Matrix Multiplication FPGA vs. CPU

Matrix Multiplication

Different algorithms can be used to perform matrix multiplication

- Iterative(naive) algorithm
- Cache behavior
- Strassen algorithm
- Divide-and-conquer algorithm

We will be using the divide-and-conquer algorithm:

- Works for all square matrices with dimensions of the power of 2
- Computes smaller matrices recursively

CPU Hardware Comparison:

Table I
Comparison of Potential (Laptop) CPUs

Name	# of Cores	# of Threads	Base Clock Frequency (GHz)	Boost Clock Frequency (GHz)	TDP (W)
AMD Ryzen 7 5800HS	8	16	2.8	4.4	35
AMD Ryzen 9 5900HS	8	16	3.0	4.6	35
Intel Core I5 7600K	4	4	3.8	4.2	91

FPGA Hardware Comparison:

Table II
Comparison of Potential FPGA Boards
(Modified From [1] Table 6.1)

Name	Total board RAM (Mb)	DSP Slices	Flip Flops	LUTs	Price
Kintex UltraScale KU115	75.9	5, 520	1, 326, 720	663, 360	$6,495.00
Nexys A7-50T	2.7	120	65,200	32,600	N/A
Nexys A7-100T	4.86	240	126,800	63,400	$349.00

Though it is significantly less robust than the Kintex UltraScale KU115 used in [1], we will be using the Nexys A7-100T due to its availability to us as students and much cheaper cost.

Metric Descriptions

n - matrix dimension for a square (n x n)

threads - number of threads the algorithm will use. Higher numbers = more parallel work

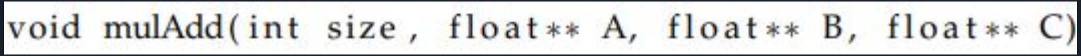
iters - number of iterations it repeated the run. Used to average the data and select the best runs

best_ms - best (minimum) elapsed time across iterations for the configuration

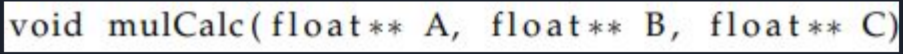
gflops - giga floating point operations per second. Higher is better.

CPU implementation

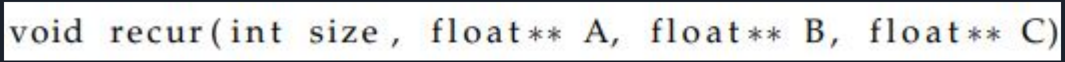
- Divide and conquer: split the matrix into smaller matrices of size *p*, the size of the kernel module
- Use the kernel to perform some multiplication operations on hardware
  - Using software only is computationally intensive
- Utilize 4 threads because parallelism is faster than serialism
- Choosing the kernel size *p* is important because different kernel sizes have differing overhead



```
void mulAdd(int size, float** A, float** B, float** C)
```



```
void mulCalc(float** A, float** B, float** C)
```



```
void recur(int size, float** A, float** B, float** C)
```

Hardware implementation

- By taking the portion that the program spends most of its time on, we optimize it using an FPGA of our choice (nexys A7-100T)
- Modify the kernel to support fixed hardware limitations
- by adding pragma HLS UNROLL, allows us to run loop iters in parallel, increasing kernel size and performance
- Depending on FPGA, can implement multiple instances of the kernel increasing performance

```
# include ”matrixmul.h”
void mulCalc ( float A[ kernal Size ] [ kernal Size ] , float
B [ kernal Size ] [ kernal Size ] , float C[ kernal Size ] [
kernal Size ] ) {
 for ( int i = 0 ; i < k e r n al Si z e ; i ++){
        for ( i n t j = 0 ; j < k e r n al Si z e ; j ++){
                #pragma HLS UNROLL
                C[ i ] [ j ] = 0 ;
                f o r ( i n t k = 0 ; k < k e r n al Si z e ; k++){
                        #pragma HLS UNROLL
                        C[ i ] [ j ] += A[ i ] [ k ] ∗ B [ k ] [ j ] ;
                        }
                 }
        }
 }
```

References

[1] G. Maan, “Hardware acceleration of matrix multiplication,” Bachelor Thesis, Leiden Institute of Advanced Computer Science, Leiden University, Leiden, The Netherlands, 2019. [Online]. Available: https://theses.liacs.nl/pdf/2018-2019-MaanGC.pdf