIDIA Tesla C2050[14] ........................................ 8
2.2 Double Precision Peak FLOP Rates[14, 16] .................. 8
4.1 Kernel Time with Locks Removed ................................. 22
List of Figures

2.1 GPU/CPU cores [9] ......................................................... 4
2.2 GPU/CPU peak performance [10] .................................... 4
2.3 Diagram of Fermi architecture: cores grouped into streaming multiprocessors
with associated resources[13] ............................................ 7
2.4 Schematic describing the GPU DBCSR implementation .................. 10
2.5 Allocation of column and row numbers .................................. 11
2.6 CPU only and GPU performance ....................................... 13
2.7 Performance of Cublas Implementation ............................... 14

3.1 Two Elements of C From Same Column Allocated to Each Thread ...... 15
3.2 Two Elements of C From Same Row Allocated to Each Thread ........... 15
3.3 Performance of Multiple Element Implementations ..................... 21

4.1 Multiple C Performance ................................................... 27

5.1 Performance of Cublas Implementation ............................... 31
Acknowledgements

Thank you to my supervisors Iain Bethune and Alan Gray for their guidance throughout the project. Also thank you to Urban Borstnik and Joost VandeVondele from the CP2K team, for giving me the opportunity to work with them in Zurich, and their time and effort in explaining the workings of the program.
Chapter 1

Introduction

CP2K is a computational chemistry package that provides a variety of methods to perform atomistic and molecular simulations [1]. One method provided is Quickstep which uses Density Functional Theory (DFT). Recently the CP2K team in Zurich implemented a version of Quickstep in which the key matrix multiplication kernels were ported to GPU. The report will look at methods of improving the performance of the GPU implementation, and widening its scope to allow it to function with larger block sizes.
Chapter 2

Background

2.1 Atomistic and Molecular Simulation

The increase in computer power over the last twenty years has grown the scope of computer simulations significantly and has put its scientific importance side by side with experimental and theoretical methods [2].

An example of the growing importance of atomistic modelling is shown in the testing of nuclear weapons. In 1995 the Advanced Simulation & Computing Program ASC was created in the United States, with its mission being to "support the U.S. Defense Programs shift in emphasis from test-based confidence to simulation-based confidence" [3]. The importance of these simulations is highlighted with them being described as "central to U.S. national security" [3].

One of the main benefits of simulation over experiment is the ability to precisely control the setting of the environment [4]. This can then allow for simulations to take place that would be expensive, or possibly infeasible to investigate experimentally. Modelling is however, "rarely a simulation of reality" [4]. It is therefore important that an appropriate model is used to provide a simulation with the required level of accuracy.

2.1.1 Density Functional Theory

Classical mechanics, based on Newton’s laws of motion, is often used in simulation. However, it is known to be only an approximation of more precise models, and only accurate enough for macroscopic objects. When dealing with the microscopic, more detailed models are required.

The most accurate model known for how particles (or any object) behaves is quantum mechanics. Experimental verification of quantum mechanics has tested its predictions to a very high level of accuracy [5]. However quantum mechanics is computationally infeasible for anything apart from the most trivial cases [6]. Density Functional Theory
uses a compact representation of the wave function, allowing for efficient algorithms [7]. This model strikes a compromise between the computability of classical mechanics and the accuracy of quantum mechanics, making it a useful model for atomistic and molecular simulations.

2.2 DBCSR library

When running a simulation using Quickstep, multiplications of large, sparse, block structured matrices are required. A sparse matrix is a matrix populated primarily with zeros. For large simulations, or simulations of high accuracy, it is the solving of the matrix multiplications that dominates the run time. To multiply sparse matrices efficiently it is usually required to take advantage of the sparse structure of the matrix [8]. To process the multiplication of the sparse matrices the CP2K team have developed a library called the DBCSR library.

DBCSR stands for the distributed, block, compressed, sparse, row library. The DBCSR library works by parsing the large, sparse block structured matrix, (for example \(5120 \times 5120\), and composing a parameter stack describing small dense matrix multiples for example \((5 \times 5), (5 \times 13)\) that will cumulatively solve the large sparse matrix multiplication. This parameter stack can then be used as an instruction list to perform the multiplications. This operation of computing a stack to be solved, and then performing the calculation may be required to be repeated multiple times for a single matrix multiply.

There are various implementations of DBCSR library allowing the parameter stack to be computed on a variety of architectures. Recently the CP2K team have implemented a version where the parameter stack is processed by a Graphics Processing Unit.
2.3 Graphics Processing Units (GPUs)

Through its demand to compute pixel values at a fast rate, the computer gaming industry has driven the development of a co-processor - the Graphics Processing Unit (GPU). To take advantage of the inherently parallel nature of pixel computation, GPUs have evolved to become devices with many (hundreds) of cores, a level of magnitude higher than CPUs (see figure 2.1)[9]. To allow the GPU to have this large number of cores, the cores are stripped down of many of the features of CPUs, caches, controllers etc. This large number of cores, available through this removal of sophistication, has allowed GPU peak flop rates to grow at a far faster rate than CPUs (see figure 2.2).

![Figure 2.1: GPU/CPU cores][9]

![Figure 2.2: GPU/CPU peak performance][10]
2.3.1 General Purpose Computing on GPU (GPGPU)

The high FLOP rates of GPUs have attracted interest from outside the world of graphics processing, including HPC. However, there are many restrictions on the type of program that can be successfully accelerated by GPUs.

Typically a whole program is not ported to GPU, only small computationally intensive parts are. These ported sections to run on the device are called kernels. Following Amdahls law, in order to significantly reduce the run time of the program, the parts to be ported to the device will have to comprise a substantial proportion of the serial run time. In the case of the DBCSR library, the majority of the run time is spent in calculating the sparse matrix multiplications, thus meeting this requirement.

In order for the kernels to be suitable for acceleration by GPU they must be able to utilise the many cores of the GPU, and therefore must be able to be decomposed into many threads. Further, any communication required between threads will greatly hinder the performance of the program, thus for effective acceleration, the program should be, or be nearly, embarrassingly parallel. Matrix multiplication is suitable for this type of parallel processing as each value can be calculated independently.

GPUs have their own dedicated memory - they do not share memory with the CPU. Data is explicitly transferred from the CPU memory to dedicated GPU memory. There is an overhead involved in this transfer of memory, and if too prevalent, can greatly reduce the gains of the parallel acceleration. For $n \times n$ matrices, matrix multiplications require $O(n^3)$ operations, again meeting the requirement.

GPUs have a high global memory bandwidth compared to CPUs. To make best use of the bandwidth, data should be loaded for multiple threads in a single transaction, this is known as memory coalescing. To allow this to happen 16 consecutive threads must access data from the same section of memory, this will happen when consecutive threads thread access consecutive elements in memory. Without this, memory accesses will be done in serial, greatly hindering performance.

Even with these restrictions, there are many programs outside of graphics processing that are suitable to be ported to GPU. Traditionally, to port these programs to GPU, the program would have to be transformed to appear like a graphics application, for example using OpenGL [11]. However, manufacturers have since realised the potential market outside of graphics applications, and catered to it through developing GPGPU specific hardware and software SDKs.


2.3.2 CUDA

CUDA is the proprietary API to the NVIDIA architecture. It allows for NVIDIA GPUs to be programmed in the high level programming languages C, C++ and FORTRAN. Other APIs are available that are capable of utilising hardware from multiple manufacturers, such as OpenCL. The original GPU porting of DBCSR was created in CUDA, and so were all the implementations produced in this project.

Kernels

CUDA allows the CPU and GPU to be used according to the "co-processing" model, with the GPU acting as an accelerator [12]. This is done by defining kernels using the language extensions provided, and then calling these kernels from the host. These kernels can then run on the device, with multiple threads executing in parallel taking advantage of the many cores available in the device.

Functions are defined as kernels in CUDA using the __global__ deceleration. The following example describes a kernel that will perform the vector addition of two arrays, with each thread performing the additions of the elements indexed by its thread ID. (Note that the data must be explicitly transferred to and from the GPU using cudaMemcpy).

(DEVICE)

__global__ void add_arrays
    (float* a,
     float* b,
     float* c)
{
    int my_index = threadIdx.x;
    c[my_index] = a[my_index] + b[my_index];
}

When calling the kernel from the host it is required to state the number of threads desired. This is done by using the <<< ... >>> syntax. The following example shows calling the add_arrays kernel with N threads.

(HOST)

<<<1,N>>> add_arrays(a,b,c);
Thread Decomposition

The NVIDIA GPU architecture has its cores organised into streaming multiproces-
sors(SMs) (see figure 2.4). Each of these SMs contains a number of cores and resources
such as shared memory (discussed later). Threads are mapped to this architecture by
being grouped into thread blocks which are then automatically assigned to an SM. The
number of thread blocks is requested as shown below.

`<<<number_of_thread_blocks,number_of_threads_per_block>>>`

In the current DBCSR GPU implementation this grouping of threads is utilised, with
each small matrix multiplication being processed by a thread block.

![Diagram of Fermi architecture: cores grouped into streaming multiprocessors with associated resources](image)

Figure 2.3: Diagram of Fermi architecture: cores grouped into streaming multiprocessors with associated resources[13]
2.3.3 NVIDIA Tesla C2050

The target hardware in this project is the NVIDIA Tesla C2050 GPU, part of the second generation of CUDA devices, codenamed Fermi. The C2050 has a large number of cores (448) giving it its high peak flop rate (see table 2.1). Being designed for general purpose computing, the device also has a number of features that would not be found on a standard GPU.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CUDA Cores</td>
<td>448</td>
</tr>
<tr>
<td>Double Precision peak FLOP rate</td>
<td>515 GFlops</td>
</tr>
<tr>
<td>Single Precision peak FLOP rate</td>
<td>1.03 TFlops</td>
</tr>
<tr>
<td>Total Dedicated Memory</td>
<td>3GB GDDR5</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>144 GB/sec</td>
</tr>
</tbody>
</table>

Table 2.1: NVIDIA Tesla C2050[14]

Local Memory

The 448 cores of the C2050 are split into 14 streaming multiprocessors (SMs), each containing 32 cores. Each of the 14 streaming multiprocessors are supported by 48K local memory, allowing fast access to a moderate amount of data [15]. In the DBCSR GPU implementation, this is utilised to allow threads in the same thread-block fast access to their A and B matrix sections.

Double Precision Performance

In the previous Tesla range, double precision operations were performed by a dedicated unit in each SM [15]. The Fermi range supports full double precision floating point performance, allowing each core to perform a double precision floating multiply-add operation in two clock cycles[15]. This gives the Fermi cards a peak double precision FLOP rate more than $6 \times$ that of the previous range (see table 2.2). As Quickstep involves double precision operations, the improved performance of the Fermi range will be of great benefit.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Double Precision Peak FLOP Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tesla C1060</td>
<td>78 GFlops</td>
</tr>
<tr>
<td>Tesla C2050 (Fermi)</td>
<td>515 GFlops</td>
</tr>
</tbody>
</table>

Table 2.2: Double Precision Peak FLOP Rates[14, 16]
2.4  Overview of Initial GPU Implementation

As mentioned in the last section, the DBCSR library creates a parameter stack describing the many small dense matrix multiples that need to be processed in order to solve the large sparse matrix multiplication.

Each entry in the parameter stack is made up of a set of seven entries that describe a small matrix multiplication \((A \times B = C)\) for example this set could be:

\((12, 6, 8, 60, 80, 100, 6)\)

The first three numbers describe the size of the small matrix multiplication to be performed, here a \(12 \times 8\) A matrix multiplied by a \(8 \times 6\) B matrix into a \(12 \times 6\) C matrix. The \(12 \times 8\) section from the A block starts at element 60 of the A element array, the \(8 \times 6\) section of B matrix block starts at element 80 of the B matrix array, and the \(12 \times 6\) product block is stored at element 100 of the C data array. The final number \(7\) is of convenience, describing the C block being referred to. There is a 1 to 1 mapping between the 6th and 7th number [17].

In the GPU implementation a thread block is created for each small multiplication that needs to be performed (one for each line of the multiplication stack). In each local multiplication every element of the C block will be assigned a unique thread to compute its value. Therefore, the thread block size must be at least as big the largest C block, and is thus set to be the size of the largest possible block size of C, ensuring enough threads to process the multiplication. To implement this, when the kernel is called from the dc_do_stack_cu function it sets the number of threads in each thread block to the largest possible block size and then creates a thread block for each line of the parameter stack.

```c
//setting size of thread block to max size of C block
maxt = MAX(MAX(m_max*n_max, m_max*k_max), k_max*n_max);
...
/*running the kernal with a thread block
 * for each local multiplication*/
stack_mm_d <<< stack_size, maxt, shared_size >>>
```

With enough thread blocks to perform each multiplication on the stack, and each block of an adequate size, the kernel call is set up so each thread block can begin its multiplication. Each thread block must then read in the data it needs to perform the multiplication, compute its value of C, and finally write the value of C to the C matrix array in global memory, all of this is done in the kernel function stack_mm_d.
2.4.1 Kernel: Reading in the Data

Firstly, each thread is required to know what multiplication (which line of the parameter stack) they are processing. This is done by using the thread block number. Each thread of the block can then use their thread ID to assign themselves a C element.

```
// sp: which stack member this thread block is processing
sp = blockIdx.x;

// cr: C matrix element of this thread
cr = threadIdx.x;

syncthreads();
```

Each thread must then know the actual details of the multiplication. To implement this, the first seven threads of the block fetch the data from the assigned line in the parameter stack. This data is stored in shared memory to allow threads fast access. Before any thread can progress to loading in the actual data it must wait to ensure all the data has been loaded.

```
__shared__ int our_params[7];
...
if (cr < 7) {
    our_params[cr] = param_stack[cr+7*sp];
}
syncthreads();
```
Using this information the first \( m \times k \) threads then each collect an element of the A matrix and store it in the SM’s shared memory, the last \( k \times n \) threads then each collect an element of the B matrix and store in shared memory. The data needed for the multiplication is then available for all threads to use. (Note the A, and B matrices are stored in shared memory as multiple threads of the same thread block may require the same row of A, or column of B, while the C remains in global memory as each thread in a thread block will be writing to unique element of it). Before the threads go on to compute the values of C, they must wait to ensure all the data has been loaded.

```c
/* Setup shared memory. */
buff = (double *) cache;
...
if (cr < mk)
    buff[cr] = a_data[our_params[3]-1+cr];
if (cr >= blockDim.x - kn)
    buff[mk+cr-(blockDim.x-kn)] =
    b_data[our_params[4]-1+(cr-(blockDim.x-kn))];
syncthreads();
```

2.4.2 Kernel: Computing C value and writing to global C Matrix

Each thread, that is required, then assigns itself an element of the local C matrix to compute. It does this by dividing its thread id by the height of C \( (m) \), and setting its row position as the remainder and column as the quotient. This can be visualised as the threads using a column major distribution (see figure 2.5).

```c
r = cr % m; //allocating row
c = cr / m; //allocating column
```

\[
\begin{pmatrix}
t_0 & t_4 & t_8 & t_{12} \\
t_1 & t_5 & t_9 & t_{13} \\
t_2 & t_6 & t_{10} & t_{14} \\
t_3 & t_7 & t_{11} & t_{15}
\end{pmatrix}
\]

Figure 2.5: Allocation of column and row numbers
To calculate the value of an element of $C$, the thread will need to multiply each element of a row of $A$ by the corresponding element of a column of $B$.

$$ C_{ij} = \sum A_{ki} \times B_{kj} $$

```c
for (l = 0; l < k; l++) {
    myc = myc +
        buff[l*m+r] * //element of A
        buff[mk+c*k+l]; //element of B
}
```

Once each thread has computed its value it can go on to update the global $C$ matrix array. Note however, that threads from different thread blocks may be trying to update the same value of the $C$ matrix, a lock on the $C$ matrix is therefore required.

```c
/* Lock the C block. */
 syncthreads();
 if (cr == 0) {
     sp_one = sp + 1;
     c_id = our_params[6]-1;
     lock_owner = 0;
     while ((lock_owner != sp_one))
         lock_owner = atomicCAS(&(c_locks[c_id]),0,sp_one);
 }
```

The threads can then go on to write the elements to the $C$ matrix in global memory and release the lock.

```c
syncthreads();
 if (cr < mn) {
     c_data[our_params[5]-1+cr] += myc;
 }

/* Release the lock on the C block. */
 syncthreads();
 if (cr == 0)
     c_locks[c_id] = 0;
```
2.4.3 Performance of Initial GPU implementation

To measure the performance of the GPU implementation a test input file was used describing a $512 \times 512$ sparse matrix multiplication, with the block sizes set to $16 \times 16$ (see appendix). This results in the DBCSR library producing 29,583, $16 \times 16$ dense matrix multiples to be computed. The test case has the parameter stack set to a maximum of 10,000, hence the kernel will be called three times.

CP2K provides a 'Total Local Multiplication Time' that measures the time taken for all the local multiplications [17], for the test this is a single $512 \times 512$ sparse matrix multiplication. A CPU only version running in serial on an Intel Xeon 2.4GHz using the BLAS library takes 0.1479 seconds to solve the multiplication. This would be expected to achieve close to linear speed up for the test case, and so with 8 cores a CPU runtime of 0.01849 seconds could be expected. The current GPU implementation computes the multiplication in 0.0054 seconds. This is roughly $3.5 \times$ faster than the theoretical 8 core CPU only version (see figure 2.6).

![Figure 2.6: CPU only and GPU performance](image)
For more accurate timing of the kernel, CP2K has a specific kernel timing option; this times each individual kernel run, which solves the multiplications of the parameter stack. This method synchronises the host and the GPU both before and after the kernel execution [17]. The overall runtime performance will therefore be affected and so this method should only be used when specifically timing the kernel. The test case makes three calls to the kernel (the first two with stacks of 9999 matrix multiples, the last with 9587). On viewing the kernel timings the first kernel is constantly slower than subsequent (full) kernels (see appendix). The timing of the second kernel was therefore used to benchmark performance.

In the initial GPU implementation the timing of the second kernel takes 0.00172 seconds, (solving 9999 matrix multiples, which involves 81911808 double precision floating point operations; achieving a flop rate of 47.6 GFLOPS). This is 9.2% of the 515 GFLOP double precision peak flop rate of the NVIDIA Tesla C2050, indicating that increased performance may well be possible.

![Figure 2.7: Performance of Cublas Implementation](image)

To investigate how the kernel could be improved, the timing of the kernel was repeated with parts removed. The timings indicate that reading in the parameters and data is relatively cheap while the calculation and especially writing to the global C matrix, are more expensive (see figure 2.7).
Chapter 3

Assigning Multiple Elements to Each Thread

In the initial version, each thread is assigned just one element of the C matrix to update (see figure 2.5). This requires a thread to make a single transfer to global memory for each element of C. As shown earlier, the timings display that it is the transfer that is the most expensive part of the kernel.

By assigning multiple elements of C to each thread, the number of writes to global memory will be reduced, potentially reducing the overhead of the data transfer. However, each thread will now be required to compute the value of multiple elements of C, hence increasing the time of computation. If the benefit of reducing the transfer times outweighed the added computation time, this method could increase the speed at which the kernel would operate and thus improve the total calculation time.
To investigate, the program was implemented using various number of elements per thread (2, 4, 8, 16). Two different versions of the program were created, one variation allocated to each thread elements from the same column of C (see figure 3.1), and the other allocated elements from the same row (see figure 3.2). The performance of the two versions was compared.

Note that these test implementations are only designed for cases where the block size is a multiple of the number of elements allocated to each thread.

The bulk of the work in this section was produced under the guidance of Urban Borstnik of the CP2K team.

### 3.1 Implementation

To allow each thread to write multiple elements of C it is first required to reduce the number of threads set by the host, this is done by dividing the size of each thread block by the required factor. This is done by entering the following line in the bridge `dc_do_stack_cu` function.

```c
maxt = ceil(maxt/elements_per_thread);
```

It is then required to change the kernel so that with the reduced number of threads the data is still read in correctly for the A and B matrices, each thread correctly computes multiple values of C, and the calculated C values are written to the correct space in global memory.

#### 3.1.1 Reading in the A and B Matrices

In the initial program the A matrix was read into shared memory by the first $m \times k$ threads each reading in a single element of the A matrix. And the B matrix was read into shared memory by the final $k \times n$ threads reading in each element of the B matrix. With a reduced number of threads, however, this method is not possible and each thread has to read in multiple elements of each matrix. To implement this, first the number of elements allocated to each thread is set.

```c
//Set the number of elements per thread
const int numberofelements = 2;
```

In the single element version, the C reference point of each thread, is simply its thread number.

```c
cr = threadIdx.x;
```
In the multiple element version the C reference point of each thread needs to be multiplied by the number of elements per thread.

```c
tn = threadIdx.x;
cr = tn*numberofelements; // compute C matrix reference
```

Now each thread can read in multiple elements of A and B using its C reference to find the starting point for its section.

```c
if (cr < mk){ //If C reference in range load in A elements
    for(i=0;i<numberofelements;i++)
        buff[cr+i] = a_data[our_params[3]-1+cr+i];
}

if (cr < kn){ //If C reference in range load in B elements
    for(i=0;i<numberofelements;i++)
        buff[mk+cr+i] = b_data[our_params[4]-1+(cr+i)];
}
```

### 3.1.2 Computing Local C Values

When computing the values for the C matrices each thread will require an array to store the values of C it will calculate.

```c
double myc[numberofelements];
```

To calculate the value of an element of C, the thread will need to multiply each element of a row of A by the corresponding element of a column of B. In the original single element program this is done as follows:

```c
if (cr < mn) {
    r = cr % m;
c = cr / m;
    myc = 0.0l;

    for (l = 0; l < k; l++) {
        myc = myc +
            buff[ l*m+r ] *
            buff[mk+c*k+l];
    }
}
```
Here, each thread finds its row position \( r \) and column \( C \) by dividing its C reference \( cr \) by the height of C \( (m) \), and finding setting its row position as remainder and column as the quotient.

Each thread can then find the corresponding A and B matrix data from its local memory, and perform the multiplications to find the result for the C element.

For each thread calculating the values for multiple elements of C, the implementation will differ depending if elements are taken from the same row, or if they are taken from the same column.

**Method 1: Elements from same column of C**

Each thread will allocate to itself elements from the same column (see figure 3.1).

To implement this each thread must, like in the single element case, assign to itself a starting column and row position. In this case it is straightforward as the thread can use its previously updated C reference \( cr \) and calculate its row and column number as in the single element method.

\[
\begin{align*}
  r &= cr \% m; \quad // \text{use C reference to find row} \\
  c &= cr / m; \quad // \text{use C reference to find column}
\end{align*}
\]

For the multiplication, each thread will still only need to use one column from the B matrix, however it will now require multiple rows from the A matrix.

```c
// loop to iterate over position of row of A, col of B
for (l = 0; l < k; l++) {

  // loop to change element
  for (i = 0; i < numberofelements; i++) {
    myc[i] = myc[i] +
    buff[l * m + r + i] * //element of row of a(changing)
    buff[mk + c * k + l];  //element of column of b
  }
}
```

**Method 2: Elements from same row of C**

Each thread will have to allocate to itself elements row-wise as shown in (see figure 3.2).

If each thread is allocated a group of elements from the same row of C, it now needs multiple columns of the B matrix but only one row of the A matrix.
The implementation is slightly more complicated than in the column-wise case. Here each thread has to calculate its row and column using its thread number.

\[
r = tn \mod m; \quad //finding \ the \ row \\
c = (tn / m) \times \text{numberofelements}; \quad //finding \ column
\]

In this case, the same row of A can be reused however multiple columns of B are required.

```c
// loop to iterate over position of row of A, col of B 
for (l = 0; l < k; l++) {
    for(i=0;i<numberofelements;i++) {// loop to change element
        myc[i] = myc[i] +
        buff[ l*\text{\texttt{m}}+r ] \times \text{element \ of \ row \ of \ a} \\
        buff[mk+(c+i)*k+l];//element \ of \ column \ of \ b(changing) 
    }
}
```
3.1.3 Writing C to global memory

When a thread has calculated its values of C, it is required to write them to the C matrix in global memory.

In the single element method each thread only has to write one element of C, this is done using its C reference \( cr \) to allocate the calculated C value to the correct place:

\[
c_{\text{data}}[\text{our\_params}[5]-1+cr] += \text{myc};
\]

When writing multiple elements from the same column, the thread can again use its C reference and just move on one place for each update.

```c
for (i=0; i<numberofelements; i++) {
    //find column and row and update C value
    c_{\text{data}}[\text{our\_params}[5]-1+cr + i] += \text{myc}[i];
}
```

Again it is slightly more complicated when dealing with multiple elements from the same row. Here each thread will need to use its row and column number, and move on the distance of the height of C \( (m) \) for each update.

```c
for(i=0; i<numberofelements; i++){
    //find column and row and update C value
    c_{\text{data}}[\text{our\_params}[5]-1+r + (c*m) + (m*i)] += \text{myc}[i];
}
```
3.2 Performance of Multiple Element Implementations

The implementation assigning threads elements for the same row greatly outperformed that of assigning elements from the same column. This is expected as the row-wise method allows consecutive threads to be allocated consecutive elements in memory, allowing efficient coalescing. The most effective implementation had two elements assigned to each thread, this reduced the kernel runtime to 0.0014 seconds, a 16% reduction from the original implementation; increasing the performance from 47.6 GFLOPS to 58.5 GFLOPS, now achieving 11.3% of the GPUs peak flop rate. When assigning more than two elements per thread, the performance quickly deteriorated (see figure 3.3).
Chapter 4

Multiple C Matrices

The initial GPU implementation had all threads writing to the same C matrix in global memory. As threads in separate blocks could potentially be writing to the same C element, a locking system was required to ensure the matrix was updated correctly, avoiding race conditions. In the initial profiling of the kernel it was shown that the writing to the C matrix was the most expensive section. By removing the locks the kernel run time reduces by 8.7% (see table 4.1).

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Kernel Run Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Kernel</td>
<td>0.001732s</td>
</tr>
<tr>
<td>Locks Removed</td>
<td>0.001581s</td>
</tr>
</tbody>
</table>

Table 4.1: Kernel Time with Locks Removed

To try to reduce the overhead of updating the C matrix an alternative method was implemented. The method removed the need for locks by having each thread block write to its own unique C matrix. However, this requires the introduction of a reduction step to merge data from the private copies of C into the final matrix.
4.1 Multiple C Matrices Implementation

To allow each thread block to have its own temporary C matrix for the test case requires 21 Gigabytes of memory on the device \((10000 \times 512 \times 512 \times 8 = 20,971,520,000)\), far larger than the 3 Gigabytes available on the target hardware. To cater for this, an implementation was created that would stagger the release of the thread blocks, requiring a smaller number of C matrices to be stored on the GPU memory.

To implement this, it is first required to allocate the memory on the GPU for the multiple C matrices. (Note that for ease of implementation this is done in the bridge \textit{dc\_do\_stack\_cu} function, so will be included in the CP2K kernel timing. To test the implementation fairly against the initial implementation the time taken for this allocation will be removed from the kernel timing)

```c
//allocate space on GPU for local C matrix arrays
cErr = cudaMalloc((void**) &c_temp,
                  numberofblocks*matsize*matsize*sizeof(double));
```

It is then required to zero this space of memory. This is done on the GPU by allocating each thread an element of C, and then each thread zeroing the assigned element for each of the C matrices. This results in consecutive threads writing to consecutive elements in memory, allowing for coalescing of writes. (Note that this implementation will only deal with cases where the size of C is a multiple of 1024).

```c
(HOST)
//zero allocate space
    zero_temp_c <<<(matsize/1024),1024>>>
    (c_temp,numberofblocks,matsize);

__global__ void zeroCextra
    (double * c_temp)
```
As mentioned earlier due to memory restrictions there are a restricted amount of C matrices stored on the device, and therefore, the multiplication kernels will have to be staggered, so that only a certain amount of thread blocks are issued at a time. To do this the host calculates how many full size thread blocks cycles are required to be called, and how many thread block cycles are required in the last cycle. It does this by dividing the stack size by the chosen number of local C matrices and then setting the number of full cycles required to the quotient and the size of the last cycle to the remainder.

```c
//calculate how many full cycles
full_cycles = stack_size/numberofblocks;
//find out how much remaining for last cycle
last = stack_size%numberofblocks;
```
The host can now send off the multiplication kernels to the device. For each group of thread blocks sent, the kernel is sent an additional piece of information telling it what its starting position in the stack is, so that they evaluate the correct values from the parameter stack and do not just compute the same values as the previous cycle of thread blocks.

(HOST CODE)

```c
//send off full cycles
for(i=0;i<full_cycles;i++){
    startingpos = i * numberofblocks;
    stack_mm_d <<< numberofblocks, maxt, shared_size >>>
        (param_stack, stack_size, nparams,
         (double *) a_data, (double *) b_data, (double *) c_data,
         startingpos,c_temp, matsize);
}

//send off last cycle if required
if (last != 0) {
    startingpos = full_cycles * numberofblocks;
    stack_mm_d <<< last, maxt, shared_size >>>
        (param_stack, stack_size, nparams,
         (double *) a_data, (double *) b_data, (double *) c_data,
         startingpos,c_temp, matsize);
}
```

(DEVICE CODE)

```c
/*starting position added so each
section of blocks gets correct params*/
sp = startingpos + blockIdx.x;
```

When writing to memory each thread now uses its block ID (0-99) so each thread block writes to a unique C matrix.

```c
//each block writes to its own C matrix
if (cr < mn) {
    c_temp[(blockIdx.x)*matsize + our_params[5]-1+cr] += myc;
}
```
When all the thread blocks are finished, it is now required to reduce the multiple C matrices into one. This is done on the GPU by assigning a thread to each element of C, and ordering each thread to sum the corresponding elements from each C matrix. This again allows for the memory updates to be coalesced.

(HOST CODE)
/perform reduction
  reduce_c <<<((matsize/1024),1024)>>(
    (double *) c_data,c_temp,
    numberofblocks,matsize);

(DEVICE CODE)
//kernel to perform reduction
__global__ void reduce_c
  (double* __restrict__ c_data,
   double * c_temp,
   int numberofblocks,
   int matsize)
{
  int i,id;

  //find out which C element I am allocated
  id = blockIdx.x * 1024 + threadIdx.x;

  //Gather all the data for that C element
  for (i=0;i<numberofblocks;i++){
    c_data[id] += c_temp[id + (matsize*i)];
  }
};
4.2 Multiple C Matrices Performance

Two implementations of the program were created and tested; one using 100 temporary C matrices and one with 1000. Performance was recorded for the test case of $512 \times 512$ sparse matrix with block sizes set to $16 \times 16$.

Note that for this program the stack size was increased from 10,000 to 30,000, large enough to process the multiplication in one stack. This was done as, for ease of implementation, the zeroing and reduction is done in the bridge `dc_do_stack_cu` function; this results in the program zeroing the temporary C matrices and gathering the information in a reduction step for every parameter stack created. This is clearly not optimal, as it is only required to do these procedures once. For the test case, with the stack size set to 10,000, it would call these procedures three times, showing an excessive expense. Having the stack size set to 30,000, allowed all the multiplications to be processed by one stack, therefore calling the procedures only once.

The CUDA profiling tool was used to view the time taken for the zeroing, the multiplications, and the reduction. Using the profiling tool also allowed for the time taken for the memory allocation to be dismissed.

![Figure 4.1: Multiple C Performance](image)

The multiple C implementations were both substantially slower than the original method; with the 100 matrix version taking twice as long and the 1000 matrix version taking five times the time.

The 100 matrix version, even without considering the time taken for zeroing and the reduction, was slower than the original implementation (see figure 4.1); this is due to the overhead of releasing the thread blocks in 100 block loads. By increasing the number of temporary C matrices to 1000, the overhead of the staggering of thread blocks was reduced. However, the larger number of local matrices greatly increased the expense of
the zeroing and reduction step, resulting in the whole process being more expensive.

The design of the reduction and the zeroing has the work load of each thread growing linearly with the number of C matrices, explaining why these procedures took $10 \times$ longer for the 1000 matrix version than the 100 matrix version.
Chapter 5

Using the Cublas library

The current GPU implementation is not completely robust; as a thread is required to process each element of a C block, it can only work if the C block size is smaller than the maximum number of threads per block. (On the Tesla C2050 this is 1536, i.e. restriction the block size to a maximum of $32 \times 32$). In the last chapter it was shown that assigning more elements to each thread becomes inefficient for more then two elements per thread, therefore an alternative method is required for larger block sizes.

The Cublas library contains a \texttt{dgemm} function to multiply matrices of double precision variables. It is expected that this function will be heavily optimised and therefore provide good performance when performing matrix multiplies. It is worth noting, however, that the Cublas implementation has a disadvantage when compared to the current implementation; it can only process one matrix multiply at a time, while the current method can process several.

The current method creates a thread block to process each line of the multiplication stack. Each of these blocks is then sent to an SM of the GPU. As there are multiple SMs in the GPU, many thread blocks can operate in parallel so many matrix multiplies from the multiplication stack can be processed in the same kernel call. However the \texttt{dgemm} function uses the whole GPU so using this method requires each matrix multiply to be done in sequence. Therefore, the Cublas implementation is not expected to compete with the current implementation when there are many small matrix multiplies. However, when there are fewer large matrix multiplies to be processed, not being able to process many matrix multiplies at the same time will not be as much of a hindrance, therefore, performance should be more competitive.

To see if the Cublas \texttt{dgemm} function would be suitable to take over from the original program when the block sizes are too large, and further, to see how its performance compares when the original method is feasible, a version of DBCSR was created having the Cublas \texttt{dgemm} function performing the matrix multiplies.
5.1 Implementing Dgemm

In the current version a parameter stack is processed by the kernel which creates a thread block to compute each of the matrix multiplies. In the Cublas version, this kernel call will be replaced by multiple calls to the Cublas dgemm function.

The Cublas dgemm function requires as parameters the sizes of the matrices it is multiplying, and the addresses of the matrices. As mentioned earlier all this information is contained in the multiplication stack (param_stack). The kernel calling function dc_do_stack_cu has access to the GPU pointer of the parameters but not the actual parameter values. To get this information it can copy the information from the device.

```c
//copy the matrix multiplication parameters from GPU memory
cErr = cudaMemcpy(params_temp, param_stack, 7*stack_size*sizeof(int), cudaMemcpyDeviceToHost);
```

For each of the lines of the parameter stack a call can now be made to Cublas dgemm function to process the multiplication. This is done by looping over the parameter stack and feeding in a new line into the dgemm function at each iteration.

```c
//loop over the parameter stack
for (i=0;i<stack_size;i++){

    //get the size of the matrices
    m= params_temp[0+ (7*i)];
    n= params_temp[1+ (7*i)];
    k= params_temp[2+ (7*i)];

    //get the location of the data
    devPtrA = (double*)a_data + params_temp[3 + (7*i)]-1;
    devPtrB = (double*)b_data + params_temp[4 + (7*i)]-1;
    devPtrC = (double*)c_data + params_temp[5 + (7*i)]-1;

    //perform multiplication
    cublasDgemm('N','N',m,n,k,1,devPtrA,m,devPtrB,k,1,devPtrC,m);
}
```

Note that the initial GPU implementation has a check to ensure the block size is less than the number of threads available per block.

```c
if (maxt > devProperties.maxThreadsPerBlock)
    return 3;
```

As this is not relevant for the DGEMM implementation this is removed.
5.2 Dgemm Performance

The original program outperforms the \textit{dgemm} implementation for all the block sizes of the test case for which it is valid. For the $16 \times 16$ block size test case the Cublas kernel takes 0.049s (0.8 GFlops), an order of magnitude slower than the original version. This highlights the benefit of processing multiple matrix multiples at the same time.

As the block sizes increase beyond $16 \times 16$, the Cublas implementation speeds up rapidly, while the original implementation slows down (see figure 5.1). The Cublas implementation performs well when the original program is not valid, showing that it is a suitable option to take over from the original program when block sizes are unsuitably large.

Figure 5.1: Performance of Cublas Implementation
Chapter 6

Further Work

By assigning multiple elements to each thread the program ran faster. However, the program created in this project would only process block sizes that were a multiple of the number of elements assigned to each thread. To implement this in a real simulation, the program would have to accept a range of block sizes, and so this restriction would have to be removed.

The multiple C implementation has the cost of the reduction and zeroing growing linearly with the number of C matrices used, as each thread is required to gather an element from each of the C matrices. This showed its overhead with the 1000 matrix zeroing and reduction each taking 10 times longer than the 100 C matrices implementation. To create a competitive version of multiple C implementation the cost of the reduction step must be reduced for large numbers of C matrices. It would be of interest therefore to investigate other reduction techniques, tree based techniques may be of particular interest due their typical $O(\log(n))$ cost.

The Cublas dgemm function showed itself to be a suitable candidate to take over from the original GPU implementation when it was not valid due to large block sizes. However, it was outperformed for small block sizes. An effective implementation would use the original implementation for small block sizes and the dgemm function for larger blocks. In real simulations, unlike the test case, the block sizes vary for each small matrix multiply, so the block size would have to be checked for each matrix multiply in order to assign it to the correct function.

In the current implementation each thread block does a single local matrix multiplication. It does not take into consideration that there may be overlap in the data that is required for these multiplications. A smarter version could create a parameter stack taking into account the data required to perform the multiplications, allowing thread blocks to do multiple matrix multiplications, getting better use of the the data loaded into their shared memory. Clearly, for this method to be beneficial, the reduction in run time of the multiplications must outweigh the expense of the parsing and modifying of the parameter stack. This operation may, therefore, be required to be performed on the GPU.
Chapter 7

Conclusion

The original GPU implementation of DBCSR performed a single $512 \times 512$ sparse matrix multiplication in less than 1/3rd of the time of the theoretical 8 core CPU only version with linear speed up. However, the implementation was only achieving 9.2% of the GPUs peak performance. On profiling the kernel it showed that it was the writing of the C values to global memory that was most expensive.

By altering the original implementation so that each thread was allocated, not one, but multiple elements of C, it was found that the run time of the kernel could be reduced. Allocation was done for up to 16 elements per thread in both a row-wise and a column-wise manner, it was found that the row-wise technique was the stronger performer as it allowed for efficient memory coalescing. The most successful implementation was that which assigned each thread two neighbouring elements from the same row of the C matrix. This resulted in reducing the time to process a multiplication stack by 16%, increasing the FLOP rate to 58.5 GFLOPS, 11.2% of peak performance.

The multiple local C matrix implementation, removed the need for locking updates to the C matrix, by giving each thread block its own local C matrix and adding a reduction step. With only a small number of private C matrices, the multiple C matrix implementation was hindered by the overhead of releasing the kernels in small loads. This was highlighted by the implementation using 100 local C implementation, which did not reduce the calculation time even after removing the time taken for the zeroing and reduction. Increasing the number of blocks reduced this overhead, but greatly increased the expense of the zeroing and reduction. To make this method competitive, more effective reduction techniques would have to be considered.

An implementation of DBCSR was created using the Cublas *dgemm* function to perform the matrix multiplies. While it performed poorly in comparison with the original implementation for small block sizes, its performance improved greatly for large block sizes; showing itself to be a suitable candidate to take over from the original implementation when the block sizes are too large for it to compute.
Appendix A

Test Input File and Kernel Timing Output
Test Input File

DBCSR test input file for a sparse 512 × 512 matrix multiplication on the GPU, with blocks set to 16 × 16, stack size set to 10,000, and kernel timings on.

```
&GLOBAL
  PRINT_LEVEL MEDIUM
  PROGRAM_NAME TEST
  RUN_TYPE NONE
&TIMINGS
  THRESHOLD 0.0000000001
&END
&DBCSR
  mm_driver CUDA
  mm_stack_size 10000
  use_cuda_host_allocation F
  DETAILED_TIMING T
  KERNEL_TIMING T
&END DBCSR
&END GLOBAL
&TEST
&CP_DBCSR
  K 512
  M 512
  N 512
  TRANS A FALSE
  TRANS B TRUE
  N_LOOP 10
  ASPARSITY 0.05
  BSPARSITY 0.05
  CSPARSITY 0.05
  bs_m 1 16
  bs_n 1 16
  bs_k 1 16
&END
&END TEST
```
Timing Output with Kernel Timings

*******************************************************************************
PERFORMANCE TESTS
*******************************************************************************

Multiplication with sizes 512 512 512 min/max block sizes 16 16 transposed? F T

<table>
<thead>
<tr>
<th></th>
<th>Index time</th>
<th>Max size time</th>
<th>Thread sync time</th>
<th>Parameter copy time</th>
<th>Kernel time</th>
<th>Index time</th>
<th>Max size time</th>
<th>Thread sync time</th>
<th>Parameter copy time</th>
<th>Kernel time</th>
<th>Index time</th>
<th>Max size time</th>
<th>Thread sync time</th>
<th>Parameter copy time</th>
<th>Kernel time</th>
<th>Index time</th>
<th>Max size time</th>
<th>Thread sync time</th>
<th>Parameter copy time</th>
<th>Kernel time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>13.651E-03</td>
<td>66.000E-06</td>
<td>36.000E-06</td>
<td>99.000E-06</td>
<td>1.748E-03</td>
<td>941.000E-06</td>
<td>65.000E-06</td>
<td>35.000E-06</td>
<td>98.000E-06</td>
<td>1.725E-03</td>
<td>591.000E-06</td>
<td>70.000E-06</td>
<td>42.000E-06</td>
<td>97.000E-06</td>
<td>1.660E-03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Segment Local Multiplication time 0.0086 and 28.3046E+03 MFLOP/s 242.3439E+06 FLOP
Stack Process Multiplication time 0.0069 and 35.1478E+03 MFLOP/s 242.3439E+06 FLOP
Kernel Process Multiplication time 0.0051 and 47.2129E+03 MFLOP/s 242.3439E+06 FLOP
Total Local Multiplication time 0.0086 and 28.3046E+03 MFLOP/s
Total Multiplication time 0.0111 and 21.7524E+03 MFLOP/s
Complete Multiplication time 0.0259 and 9.3472E+03 MFLOP/s
Total Multiplication 1 processors: 6.495 s, 37.312E+06 FLOP/s 0.3525360E+10 0.3188850E+09
Bibliography


