Master's Degree in
Artificial Intelligence and Data Engineering

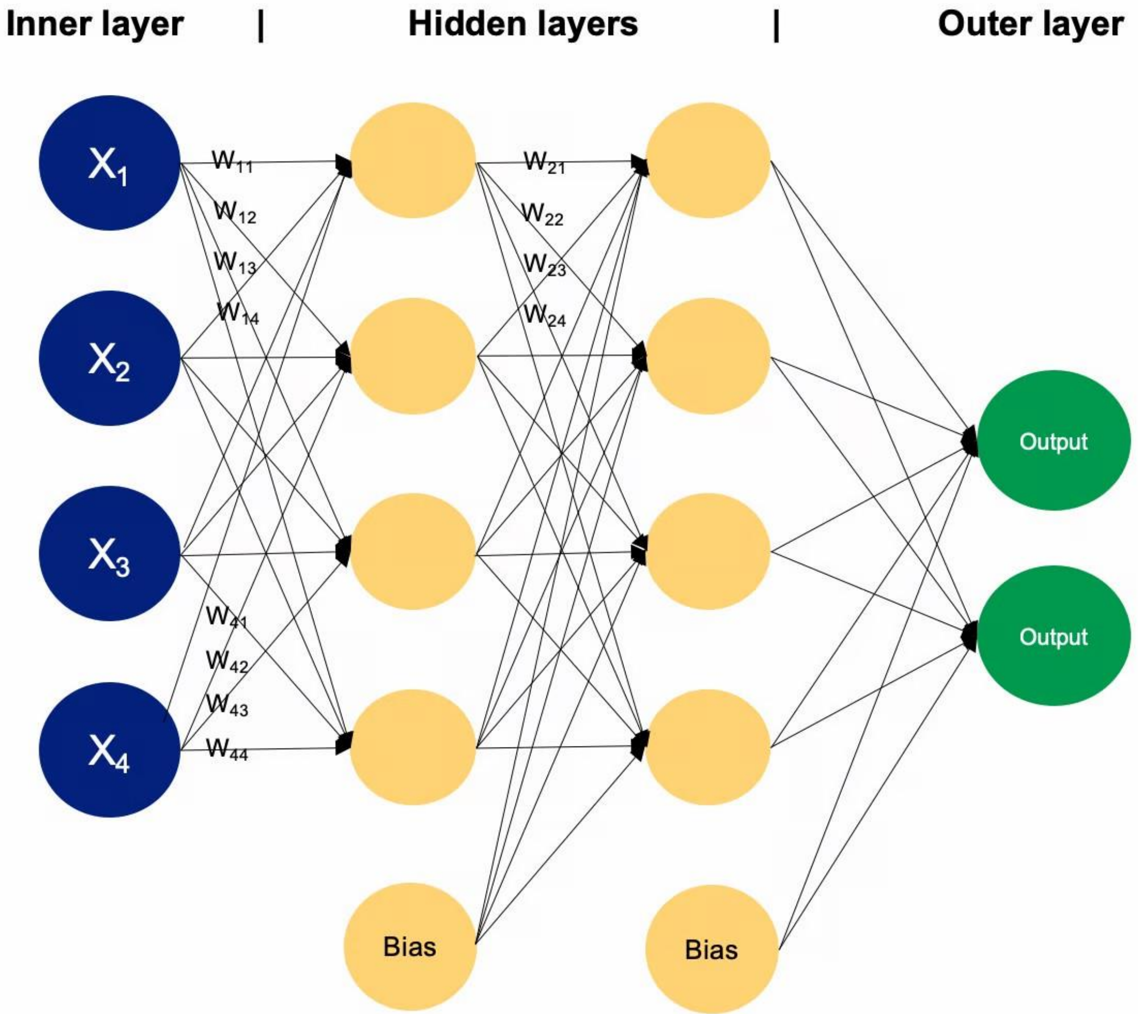
Multi Layer Perceptron for Classification

Academic Year 2024/2025

M. Gemelli – F. Nocella

MLP

A multi-layer perceptron (MLP) is a type of artificial neural network consisting of multiple layers of neurons.



Software Architecture

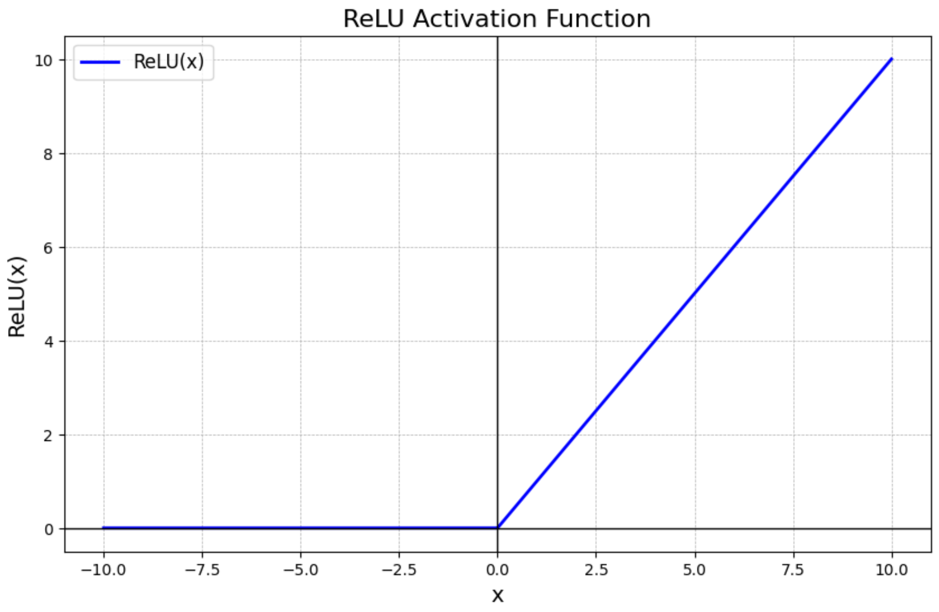
Operations involved:

- Linear layer:

$$\mathbf{Weighted\ sum} = \sum_{\mathbf{i=1}}^{\mathbf{n}} (\mathbf{w_i} \cdot \mathbf{x_i}) + \mathbf{b}$$

- Activation function (ReLU):

$$\boldsymbol{f(x) = \max(0, x)}$$



Case Study: Classification 1/2

X: input data (matrix organization)

- **B rows**: number of records (batch size)
- **512 columns**: number of input features

1 Hidden layer

- **512 input features**
- **2048 output fetaures**

- **Output layer**
  - **2048 input features**
  - **100 output features** (number of classes)

Case Study: Classification 2/2

Parameter	Value/Description
Batch size	$256, 512, 1024, 2048, 4096$
Threads per block	$32, 64, 256, 512, 1024$
Number of Blocks (2D grid)	$\left(\frac{N_x + M_x - 1}{M_x}, \frac{N_y + M_y - 1}{M_y}\right)$
	N: number of elements to be processed
	M: number of threads per block
Hidden layer	X_1: hidden neurons (2048)
	Y_1: batch size
Output layer	X_2: output neurons (100)
	Y_2: batch size

Hardware

Versione 572.61
NVIDIA GeForce GTX 1650

Memory

- Memory Size: 4GB
- Memory Type: GDDR6
- Memory Bus: 128 bit
- Bandwidth: 192.03 GB/s

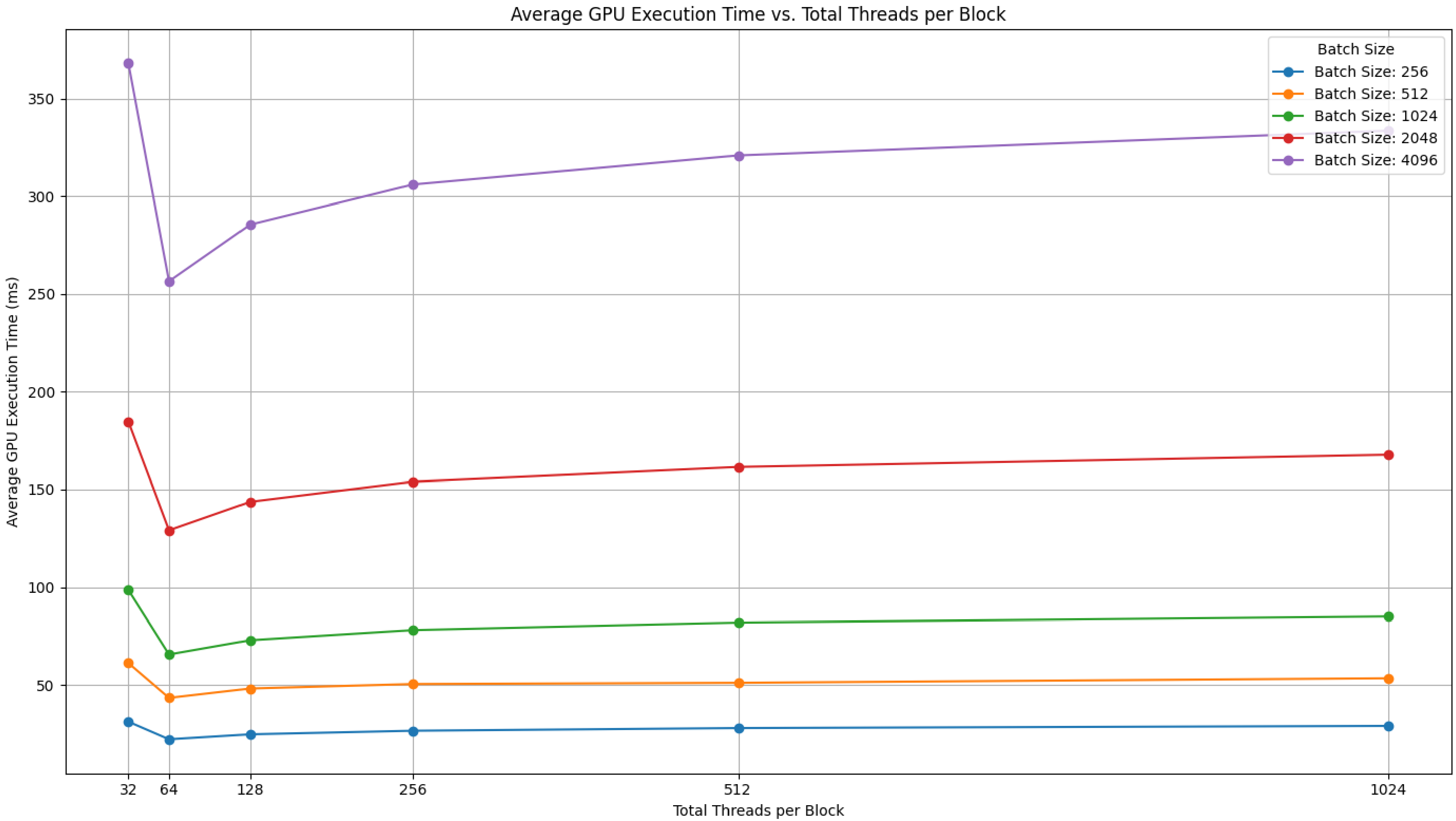
Render Config

- Shading Units: 896
- Streaming Multiprocessor (SM): 14
- L1 Cache: 64 KB (per SM)
- L2 Cache: 1024 KB

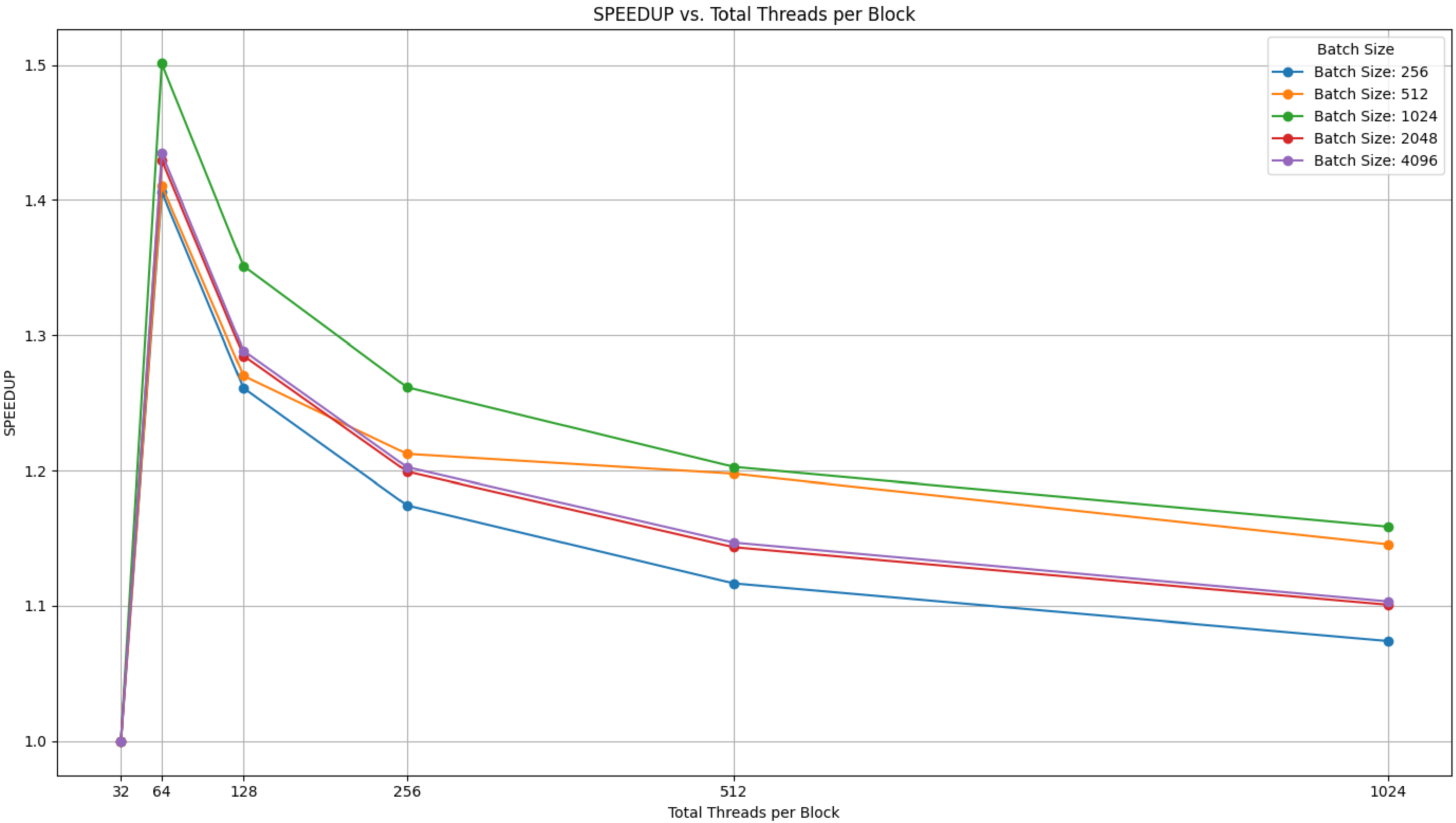
Goals

- Optimize a **MLP**'s performance.
- **Maximize** hardware **resource utilization** (GPU).
- Optimize algorithm for **GPU processing**
- **Maximize** throughput

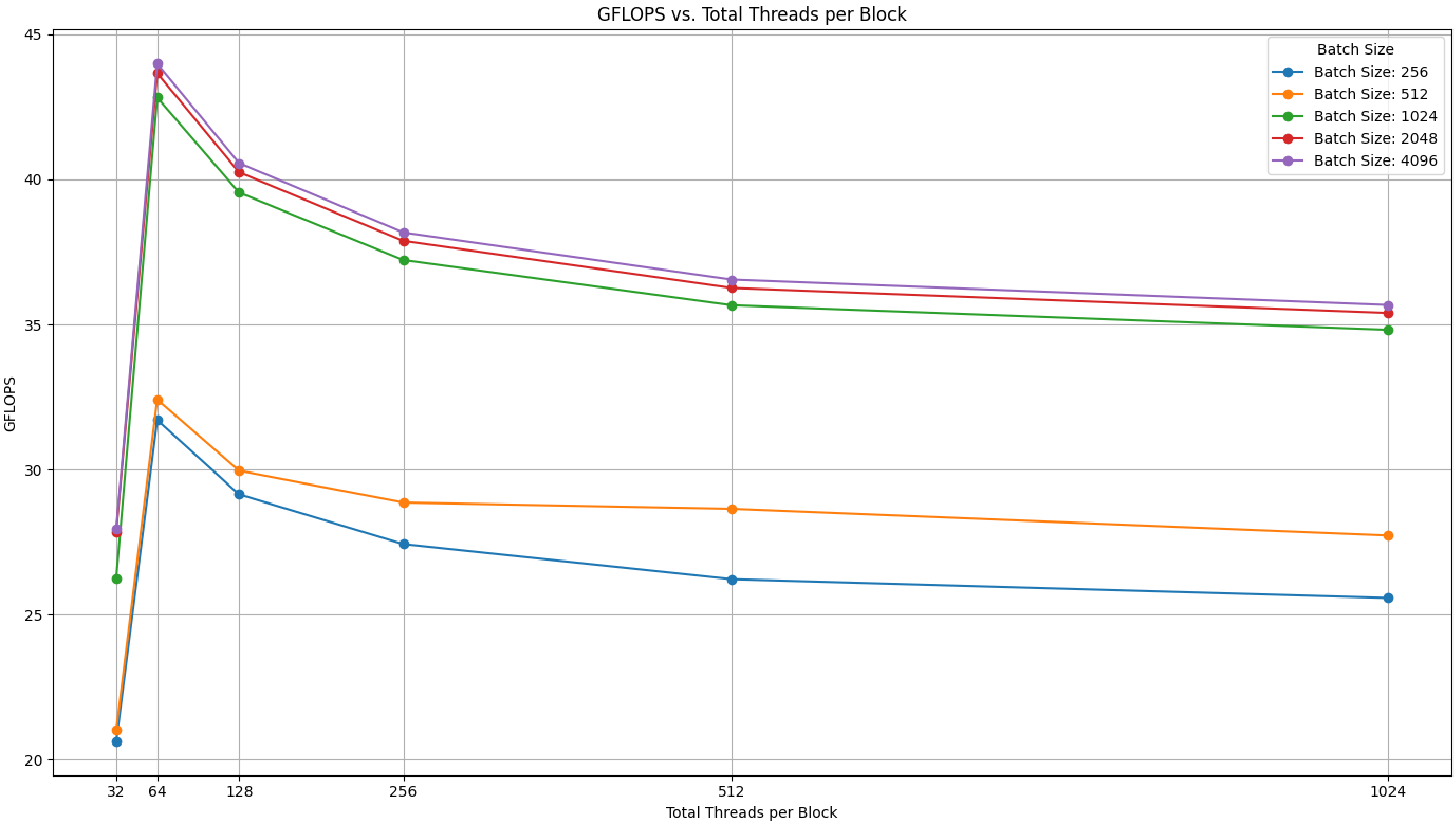
Initial version: Avg. Exec. Time



Initial version: Speed-up



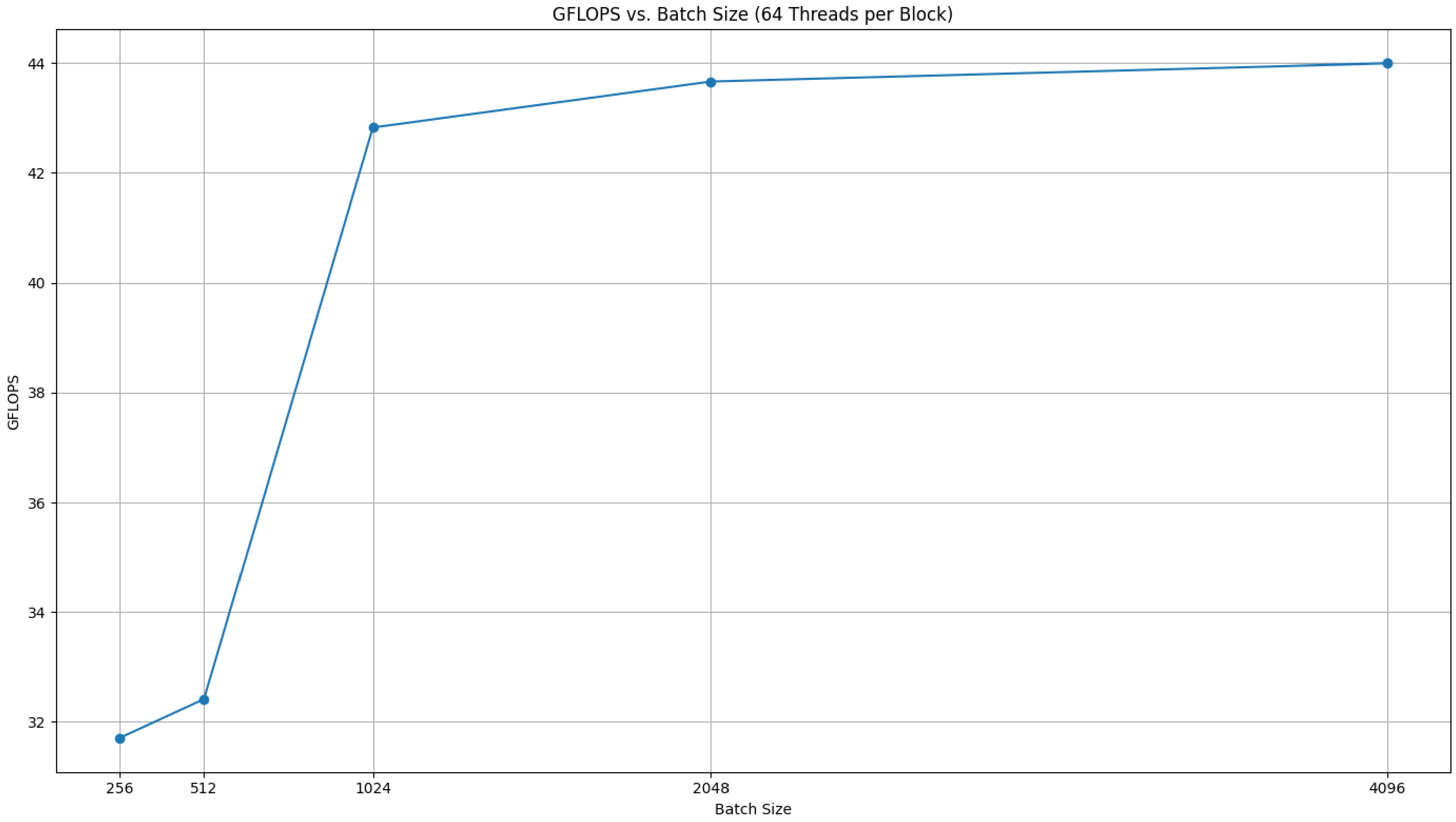
Initial version: GFLOPs



Summary

- Ideal number of threads per block: **64**
- Avg. Exec. Time (ms) with a batch size of 4096 records: **~250 ms**
- Speed-up (with respect to 32 threads per block): **~1.5**
- GFLOPs with a batch size of 4096 records: **~45**

Initial version: Batch scaling



Profiling tool

NVIDIA Nsight Compute

Issues

	L2 Theoretical Sectors Global Excessive
`// W1_global is typically indexed [hidden_idx, k] (or [k, hidden_idx] if column-major, here it impl`	
`// The current W1_global access W1_global[hidden_idx * input_features + k] implies W1 is stored as`	
`sum += X_global[batch_idx * input_features + k] * W1_global[hidden_idx * input_features + k];`	100.00%

⚠ 50.00% of this line's global accesses are excessive.

Memory Throughput [Gbyte/s]	11,80

*** 64 threads per block & batch size of 4096**

Issues

L2 Theoretical Sectors Global Excessive

```
// W1_global is typically indexed [hidden_idx, k] (or [k, hidden_idx] if column-major, here it impl
// The current W1_global access W1_global[hidden_idx * input_features + k] implies W1 is stored as
sum += X_global[batch_idx * input_features + k] * W1_global[hidden_idx * input_features + k];    100.00%
```

⚠ 75.00% of this line's global accesses are excessive.

Memory Throughput [Gbyte/s]	4,43 (-62,49%)

* **256 threads per block & batch size of 4096**

```cuda
__global__ void forward_layer1_naive_kernel(const float *X_global,  // Pointer to the global memory holding the input batch
                                            const float *W1_global, // Pointer to the global memory holding the weights of
                                            const float *b1_global, // Pointer to the global memory holding the biases of t
                                            float *H_global,        // Pointer to the global memory where the output of the
                                            int current_batch_size, // The actual batch size for this specific kernel launc
                                            int input_features,     // Number of input features, matches INPUT_FEATURES.
                                            int hidden_neurons)
{ // Number of hidden neurons, matches HIDDEN_NEURONS.
    // Calculate the global thread indices for the batch and hidden neuron dimensions.
    // blockIdx & threadIdx are CUDA built-in variables.
    // blockDim is the dimension of a thread block.
    int batch_idx = blockIdx.y * blockDim.y + threadIdx.y;  // Identifies the sample in the batch.
    int hidden_idx = blockIdx.x * blockDim.x + threadIdx.x; // Identifies the neuron in the hidden layer.

    // Boundary check: ensure the thread is within the valid range of batch size and hidden neurons.
    // This is important if the number of threads launched exceeds the actual work items.
    if (batch_idx < current_batch_size && hidden_idx < hidden_neurons)
    {
        float sum = 0.0f; // Accumulator for the weighted sum.
        // Compute the dot product of the input features for the current batch sample
        // and the weights corresponding to the current hidden neuron.
        for (int k = 0; k < input_features; ++k)
        {
            // X_global is indexed [batch_idx, k]
            // W1_global is typically indexed [hidden_idx, k] (or [k, hidden_idx] if column-major, here it implies row-majo
            // The current W1_global access W1_global[hidden_idx * input_features + k] implies W1 is stored as (hidden_neur
            sum += X_global[batch_idx * input_features + k] * W1_global[hidden_idx * input_features + k];
        }
        sum += b1_global[hidden_idx]; // Add the bias for the current hidden neuron.
        // Apply the ReLU activation function and store the result in the hidden layer output matrix.
        H_global[batch_idx * hidden_neurons + hidden_idx] = relu(sum);
    }
}
```

Optimization: cuBLAS

- **What it is:** cuBLAS is NVIDIA's high-performance library for Basic Linear Algebra Subprograms (BLAS), specifically optimized for GPUs.
- **Core Function:** It provides a rich set of routines for vector and matrix operations (e.g., dot products, matrix-vector multiplication, matrix-matrix multiplication).
- **Primary Goal:** To accelerate linear algebra computations significantly by leveraging the massive parallel processing power of NVIDIA GPUs.

cuBLAS kernel

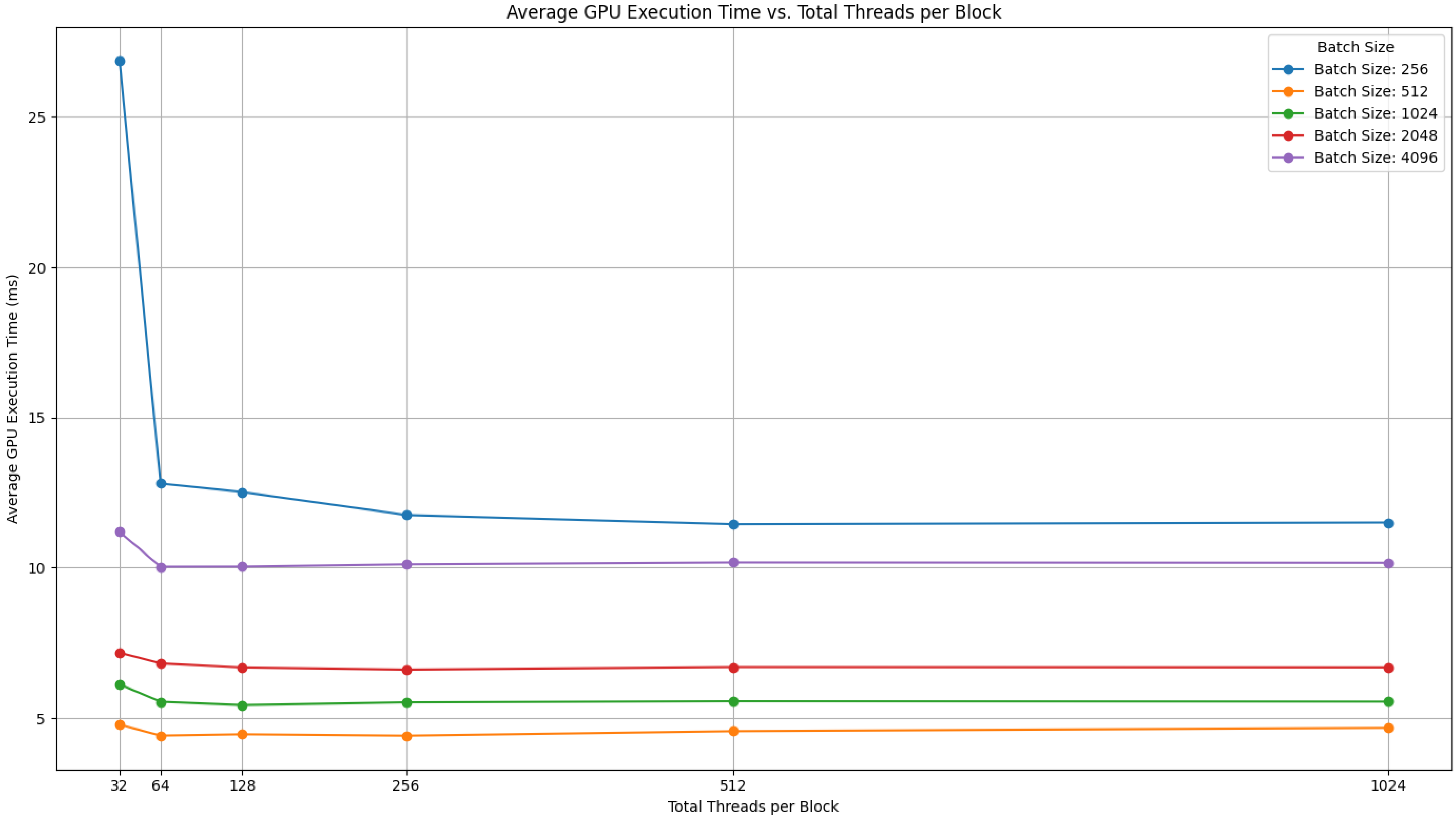
```
cublasSafeCall(cublasSgemm(cublasH,
                           CUBLAS_OP_N, CUBLAS_OP_N,  // No transpose for W1 and X
                           HIDDEN_NEURONS_DIM,        // n (cols of W1, cols of d_H)
                           batch_size_arg,            // m (rows of X, rows of d_H)
                           INPUT_FEATURES_DIM,        // k (cols of X / rows of W1)
                           &alpha,                    // Scalar alpha
                           d_W1, HIDDEN_NEURONS_DIM,  // B_math (W1) and its leading dimension (num_cols of W1)
                           d_X, INPUT_FEATURES_DIM,   // A_math (X) and its leading dimension (num_cols of X)
                           &beta,                     // Scalar beta
                           d_H, HIDDEN_NEURONS_DIM)); // C_math (d_H) and its leading dimension (num_cols of d_H)
cudaSafeCall(cudaGetLastError());                     // Check for errors after cuBLAS call.
```

Custom ReLU & bias kernel

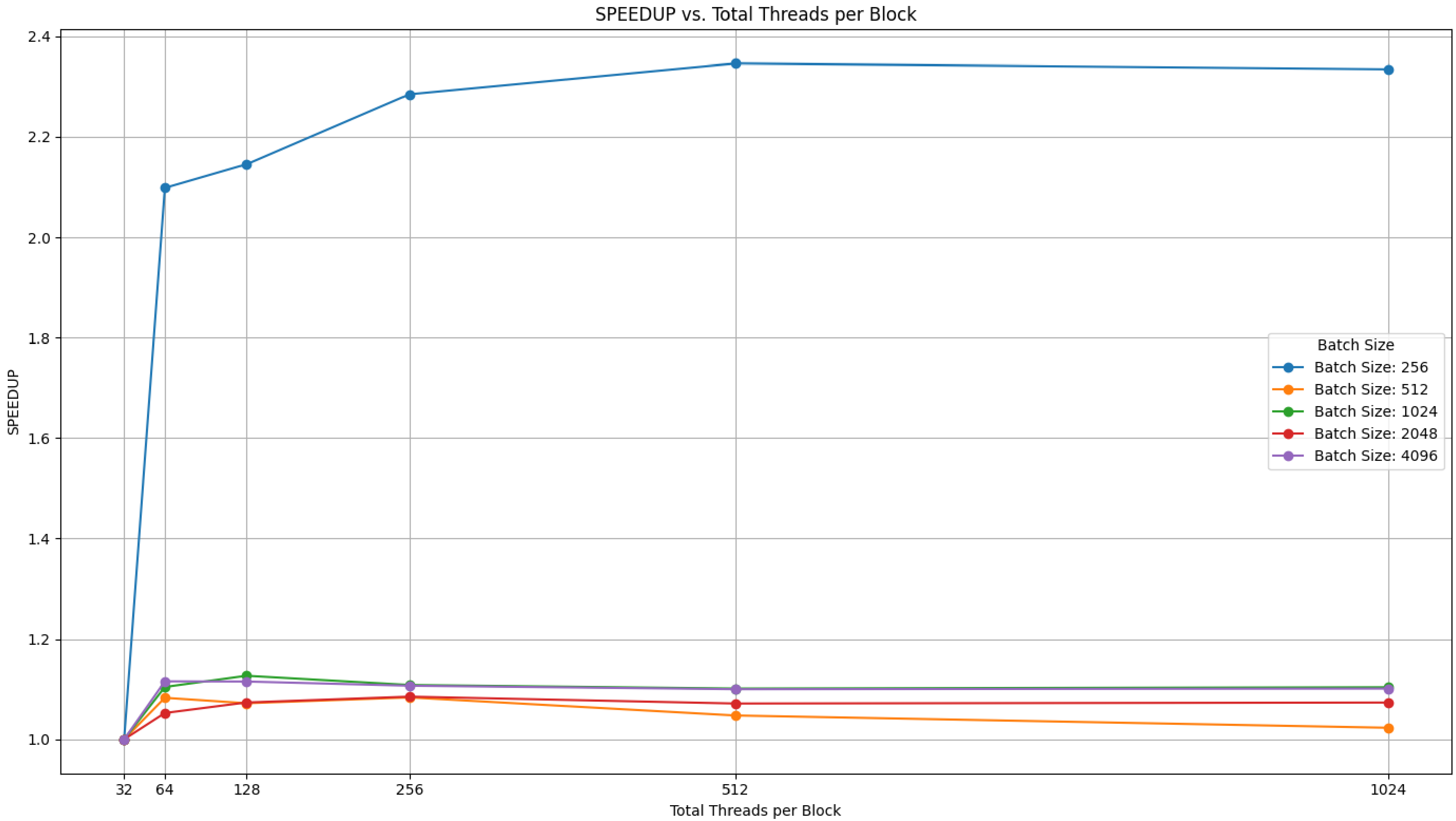
```
__global__ void add_bias_relu_kernel(float *matrix_out, // Pointer to the input/output matrix (num_rows x num_cols) on the device.
                                     const float *bias, // Pointer to the bias vector (1 x num_cols) on the device.
                                     int num_rows,      // Number of rows in matrix_out.
                                     int num_cols)
{ // Number of columns in matrix_out (and size of bias vector).
    // Calculate global thread indices for row and column.
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // Boundary check: ensure thread is within matrix dimensions.
    if (row < num_rows && col < num_cols)
    {
        int index = row * num_cols + col; // Linear index for row-major matrix.
        // Add bias and apply ReLU: matrix_out[index] = max(0, matrix_out[index] + bias[col]).
        matrix_out[index] = fmaxf(0.0f, matrix_out[index] + bias[col]);
    }
}
```

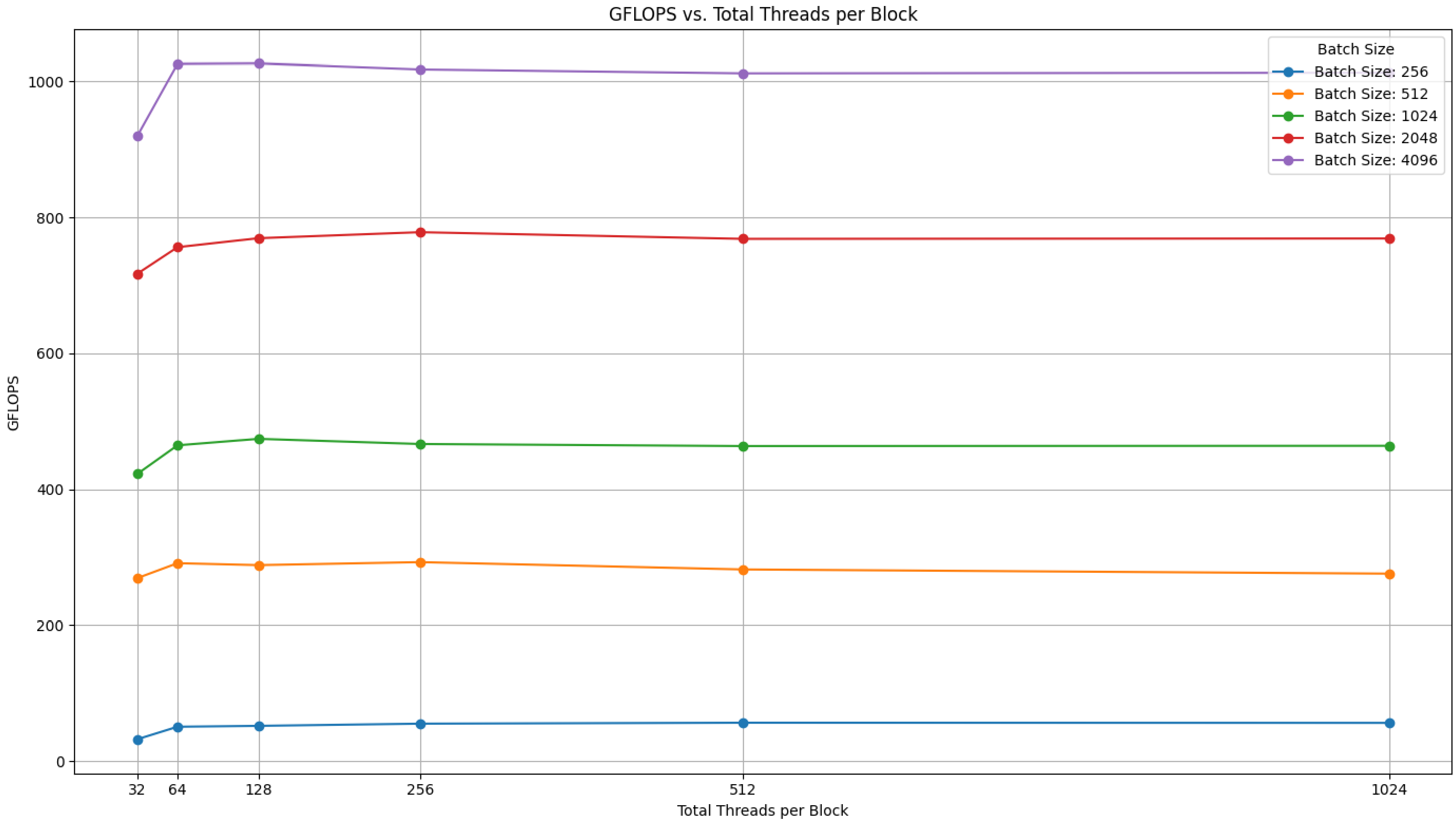
Opt. version: Avg. Exec. Time



Opt. version: Speed-up



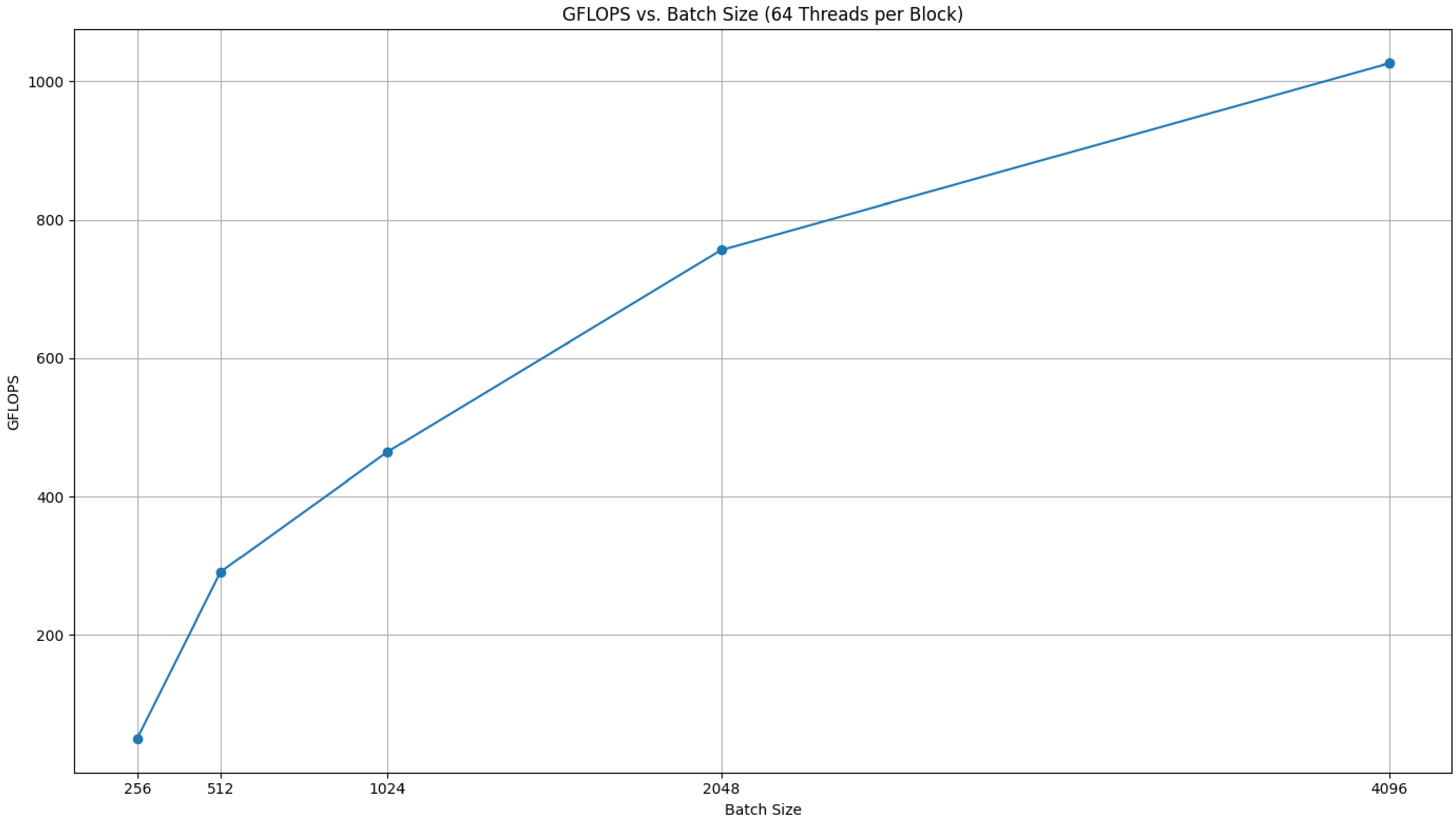
Opt. version: GFLOPs



Summary

- Ideal number of threads per block: **64**
- Avg. Exec. Time (ms) with a batch size of 4096 records: **~10 ms**
- Speed-up (with respect to 32 threads per block): **~1.2**
- GFLOPs with a batch size of 4096 records: **> 1000**

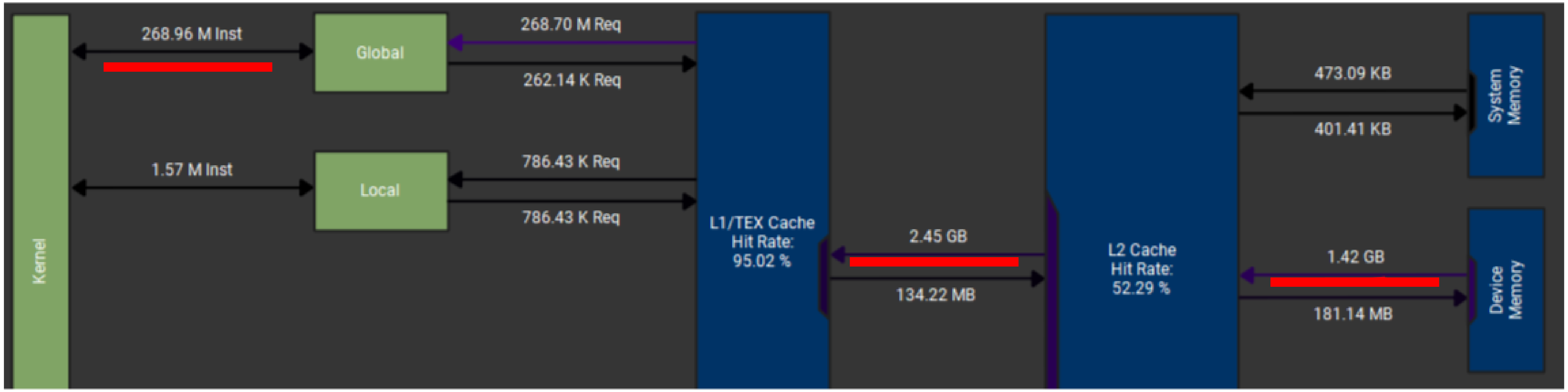
Opt. version: Batch scaling



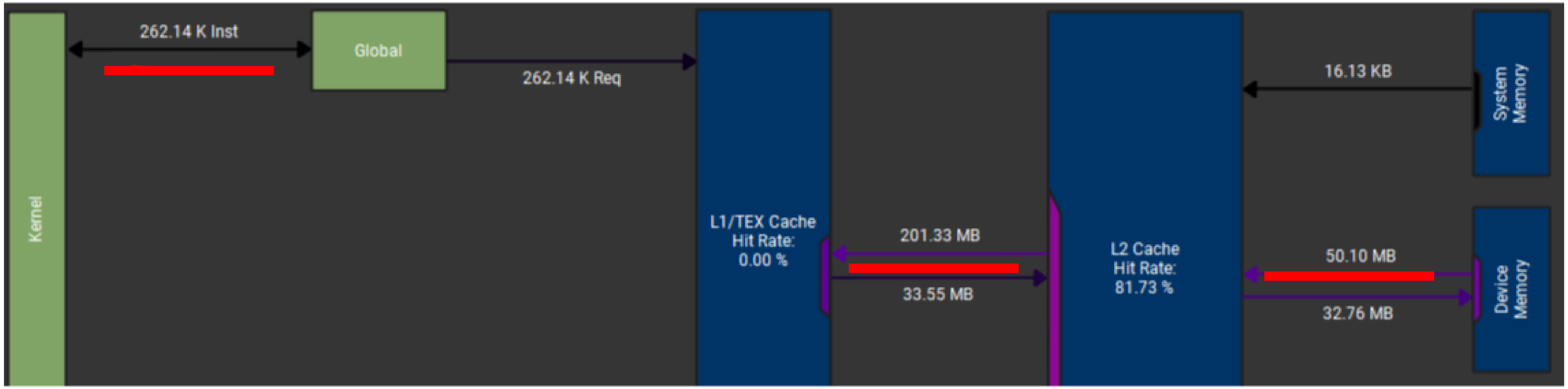
Comparison with old version

Memory Throughput [Gbyte/s]	84,42 (+615,35%)

Naive version



Cublas version



Conclusion

Metric	Result
Execution Time (Duration in ms)	Approximately **25 times** faster (from 250ms to ~10ms)
Computational Throughput (GFLOPS)	Increased by over **22 times** (from ~44 to >1000)
Scaling with batch size	Optimized performance scales effectively with increasing batch size up to 4096, **whereas the initial version stabilized at batch size of 1024.**

*Results related to **64** threads per block & batch size of **4096**