EE451
Accelerating Matrix Multiplication Algorithms on Cerebras CS-2 Wafer-Scale Engine

Felicia Lo and Kaley Tien

May 3, 2024

1 Introduction

From 2018 to 2020, the parameters of state of the art neural networks increased from 100 million to 175 billion parameters. As neural networks continue to grow, the demand for hardware capable of efficiently handling large neural networks also increased. Cerebras created the Wafer-Scale Engine, with architecture designed to optimize machine learning and AI computations. Being the second largest computer chip thus far with 850,000 cores, the WSE-2 enables new levels of parallelism which can be used to accelerate complex computations.

The Cerebras CS-2 System, embodying the WSE-2 architecture, features a 2D-array of individually programmable processing elements (PEs) with routers, processors, and 48kB memory each. A key optimization focuses on memory bandwidth, where unlike most processors matches the operand bandwidth by distributing memory alongside processing elements, thus minimizing memory access latency. Unlike traditional systems, Cerebras accommodates even the largest layers onto a single chip, exploiting processors to run processes as purely data-parallel tasks, thereby eliminating interchip latencies and complexities and greatly enhancing parallelization.

The three primary matrix multiplication kernels, Dense-Dense, Sparse-Dense, and Sparse-Sparse, serve as fundamental compute primitives across various applications, from deep neural network training and inference to Fast Fourier Transforms in signal processing. The Dense-Dense kernel handles multiplication between two dense matrices, while Sparse-Dense and Sparse-Sparse kernels manage multiplication involving a sparse matrix and a dense matrix, and multiplication of two sparse matrices, respectively.

In our project, we will explore and enhance accelerating matrix multiplication kernels on the Cerebras systems. We aim to optimize the Dense-Dense, Sparse-Dense, and Sparse-Sparse matrix-multiplication kernels by exploiting the massive parallelism available via the large number of Cerbras cores, capable of storing all

data required for processing on-chip, and show that these operations are compute-bound, using the roofline model, due to limitations of the computational resources on the Cerebras system.

2 Background & Related Work

The naive algorithm for matrix multiplication has a computational runtime of $\theta(n^3)$ which exponentially increases as n grows. With the need for more scalable solutions, many matrix multiplication accelerations have been created that take advantage of data locality, memory access patterns, vectorization, parallelism, or reducing overhead.

The architecture of a GPU is designed to support high-data throughput, which is necessary for large matrix computations. There are many optimized libraries and frameworks for the GPU such as CUDA that take advantage of the multithreading capabilities. A GPU utilizes shared memory for communication

CPUs are also highly capable of computing matrix multiplication, but is quickly outperformed by the GPU and FPGA as the matrix sizes increase.

Field-Programmable Gate Arrays (FPGAs) are suited to optimize matrix multiplication since its architecture can be customized or reconfigured to be tailored to specific tasks. The Scalable Macro-pipelined Architecture (SMPA) is an accelerator system that uses a ring of connected PEs to overlap communication and computation. Along with the customized architecture, there is an algorithm that fully exploits the set-up that is similar to block matrix multiplication.

Cerebras is set apart due to the Wafer-Scale Engine (WSE) which is the entire silicon wafer for a single chip. Because the WSE is not limited by the number of cores, having 850,000, which for a CPU/GPU might only be in the thousands, the parallel capabilities are unmatched and can be exploited. The specialized fabric, which is the interconnection system between cores, allows for effective and quicker communication. The enormous on-chip memory reduces overhead and latency compared to a system where multiple chips need to communicate with each other.

3 Problem Definition

Naive matrix multiplication computation of c_{ij} in C:

$$c_{ij} = \sum_{k=1}^{K} a_{ik} \times b_{kj}$$

Block matrix multiplication run-time, where:

- A_{rs} is the block in the r-th block row and s-th block column of A
- B_{st} is the block in the s-th block row and the t-th block column of B

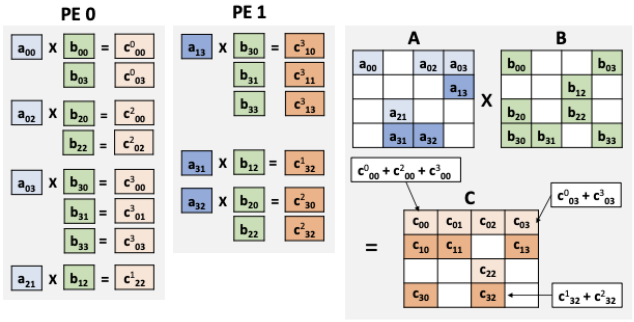

Figure 1: Row-wise Products

- C_{rt} is the block in the r-th block row and the t-th block column of C

$$C_{rt}(i', j') = \sum_{s=1}^{S} \sum_{\alpha=1}^{k} A_{rs}(i', \alpha) B_{st}(\alpha, j')$$

Sparse matrix Multiplication computation using row-wise Products:

$$C[i, :] = \sum_{k=0}^{N} A[i, k] \cdot B[k, :]$$

Sparse matrix multiplication runtime, given:

- Input and output matrices have dimensions = N × N
- Input matrices number of non-zeros = nnz
- Output matrix number of non-zeros = nnz'
- Non-zero elements for each row/col of input matrices for input matrices = $\frac{nnz}{N}$
- Non-zero elements for each row/col of input matrices for output matrix = $\frac{nnz'}{N}$

Computational complexity of sparse matrix multiplication, given above parameters, is $O(\frac{nnz}{N})$.

4 Method

4.1 Cerebras Architecture

The architecture of Cerebras is made up of a 2D mesh of processing elements (PEs) interconnected by a high-bandwidth fabric. These PEs are independently programmable and are capable of communicating with neighboring PEs through the fabric. The layout of the PEs allows for efficient communication and data exchange, facilitating parallel processing and scalability.

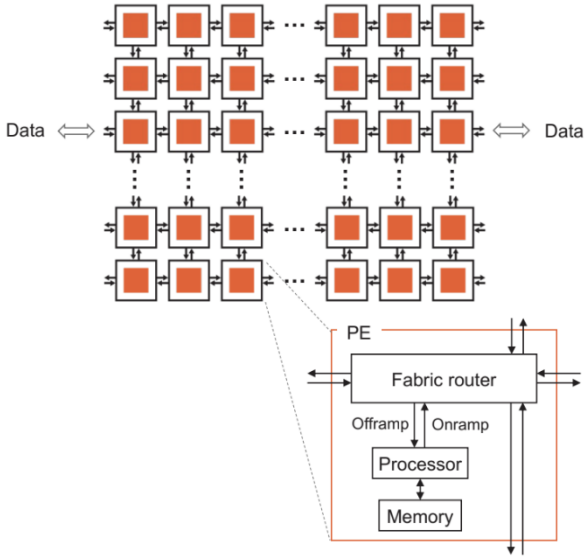

Figure 2: CS-2 2D mesh of Processing Elements

Facilitating the communication within this architecture are high-speed interconnects, ensuring swift and efficient data transfer between PEs. Each PE is equipped with a router, a 24-entry table associating colors with directions. This router system enables complex communication patterns, dynamically updating routes at runtime to adapt to changing computational requirements. With multiple routing table entries per color, multicast capabilities allow for data to be broadcasted in multiple directions simultaneously each cycle.

In our experiment, we utilized the Cerebras simulator and focused on evaluating the performance of their GEMM (General Matrix Multiply) collectives operation program. We varied the size of the fabric and the number of processing elements (PEs) employed, to see how it would impact the overall execution time and efficiency of the program. To achieve this, we implemented a blocked matrix algorithm and adjusted the sizes of the blocks assigned to each PE for computation. These results will provide insights into optimizing the performance of parallelized matrix multiplication operations on Cerebras systems and by comparing the results to that of GPUs we may see the speedup that can be achieved.

4.2 GEMM on Cerebras

A complete program on Cerebras that computer General Matrix Multiplication (GEMM) will contain a host program (run.py), a layout file (layout.csl), and a file that defines programs running on each PE (pe.csl). For ease, there is also a commands.sh file used to compile and run the code, which also holds the dimension for PEs (defined as P) as well as matrix dimensions.

Algorithm 1 is the script that launches the code from a host machine to the

device. The device code is loaded, run and launched using the runner object. At the end of the program, the runner object copies the back the output C from the device back to the host.

Algorithm 1 run.py pseudocode

Ensure: $Mt = Kt = Nt$ ▷ Square matrices for simplicity
$P \leftarrow P$ from commands.sh ▷ number of PEs = PxP
$Mt, Kt, Nt \leftarrow Mt, Kt, Nt$ from commands.sh

$M = N = K = Mt \times P$
$A = RandomMatrixGenerate(M, K)$
$B = RandomMatrixGenerate(K, N)$
▷ Load model on hardware and begin processing
runner.load()
runner.run()
▷ Prepare matrices A and B
$A1 = A.reshape()$
$A2 = A1.transpose()$
$A3 = A2.reshape()$

$B1 = B.reshape()$
$B2 = B1.transpose()$
$B3 = B2.reshape()$

runner.launch() ▷ Start main computation kernel

runner.memcpyd2h(C3fromDevice) ▷ Transfer result from device to host

$C3 = C3fromDevice.reshape()$
$C2 = C3.transpose()$
$C1 = C2.reshape()$
$C = C1.view()$

Algorithm 2 implements SUMMA to perform GEMM. Matrix A and B are arranged in column major order. At each iteration 'i', the PEs in the ith column broadcast their A tiles to neighboring PEs in their row; PEs in the ith row will broadcast their B tiles to PEs in their respective column. The tiles passed by neighboring PEs are stored in a buffer to be used to execute FMAC function to update the values in C.

Algorithm 2 Matrix Multiplication

```
// Define dimensions for each block
param Mt, Kt, Nt;
// Initialize arrays for matrices A_tile, B_tile, and C_tile, stored in column-major format
// Initial state setup
procedure MAIN
    if (step == 0) then
        mpi_x.init();
        mpi_y.init();
        px = mpi_x.pe_id;
        py = mpi_y.pe_id;
    end if
    // Communicate along both rows and columns at each step 'i'
    const Ap = if (px == step) then &A_tile else &A_buffer;
    const Bp =  if (py == step) then &B_tile else &B_buffer;
    PEs in the ith column broadcast their home tiles of A
    PEs in the ith row broadcast their home tiles of B
end procedure
procedure COMPUTE
    // Task receive tiles from neighbors
    const Ap = if (px == step) then &A_tile else &A_buffer;
    const Bp = if (py == step) then &B_tile else &B_buffer;

    // Retrieve matrix A from memory and set up its Distributed Sparse Data (DSD) representation
    for each index k in the range 0 to Kt (exclusive) do
        // Retrieve matrix C from memory and set up its DSD representation
        C_dsd = GetDSD(mem1d_dsd, {tensor_access = λfunction});

        for each index j in the range 0 to Nt (exclusive) do
            // Calculate the scalar value b from matrix B based on indices j and k
            b = Bp[j × Kt + k];

            // Perform FMAC using elements from matrices A and B
            FMAC(C_dsd, C_dsd, A_dsd, b);        ▷ Update the corresponding element in matrix C

            // Update the offset matrix C to along the inner dimension
            C_dsd = IncrementDSDOffset(C_dsd, Mt, f32);
        end for
        // Update the offset matrix A along the inner dimension
        A_dsd = IncrementDSDOffset(A_dsd, Mt, f32);
    end for
    // Increment the step counter by 1
    // Block until the compute task identified by compute_task_id completes
    if (step ≠ P) then
        Call the main() function;
    else
        // If it is, activate the exit signal or function
        Activate the EXIT process;
    end if
end procedure
```

5 Experiments & Results

In order to test our claims, we ran algorithm 1 on Cerebras using different block-size values, number of processors, and matrix sizes. As a standard across all the systems, we used block-sizes = 8, 12, 16, 20, 32 against numbers of processors = 4, 8, 16, 32. The two values also multiply to create the matrix size for that run. The data from the Cerebras, GPU, and CPU experiments are below in Tables 1-3.

The algorithm used to test the performance of Cerebras executing Matrix multiplication is a given example program that uses the collectives2d library from Cerebras. We used the GUI of the simulator to find the number of cycles used in each execution.

To test the performance of the CPU, we implemented Cannon's Algorithm using MPI. Note that multi-node processors were used rather than single node processors in this implementation.

In order to create comparable results, blocked matrix multiplication using shared memory was executed on the GPU using a grid dimension of 1×1 and a thread block dimension equivalent to the PE mesh size.

5.1 Platform specifications

The specifications for the utilized Cerebras, GPU and CPU platforms used to run the experiments are as below.

Cerebras System Specifications	
Base-Frequency	1.1GHz
On-chip Memory	40 GByte
Memory Bandwidth	20 PByte/s
Fabric Bandwidth	220 Pbit/s

Table 1: Cerebras Specification

CPU: xeon-4116 Specifications	
Base-Frequency	2.1GHz
On-chip Memory	768 GBytes
Memory Speed	2400 MHz
Fabric Bandwidth	-

Table 2: CPU Specifications

GPU: v100 Specifications	
Base-Frequency	1200 MHz
On-chip Memory	32GB /16GB HBM2
Memory Bandwidth	900GB/s
Fabric Bandwidth	32GB/s

Table 3: GPU Specifications

5.2 Results

Table 4: Results of Cerebras Experiment(s)

PE Mesh	**Matrix Size**	**Block Size**	**Latency @ 1.1 GHz (ms)**	**Cycles**
4	32×32	8	0.00593	6523
4	48×48	12	0.00092	1012
4	64×64	16	0.0446409	49105
4	80×80	20	0.01636363	18000
4	128×128	32	0.03397363	37371
8	64×64	8	0.0162509	17876
8	96×96	12	0.04390363	48294
8	128×128	16	0.01018818	11207
8	160×160	20	0.04390363	55652
8	256×256	32	0.05468727	60156
16	128×128	8	0.04073363	44807
16	192×192	12	0.12184363	134028
16	256×256	16	0.08747363	96221
16	320×320	20	0.06455636	71012
16	512×512	32	-	-
32	256×256	8	0.13181818	145000
32	384×384	12	0.10744454	118189
32	512×512	16	-	-
32	640×640	20	-	-
32	1024×1024	32	-	-

The following plots represent the data from the tables with matrix size on the x-axis and normalized speed-up on the y-axis. Due to different frequencies among the different systems, the different calculated latencies are not comparable unless normalized. The speed up of Figure 3, 4, 5, 6 are normalized with respect to the base frequency of the CPUs and figure 7 with respect to the base frequency of GPUs.

Figure 3 shows that Cerebras has a significant speed-up with respect to GPU. Although the graph does not visually represent a speed-up on the GPU, there is a speed-up though not as significant.

Figure 4 has a similar observation as Figure 3.

Figure 5 is a comparison in speed-up between Cerebras and GPU with PE meshes of 16×16. The Cerebras plotted line is not complete up to matrix size 500 due to memory limitations on the Cerebras server causing errors for for larger matrix dimensions.

Table 5: Results of GPU Experiment(s)

Thread Block	**Matrix Size**	**Block Size**	**Latency @ 1.23 GHz (ms)**
4	32×32	8	0.05895
4	48×48	12	0.061075
4	64×64	16	9.429825
4	80×80	20	9.702106
4	128×128	32	9.829359
8	64×64	8	0.064673
8	96×96	12	9.392509
8	128×128	16	9.592221
8	160×160	20	9.804902
8	256×256	32	9.599008
16	128×128	8	0.081014
16	192×192	12	0.108829
16	256×256	16	0.149937
16	320×320	20	9.398144
16	512×512	32	9.370005
32	256×256	8	0.394679
32	384×384	12	0.416162
32	512×512	16	1.04053
32	640×640	20	1.499249
32	1024×1024	32	3.647162

Table 6: Results of CPU Experiment(s)

PE Mesh	**Matrix Size**	**Block Size**	**Latency @ 2.1 GHz (ms)**
4	32×32	8	3.53
4	48×48	12	3.87
4	64×64	16	4.95
4	80×80	20	5.31
4	128×128	32	5.97
8	64×64	8	7.80
8	96×96	12	7.21
8	128×128	16	7.69
8	160×160	20	7.81
8	256×256	32	10.56
16	128×128	8	10.35
16	192×192	12	10.24
16	256×256	16	10.34
16	320×320	20	10.56
16	512×512	32	14.65
32	256×256	8	38.11
32	384×384	12	35.47
32	512×512	16	37.24
32	640×640	20	40.83
32	1024×1024	32	46.44

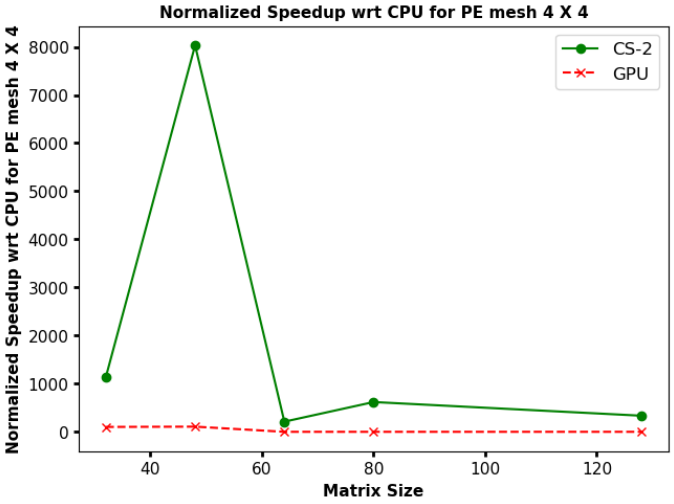

Figure 3: Normalized wrt CPU for PE mesh 4X4

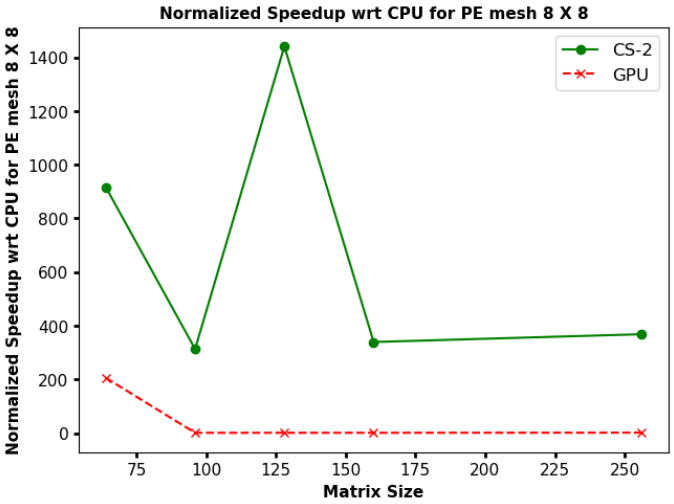

Figure 4: Normalized wrt CPU for PE mesh 8×8

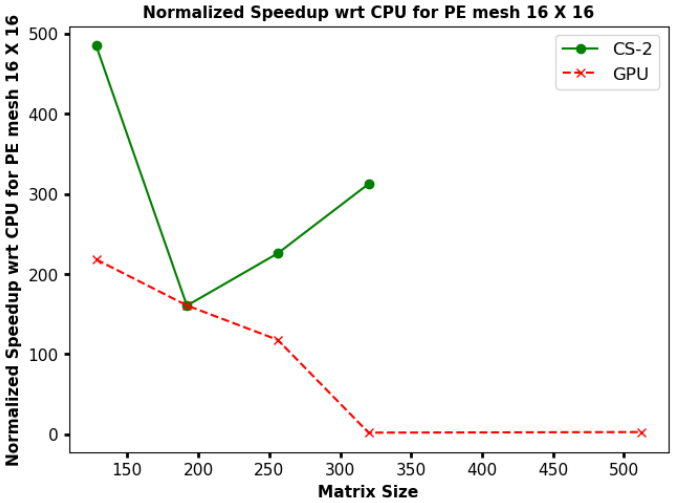

Figure 5: CPU vs GPU Normalized Speedup for PE mesh 16x16

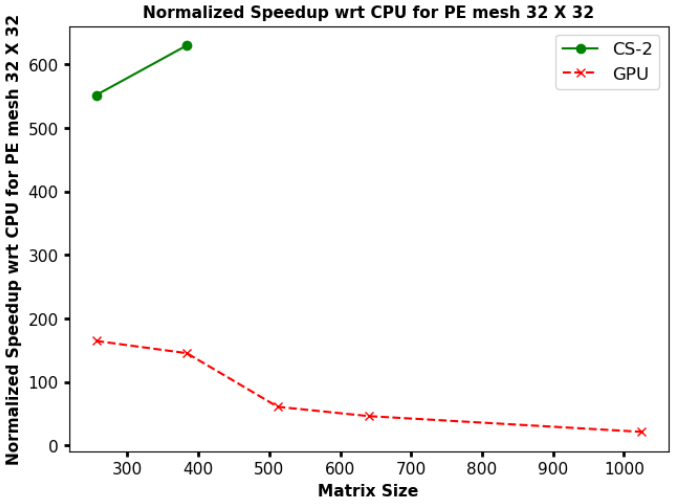

Figure 6: CPU vs GPU Normalized Speedup for PE mesh 32x32

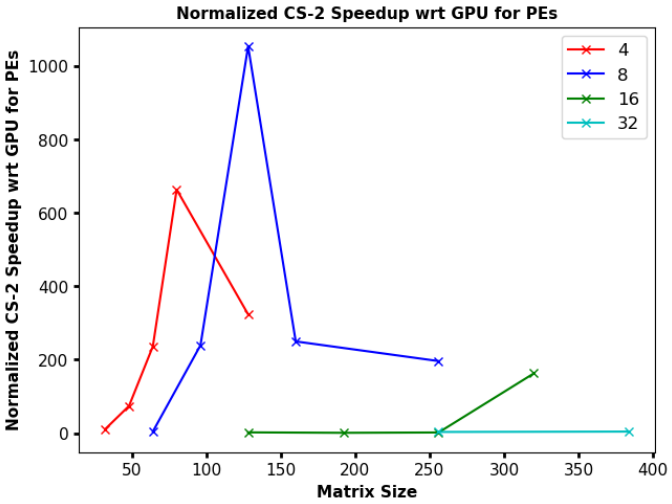

Figure 7: CS-2 Normalized Speedup

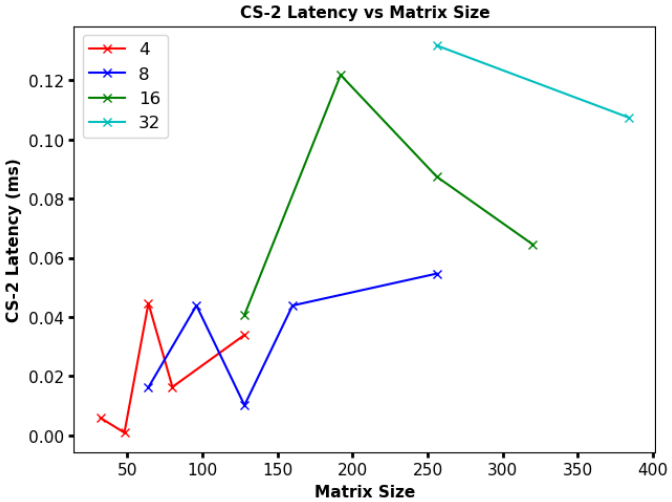

Figure 8: CS-2 Latency

Again in Figure 6, there is significant observable speed-up for Cerebras. Note that there are more missing data points for Cerebras due to the larger PE mesh size, therefore the larger matrices at the lower block sizes.

Figure 7 shows a peak in speedup around matrix size 75 to 150. Latencies of GPUs spikes for larger block sizes likely due to overhead caused by fetching data from main memory. This does not occur for Cerebras because the data resides on the local memory of each PE.

Figure 8 shows that 8×8 and 32×32 processors with dimensions over 200x200 matrix has one of the higher latencies on the graph. This is expected with increased number of PEs and matrix size, there will be an increased number of cycles used to compute each block.

6 Conclusion & Future Work

When matrix multiplication was executed on the Cerebras system, it showed superior performance metrics in terms of latency and cycle counts, confirming the theoretical advantages of the Cerebras architecture. These findings not only support the hypothesis that the Cerebras system can outperform conventional architectures in specific tasks but also highlight the practical implications of using such specialized hardware in real-world applications.

While the current results are promising, we could further explore the acceleration of sparse-sparse matrices to further the understanding and capabilities of matrix multiplication on the Cerebras system. Specifically with sparse-sparse, since the dimensions can be much larger yet there is much room for improvement due to all the zero elements.

7 References

1. J. Jiang, V. Mirian, K. P. Tang, P. Chow and Z. Xing, "Matrix Multiplication Based on Scalable Macro-Pipelined FPGA Accelerator Architecture," 2009 International Conference on Reconfigurable Computing and FPGAs, Cancun, Mexico, 2009, pp. 48-53, doi: 10.1109/ReConFig.2009.30.

2. N. Srivastava, H. Jin, J. Liu, D. Albonesi and Z. Zhang, "MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product," 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Athens, Greece, 2020, pp. 766-780, doi: 10.1109/MICRO50266.2020.00068.

3. Lie, S. (2023, July 23). Cerebras Architecture Deep Dive: First look inside the HW/SW co-design for Deep Learning [updated]. Cerebras.

4. Cerebras Architecture Deep Dive: First Look Inside the HW/SW Co-Design for Deep Learning. (2023, May 22). cerebras.net. April 15, 2024, https://www.cerebras.architecture-deep-dive-first-look-inside-the-hw/sw-co-design-for-deep-learning

5. The Complete Guide to Scale-Out on Cerebras Wafer-Scale Clusters. (2022, September 14). cerebras.net. April 15, 2024, https://www.cerebras.net/blog/the-complete-guide-to-scale-out-on-cerebras-wafer-scale-clusters