Large Matrix Multiplication on FPGA

Naimul Hoque
nh15775@my.bristol.ac.uk

Goce Dimitrov
gd14470@my.bristol.ac.uk

***Abstract*—This report looks into various methods of performing matrix multiplication, the advantages and disadvantages of each method, and the performance of one particular method, matrix-vector multiplication. Testing has shown that matrix-vector multiplication can be very fast compared to the naive method and can multiply in parallel much larger matrices, and thus the ability to use larger block sizes can lend itself to greater performance gains.**

I. Introduction

Matrix multiplication is a complex and fundamental matrix operation that is used in many algorithms for scientific computations. Because of this, much work has been invested in making matrix multiplication algorithms as efficient as possible.

Traditionally, matrix multiplication operations are implemented on PCs or DSPs (Digital Signal Processors) which are based on a serial structure and often become the bottleneck of the overall system. This is due to the increasing density and massive computing performance that is especially needed in floating-point calculations.

Field Programmable Gate Arrays (FPGAs) with their hardware parallelism are becoming a promising way to speed-up the floating-point matrix multiplications especially when large matrices are involved.

One of the pitfalls of FPGA design is the relatively long implementation time when compared to alternative architectures such as CPUs, GPUs or DSPs. However, this can be greatly reduced by tools that generate hardware systems in the form of a hardware descriptive language (HDL) from high-level languages such as C or C++. Such implementations can be optimized by applying special directives called pragmas that focus the high-level synthesis (HLS) effort on particular objectives, such as performance, area and throughput.

In this report, various matrix multiplication methods have been outlined and each corresponding result has been plotted and compared with a naive software version of matrix multiplication, in order to find the version that utilises the most of the FPGA's resources whilst gaining the most speed up over the software.

II. Naive Multiplication

Matrix multiplication is defined as follows:
For two matrices A and B of sizes NxM and MxP:

$$C = A * B; \tag{1}$$

In element-wise form:

$$C_{ij} = \sum_{k=0}^{M} A_{ik} B_{kj} \tag{2}$$

For the purposes of this report, the matrices are square matrices of size NxN, where $N = 1024$.

Implementing this naive algorithm is very simple, as shown below.

```
void mmult (float A[N*N], float B[N*N], float C[N*N])
{
    for (int row = 0; row < N; row++) {
            for (int col = 0; col < N; col++) {
                float result = 0.0;
                for (int k = 0; k < N; k++) {
                    result += A[row*N+k] * B[k*N+col];
                }
                C[row*N+col] = result;
            }
        }
}
```

Fig. 1. Code for Naive Multiplication

Whilst the process of matrix multiplication is quite easy to perform, this naive method is extremely slow. The number of multiplications required is N^3 and the number of additions is $N^2(N-1)$. The worst-case computational complexity then would be $O(N^3)$.

It is clear then that a naive approach would not make good use of the FPGAs resources. Larger matrix sizes means more data to iterate through one by one, and the lack of parallelism exploited here is evident from the diagram below.

Performance estimates for functions 'madd in main.cpp:128 ...

Hardware accelerated (Estimated cycles)	230729109

Resource utilization estimates for Hardware functions

Resource	Used	Total	% Utilization
DSP	162	220	73.64
BRAM	2080	140	1485.71
LUT	157918	53200	296.84
FF	152098	106400	142.95

Fig. 2. Performance estimate and utilisation for naive method

Attempting to run the naive algorithm showed that the FPGA would spend a very long time trying to compute the results, to the point that there would be no speedup factor if it ever finished.

III. Block-Matrix Multiplication

To take better advantage of the FPGA, one approach is to use "blocks" from the matrices. Rather than copying the entire

matrix over to process, take blocks from A and B and perform multiplication over those blocks to produce a block for C. In other words:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad (3)$$

The same is true for B. Multiplication then would be defined similarly to the element wise multiplication, but over the blocks instead. The size of these blocks would be less than N, and as these would be deemed "sub-matrices": $S < N$.

There's an immediate advantage then to using blocks instead of iterating over every element: you only need to load, process and store what's necessary at a particular time. This means that storing the whole matrices on the FPGA will not be necessary and its resources will not be overused.

In the picture below timings for different algorithms has been shown. Testing has been done in serial using a normal C++ compiler.

```
Enter matrix size:
1024
Enter block size:
32
Init has begun
Computation has begun(serial)
Time elapsed: 9.75358
Computation has begun(serial transposed)
Time elapsed: 4.84315
Computation has begun(serial block)
Time elapsed: 4.01327
Computation has begun(serial block, transposed)
Time elapsed: 3.72173
```

Fig. 3. Timings for different Matrix Multiplication methods

As it can be seen, there was an evident speed up in comparison to the naive method and furthermore you can specify the size of the blocks that would be processed, so that it would not use more than the available resources from the FPGA. Unfortunately, we were unable to get block matrix multiplication to work correctly on the FPGA in parallel. Presumably due to logic errors with the implementation.

IV. Matrix-Vector Multiplication

There's an alternative variant of Block-Matrix multiplication, and that's Matrix-Vector multiplication. The blocks stated above have size SxS, and while this method can be quite effective, a more intuitive way of using these blocks is through loading entire rows and large blocks of matrices, rather than equal sized blocks each time.

For a row from A of length N, a block from B of size NxS, the resultant segment in C is of size S, where each element C_j is computed as $C_j = \sum_{k=0}^{N} A_k B_{kj}$, where $j < S$.

```
#pragma SDS data zero_copy(A[0:N * N], B[0:N * N], C[0:N * N])
void vecmat_mmult(float A[N * N], float B[N * N], float C[N * N])
{
    float b[N][S]; //Create buffer for B
#pragma HLS ARRAY_PARTITION variable = b block factor = 16 dim = 2

    int block_offset;
    int offset;
    /*
     * N / S is the block count. This allows us to sweep across
     * an N x S region in B and store that in fast memory.
     */
    for (int k = 0; k < N / S; k++)
    {
        block_offset = k * S;
        for (int i = 0; i < N; i++)
        {
            offset = i * N + block_offset;
            for (int j = 0; j < S; j++)
            {
#pragma HLS PIPELINE
                b[i][j] = B[offset + j]; //Load block
            }
        }
        for (int p = 0; p < N; p++)
        {
            /*
             * Using pointers allows us to specify where exactly in A
             * we should start reading from, and where in C we should
             * start writing to.
             */
            matxvec(A + p * N, b, C + p * N + S * k);
        }
    }
}
```

Fig. 4. Code for the vecmat_mult function

```
void matxvec(float A[N], float B[N][S], float C[S])
{
    int j;
    float a[N];
    float c[S];
#pragma HLS ARRAY_PARTITION variable = c block factor = 16 dim = 1
    /*
     * Normally a for loop would be used to copy data
     * to and from buffers and matrices.
     * However, memcpy can do this process very efficiently.
     */
    memcpy(a, (float*)A, N * sizeof(float));
    memset(c, 0, S * sizeof(float));
    for (int k = 0; k < N; k++)
    {
        for (j = 0; j < S; j++)
        {
#pragma HLS UNROLL
#pragma HLS PIPELINE
            c[j] += a[k] * B[k][j];
        }
    }
    memcpy((float*)C, (float*)c, S * sizeof(float));
}
```

Fig. 5. Code for the matxvec function

The code described above is all that's needed to perform matrix-vector multiplication. The vecmat_mult function is the hardware accelerated function, as this is where the buffering and primary data transfer occurs.

A. Implementation Details

To make effective usage of an FPGA, several pragmas were used. At the top of the vecmat_mult function is a data transfer pragma called zero_copy. Information on this pragma was derived from [1]. Hardware by default copies all data across before processing occurs. zero_copy ignores this and instead uses shared memory to read data from. This is advantageous as it reduces the time required to read in the data.

Within every loop of both functions, there's either a pipeline or unroll pragma, or both.

The unroll pragma will take a loop and unroll it into several function calls, each call occurring at a different index offset.

The pipeline pragma on the other hand enables immediate sequential calls within the loop, since otherwise there would be some latency between loop calls.

From 4, the buffer for B uses a pipelined loop, whereas the multiplication in 5 uses unrolling and pipelining to fill the c buffer there. This is due to the way both buffers are set up, as c is a linear block whilst b is a 2D block, and unrolling does not behave nicely with the 2D block.

Using pointer arithmetic, it's possible to provide matrix A and C directly to the matxvec function, specifying the location where reads and writes should occur. A buffer could have been used in place of using pointers, but this method allows the usage of two methods from the string.h header file, memset and memcpy, where memset will fill a block of memory with a particular value and memcpy will fill a block with values from another block of memory. Loops could have been used to fill both the buffer and the C matrix, but this function call does that provide succinctly and efficiently.

B. Testing methodology

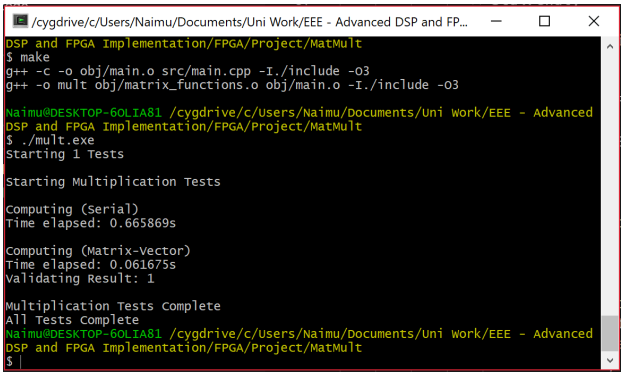

Fig. 6. Output of test for matrix-vector multiplication

It was important for us to test the code to make sure it produced valid results. As such, we tested the code using a normal C++ compiler for this purpose, as shown in Figure 6 above.

To test the FPGA, we built the algorithm using Xilinx SDx, a High-Level Synthesis program that produces hardware from C/C++ code. We used a matrix size $N = 1024$, a block size $S = 64$ and the number of iterations ran was just one so that we could test variations of the algorithm faster. An output of this is shown below in 7.

```
Testing 1 iterations of 1024x1024 floating point mmultadd...
Average number of CPU cycles running mmultadd in software: 43422306448
Average number of CPU cycles running mmultadd in hardware: 1515300930
Speed up: 28.6559
TEST PASSED
```

Fig. 7. Output from SDx Hardware Run

C. Results

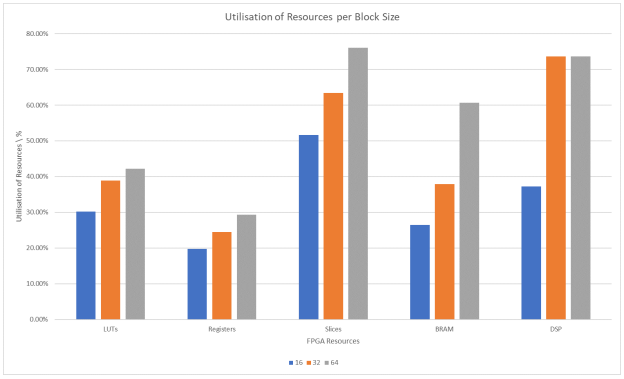

Fig. 8. Utilisation of FPGA Resources per Block Size

From Figure 8 above, as block size increases so does the utilisation of resources. The rate of increase per resource varies per block size, with LUT usage increasing only a small amount, but BRAM usage grows at a much larger rate. Interestingly, DSP usage appears to max out after a block size of 32, suggesting that the DSPs are being used as efficiently as possible. Further testing then would have shown larger resource utilisation for all other resources barring DSPs.

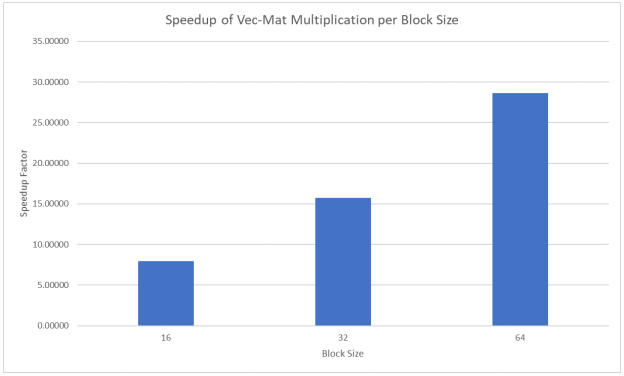

Fig. 9. Speedup of Vector-Matrix Multiplication per Block Size

Similar to Figure 8, Figure 9 shows greater performance gains as block size increases. Given the small sample size of only three different block sizes, it's hard to say if the performance gains would continue to grow at the same rate, or if diminishing returns would show up early.

V. Conclusion

The long build times drastically reduced the amount of testing that could be performed. However, it was possible to demonstrate matrix-vector multiplication working well and scaling well for varying block sizes. Given more time, it would've been possible to explore the performance gains for larger matrix sizes and test using various different directives (pragmas).

FPGAs are well suited for the task of matrix multiplication, but it's important that the algorithms supplied take advantage

of their highly parallel nature. This means writing code that can lend itself to a strong hardware implementation. Buffering data is necessary to minimise latency and maximise throughput, alongside using the unroll and pipeline pragmas.

REFERENCES

[1] "SDx Pragma Reference Guide." https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug1253-sdx-pragma-reference.pdf, 2017. [Online; Accessed 15-03-2019].