Optimizing Global Memory Accesses in LSTM implementations on GPU

Thesis submitted by

ANANT JAIN
2019MCS2557

under the guidance of

DR. KOLIN PAUL

in partial fulfilment of the requirements for the award of the degree of

Master of Technology

Department Of Computer Science and Engineering
INDIAN INSTITUTE OF TECHNOLOGY DELHI

JUNE 2021

CERTIFICATE

This is to certify that the thesis titled **Optimizing Global Memory Accesses in LSTM implementations on GPU**, submitted by **Anant Jain (2019MCS2557)**, to the Indian Institute of Technology, Delhi, for the award of the degree of **Master of Technology in Computer Science and Engineering**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

DR. KOLIN PAUL
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to our supervisor, Dr Kolin Paul for his valuable guidance and support throughout the project. His patience, motivation and knowledge have provided me a way to do this work. He has introduced me to the field of GPU and CUDA programming. His guidance has helped me to learn this field well. I am thankful for him giving the freedom to explore various aspects of the problem.

I also like to thank Saurabh Tewari for his valuable support. He has helped me to understand various ideas about weight reuse methods for LSTM algorithms. His guidance has helped me in many implementations and experiments. In addition, I also like to thank IIT Delhi for providing all the resources to support this work.

Lastly, I would like to thank my family members and friends for their love and support.

ABSTRACT

Long Short-Term Memory (LSTM) algorithm, a variant of Recurrent Neural Network (RNN) algorithm, is widely used in many applications like natural language processing, speech recognition etc. It is mainly used to handle sequential data and their long data dependencies. As, It achieves better result than RNNs. It is widely used algorithm.

Due to this, Large LSTM models are in trend which have large computational complexities, high energy and power consumption. Also, These large models take large time to process. Due to this, Researchers are building LSTM accelerators for devices like GPUs, AISCs and FPGAs to get better result.

The objective of the thesis was to reduce time taken by LSTM algorithm to process, by reducing the amount of the memory accesses from global memory in GPUs. I have used element-wise and block-wise weight reuse methods to reduce memory accesses from global memory. Though, we are successful in reducing memory accesses (by significant amount,in many cases, about more than 50%) by dependent weight matrices, but, still donot able to reduce time. It is due to sequential nature of weight reuse method. We have implemented various versions of the weight reuse method and have reduced time taken as much as possible (but, not less than desired value). The concepts of CUDA and GPU programming are used for doing all these implementations. Overall, since, global memory accesses are decreasing by these methods in GPUs, this can be used to reduce power consumption or energy consumption in future.

Contents

List of Tables

List of Figures

ABBREVIATIONS

IITD	Indian Institute of Technology, Delhi
MXV	Matrix Vector Multiplication
DNN	Deep Neural Network
LSTM	Long Short-Term Memory
RNN	Recurrent Neural Network
GPU	Graphics Processing Unit
CUDA	Compute Unified Device Architecture
KB	Kilobyte
MB	Megabyte
GB	Gigabyte
FPGA	Field-programmable gate array
ASIC	Application-specific integrated circuit

Chapter 1

INTRODUCTION

1.1 Background

The quest for Artificial Intelligence has lead to growth of the field of the Machine learning. Due to Machine learning, it is possible for computers to do certain tasks without being explicitly programmed to do so. Now, Machine learning is very important field of the computer technology. Also, Deep learning is also a subset of Machine learning, which has helped in various fields like computer vision, speech recognition, medical image analysis, natural language processing, drug design etc.

The Artificial Neuron Networks (ANNs) is the machine learning method which is widely used in many applications. Their structure is like that of the structure of the human brain. Simple neural networks donot have any feedback connection. The Recurrent Neural Network (RNNs), a class of Artificial Neural Network, have feedback connection which is used to remember past information. RNNs use current input and previous output to give present output value. Due to this, RNNs are suitable for processing sequential data. And, they are mostly used in applications like speech recognition, natural language processing etc.

But, due to the vanishing gradient problem, RNNs donot give very good performance. This result in poorer accuracy for the various applications. In such cases, Long Short-Term Memory (LSTM) Network, a class of Recurrent Neural Network are used. It can handle long range dependencies and has very good accuracy. LSTMs are suitable for tasks of speech recognition, time series prediction, anamoly detection in network traffic etc.

Due to its good performance, bigger LSTM model are mostly used with large number of layers or large value of time-steps. Since, LSTM network has very complex network. So, By this, the computational complexity of LSTM network increases further. It contain lots of parameters and complex computation logics. Processing such large LSTM network take lots of time and high power and energy consumption.

Since, Many normal computing system cannot run such big models. There is a need for customized LSTM accelerators. Researchers are proposing ASIC, FPGA and GPU based LSTM accelerators for either optimizing time, power and energy by the LSTM network.

1.2 Motivation and Related Work

The two major problem with the LSTM network in most of the systems (FPGA, ASIC and GPU) are :-

- **Redundant Data Movements :-** A feature of LSTM that some weight matrices are shared by all the cells in LSTM network. And, all these cells have to be processed in the sequential manner, as there is an dependency between two adjacent cells. Given the system has limited on-chip memory, the sequential execution causes repeated loading of same matrices from the off-chip memory (or global memory).
- **Limited off-chip Bandwidth :-** Large working load for each LSTM cell causes severe pressure on off-chip memory bandwidth, thus leading to high execution time, high power consumption and high energy consumption in the systems.

Researchers have developed various FPGA, ASIC and GPU based LSTM accelerators. Due to wide applications of LSTM network, they are needed. The smartphones that we are using, contain small GPUs (or mobile GPUs). Now, Our smartphones[4][7][9] have functionalities of face detection and recognition, image classification and enhancement, natural language translation, voice assistants, human activity recognition, optimal character recognition etc. These hardware LSTM accelerators are used in many handheld devices like smartphones, medical machines etc.

Since, The memory and processing power of many handheld devices are very limited, which causes lots of overheads while processing bigger LSTM model. So, these LSTM accelerators are made in such a way that the required tasks can be done in given resources. Due to this, some FPGA-based LSTM accelerators make use of tiling techniques(or blocking)[3]. In 2016, Jeremy Appleyard[5] gave a method of optimizing LSTM network on GPUs. It contain method of using parallel resources of GPUs efficiently and fusions of various point-wise operations.

Also, In 2020, N.Park[6] introduced a Time-Step Inter-leaved Weight Reuse (TSI-WR) method which resues the weight matrix by interleaving the computations of two adjacent time-steps of a LSTM cell. This method is reducing the amount of DRAM (or off-chip memory) access, thus leading to reduction in power consumption and better throughput.

In this project, The works are done to reduce global memory acceses (or global transactions) of LSTM network on the GPUs. I have used CUDA language for working on GPUs. I have obtained reductions in global transactions, but this didnot translate into reduction in time.

1.3 Thesis Organization

The remaining chapters of this thesis are organized as follows :-

In Chapter 2, we have discussed about the problem definition of this thesis.i.e. reducing time taken by the LSTM model by decreasing the memory accesses of the LSTM network during LSTM inferencing period. We have described about the LSTM neural network and functions of each gates. Also, structure of the Basic LSTM unit is also discussed. Finally, the concepts of GPU and CUDA programming are also described.

In Chapter 3, Weight Reuse method for LSTM model are introduced. For this methods, I have implemented four versions. All these versions are discussed in this chapter.

In Chapter 4, All the results obtained from the weight reuse method, are mentioned. From this result, I have made observations which are discussed in this chapter.

Chapter 5 contains the conclusions, brief summary and future works that can be done to make this work better and useful.

Chapter 2

Problem Definition

2.1 Objective of the Project

Since, large LSTM neural network has lots of parameters and complex computational logics. Processing large LSTM model results in high execution time, high power consumption and high energy consumption. Also, many devices or systems have limited on-chip memory and other resources. The performance of LSTM network in such cases are affected due to redundant data movements and limited off-chip memory bandwidth.

Optimizing LSTM network for all the parameters - time, power, energy, throughput, latency etc simultaneously, may not always possible. So, mostly, according to requirements, LSTM accelerators are designed for optimizing required parameters.(any of above parameters).

In this project, I have tried to reduce execution time by reducing global memory accesses (or global transactions) of LSTM network on the GPU machine.

2.2 Long Short-term Memory Neural Network

Long Short-term Memory (LSTMs) Neural Network, a class of Recurrent Neural Network(RNNs), is widely used algorithm in the field of deep learning. As, RNN has vanishing gradient problem, thus donot give very good accuracy. Due to this, LSTM algorithm is introduced. It is mostly used for processing sequential data. Many Natural language applications uses LSTM algorithm.

LSTM perform better than most of the artificial neural networks and recurrent neural network because it has property of remembering patterns for long duration of time. i.e. it can remember long term dependencies. Thus, LSTM networks are suited for classifying, processing and making predictions based on time series data.

A LSTM unit contain a cell which help in deciding what to remember, what to forget and what to output. A common LSTM unit consists of 4 gates (input gate, forget gate, cell input gate and output gate) and cell state. This cell remembers value over arbitrary times and the gate regulates the flow of information into and out of the cell. The gates help in storing useful information in the cell.

2.2.1 Function of different LSTM gate :-

- **Input gate :-** It is used for adding new information into the cell.
- **Forget gate :-** It is used to remove information from the cell state. The information which is not useful, is removed by this gate.
- **Cell Input gate :-** It is used for inputting values to cell state.
- **Output gate :-** It is used for selecting useful information from current cell state. Then, that useful information is presented in output or hidden state by use of this cell.

2.2.2 Equations representing basic LSTM unit :-

Each LSTM unit follow following six equations :-

1. $i_t = sigmoid(W_{x(i)}.x_t + W_{h(i)}.h_{t-1} + b_i)$
2. $f_t = sigmoid(W_{x(f)}.x_t + W_{h(f)}.h_{t-1} + b_f)$
3. $g_t = tanh(W_{x(g)}.x_t + W_{h(g)}.h_{t-1} + b_g)$
4. $c_t = f_t * c_{t-1} + i_t * g_t$
5. $o_t = sigmoid(W_{x(o)}.x_t + W_{h(o)}.h_{t-1} + b_o)$
6. $h_t = o_t * tanh(c_t)$

where

- **x** - input vector of the layer
- **h** - output vector of the layer
- **i** - vector for input gate value
- **f** - vector for forget gate value
- **g** - vector for cell input value
- **c** - vector for cell state value
- **o** - vector for output gate value
- W_x - weight matrix which is multiplied by the input vector of the layer
- W_h - weight matrix which is multiplied by the previous output vector of the layer
- **b** - bias vector
- **t** - current time step
- **t-1** - previous time step

2.2.3 Diagram of a basic LSTM unit :-

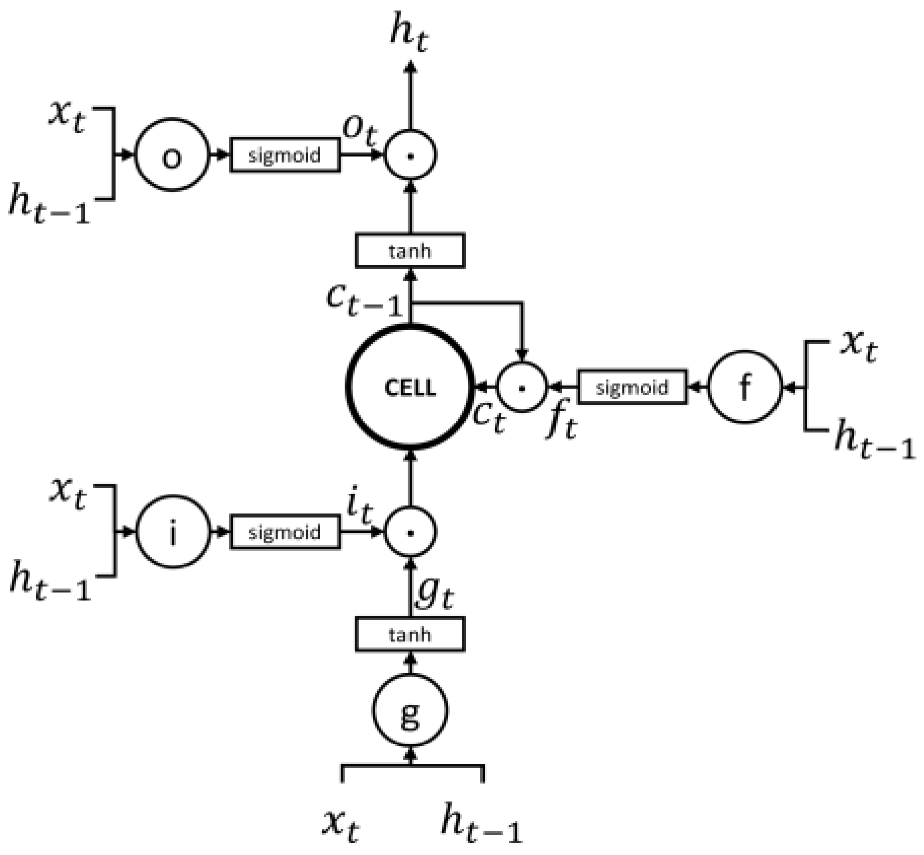


Figure 2.1: The structure of basic LSTM unit

2.2.4 LSTM processing on the GPUs

When a time step of LSTM network is computed, two main types of operations take place, matrix-vector multiplications and element-wise operations. Matrix-vector multiplications are used for computing values for each types of the gates. There are two type of matrix-vector multiplications, $W_x.x$ and $W_h.h_{t-1}$. And, element-wise operations (or point-wise operations) include element-wise additions, element-wise multiplications and computation of activation functions. Here, activation functions are sigmoid and tanh functions. In most cases, weight matrices are very big. Due to this, matrix-vector multiplications take most of the computations in comparison to that of element-wise operations in the LSTM models. These element-wise operations can be computed easily in the GPUs by simply calling a kernel function.

Two matrix-vector multiplications, $W_x.x$ and $W_h.h_{t-1}$ are very different from each other. The multiplications, $W_x.x$ can be done in parallel for all the time steps because these multiplications have no data dependency for x_t between adjacent time steps. But, The multiplications, $W_h.h_{t-1}$ have data dependency because value of h_t depend on the multiplication of W_h and h_{t-1}. So, This multiplication operations cannot be done in parallel. It has to be executed in the serial order. Doing such implementation for the LSTM model, will not be able to utilize the resources of the GPUs.

Since, all the weight matrices, W_x and W_h are stored in the off-chip (or global memory) of the GPUs. And, the on-chip memory (L1 or L2 cache) are of limited size, not enough to store weight matrices fully. Weight matrices W_h and W_x are accessed in each time step, resulting in large memory accesses (or global transactions). As, at each time step, same weight matrices are used. By using proper methods, these global transactions can be reduced. In our work, I have implemented block-wise and element-wise weight reuse method to reduce these global transactions in the GPUs.

2.3 GPU and CUDA programming

Earlier, The Graphics Processing Unit (GPU) are used mostly for graphical processing in the computer devices. As, they can perform large number of operations in parallel, thus giving large throughput. And, the Central Processing Unit (CPUs) have architectural limitation. So, it is decided to use GPUs for general purpose, introducing the concept of General Purpose Graphics Processing Units (GPGPUs). Nowadays, these GPGPUs are used in many fields, like machine learning, deep learning, image processing, 3D reconstruction etc.

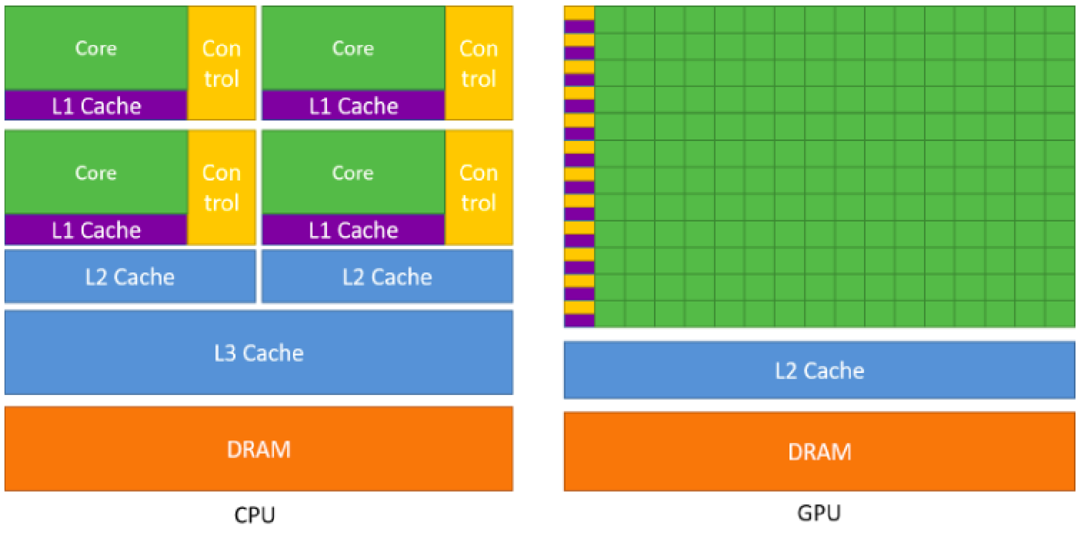


Figure 2.2: The GPU uses more transistors for data processing than CPU

The main advantage of the GPU over CPU is that it provide higher instruction throughput and memory bandwidth with similar price and power resources. There are differences between the GPU and CPU as both are designed with different goals. The CPU can execute a sequence of operations, called threads, as fast as possible and can execute few threads (about 10) in parallel. But, the GPU can execute thousands of threads in parallel. Since, the GPUs are good for highly parallel operations. Due to this, more transistors are devoted for data processing than data caching and data flow.

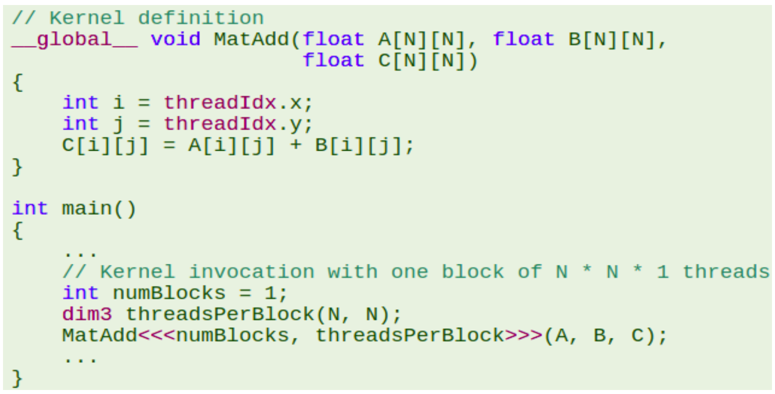
Generally, systems are designed with combining parts of CPUs and GPUs, which help in maximizing the throughput and overall performance. In November 2006, NVIDIA (a leading chip manufacturer company) introduced CUDA (Compute Unified Device Architecture), a general purpose parallel computing platform and programming model. It is used for using parallel compute engine in NVIDIA GPUs to solve very complex computations in an efficient way.

2.3.1 Programming Model in CUDA programming

CUDA[1] provide software developers and software engineers a platform to use CUDA enabled GPUs for general purpose computing. CUDA is designed in a way to work with programming languages like C, C++ and Fortran. This makes it developers to use CUDA for doing parallel programs.

The programming model of CUDA is very different from that of other languages. It has various features like kernel functions, thread organisation, shared memory and specialized synchronization functions. All these concepts of CUDA are discussed below :-

Kernel function



```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Figure 2.3: Example of kernel function. Here, Each of the N threads perform VecAdd() which performs pairwise addition

In CUDA, the function defined for doing certain parallel task is called kernel. It is defined using **__global__** declaration. And, the number of threads (light weight process) that can execute for a given kernel call is specified by the $\lll Blocks, ThreadPerBlock \ggg$.Here, each thread executes its work, with its unique thread id.

Thread Organisation :-

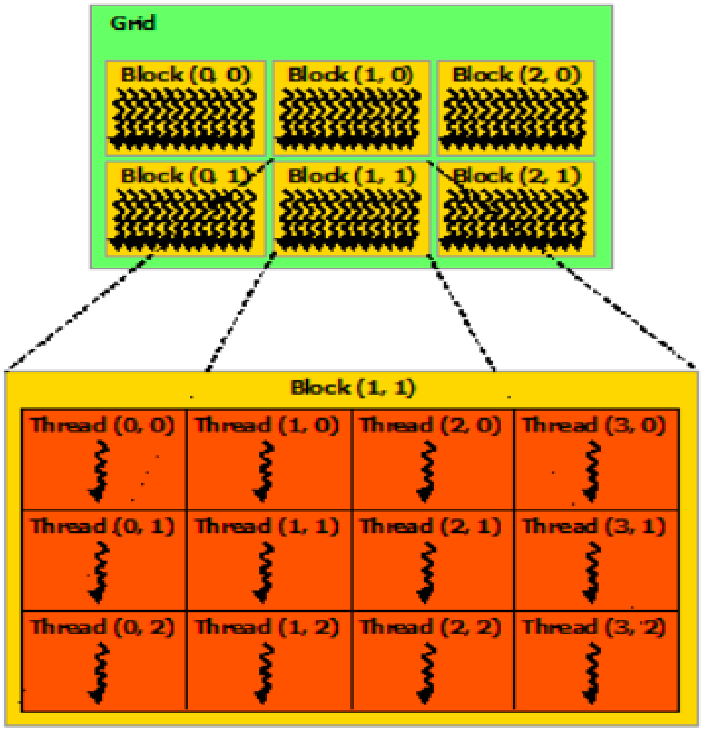


Figure 2.4: Grid of Thread Blocks

Each kernel function is launched as a grid of thread blocks. The thread organization of kernel function is as follows :-

1. A grid is a 3D array of thread-blocks (gridDim.x, gridDim.y and gridDim.z)
2. A thread-block is a 3D array of threads (blockDim.x, blockDim.y and blockDim.z).
3. A thread is an abstract entity that represents the execution of the kernel. Each thread is identified by the values of threadIdx.x, threadIdx.y and threadIdx.z.

The numbers of threads that can be present in a thread block has a upper limit. In most GPUs, the maximum number of threads in a thread block is 1024.

Memory Organisation :-

CUDA programming model has different types of memories.

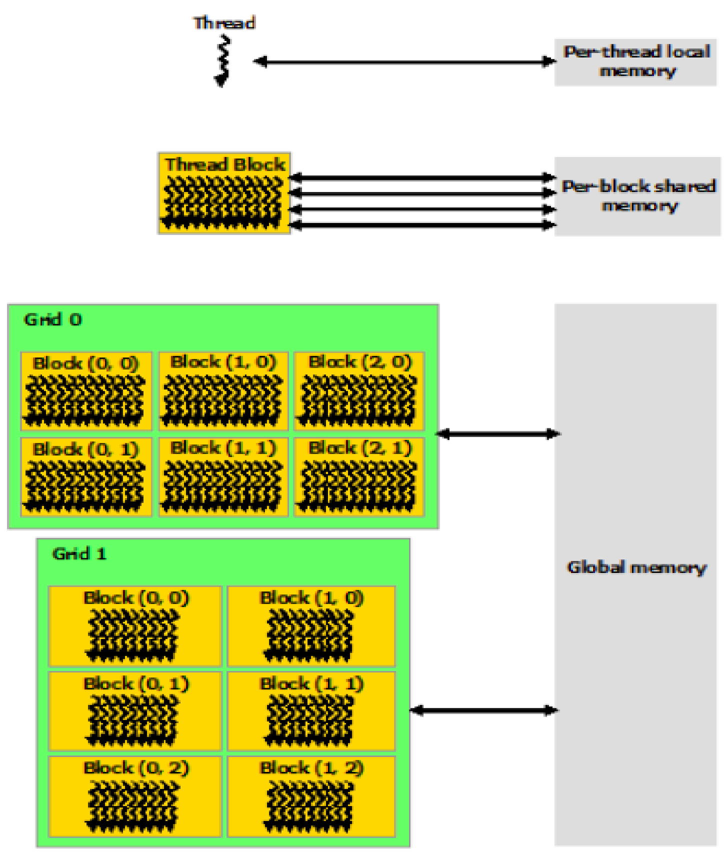


Figure 2.5: Memory Hierarchy in CUDA programming Model

- Each thread has its own private local memory (registers). It is the fastest memory in GPU. It is used for faster data processing.
- Each thread block has shared memory. This shared memory can be shared by all the threads in a block. It is used for data communication and synchronization.
- Also, There is a global memory (called as off-chip memory). It is very large. i.e. in GBs. It is the main means of communicating data between the host and the device. It is visible to all the threads of all the blocks.
- Texture memory is read-only memory and used for 2D spatial locality. They are very small in size.(in KBs)
- Constant memory is also read-only memory. It is used to store constant values which will not change throughout the program. They are very small in size.(in KBs)

All these memories are used only when kernel function is called.

Heterogeneous programming

CUDA programming model has feature of heterogeneous programming. In heterogeneous programming, multiple types of devices can work together. In case of CUDA, CPU and

GPU can work together. It work together in asynchronous form. Here, CUDA threads work on different device which operates as a coprocessor to host running on C++ (or CPU) program. i.e When, kernel executes on the GPU, then rest of the program run on the CPU.

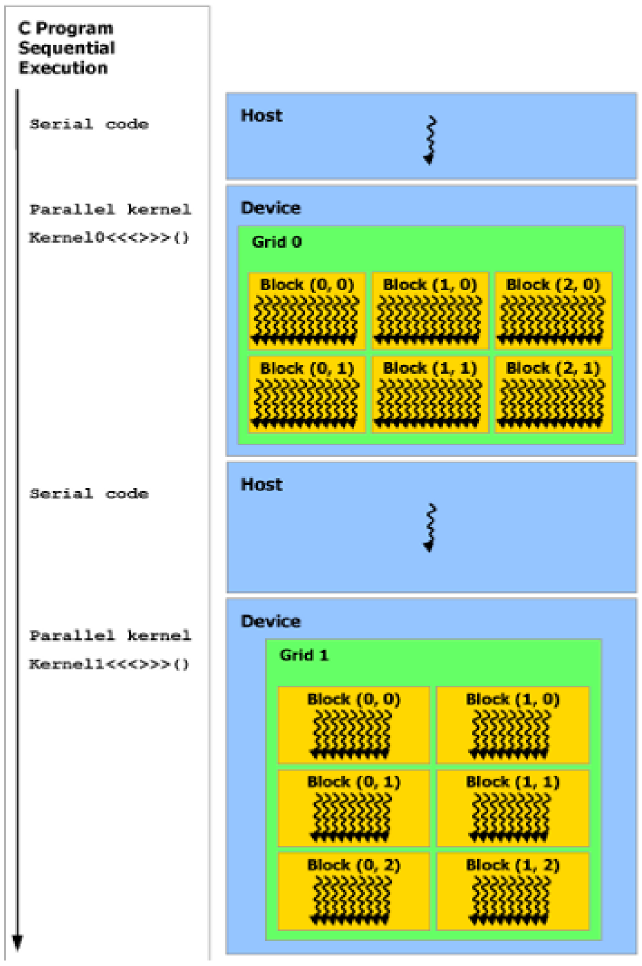


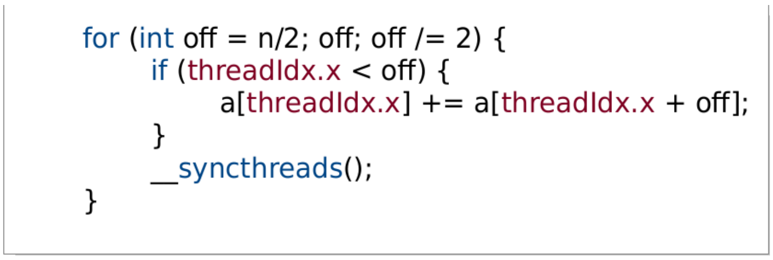
Figure 2.6: Heterogeneous programming in CUDA programming model. Here, serial code run on the host, while parallel kernel code run on the device.

Both the host and device maintain their own separate memories , called as host memory and device memory respectively. Programmers have to manage all these memories to get benefit of processing power of GPU.

Barrier Functions :-

In CUDA programming model, barrier functions are used to provide the facility of synchronization. Here, barrier is a point in the program where all the threads have to reach so that other threads can proceed. The mostly used barrier function is __syncthreads().

The __syncthreads() help in synchronizing threads in the same thread block. This function is mostly used for calculation of sum of all the array elements, prefix sum and other operations. Explicitly, there is no global barrier function in CUDA model which can



```
for (int off = n/2; off; off /= 2) {
      if (threadIdx.x < off) {
            a[threadIdx.x] += a[threadIdx.x + off];
      }
      __syncthreads();
}
```

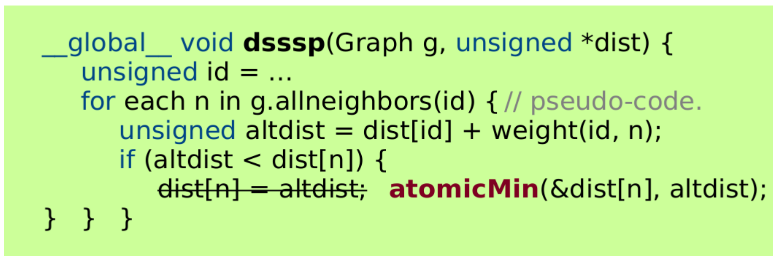
Figure 2.7: Using __syncthreads() function for calculation of sum of all the elements of array

be applicable on all the threads of all the blocks. But, the global barrier function can be simulated by using proper method.

Other barrier functions are __threadfence_block and __threadfence().

Atomic Functions :-

In GPU, many threads execute simultaneously. If these threads operate on same variable (shared or global), then there can be data race, which will lead to data inconsistency and incorrect output. To avoid this, CUDA programming model has atomic functions. These atomic functions operate an atomic operation on shared or global variable in such a way that variable is read first, then it is updated and its new value is written to it. In whole process, no other thread has the access of the variable except the thread which is applying atomic function on it.



```
__global__ void dsssp(Graph g, unsigned *dist) {
    unsigned id = ...
    for each n in g.allneighbors(id) { // pseudo-code.
        unsigned altdist = dist[id] + weight(id, n);
        if (altdist < dist[n]) {
            ~~dist[n] = altdist;~~  atomicMin(&dist[n], altdist);
}  }  }
```

Figure 2.8: Using Atomic Function, atomicMin for shortest path computation

They need hardware support. The common atomic functions are atomicAdd, atomicInc, atomicMin, atomicCAS etc.

- **atomicAdd :-** It is used to add given value to value of the variable atomically.
- **atomicInc :-** It increments the value of given variable by 1 atomically. If the value reach the threshold value, the value of variable becomes 0.
- **atomicMin :-** It compares the value of variable with given value atomically. If the given value is less than that of value of variable, then value of variable is updated with new value.

- **atomicCAS :-** It compares the value of variable with given first value atomically, if they are equal. Then, the value of variable is updated with given second value. And, The old value is returned.

These atomic functions are used for implementing global barrier function, for ensuring mutual exclusion, for implementing locks, shortest path calculation between two points etc.

2.3.2 Flow of CUDA program

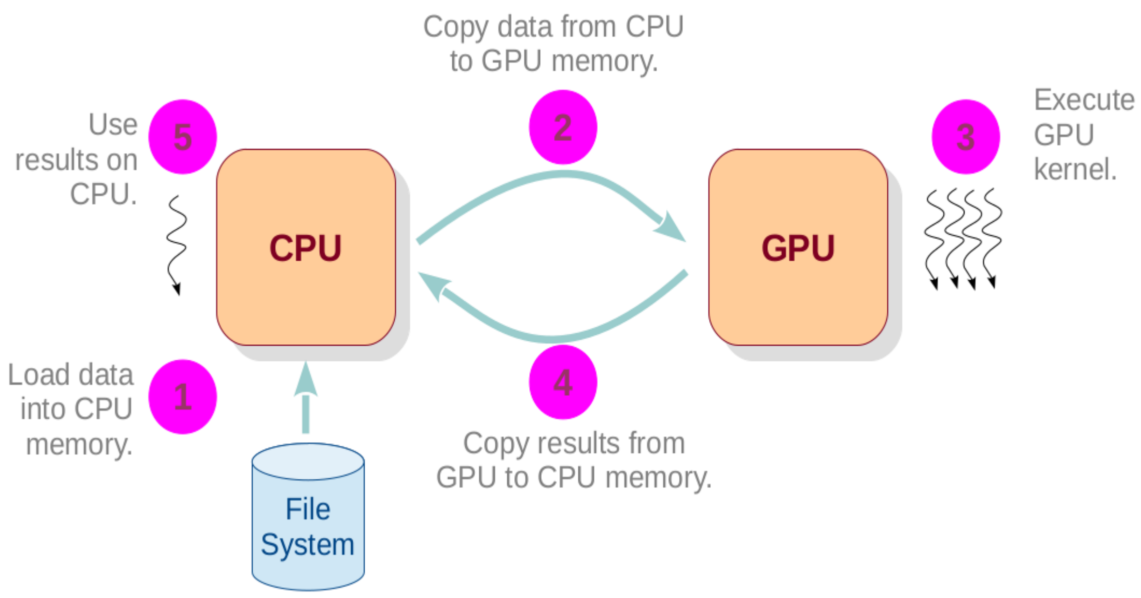

Copy data from CPU to GPU memory.
2
Use results on CPU.
5
Execute GPU kernel.
3
CPU
GPU
Load data into CPU memory.
1
4
Copy results from GPU to CPU memory.
File System

Figure 2.9: Typical Flow of CUDA program

The typical flow of CUDA program is as follows :-

1. **Loading data into CPU memory** :- In CUDA program, firstly, the required data is loaded from file system or disk into the CPU memory.
2. **Transferring data from CPU memory into GPU memory :-** After loading of the data into CPU memory, the required amount of memory is allocated into the GPU memory. Then, the loaded data is copied into the allocated memory of the GPU.
3. **Executing GPU kernels :-** After getting the required data in the GPU memory, parameter of calling kernel function like number of blocks and threads per block are calculated. Then, the GPU kernels are executed.
4. **Copying result from GPU memory into CPU memory :-** After execution of kernel functions, its results are stored into GPU memory. As, this results are to used by CPU, so they are copied into the CPU memory from the GPU memory.
5. **Using results on CPU :-** After getting the results from the GPU memory, the CPU can use these results for further processing.

After all the processing in the CUDA program, the allocated GPU and CPU memory need to be deallocated.

2.3.3 Running and Profiling CUDA programs :-

In this program, we have implemented Weight Reuse LSTM algorithm in CUDA language on GPU machine. We have described how to run and profile CUDA program[2] on GPU machine below :-

Running CUDA program :-

For running CUDA program, commands are as follows :-

- **For compilation :-** *nvcc -o a.out answer.cu*
- **For execution :-** *./a.out*

Computing time taken by CUDA program to run :

The time for particular code or function can be computed in following manner :-

```
struct timeval t1, t2;
gettimeofday(&t1, 0);

// some important kernel function or code //

gettimeofday(&t2, 0);
double time = (1000000.0*(t2.tv_sec-t1.tv_sec) + t2.tv_usec-t1.tv_usec)/1000.0;
```

Here, **time** = time taken for running the program in milliseconds (approx value)

Important commands for profiling of CUDA programs on the GPUs :-

Nvprof profiler tool of CUDA tootkit is used for knowing various statistics related to CUDA program.

- **metrics** :- used for computing the given metrics for the given program
  nvprof –metrics all ./a.out

This command give the value of all the metrics like global load transaction, global store transaction, shared load transaction, shared store transaction etc. This commands give the value of about 162 metrics. We can do profiling for only useful metrics and required metrics as per our requirements.

- **unified-memory-profiling** :- It counts each process on device. Also, it compute the time taken by each process.
  nvprof –unified-memory-profiling off ./a.out

Chapter 3

Weight Reuse Method

3.1 Introduction

As discussed in the previous chapter, our objective is to reduce time taken by LSTM model by reducing global memory accesses (or global transactions) of the output weight matrices W_h of the LSTM network. For this, Saurabh Tewari has proposed a weight reuse method[8] which can be applied element-wise and block-wise on the LSTM network. He has applied this method on FPGA. I have applied this method on GPU. The architecture of the FPGA is different from that of GPU. The CUDA platform is used for applying this method on GPU. For applying this method, four versions are made. In this chapter, conventional method of LSTM algorithm for GPU is mentioned first. Then, this weight reuse method is discussed. In the end, the details and implementations of all the four versions are mentioned.

3.2 Conventional method for LSTM algorithm on GPU

The Conventional LSTM algorithm for GPU call kernel functions for each operation. In LSTM algorithm, there are eight matrix vector multiplications for one time step, four using input weight matrices(W_x) and four using output weight matrices(W_h). First, eight kernel functions are called for doing these multiplications. Then, kernel functions are called for calculating values of all the gates,i.e. input, forget, cell input and output gate. After the calculation of these gates, the kernel functions are called for calculating cell state vector and present output vector. Then, this process is repeated for all the time steps. The LSTM equations for a single time steps are as follows :-

1. $i_t = sigmoid(W_{x(i)}.x_t + W_{h(i)}.h_{t-1} + b_i)$
2. $f_t = sigmoid(W_{x(f)}.x_t + W_{h(f)}.h_{t-1} + b_f)$
3. $g_t = tanh(W_{x(g)}.x_t + W_{h(g)}.h_{t-1} + b_g)$
4. $c_t = f_t * c_{t-1} + i_t * g_t$
5. $o_t = sigmoid(W_{x(o)}.x_t + W_{h(o)}.h_{t-1} + b_o)$
6. $h_t = o_t * tanh(c_t)$

where * represent the element-wise multiplication. **x** is the input vector. **c** and **h** are the cell state and output vector. **t** and **t-1** represent the current time step and previous time step. **i**,**f**,**g** and **o** are the input, forget, cell input and output gate vector. W_x and W_h represent the input weight matrices and output (or hidden) weight matrices. **b** represent the bias vector. Here, The dimension of input vector **x** and output vector **h** are L and N. The dimension of input weight matrices W_x, output weight matrices W_h and bias vector are $N \times L$, $N \times N$ and N respectively.

3.3 Weight Reuse method for LSTM algorithm on GPU

This method reduces the global memory accesses (or global transactions) of output weight matrices W_h of the LSTM network. In each time step, computation for current output vector h_t is done completely and computation for next output vector h_{t+1} is done partially. For time step t, it is assumed that partial computation for present output vector h_t is done in previous time step $t-1$. And, in this time step t, remaining computation is done to get full value of present output vector h_t by using output weight matrices W_h. At this time step t, while using full computation of h_t, partial computation for next output vector h_{t+1} also get done by reusing elements of W_h. It saves about half of global memory accesses of output weight matrices W_h.

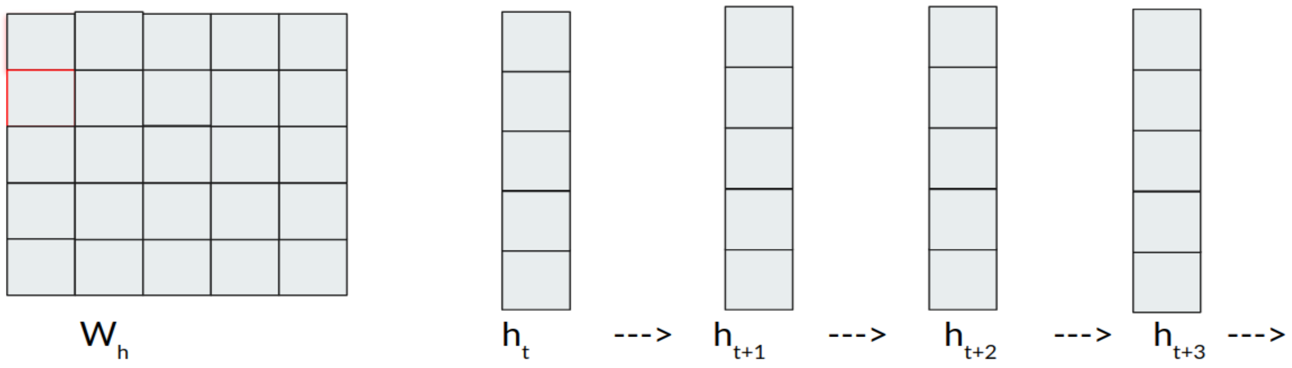


Figure 3.1: Output weight matrix W_h. And, Here, current output vector depend upon previous output vector

In this method, there are two conditions. In first condition, the lower diagonal and diagonal elements of weight matrices W_h are used for computing present output vector. In second condition, the upper diagonal elements of weight matrices W_h are used for computing present output vector. After completion of each time step, value of condition changes. Initially, at first time step, first condition is followed.

3.3.1 Steps in the Weight-Reuse Method :-

Here, the dimension of weight matrix W_h is N * N, and the dimension of output vector (hidden vector) h_t is N. The matrix vector multiplication of $W_x.x$ is done before the matrix vector multiplication of $W_h.h_{t-1}$. **Initially, the cell state and previous output vector (hidden vector) are zero vector.** Assume , the variable **cond** represent condition of LSTM layer. If *cond* is 0, first condition is followed. In *cond* is 1, second condition is followed. Initially, value of *cond* is 0. Following are the steps in the weight reuse method :-

1. When **cond** $=$ **0** :-

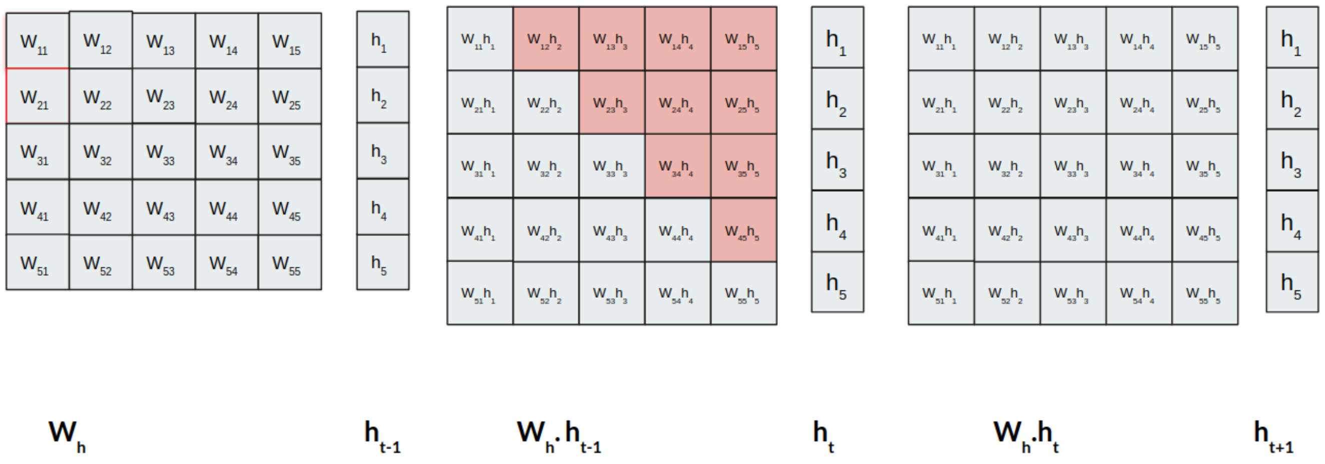


Figure 3.2: (Initial) The only lower diagonal and diagonal elements of weight matrix W_h will be used for computing h_t. The upper diagonal elements of W_h have been multiplied with elements of h_{t-1} in earlier time step.

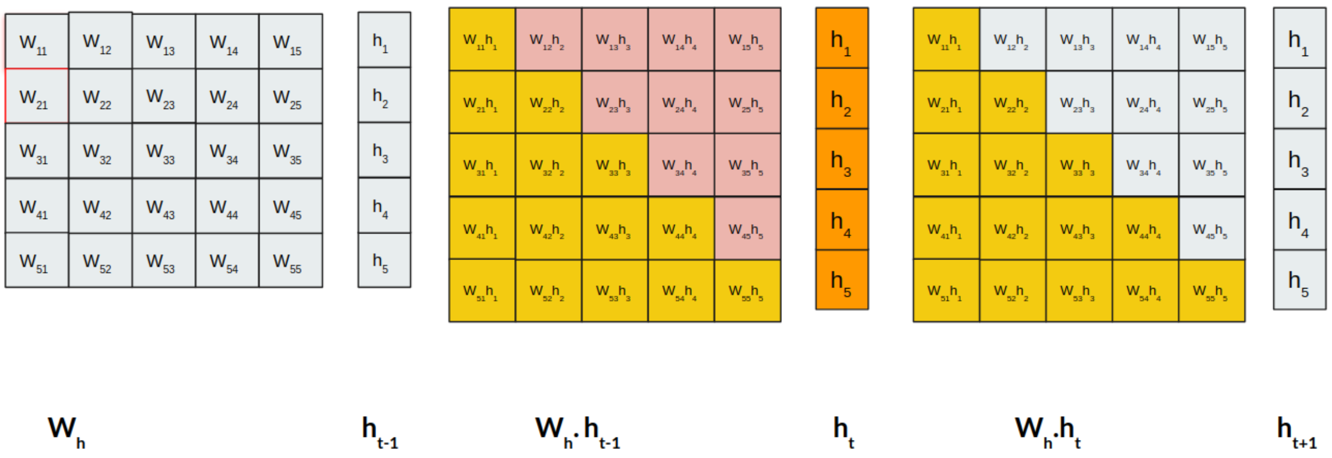


Figure 3.3: h_t is obtained after the matrix multiplication $W_h.h_{t-1}$. And, for h_{t+1}, The lower diagonal and diagonal operations of matrix multiplication $W_h.h_t$ are done in this time step.

In matrix vector multiplication of $W_h.h_{t-1}$, only lower diagonal and diagonal elements of W_h are used, because the upper diagonal elements have been used. So, collectively, whole multiplication of $W_h.h_{t-1}$ get done to produce present output vector h_t. As, the computation of h_t element with particular index get done, then that h_t element of particular index can be used for partial computation of h_{t+1} (next output vector)element with that particular index. Here, elements of matrices W_h are reused. When, the computation of whole h_t get done, then, value of *cond* get changed to 1. And, in next iteration, h_{t+1} will be our current output vector. And, h_{t+2} will be next output vector.

2. When **cond** = **1 :-**

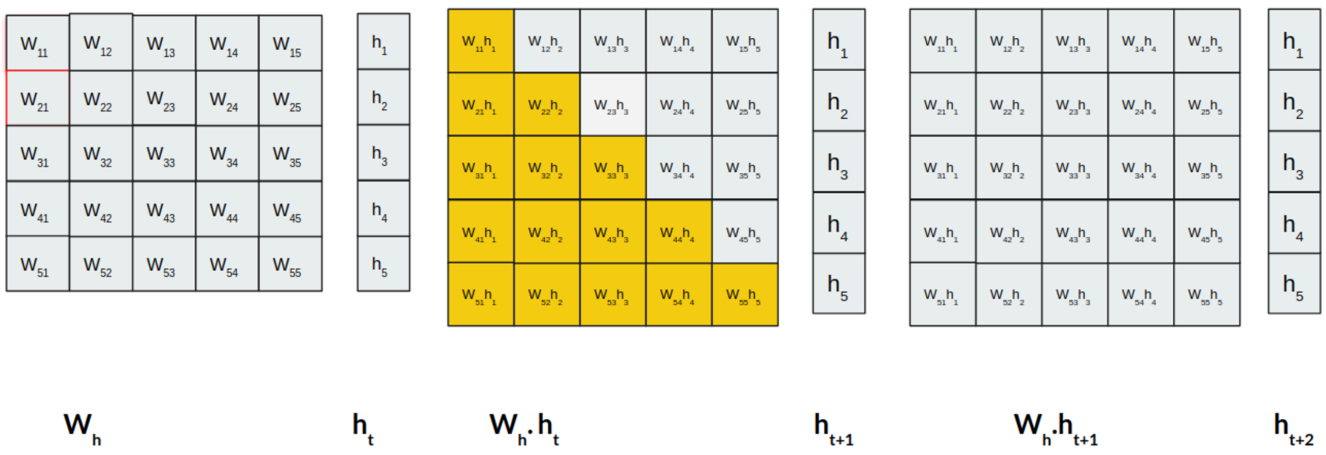


Figure 3.4: (Initial) The upper diagonal elements of Weight matrix W_h will be used for computing h_{t+1}. The lower diagonal and diagonal elements of W_h have been multiplied with elements of h_t in earlier time step.

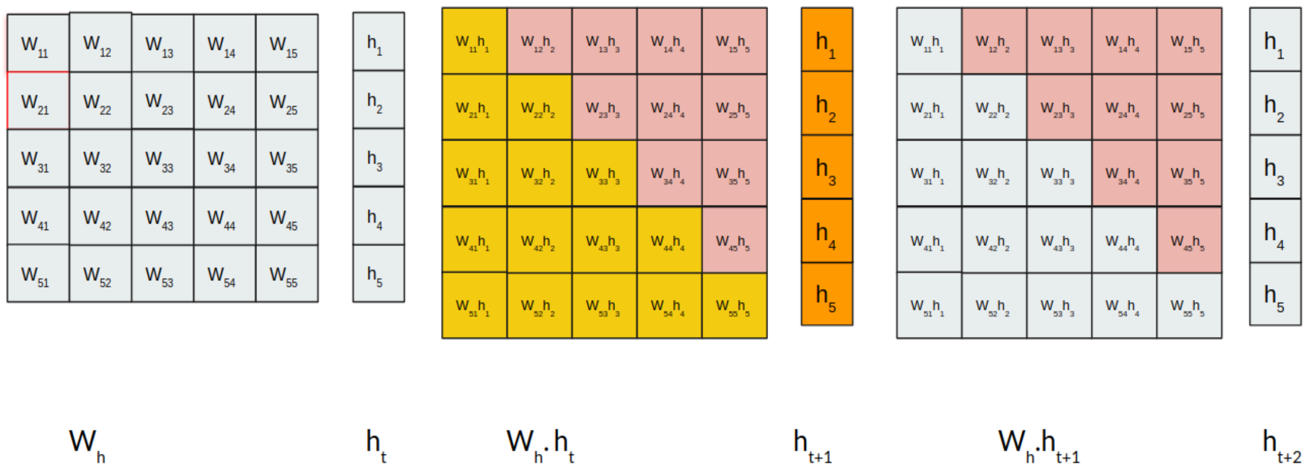


Figure 3.5: h_{t+1} is obtained after the matrix multiplication $W_h.h_t$. And, for h_{t+2}, The upper diagonal operations of matrix multiplication $W_h.h_{t+1}$ are done in this time step.

In matrix vector multiplication of $W_h.h_t$, only upper diagonal elements of W_h are used, because the lower diagonal and diagonal elements of W_h have been used. So, collectively, whole multiplication of $W_h.h_t$ get done to produce the current output vector h_{t+1}. Similar to first step, when the computation of particular index of h_{t+1} get done, then that h_{t+1} element of particular index can be used for partial computation of h_{t+2} element with that particular index. Here, elements of matrices W_h are reused. When, the computation of h_{t+1} output vector get done fully , then, value of *cond* get changed to 0. And, in next iteration, h_{t+2} will be our current output vector. And h_{t+3} will be next output vector.

3. Until the end of time steps, above process (step1 and step2) will repeat. And, Then, The desired result will be obtained.

3.3.2 Block Weight Reuse Method :-

The Weight Reuse method can be used element-wise and block-wise. In block-wise method, the weight matrices W_h(of dimension $N \times N$) are divided into blocks of dimension $B \times B$. Each block of W_h is multiplied with some chunk or partition of previous output vector h_{t-1}. In this method, the way of computation is little different from that of element-wise method. But, the approach is same. In some cases, this method utilize the hardware resources better and increase the speed of the computations, thus reducing the time taken.

3.4 Different versions for the weight reuse method :-

For Weight Reuse method, the reductions in global memory accesses (global transactions) are for only output weight matrices W_h. So, the kernel functions for matrix vector multiplications using W_h are changed in this method. And, the kernel functions for matrix vector multiplications using input weight matrices W_x remain same as in conventional method. All the four matrix vector multiplications using W_h are fused in a single kernel function. This fused kernel function also perform the task of doing all the point-wise operations like calculation of element-wise addition, element-wise multiplication, sigmoid functions and tanh functions. For different type of condition, this fused function has different implementation. I have implemented four version of the Weight Reuse Method. They can be applied both element-wise and block-wise. They are discussed below :-

3.4.1 Version1

It has following features :-

- For computation of each element (or chunk) of current output vector, a kernel function is called with threads depending upon the index of output element.i.e **In each time step, kernel functions are called number of hidden units times.**
- When **cond** $=$ **0**, the elements of W_h are accessed in top to bottom manner. And, When **cond** $=$ **1**, the elements of W_h are accessed in bottom to top manner.

- While computing any element of current output vector, different threads access elements of W_h and do required operations. All these computations are done parallelly.
- More than one element (or chunk) of current output vector cannot be computed simultaneously. As, there is a dependency between between two elements of output vector.
- This version make use of __syncthreads() function, atomic functions and shared variables for processing.

Algorithm for Version1 :-

Algorithm 1 Block-wise Weight Reuse Version1 algorithm

```
procedure LSTMFUNCTION($W_x, W_h, G, x_t, b, cond, h_{t-1}, c_{t-1}$)
    $V_x \leftarrow$M $\times V(W_x, x_t, N, L)$
    $rCeil = \lceil \frac{N}{B} \rceil$

    if $cond = 0$ then
        for $i \leftarrow 0$ to $rCeil - 1$ do
            $threads \leftarrow i + 1$
            $(c_t^i, h_t^i) \leftarrow$ GateCompCond0 $\lll 1, Threads \ggg (W_h, V_x, G, b, h_{t-1}, c_{t-1}, i, rCeil)$
        end for
    else
        for $i \leftarrow rCeil - 1$ to $0$ do
            $threads \leftarrow rCeil - 1 - i$
            if $threads = 0$ then
                $(c_t^i, h_t^i) \leftarrow$ OutElemComp $\lll 1, 1 \ggg (G, h_{t-1}, c_{t-1}, i)$
            else
                $(c_t^i, h_t^i) \leftarrow$ GateCompCond1 $\lll 1, Threads \ggg (W_h, V_x, G, b, h_{t-1}, c_{t-1}, i, rCeil)$
            end if
        end for
    end if
        $cond = 1 - cond$
return $(c_t, h_t)$
    end procedure

    procedure GATECOMPCOND0($W_h, W_x, G, b, h_{t-1}, c_{t-1}, r, rCeil$)
        __shared__ $tPres, tNext$
        $i = threadId$
        InitailizeParameter($tPres, tNext$)
        Barrier Function
        if i < r then
```

```
            $tHr \leftarrow getVector(\mathrm{h}_{t-1}, i)$
            $tHval \leftarrow getVector(\mathrm{h}_t, i)$
            $tWh \leftarrow getBlock(\mathrm{W}_h, r, i)$
            $tPres \leftarrow$tPres+M $\times V(tWh, tHr)$
            $tNext \leftarrow$tNext+M $\times V(tWh, tHval)$
        end if
        Barrier Function
        if i=r then
            $tHr \leftarrow getVector(\mathrm{h}_{t-1}, i)$
            $tWh \leftarrow getBlock(\mathrm{W}_h, r, i)$
            $tPres \leftarrow$tPres+M $\times V(tWh, tHr)$
            $tPres \leftarrow$tPres + $\mathrm{V}_x + b + G$
            $(c_t^r, h_t^r) \leftarrow$ LstmEquations(tPres, $\mathrm{h}_{t-1}, c_{t-1}, r)$
            $tHval \leftarrow getVector(\mathrm{h}_t, i)$
            $tNext \leftarrow$tNext+M $\times V(tWh, tHval)$
            $G^r \leftarrow tNext$
        end if
  return $(c_t^r, h_t^r)$
    end procedure

    procedure GateCompCond1$(W_h, W_x, G, b, h_{t-1}, c_{t-1}, r, rCeil)$
        __shared__ $tPres, tNext$
        $i = rCeil - 1 - threadId$
        InitailizeParameter$(tPres, tNext)$
        Barrier Function
        if i > r then
            $tHr \leftarrow getVector(\mathrm{h}_{t-1}, i)$
            $tHval \leftarrow getVector(\mathrm{h}_t, i)$
            $tWh \leftarrow getBlock(\mathrm{W}_h, r, i)$
            $tPres \leftarrow$tPres+M $\times V(tWh, tHr)$
            $tNext \leftarrow$tNext+M $\times V(tWh, tHval)$
        end if
        Barrier Function
        if i=rCeil-1 then
            $tPres \leftarrow$tPres + $\mathrm{V}_x + b + G$
            $(c_t^r, h_t^r) \leftarrow$ LstmEquations(tPres, $\mathrm{h}_{t-1}, c_{t-1}, r)$
            $G^r \leftarrow tNext$
        end if
  return $(c_t^r, h_t^r)$
    end procedure
```

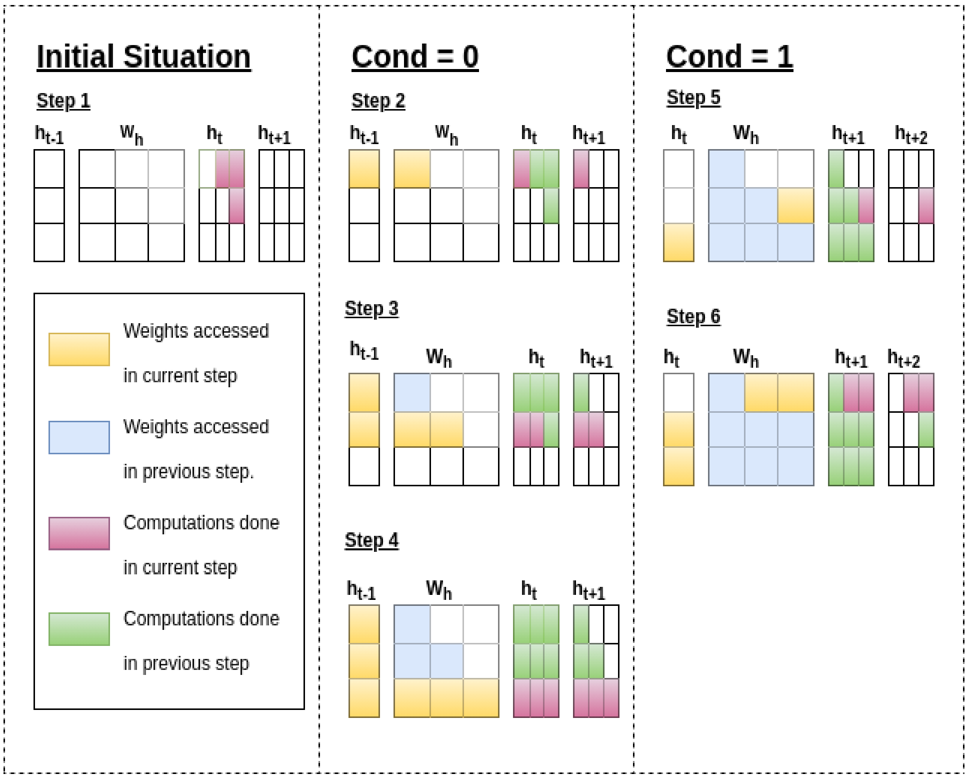

Initial Situation
Step 1
Cond = 0
Step 2
Step 3
Step 4
Cond = 1
Step 5
Step 6
h_{t-1}
W_h
h_t
h_{t+1}
h_{t+2}
Weights accessed in current step
Weights accessed in previous step.
Computations done in current step
Computations done in previous step

Figure 3.6: For Version1 and Version2, this diagram show the computation of output vectors h_t, h_{t+1} and h_{t+2} while accessing W_h matrix from global memory. Initially, it is assumed that h_t is partially computed. When cond $= 0$, elements of W_h are accessed in top to bottom manner. When cond $= 1$, elements of W_h are accessed in bottom to top manner.

3.4.2 Version2

This version is extension of Version1. It has following features :-

- It is inspired by the implementation of array sum calculation and prefix sum calculation in CUDA on GPU. Here for any condition in a time step (either first or second),**kernel function is called once** which compute all the elements (or chunks) of output vector in it.
- When **cond** $= \mathbf{0}$, the elements of W_h are accessed in top to bottom manner. And, when **cond** $= \mathbf{1}$, the elements of W_h are accessed in bottom to top manner.
- Calling kernel function has some overhead on the system. In version1, for each element of output vector, kernel function is called, thus leading to large overhead. In this version, kernel is called once, thus avoiding lots of overheads.
- In kernel function, each element of current output vector is computed iteratively.
- This version make use of __syncthreads() functions, atomic functions and shared variables for processing.

Algorithm for Version2 :-

Algorithm 2 Block-wise Weight Reuse Version2 algorithm

```
procedure LstmFunction($W_x, W_h, G, x_t, b, cond, h_{t-1}, c_{t-1}$)
    $V_x \leftarrow M \times V(W_x, x_t, N, L)$
    $rCeil = \lceil \frac{N}{B} \rceil$

    if $cond = 0$ then
        $(c_t, h_t) \leftarrow$ GateCompCond0 $\lll 1, rCeil \ggg (W_h, V_x, G, b, h_{t-1}, c_{t-1}, rCeil)$
    else
        $(c_t^i, h_t^i) \leftarrow$ OutElemComp $\lll 1, 1 \ggg (G, h_{t-1}, c_{t-1}, rCeil - 1)$
        $(c_t, h_t) \leftarrow$ GateCompCond1 $\lll 1, rCeil - 1 \ggg (W_h, V_x, G, b, h_{t-1}, c_{t-1}, rCeil)$
    end if
        $cond = 1 - cond$
return $(c_t, h_t)$
    end procedure

    procedure GateCompCond0($W_h, W_x, G, b, h_{t-1}, c_{t-1}, rCeil$)
        __shared__ $tPres, tNext$
        $i = threadId$
        InitailizeParameter($tPres, tNext$)
        Barrier Function
        for $r \leftarrow 0$ to $rCeil - 1$ do
            if i $<$ r then
                $tHr \leftarrow getVector(h_{t-1}, i)$
                $tHval \leftarrow getVector(h_t, i)$
                $tWh \leftarrow getBlock(W_h, r, i)$
                $tPres \leftarrow tPres + M \times V(tWh, tHr)$
                $tNext \leftarrow tNext + M \times V(tWh, tHval)$
            end if
            Barrier Function
            if i=r then
                $tHr \leftarrow getVector(h_{t-1}, i)$
                $tWh \leftarrow getBlock(W_h, r, i)$
                $tPres \leftarrow tPres + M \times V(tWh, tHr)$
                $tPres \leftarrow tPres + V_x + b + G$
                $(c_t^r, h_t^r) \leftarrow$ LstmEquations(tPres, $h_{t-1}, c_{t-1}, r$)
                $tHval \leftarrow getVector(h_t, i)$
                $tNext \leftarrow tNext + M \times V(tWh, tHval)$
                $G^r \leftarrow tNext$
            end if
            Barrier Function
        end for
return $(c_t, h_t)$
    end procedure

    procedure GateCompCond1($W_h, W_x, G, b, h_{t-1}, c_{t-1}, r, rCeil$)
        __shared__ $tPres, tNext$
        $i = rCeil - 1 - threadId$
        InitailizeParameter($tPres, tNext$)
        Barrier Function
        for $r \leftarrow rCeil - 2$ to $0$ do
            if i $>$ r then
                $tHr \leftarrow getVector(\mathrm{h}_{t-1}, i)$
                $tHval \leftarrow getVector(\mathrm{h}_t, i)$
                $tWh \leftarrow getBlock(\mathrm{W}_h, r, i)$
                $tPres \leftarrow$tPres+M $\times V(tWh, tHr)$
                $tNext \leftarrow$tNext+M $\times V(tWh, tHval)$
            end if
            Barrier Function
            if i=rCeil-1 then
                $tPres \leftarrow$tPres $+ \mathrm{V}_x + b + G$
                $(c_t^r, h_t^r) \leftarrow$ LstmEquations(tPres, $\mathrm{h}_{t-1}, c_{t-1}, r$)
                $G^r \leftarrow tNext$
            end if
            Barrier Function
        end for
return $(c_t, h_t)$
    end procedure
```

3.4.3 Version3

This version is very different from version1 and version2. It has following features :-

- Like Version2, the kernel function is called once for each condition.
- When **cond** $= \mathbf{0}$, only lower diagonal and diagonal elements of W_h are required. They are accessed in **left to right manner**. When **cond** $= \mathbf{1}$, only upper diagonal elements of W_h are required. They are accessed in **right to left manner**.
- For each block of kernel function, the maximum amount of shared memory is 48KB. And, we store values of all the present and next gate vectors in this memory.

- In our experiment, datatype taken is **double** which is of 8 bytes. And, there are eight vectors of gate values (present and next gate vector of all the four types - input, forget, cell input and output gates). So, maximum size of each array is $((48 \times 1024)/8) = 768$.
- As, maximum size of each vector is 768. So, maximum number of hidden units is 768.
- In this version, the maximum size of vector depend upon the **datatype** i.e. (int, float, double etc). If, we are doing computation for different datatypes, then maximum size can differ.

Algorithm for Version3 :-

Algorithm 3 Block-wise Weight Reuse Version3 algorithm

```
global variables
    $SS$, Global variable for maximum size of shared array
end global variables
procedure LstmFunction($W_x, W_h, G, x_t, b, cond, h_{t-1}, c_{t-1}$)
    $V_x \leftarrow$M $\times V(W_x, x_t, N, L)$
    $rCeil = \lceil \frac{N}{B} \rceil$

    if $cond = 0$ then
        $(c_t, h_t) \leftarrow$ GateCompCond0 $\lll 1, rCeil \ggg (W_h, V_x, G, b, h_{t-1}, c_{t-1}, rCeil)$
    else
        $(c_t^i, h_t^i) \leftarrow$ OutElemComp $\lll 1, 1 \ggg (G, h_{t-1}, c_{t-1}, rCeil - 1)$
        $(c_t, h_t) \leftarrow$ GateCompCond1 $\lll 1, rCeil - 1 \ggg (W_h, V_x, G, b, h_{t-1}, c_{t-1}, rCeil)$
    end if
        $cond = 1 - cond$
return $(c_t, h_t)$
    end procedure

    procedure GateCompCond0($W_h, W_x, G, b, h_{t-1}, c_{t-1}, rCeil$)
        __shared__ $tPres[SS], tNext[SS]$
        $i = threadId$
        InitailizeParameter($tPres, tNext$)
        Barrier Function
        for $r \leftarrow 0$ to $rCeil - 1$ do
            if i == r then
                $tHr \leftarrow getVector(\mathrm{h}_{t-1}, r)$
                $tWh \leftarrow getBlock(\mathrm{W}_h, i, r)$
                $tPres \leftarrow$tPres+M $\times V(tWh, tHr)$
                $(c_t^r, h_t^r) \leftarrow$ LstmEquations(tPres, $\mathrm{h}_{t-1}, c_{t-1}, r$)
                $tHval \leftarrow getVector(\mathrm{h}_t, r)$
                $tNext$ ←tNext+M $\times V(tWh, tHval)$
                $G^r \leftarrow tNext$
            end if
            Barrier Function
            if i>r then
                $tHr \leftarrow getVector(\mathrm{h}_{t-1}, r)$
                $tHval \leftarrow getVector(\mathrm{h}_t, r)$
                $tWh \leftarrow getBlock(\mathrm{W}_h, i, r)$
                $tPres$ ←tPres+M $\times V(tWh, tHr)$
                $tNext$ ←tNext+M $\times V(tWh, tHval)$
            end if
            Barrier Function
        end for
return $(c_t, h_t)$
    end procedure

    procedure GateCompCond1$(W_h, W_x, G, b, h_{t-1}, c_{t-1}, r, rCeil)$
        __shared__ $tPres[SS], tNext[SS]$
        $i = threadId$
        InitailizeParameter$(tPres, tNext)$
        Barrier Function
        for $r \leftarrow rCeil - 1$ to 1 do
            if i < r then
                $tHr \leftarrow getVector(\mathrm{h}_{t-1}, r)$
                $tHval \leftarrow getVector(\mathrm{h}_t, r)$
                $tWh \leftarrow getBlock(\mathrm{W}_h, i, r)$
                $tPres$ ←tPres+M $\times V(tWh, tHr)$
                $tNext$ ←tNext+M $\times V(tWh, tHval)$
            end if
            Barrier Function
            if i=r-1 then
                $(c_t^r, h_t^r) \leftarrow$ LstmEquations(tPres, $\mathrm{h}_{t-1}, c_{t-1}, r)$
                $G^r \leftarrow tNext$
            end if
            Barrier Function
        end for
return $(c_t, h_t)$
    end procedure
```

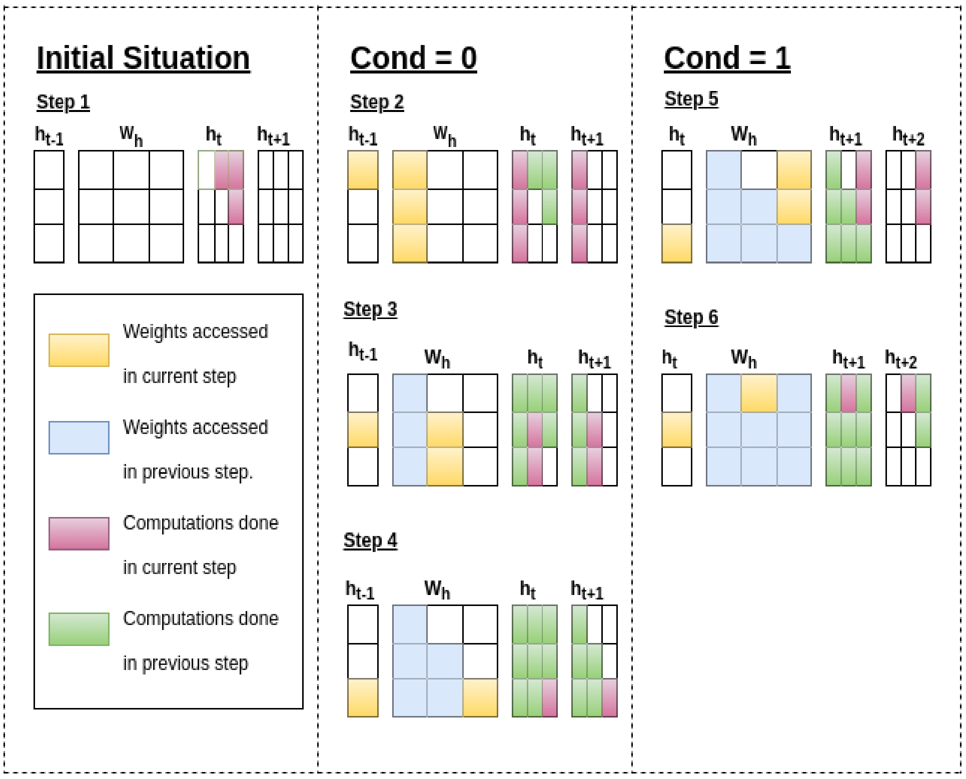

Initial Situation
Step 1
Cond = 0
Step 2
Step 3
Step 4
Cond = 1
Step 5
Step 6
h_{t-1}
W_h
h_t
h_{t+1}
h_{t+2}
Weights accessed in current step
Weights accessed in previous step.
Computations done in current step
Computations done in previous step

Figure 3.7: For Version3 and Version4, this diagram show the computation of output vectors h_t, h_{t+1} and h_{t+2} while accessing W_h matrix from global memory. Initially, it is assumed that h_t is partially computed. When cond = 0, elements of W_h are accessed in left to right manner. When cond = 1, elements of W_h are accessed in right to left manner. In Version4, the element (or segment) of previous output vector and present output vector are stored in shared variable.

3.4.4 Version4

This version is extension of Version3. It has following features :-

- Like Version2 and Version3, the kernel function is called once for each condition.
- When **cond = 0**, only lower diagonal and diagonal elements of W_h are required. They are accessed in **left to right manner**. When **cond = 1**, only upper diagonal elements of W_h are required. They are accessed in **right to left manner**.
- For each thread-block of kernel function, the maximum amount of shared memory is 48KB. And, we store values of all the present and next gate vector in this memory.
- In each iteration of kernel function, **element (or chunk) of previous output vector and present output vector is common** to all the threads. So, it can be stored in a **shared variable**. They will be used by all the threads, thus saving more global memory accesses.
- In our experiment, datatype taken is **double** which is of 8 bytes. And, there are eight vectors of gate values (present and next gate vector of all the four types - input, forget, cell input and output gates). And, two extra shared variable are also used. .

- As, For element-wise method, maximum size of shared vector is 767. So, Here, maximum value of hidden units is 767.
- For different value of block sizes, following are maximum size of shared vector and maximum value of hidden units :-
    1. **Block size = 4 :-** Maximum size of shared vector and maximum value of hidden units is 764.
    2. **Block size = 8 :-** Maximum size of shared vector and maximum value of hidden units is 760.
    3. **Block size = 16 :-** Maximum size of shared vector and maximum value of hidden units is 752.
    4. **Block size = 32 :-** Maximum size of shared vector and maximum value of hidden units is 736.
    5. **Block size = 64 :-** Maximum size of shared vector and maximum value of hidden units is 704.
- In this version, the maximum size of vector depend upon the **datatype and size of the block**. If, we are doing computation for different datatype or with different size of block, then maximum size can differ.

Algorithm for Version4 :-

Algorithm 4 Block-wise Weight Reuse Version4 algorithm

```
global variables
    $SS$, Global variable for maximum size of shared array
    $BS$, Global variable for block size used.
end global variables
procedure LstmFunction($W_x, W_h, G, x_t, b, cond, h_{t-1}, c_{t-1}$)
    $V_x \leftarrow$M $\times V(W_x, x_t, N, L)$
    $rCeil = \lceil \frac{N}{B} \rceil$

    if $cond = 0$ then
        $(c_t, h_t) \leftarrow$ GateCompCond0 $\lll 1, rCeil \ggg (W_h, V_x, G, b, h_{t-1}, c_{t-1}, rCeil)$
    else
        $(c_t^i, h_t^i) \leftarrow$ OutElemComp $\lll 1, 1 \ggg (G, h_{t-1}, c_{t-1}, rCeil - 1)$
        $(c_t, h_t) \leftarrow$ GateCompCond1 $\lll 1, rCeil - 1 \ggg (W_h, V_x, G, b, h_{t-1}, c_{t-1}, rCeil)$
    end if
        $cond = 1 - cond$
return $(c_t, h_t)$
    end procedure

    procedure GateCompCond0($W_h, W_x, G, b, h_{t-1}, c_{t-1}, rCeil$)
```

```
            __shared__ $tPres[SS], tNext[SS], tHr[BB], tHval[BB]$
            $i = threadId$
            InitailizeParameter($tPres, tNext$)
            **Barrier Function**
            **for** $r \leftarrow 0$ to $rCeil - 1$ **do**
                **if** i == r **then**
                    $tHr \leftarrow getVector(\mathrm{h}_{t-1}, r)$
                    $tWh \leftarrow getBlock(\mathrm{W}_h, i, r)$
                    $tPres \leftarrow$tPres+M $\times V(tWh, tHr)$
                    $(c_t^r, h_t^r) \leftarrow$ **LstmEquations**(tPres, $\mathrm{h}_{t-1}, c_{t-1}, r$)
                    $tHval \leftarrow getVector(\mathrm{h}_t, r)$
                    $tNext \leftarrow$tNext+M $\times V(tWh, tHval)$
                    $G^r \leftarrow tNext$
                **end if**
                **Barrier Function**
                **if** i>r **then**
                    $tWh \leftarrow getBlock(\mathrm{W}_h, i, r)$
                    $tPres \leftarrow$tPres+M $\times V(tWh, tHr)$
                    $tNext \leftarrow$tNext+M $\times V(tWh, tHval)$
                **end if**
                **Barrier Function**
            **end for**
    **return** $(c_t, h_t)$
        **end procedure**

        **procedure** GateCompCond1($W_h, W_x, G, b, h_{t-1}, c_{t-1}, r, rCeil$)
            __shared__ $tPres[SS], tNext[SS], tHr[BB], tHval[BB]$
            $i = threadId$
            InitailizeParameter($tPres, tNext, tHr, tHval$)
            **Barrier Function**
            **for** $r \leftarrow rCeil - 1$ to 1 **do**
                **if** i < r **then**
                    $tWh \leftarrow getBlock(\mathrm{W}_h, i, r)$
                    $tPres \leftarrow$tPres+M $\times V(tWh, tHr)$
                    $tNext \leftarrow$tNext+M $\times V(tWh, tHval)$
                **end if**
                **Barrier Function**
                **if** i=r-1 **then**
                    $(c_t^r, h_t^r) \leftarrow$ **LstmEquations**(tPres, $\mathrm{h}_{t-1}, c_{t-1}, r$)
                    $G^r \leftarrow tNext$
                    $tHr \leftarrow getVector(\mathrm{h}_{t-1}, r-1)$
                    $tHval \leftarrow getVector(\mathrm{h}_{t}, r-1)$
                end if
                Barrier Function
            end for
        return $(c_t, h_t)$
        end procedure
```

There are some common functions with same implementation which are used in all the versions of the Weight Reuse method. They are given below:-

Algorithm 5 Common Functions

```
procedure LstmWeightResue($W_x, W_h, G, x, b, T$)
    $cond = 0$
    for $t \leftarrow 0$ to $T-1$ do
        $(c_t, h_t) \leftarrow$ LstmFunction($\mathrm{W}_x, W_h, G, x_t, b, cond, h_{t-1}, c_{t-1}$)
    end for
return $h$
end procedure

procedure OutElemComp($G, h_{t-1}, c_{t-1}, r$)
    $(c_t^r, h_t^r) \leftarrow$ LstmEquations($\mathrm{G}, \mathrm{h}_{t-1}, c_{t-1}, r$)
    $G^r = 0$
return $(c_t^r, h_t^r)$
end procedure

procedure LstmEquations($G, h_{t-1}, c_{t-1}, r$)
    $i^r \leftarrow Sigmoid(\mathrm{G}_i^r)$
    $f^r \leftarrow Sigmoid(\mathrm{G}_f^r)$
    $g^r \leftarrow Tanh(\mathrm{G}_g^r)$
    $o^r \leftarrow Sigmoid(\mathrm{G}_o^r)$
    $c_t^r \leftarrow \mathrm{f}^r * c_{t-1}^r + i^r * g^r$
    $h_t \leftarrow \mathrm{o}^r * Tanh(c^r)$
return $(c_t^r, h_t^r)$
end procedure
```

Chapter 4

Results and Discussion

4.1 Introduction

In this Chapter, I have mentioned the result of the all the versions of Weight Reuse method. Global transactions of output weight matrices W_h are reduced in all the versions. It is shown in the result. Also, effect of reduction in global transactions on time taken is also shown. I have also mentioned the datasets and gpu machine used for experiment.

4.2 Experiments and Results

I have conducted experiments for reducing time taken by reducing memory accesses for LSTM algorithm on GPU machine. The results obtained from the Weight reuse method are shown in the given subsections.

In GPU profiling, there are two types of global transactions, global load transactions(memory read) and global store transactions(memory write). In our experiments, the value of global store transactions are very less than that of global load transactions. So, I have considered the two values together by taking their sum as global transactions. The global transactions consist of all the transactions from L1 Cache, L2 Cache and DRAM memory for kernel function. Thus, this value represent all over memory accesses in the GPU Machine.

In the Weight Reuse method, the memory accesses of output weight matrices(W_H) are reducing. The memory accesses of input weight matrices(W_x) remain same. In each version of our implementations, operation like matrix vector multiplication containing W_h, calculation of values of all the gates(tanh and sigmoid functions) and calculation of cell state and present output vectors are fused in the same kernel function. So, while profiling of this kernel function, one value is obtained. But, for conventional method, all the above mentioned operations are in different kernel function. So, the values for different kernel function in conventional method are summed up. Then, the values of these different methods are compared.

4.2.1 Datasets used

For our experiments, I have generated random datasets of appropriate dimensions. Before running for any configuration of particular version, I have generated four files - input vector file, input weight matrix file, output weight matrix file and bias vector file. When, program of any version is run, it generates an output vector file. In our work, we are concerned about doing profiling on the program. In such work, random datasets[5] can be used. Saurabh has given me CPU LSTM program. I have tested the working of all the versions with his program. The implementations of all the versions are working fine. He has matched the results of his program with the results of program on the Matlab.

4.2.2 About GPU machine used

I have conducted experiments on the GeForce GTX 1070 GPU machine. This machine has following properties.

Memory clock rate (khz)	4004000
Memory bus width (bits)	256
Peak Memory Bandwidth (GB/sec)	256.256
Maximum dimension 0 of block	1024
Maximum dimension 1 of block	1024
Maximum dimension 2 of block	64
Maximum dimension 0 of grid	2147483647
Maximum dimension 1 of grid	65535
Maximum dimension 2 of grid	65535
Number of threads in warp	32
Maximum threads per block	1024
Global Memory	8510701568 bytes (approx. 8GB)
Shared memory per block	49152 bytes (48KB)
Number of registers per block	65536
Available constant memory	65536 bytes(64KB)
Wrap size	32
No. of multiprocessors	15

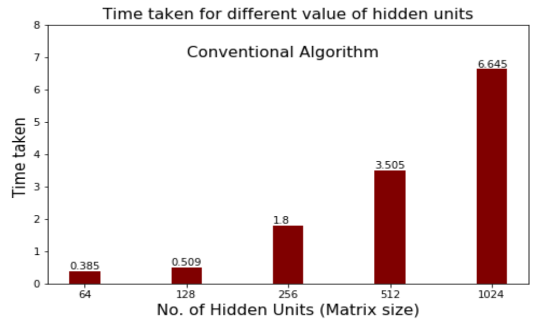
Table 4.1: Table shows the properties of the used GPU machine

4.2.3 Results

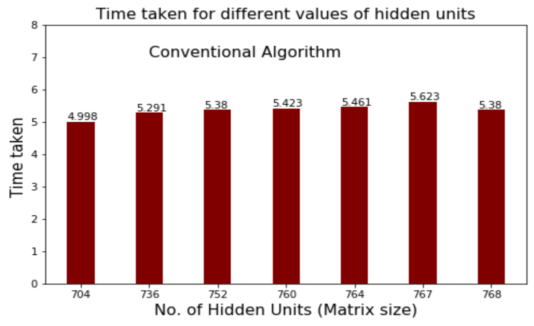
The Weight Reuse method is for LSTM inferencing. I have developed four versions. I have run the conventional method and all the four versions for different parameters. Each version run element-wise and block-wise. In block-wise method, the block-size taken are 4, 8, 16, 32 and 64. In first two versions, I have experimented with hidden units of 64, 128, 256, 512 and 1024 for four time steps in each case. In third version, the maximum value of hidden unit is 768 which depend upon the size of shared memory. In fourth version, the maximum value of hidden unit depend upon the size of block-size used. We have shown the affect of reduction of global transactions on time taken for each case.

Conventional Method

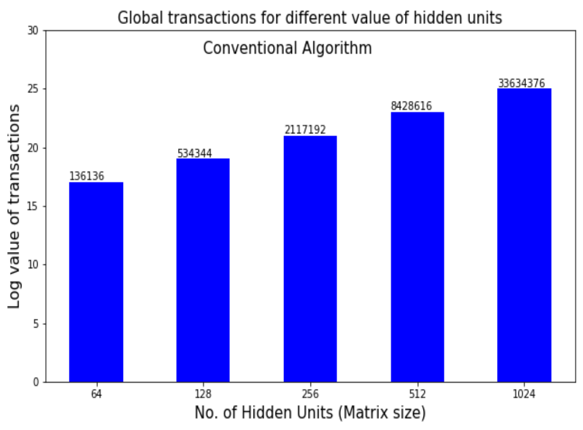
The value of time taken and global transactions for different values of hidden units are shown below:-



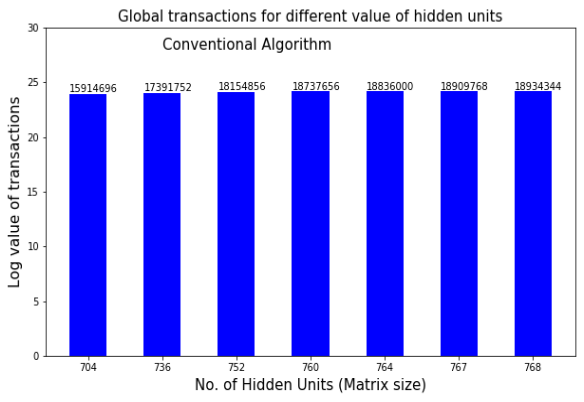
(a)



(b)



(c)

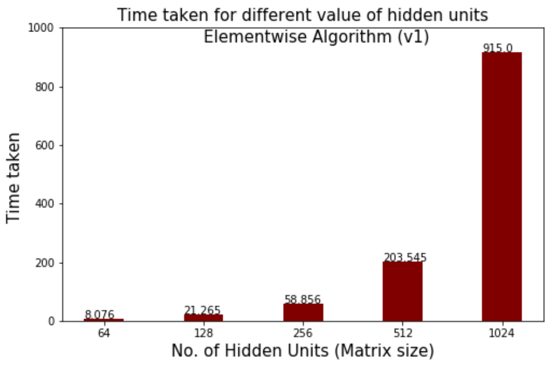


(d)

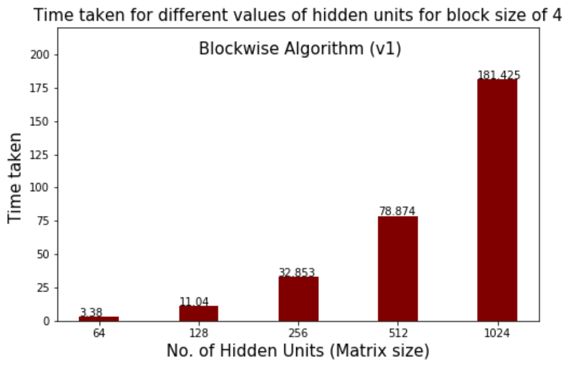
Figure 4.1: Values of time taken ((a),(b)) and global transactions ((c),(d)) for different values of number of hidden units for conventional method.

Version1

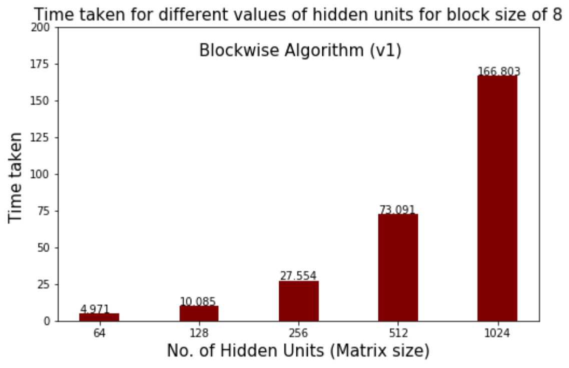
The value of time taken for different values of hidden units for element-wise and block-wise method are shown below :-



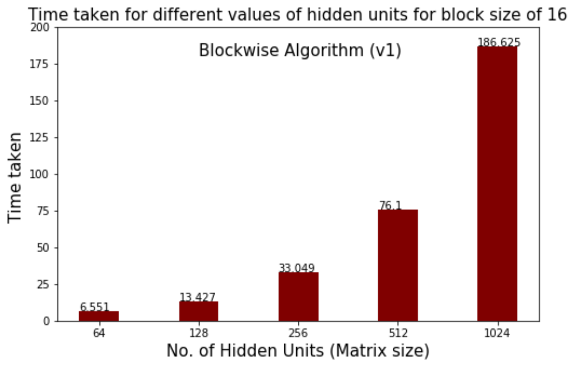
(a)



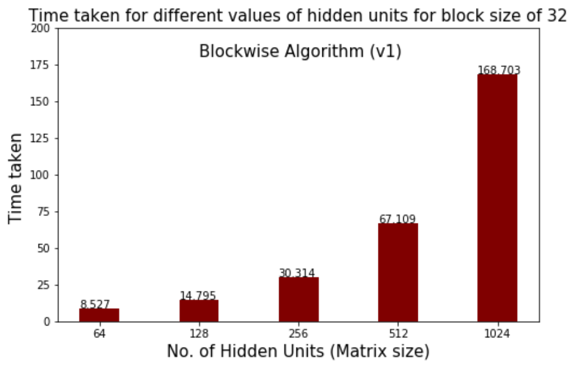
(b)



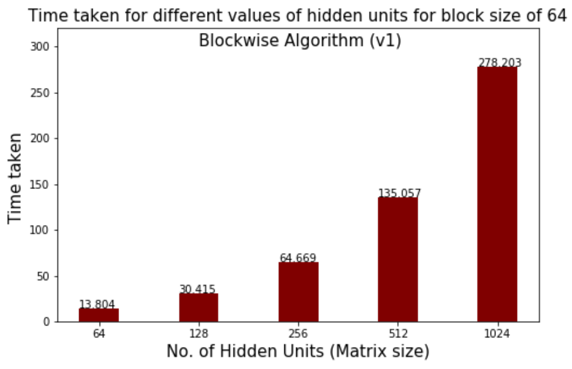
(c)



(d)



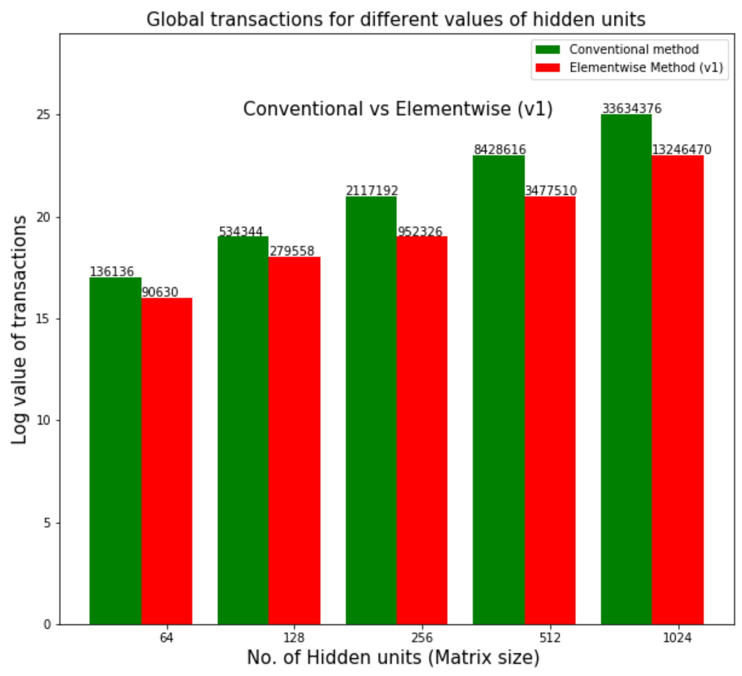
(e)



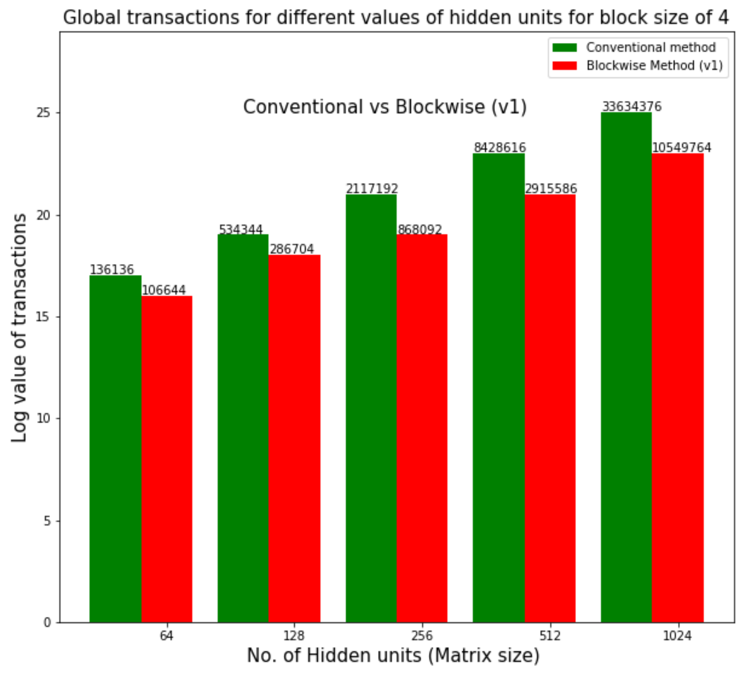
(f)

Figure 4.2: Time taken for different configurations in Version1

The value of global transactions for different values of hidden units for element-wise and block-wise method are shown below :-

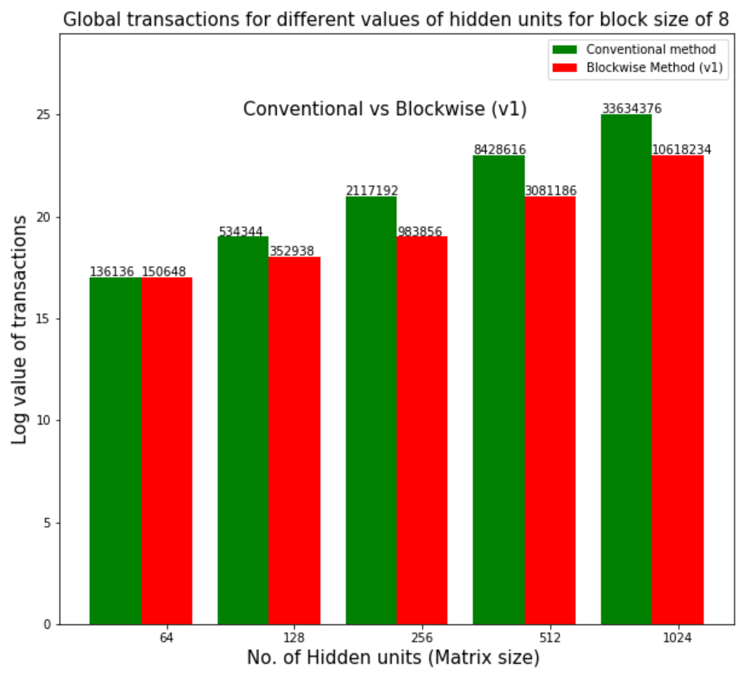


(a) For Element-wise approach

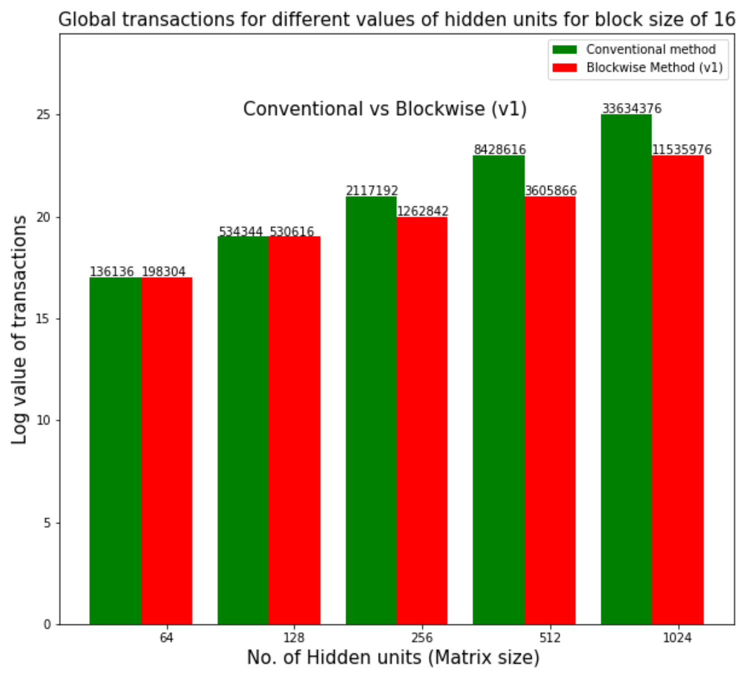


(b) For Block size of 4

Figure 4.3: Global transactions in Version1

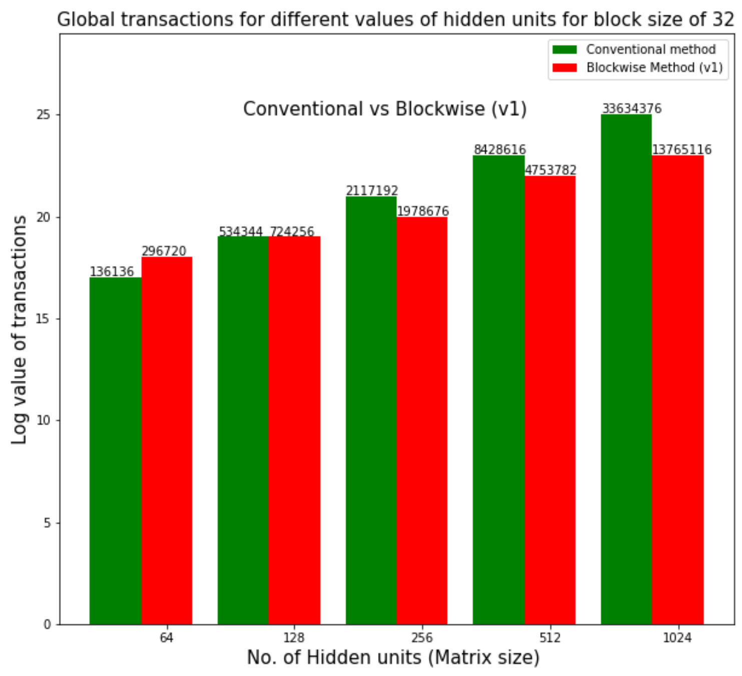


(c) For Block size of 8

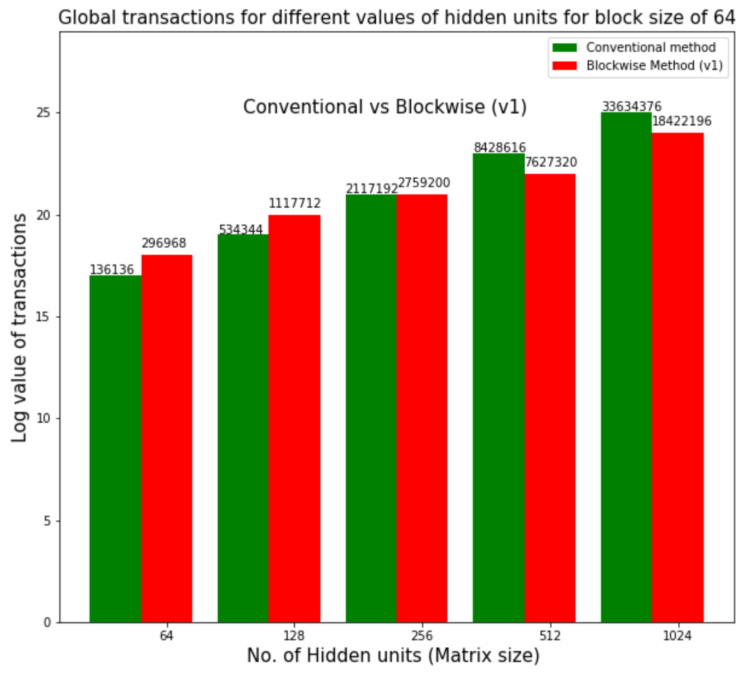


(d) For Block size of 16

Figure 4.3: Global transactions in Version1 (cont.)



(e) For Block size of 32



(f) For Block size of 64

Figure 4.3: Global transactions in Version1 (cont.)

The comparison of the weight reuse method version1 with conventional method for the values of time and global transactions are given below in these tables.

Hidden Units	**Elememt-wise**	**BS = 4**	**BS = 8**	**BS = 16**	**BS = 32**	**BS =64**
64	-33.43%	-21.66%	+10.66%	+45.67%	+117.96%	+118.14%
128	-47.68%	-46.34%	-33.95%	-0.70%	+35.54%	+109.17%
256	-55.02%	-59.00%	-53.53%	-40.35%	-6.54%	+30.32%
512	-58.74%	-65.41%	-63.44%	-57.22%	-43.60%	-9.51%
1024	-60.62%	-68.63%	-68.43%	-65.70%	-59.07%	-45.23%

Table 4.2: Table shows change in the global transactions of Version1 with respect to the conventional method. Here, + sign represent increase percentage and − sign represent reduction percentage

Hidden Units	**Elememt-wise**	**BS = 4**	**BS = 8**	**BS = 16**	**BS = 32**	**BS =64**
64	+20.98	+8.78	+12.91	+17.02	+22.15	+35.85
128	+41.78	+21.69	+19.81	+26.38	+29.07	+59.75
256	+32.70	+18.25	+15.31	+18.36	+16.84	+35.93
512	+58.07	+22.50	+20.85	+21.71	+19.15	+38.53
1024	+137.70	+27.30	+25.10	+28.09	+25.39	+41.87

Table 4.3: Table shows change in the time taken of Version1 with respect to the conventional method. Here, + sign represent multiplicative increase for version1 as compared to conventional method

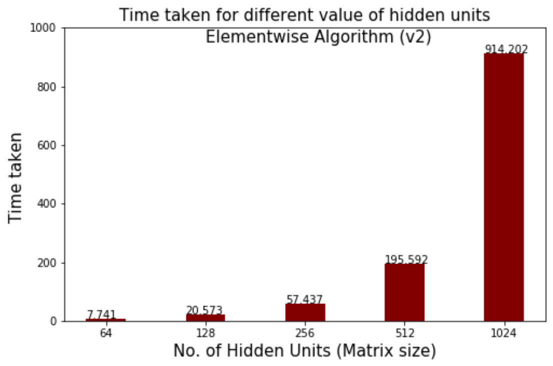
From the graphs and tables shown, it is observed that time taken by element-wise method is most. For case of hidden unit of 1024, it is 137 times of that of conventional method. For block-wise method, time taken is better than that of element-wise version1. But, For case of block size of 64, it has worst time performance.

For any case in version1, as the size of hidden unit increases, the amount of reduction in global transactions also increases. But, for bigger block sizes, there is no reduction for smaller value of hidden units. In most cases, the amount of reduction in global transactions is more than 50%.

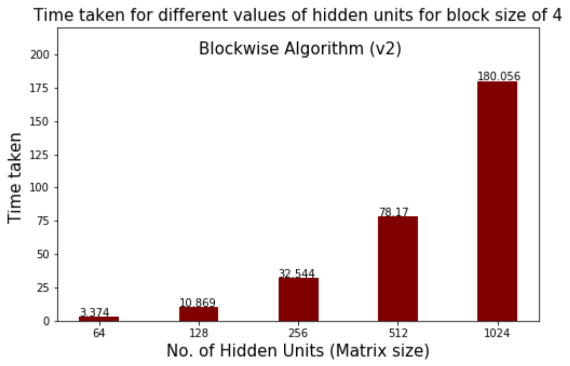
Overall, although, there is a significant decrease in amount of global transactions in most cases. The values of time taken for processing LSTM algorithm do not decrease.

Version2

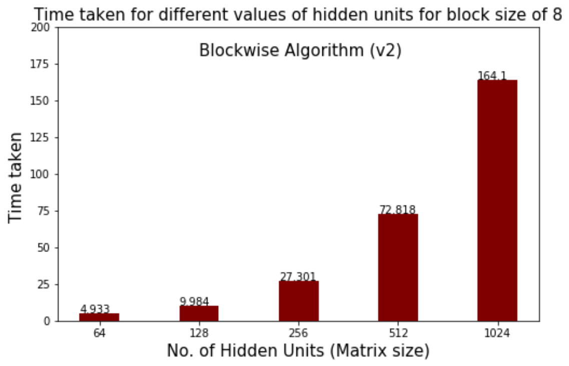
The value of time taken for different values of hidden units for element-wise and block-wise method are shown below :-



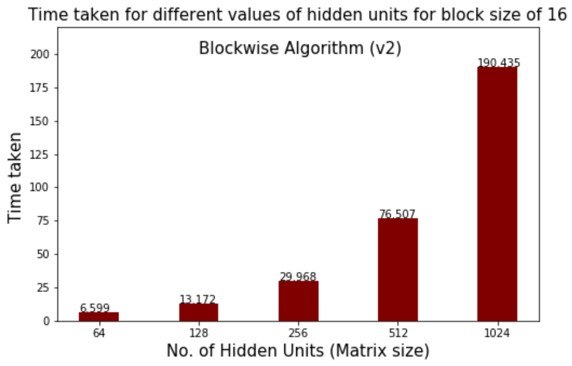
(a)



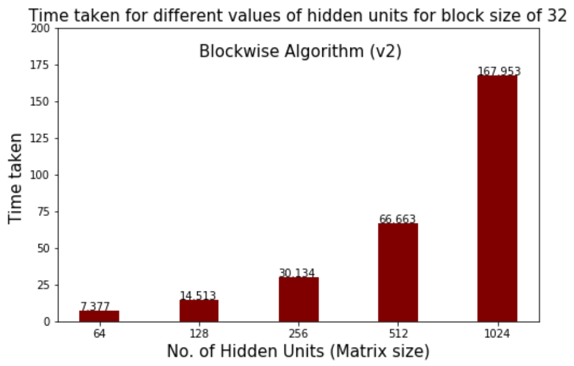
(b)



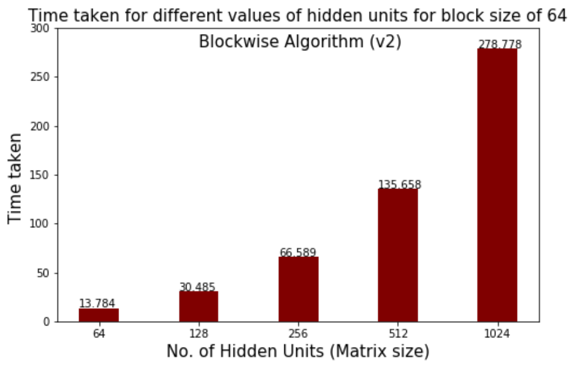
(c)



(d)



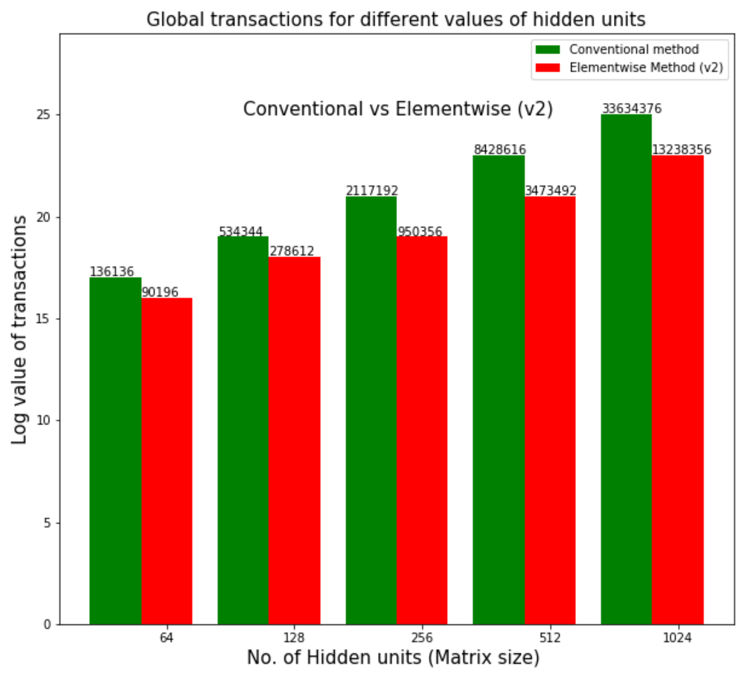
(e)



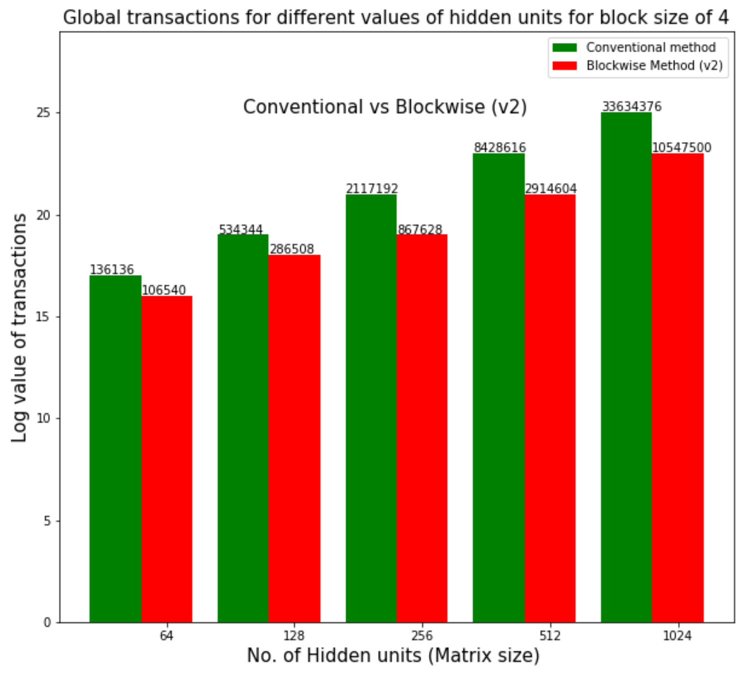
(f)

Figure 4.4: Time taken for different configurations in Version2

The value of global transactions for different values of hidden units for element-wise and block-wise method are shown below :-

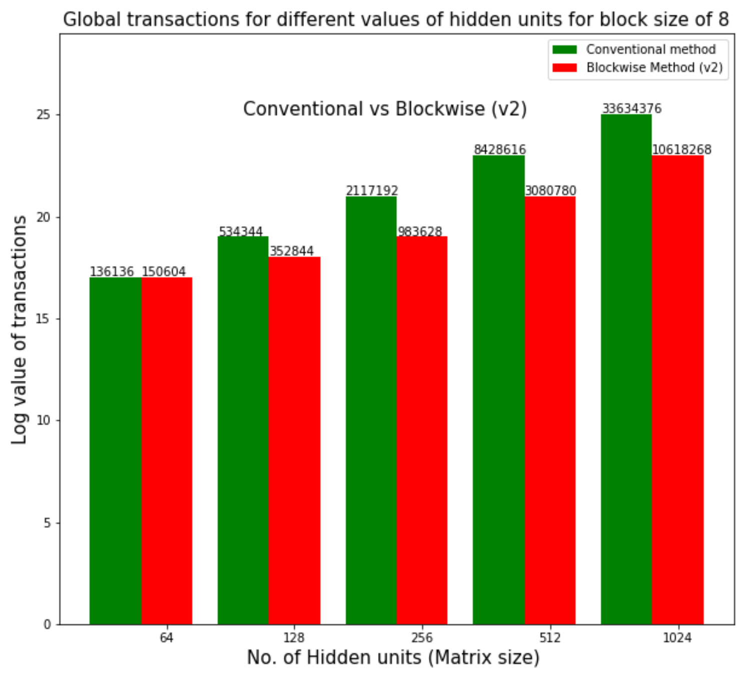


(a) For Element-wise approach

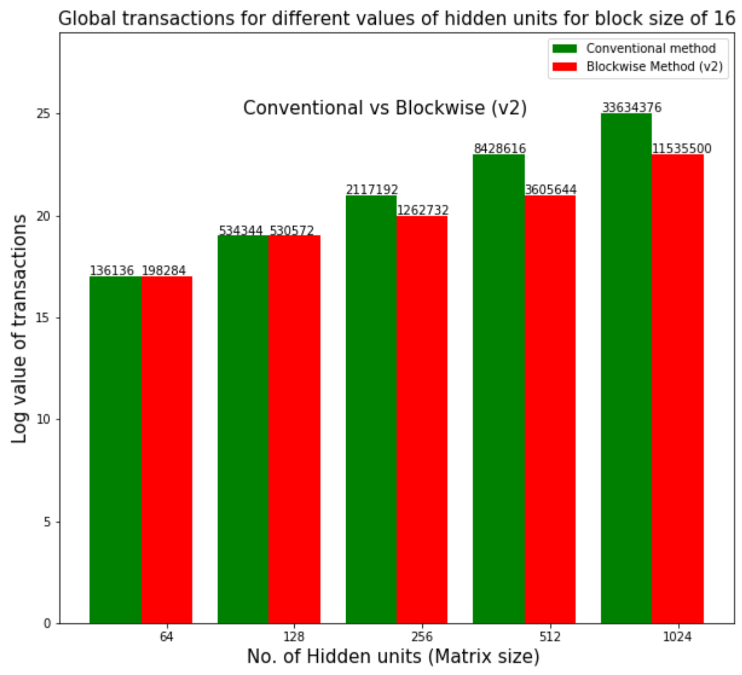


(b) For Block size of 4

Figure 4.5: Global transactions in Version2

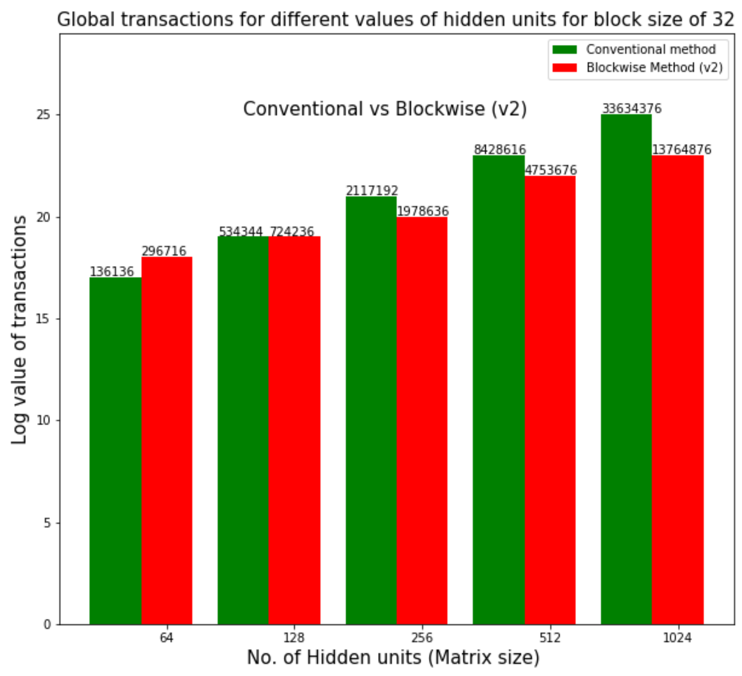


(c) For Block size of 8

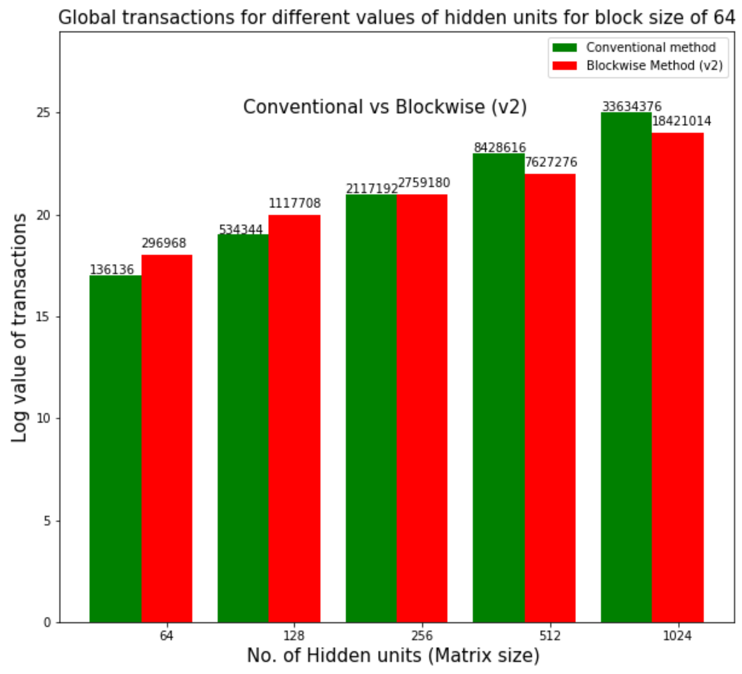


(d) For Block size of 16

Figure 4.5: Global transactions in Version2 (cont.)



(e) For Block size of 32



(f) For Block size of 64

Figure 4.5: Global transactions in Version2 (cont.)

The comparison of the weight reuse method version2 with conventional method for the values of time and global transactions are given below in these tables.

Hidden Units	**Elememt-wise**	**BS = 4**	**BS = 8**	**BS = 16**	**BS = 32**	**BS =64**
64	-33.75%	-21.74%	+10.63%	+45.65%	+117.96%	+118.14%
128	-47.86%	-46.38%	-33.97%	-0.71%	+35.54%	+109.17%
256	-55.11%	-59.02%	-53.54%	-40.36%	-6.54%	+30.32%
512	-58.79%	-65.42%	-63.45%	-57.22%	-43.60%	-9.51%
1024	-60.64%	-68.64%	-68.43%	-65.70%	-59.07%	-45.23%

Table 4.4: Table shows change in the global transactions of Version2 with respect to the conventional method. Here, + sign represent increase percentage and − sign represent reduction percentage

Hidden Units	**Elememt-wise**	**BS = 4**	**BS = 8**	**BS = 16**	**BS = 32**	**BS =64**
64	+20.11	+8.76	+12.81	+17.14	+19.16	+35.80
128	+40.42	+21.35	+19.61	+25.88	+28.51	+59.89
256	+31.91	+18.08	+15.17	+16.65	+16.74	+36.99
512	+55.80	+22.30	+20.78	+21.83	+19.02	+38.70
1024	+137.58	+27.10	+24.70	+28.66	+25.28	+41.95

Table 4.5: Table shows change in the time taken of Version2 with respect to the conventional method. Here, + sign represent multiplicative increase for version2 as compared to conventional method

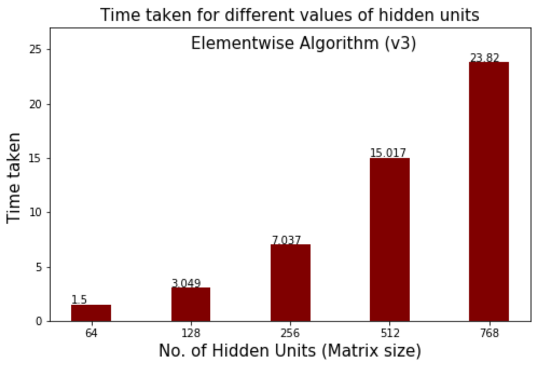
From the above graphs and tables shown, it can be observed that the results of version2 is same as that of the version1. This version2 is designed to avoid repeated calling of kernel function for each element of present output vector. We have not obtained any change in time performance. Here, the element-wise method has worst time performance. And, for block-wise method, block size of 64 has worst time performance.

For any case in version2, As the value of hidden units increases, the amount of reduction in global transactions also increases. But, for bigger block sizes, there is no reduction for smaller value of hidden units. In most cases, the amount of reduction in global transactions is more than 50%.

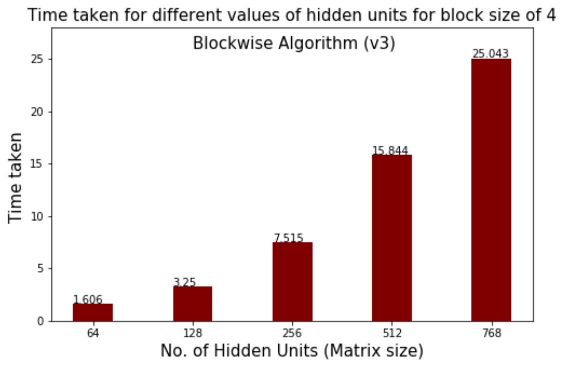
Overall, although, there is a significant decrease in amount of global transactions in most cases. The values of time taken for processing LSTM algorithm do not decrease.

Version3

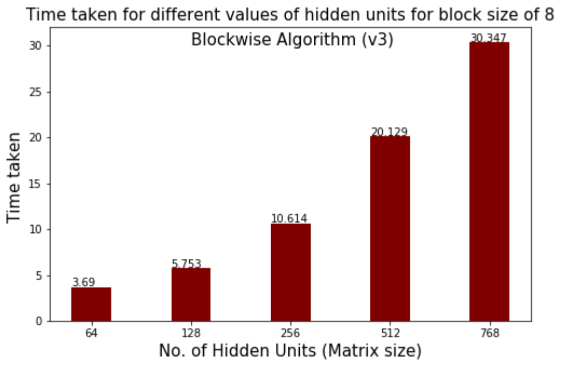
The value of time taken for different values of hidden units for element-wise and block-wise method are shown below :-



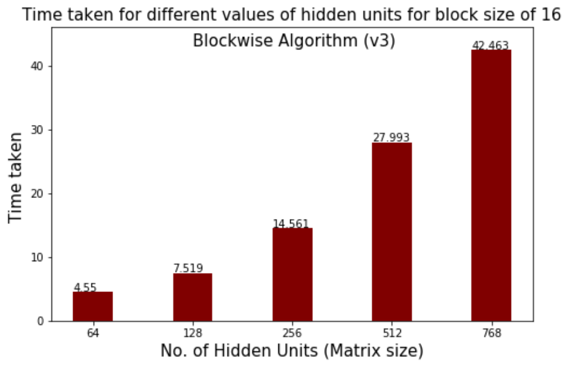
(a)



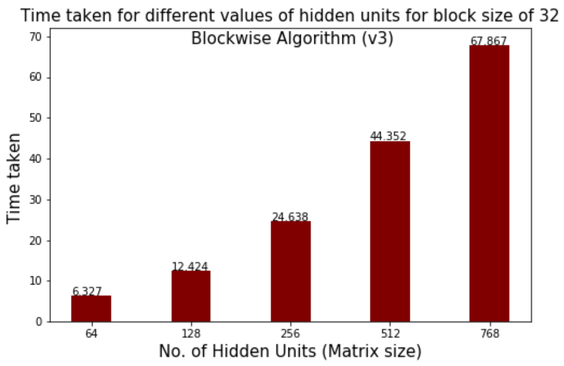
(b)



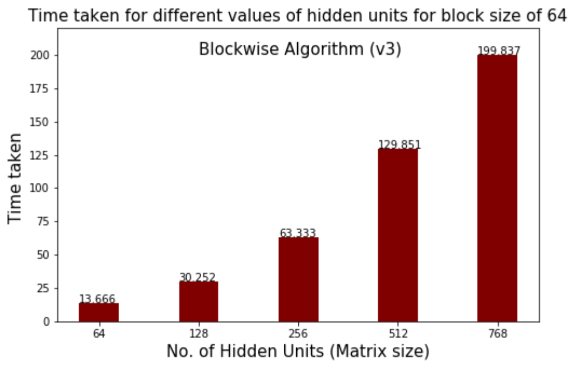
(c)



(d)



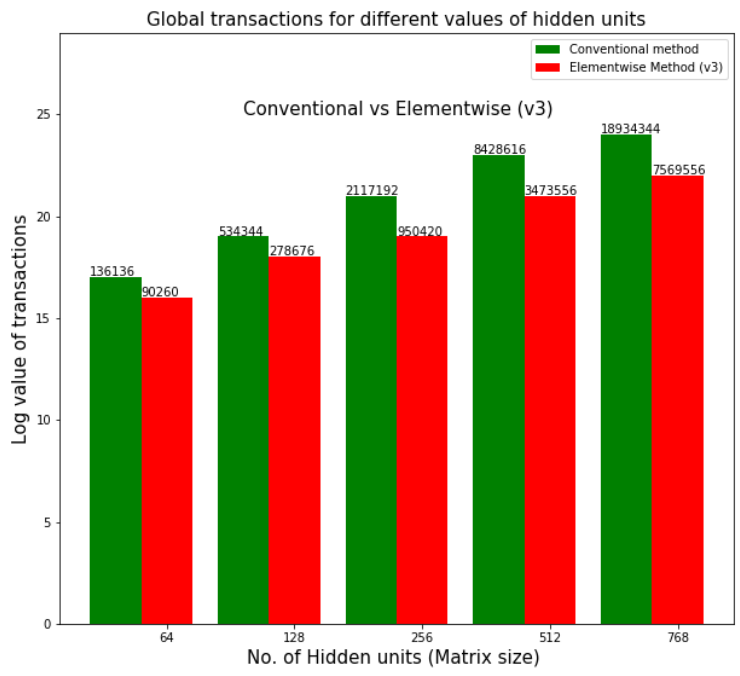
(e)



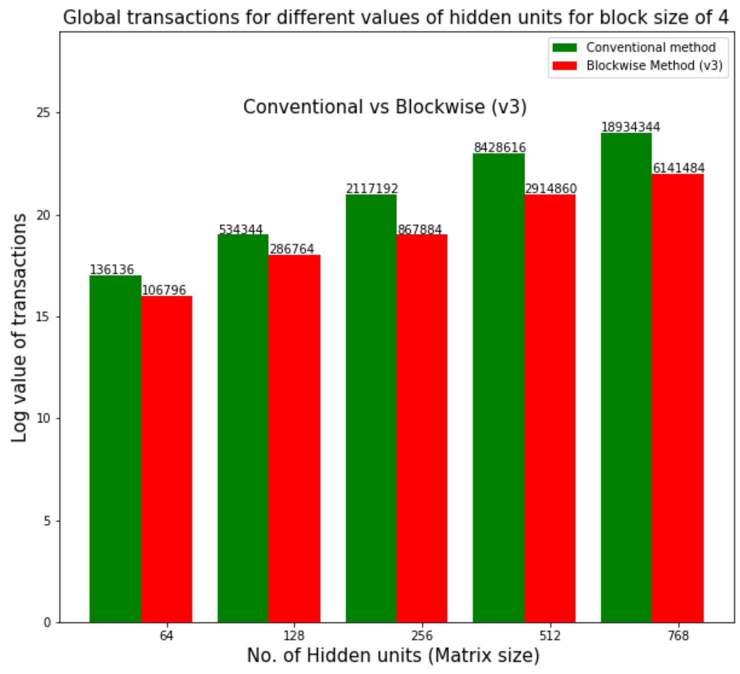
(f)

Figure 4.6: Time taken for different configurations in Version3

The value of global transactions for different values of hidden units for element-wise and block-wise method are shown below :-

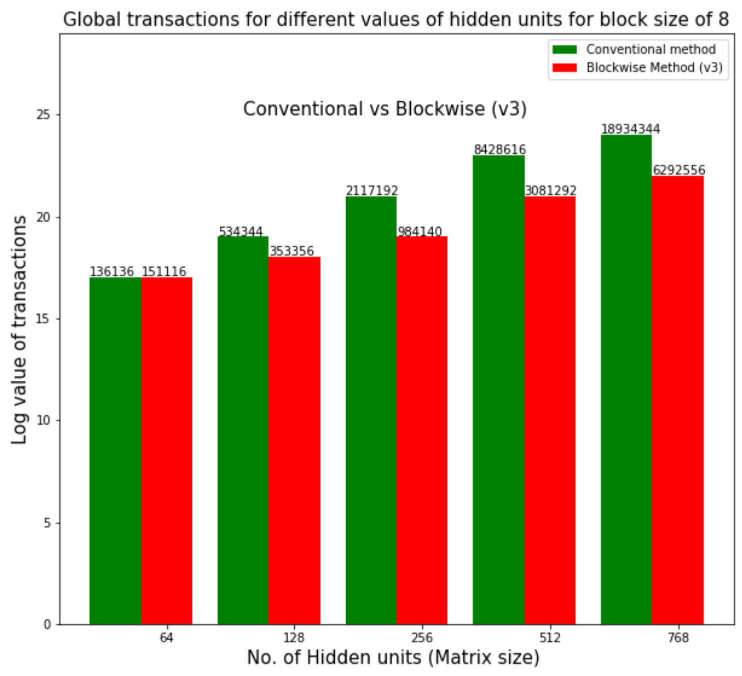


(a) For Element-wise approach

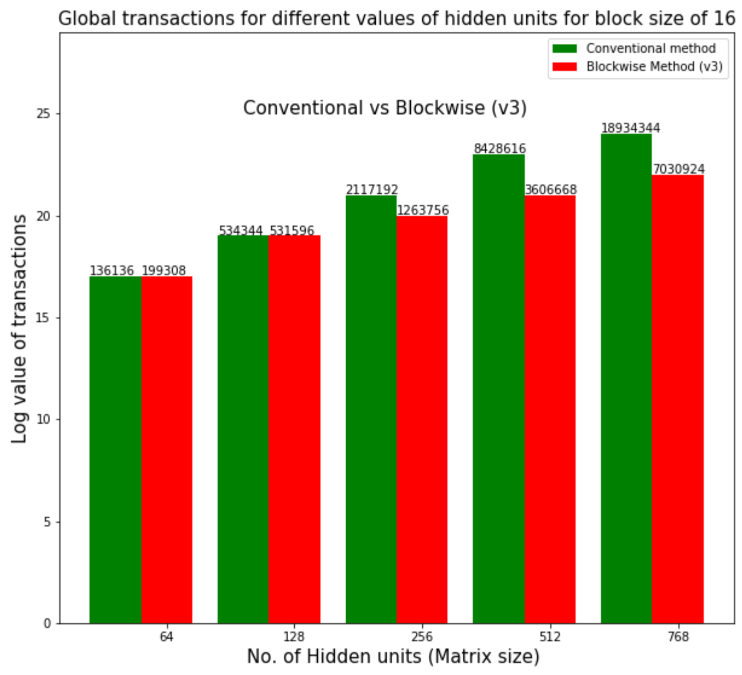


(b) For Block size of 4

Figure 4.7: Global transactions in Version3

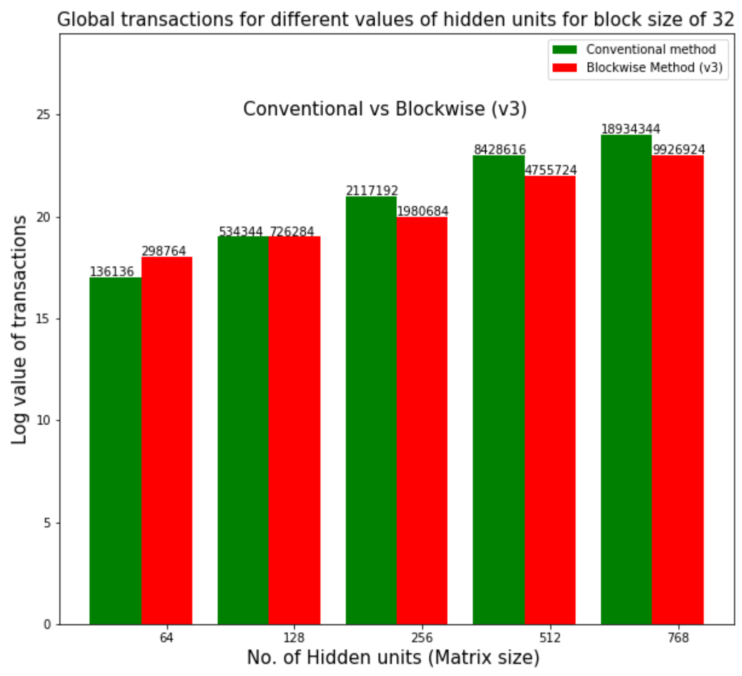


(c) For Block size of 8

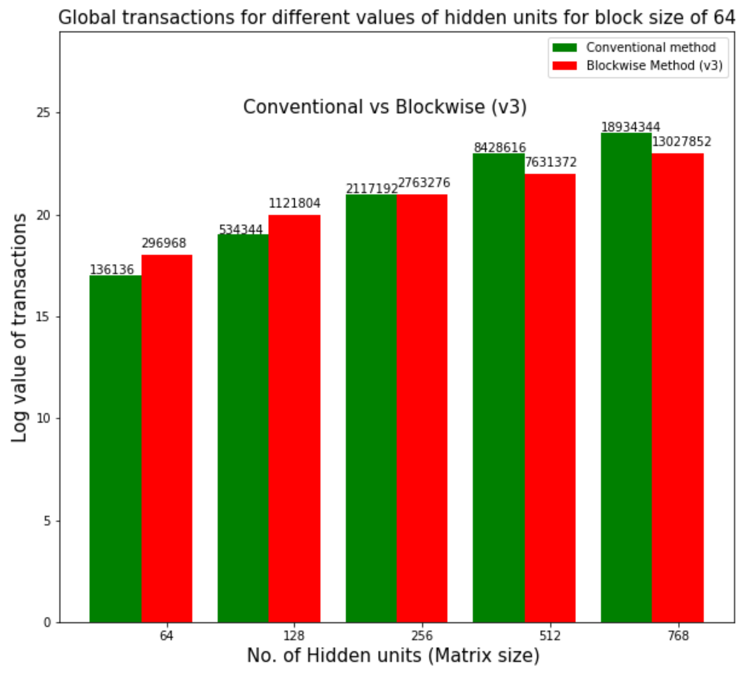


(d) For Block size of 16

Figure 4.7: Global transactions in Version3 (cont.)



(e) For Block size of 32



(f) For Block size of 64

Figure 4.7: Global transactions in Version3 (cont.)

The comparison of the weight reuse method version3 with conventional method for the values of time and global transactions are given below in these tables.

Hidden Units	**Elememt-wise**	**BS = 4**	**BS = 8**	**BS = 16**	**BS = 32**	**BS =64**
64	-33.70%	-21.55%	+11.00%	+46.40%	+119.46%	+118.14%
128	-47.85%	-46.33%	-33.87%	-0.51%	+35.92%	+109.94%
256	-55.11%	-59.01%	-53.52%	-40.31%	-6.45%	+30.52%
512	-58.79%	-65.42%	-63.44%	-57.21%	-43.58%	-9.46%
768	-60.02%	-67.56%	-66.77%	-62.87%	-47.57%	-31.19%

Table 4.6: Table shows change in the global transactions of Version3 with respect to the conventional method. Here, + sign represent increase percentage and − sign represent reduction percentage

Hidden Units	**Elememt-wise**	**BS = 4**	**BS = 8**	**BS = 16**	**BS = 32**	**BS =64**
64	+3.90	+4.17	+9.58	+11.82	+16.43	+35.50
128	+5.99	+6.39	+11.30	+14.77	+24.41	+59.43
256	+3.91	+4.17	+5.90	+8.09	+13.69	+35.18
512	+4.28	+4.52	+5.74	+7.99	+12.65	+37.05
768	+4.43	+4.65	+5.64	+7.89	+12.61	+37.14

Table 4.7: Table shows change in the time taken of Version3 with respect to the conventional method. Here, + sign represent multiplicative increase for version3 as compared to conventional method

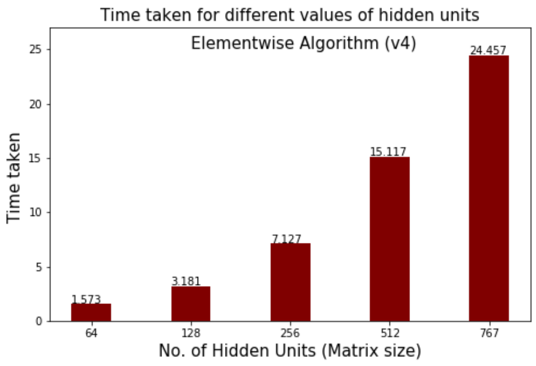
From the above graphs and tables, it can be observed that version3 has better time performance that that of version1 and version2. Here, The element-wise method and block-wise method for smaller block sizes of 4 and 8 has similar time performance. For bigger block sizes of 32 and 64, there is a high increase in value of time taken in comparison to conventional method.

For global transactions, version3 has similar performance to that of version1 and version2. As, the value of hidden units increases, the reduction percentage of global transactions also increases. In case of block-wise method, there is no reduction in global transactions for smaller value of hidden units.

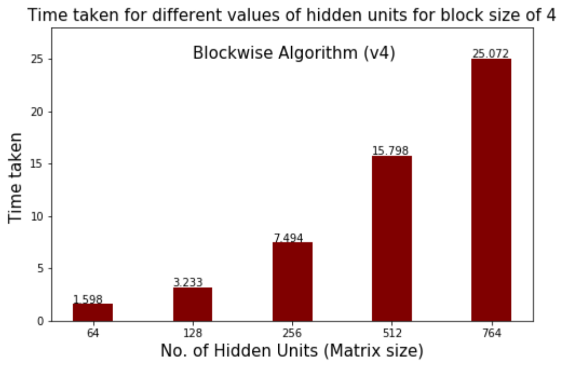
Although, this version has better time performance than that of version1 and version2. In previous versions, for hidden unit of 512, the least multiplicative increase in time taken is about 19-20 times. And, In this version, for same value, this value is about 4-5 times. So, this is a good improvement as compared to previous versions. Still, we cannot achieve the desired results.i.e. decrease in time taken.

Version4

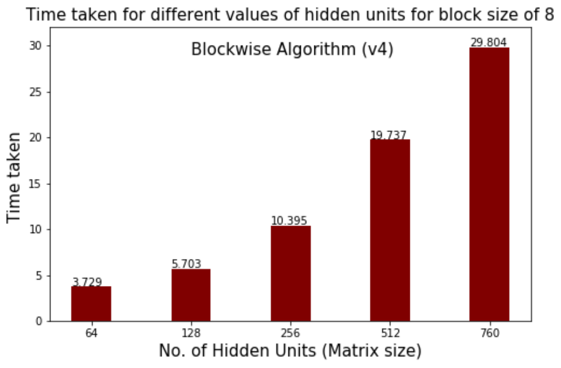
The value of time taken for different values of hidden units for element-wise and block-wise method are shown below :-



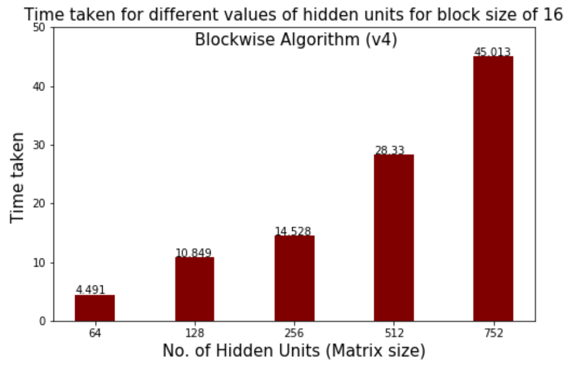
(a)



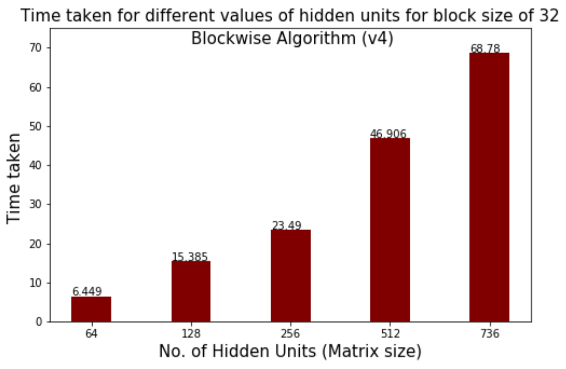
(b)



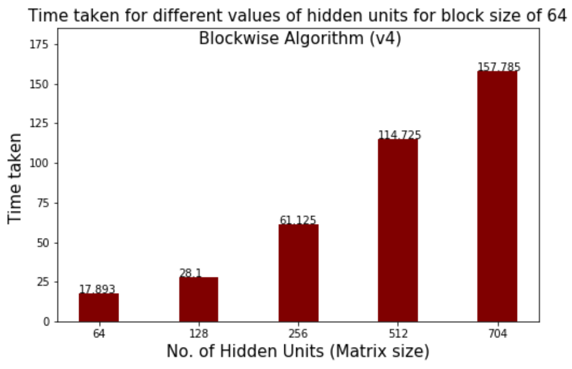
(c)



(d)



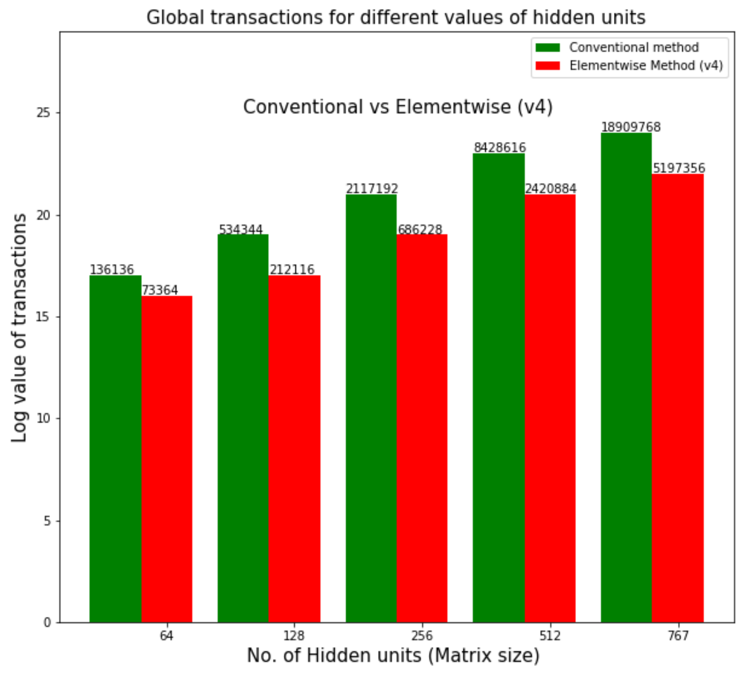
(e)



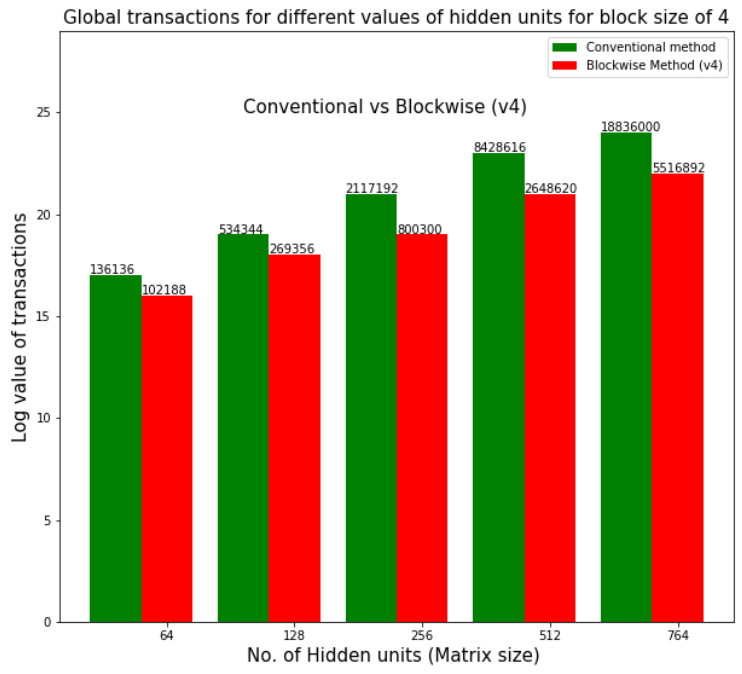
(f)

Figure 4.8: Time taken for different configurations in Version4

The value of global transactions for different values of hidden units for element-wise and block-wise method are shown below :-

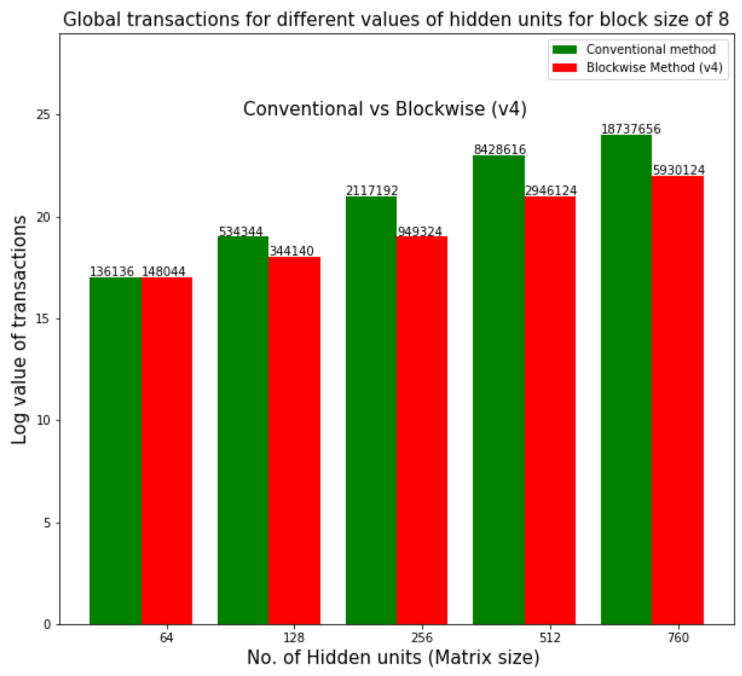


(a) For Element-wise approach

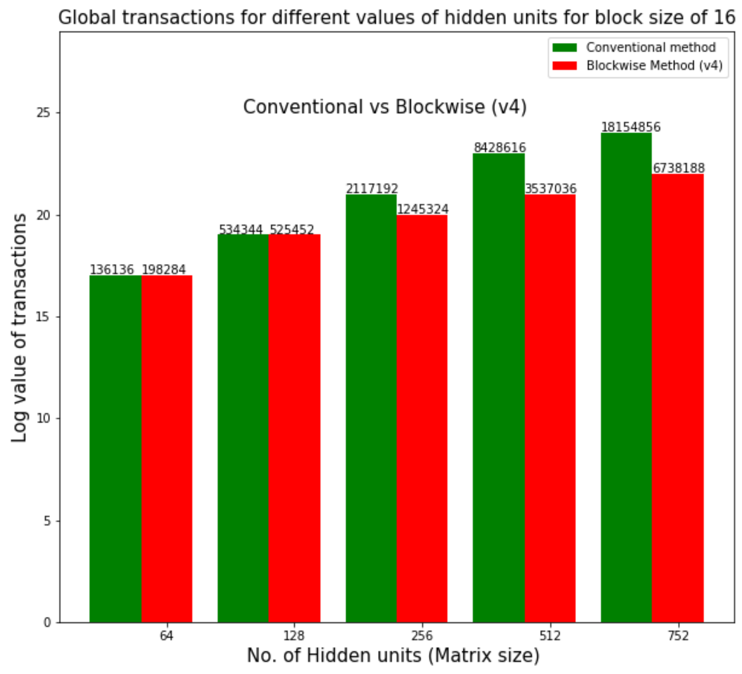


(b) For Block size of 4

Figure 4.9: Global transactions in Version4

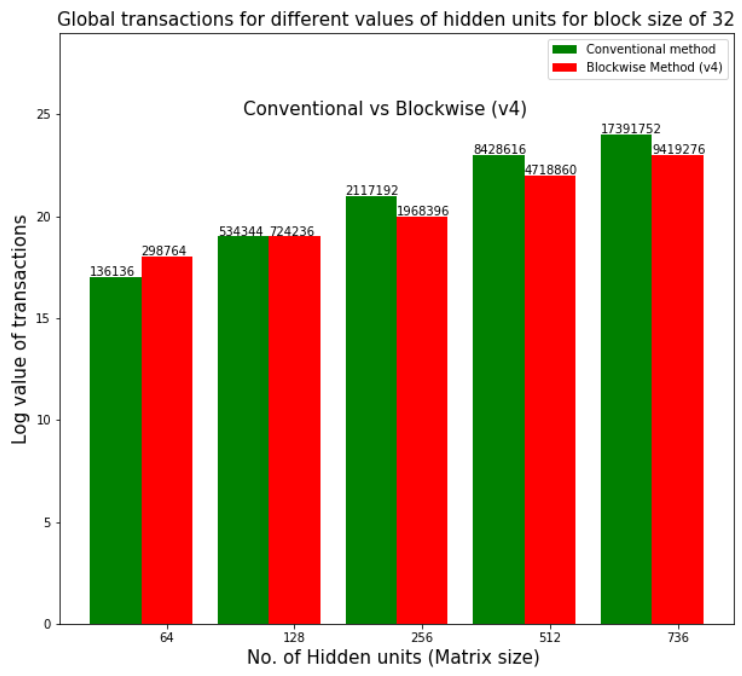


(c) For Block size of 8

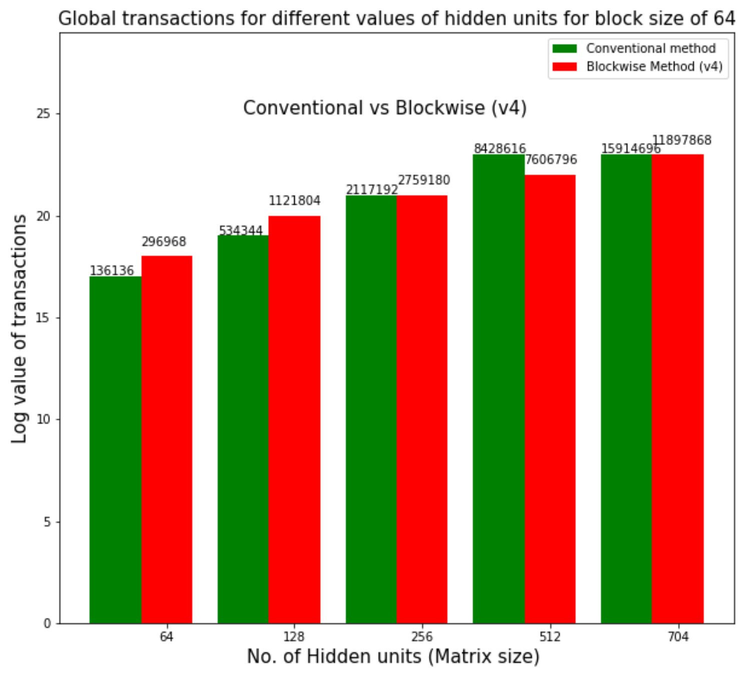


(d) For Block size of 16

Figure 4.9: Global transactions in Version4 (cont.)



(e) For Block size of 32



(f) For Block size of 64

Figure 4.9: Global transactions in Version4 (cont.)

The comparison of the Weight reuse method version4 with conventional method for the values of time and global transactions are given below in these tables.

- **For Element-wise method**

Hidden Units	Change % in Global Transactions	Change in Time
64	-46.11%	+4.09
128	-60.30%	+6.25
256	-67.59%	+3.96
512	-71.28%	+4.31
767	-72.51%	+4.35

Table 4.8: Table shows change in the global transactions and time taken of Version4 with respect to the conventional method for case of Element-wise method. Here,In case of global transactions, + sign represent increase percentage and − sign represent reduction percentage. And, for time, + sign represent multiplicative increase.

- **For Block size of 4**

Hidden Units	Change % in Global Transactions	Change in Time
64	-24.94%	+4.15
128	-49.59%	+6.35
256	-62.20%	+4.16
512	-68.58%	+4.51
764	-70.71%	+4.59

Table 4.9: Table shows change in the global transactions and time taken of Version4 with respect to the conventional method for block size of 4. Here,In case of global transactions, + sign represent increase percentage and − sign represent reduction percentage. And, for time, + sign represent multiplicative increase.

- **For Block size of 8**

Hidden Units	Change % in Global Transactions	Change in Time
64	+8.75%	+9.69
128	-35.60%	+11.20
256	-55.16%	+5.77
512	-65.05%	+5.63
760	-68.35%	+5.50

Table 4.10: Table shows change in the global transactions and time taken of Version4 with respect to the conventional method for case of block size of 8. Here,In case of global transactions, + sign represent increase percentage and − sign represent reduction percentage. And, for time, + sign represent multiplicative increase.

- **For Block size of 16**

Hidden Units	Change % in Global Transactions	Change in Time
64	+45.65%	+11.66
128	-1.66%	+21.31
256	-41.18%	+8.07
512	-58.04%	+8.08
752	-62.88%	+8.37

Table 4.11: Table shows change in the global transactions and time taken of Version4 with respect to the conventional method for block size of 16. Here,In case of global transactions, + sign represent increase percentage and − sign represent reduction percentage. And, for time, + sign represent multiplicative increase.

- **For Block size of 32**

Hidden Units	Change % in Global Transactions	Change in Time
64	+119.46%	+16.75
128	+35.54%	+30.23
256	-7.03%	+13.05
512	-44.01%	+13.38
736	-45.84%	+13.00

Table 4.12: Table shows change in the global transactions and time taken of Version4 with respect to the conventional method for block size of 32. Here,In case of global transactions, + sign represent increase percentage and − sign represent reduction percentage. And, for time, + sign represent multiplicative increase.

- **For Block size of 64**

Hidden Units	Change % in Global Transactions	Change in Time
64	+118.14%	+46.48
128	+109.94%	+55.21
256	+30.32%	+33.96
512	-9.75%	+32.73
704	-25.24%	+31.57

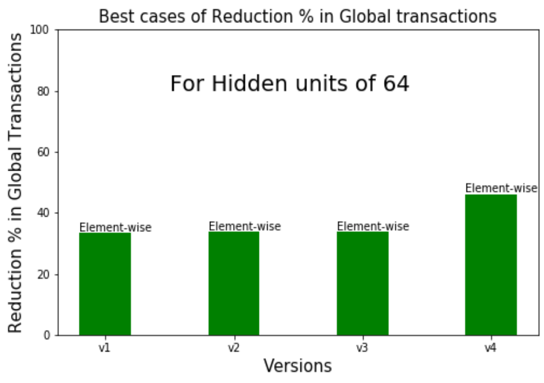
Table 4.13: Table shows change in the global transactions and time taken of Version4 with respect to the conventional method for block size of 64. Here,In case of global transactions, + sign represent increase percentage and − sign represent reduction percentage. And, for time, + sign represent multiplicative increase.

From the above graphs and tables, it can be observed that version4 has similar time performance as that of version3. But, there is more reduction in global transactions in most of the cases as compared to that of all the previous three versions. Here, the element-wise and block-wise method for smaller block size of 4 and 8 have similar time performance. For bigger block sizes of 32 and 64, there is a high increase in value of time taken in comparison to that of conventional method.

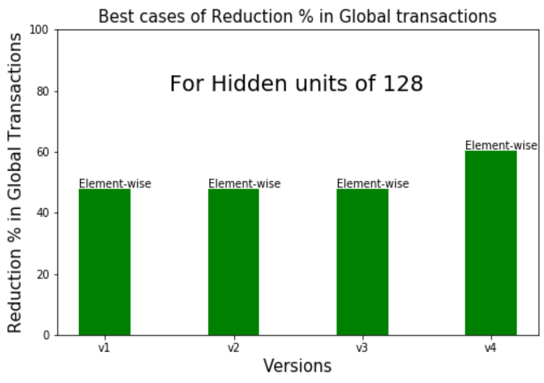
For global transactions, its trends are similar to that of all the previous three version3. As the value of hidden units increases, the reduction percentage of global transactions increases. In case of block-wise method, there is no reduction in global transactions for smaller value of hidden units.

Similar to version3, this version has better time performance than that of version1 and version2. In version1 and version2 for hidden unit of 512, for hidden unit of 512, the least multiplicative increase in time taken is about 19-20 times. And, in this version, for same value, this value is about 4-5 times. We have obtained significant reduction in global transactions, but we are not able to reduce the time taken to process the LSTM algorithm.

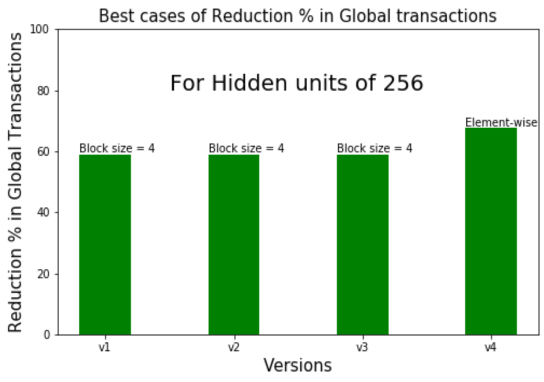
4.3 Best cases of all the weight resue versions



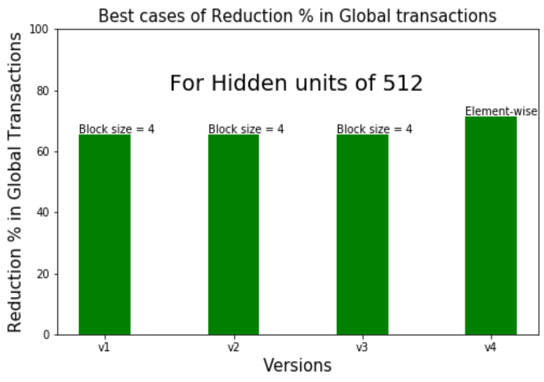
(a)



(b)

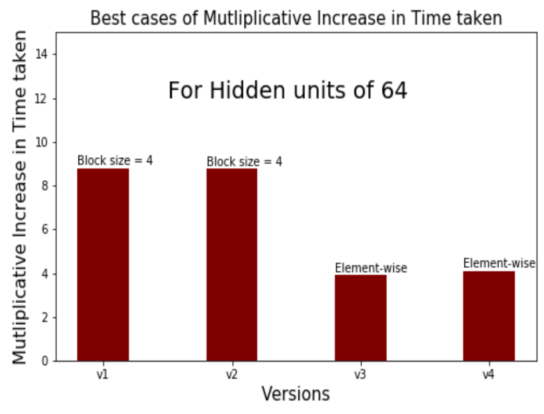


(c)

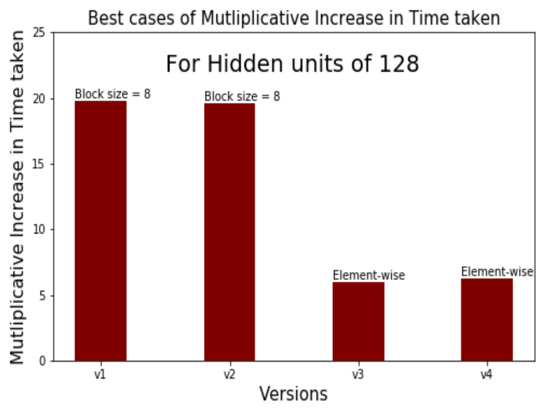


(d)

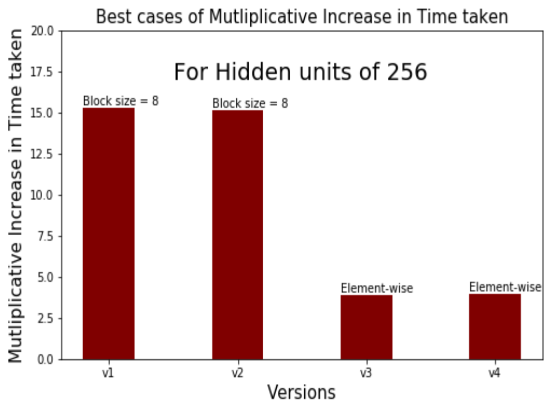
Figure 4.10: Best cases of Reduction % in Global Transactions in all the versions



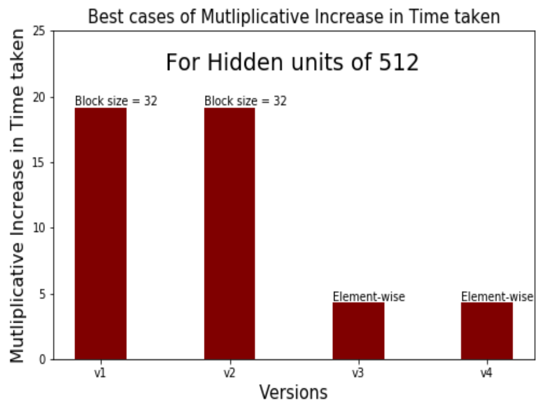
(a)



(b)



(c)



(d)

Figure 4.11: Best cases of Multiplicative increase(i.e least increase) in Time taken in all the versions

The best cases of all the versions on the basis of global transactions and time taken, are shown in above graphs. From them, it can be observed that :-

- The reduction in global transactions for Version4 is more than that of all the other versions.
- The multiplicative increase in time taken for Version3 and Version4 is very less than that of Version1 and Version2. For hidden unit of 512, the least multiplicative increase in time taken for Version1 and Version2 is about 19-20 times. And, for Version3 and Version4, it is about 4-5 times. This is a good improvement.
- For these versions, it is not necessary that the best case for global transactions is same as that of time taken for any value of hidden unit.

4.4 Discussion about the results obtained :-

We have tried to reduce time taken by reducing global transactions (or global memory accesses) of output weight matrices(W_h) of the LSTM network. For this, we have implemented four different versions of the method. But, we are able to reduce amount of global transactions, but not time taken. In each section, most of the results obtained are similar. We have discussed about the results obtained in all the versions in this section.

4.4.1 No reduction in time taken :-

In each version, we have obtained the reduction in global transactions, but not in the time taken. The reasons for this thing is discussed below :-

- For Weight Reuse method, in each time step, the present output vector is computed completely and the next output vector is computed partially. On the other hand, for conventional method, only present output vector is computed in each time step.
- In this method, the previous output vector(h_{t-1}) is required for computation of the present output vector(h_t). And, when **cond** $=$ **0**, for the partial computation of any element with particular index i of next output vector(h_{t+1}), the fully computed elements of present output vector(h_t) from index 1 to i are required. When **cond** $=$ **1**, for the partial computation of any element with particular index i of next output vector(h_{t+1}), the fully computed elements of present output vector(h_t) from index $numHiddenUnits$ to $i+1$ are required. So, when we are doing computation of elements of present output vector and next output vector, we apply weight reuse method by storing element of W_h in the local memory or registers. Then, they are used for the computation. This causes time overhead.
- As discussed in above point, there is a dependency in the weight reuse method. So, more than one element (or chunk) of present output vector(h_t) cannot be calculated simultaneously, which is possible in conventional method. Here, when **cond** $=$ **0**, we cannot start partial computation of any higher index element of next output vector(h_{t+1}) without completely calculating all its lower index elements of present output vector(h_t). When **cond** $=$ **1**, we cannot start partial computation of any lower index element of next output vector(h_{t+1}) without completely calculating all its higher index elements of present output vector(h_t). And, as, we are using Weight reuse method, For particular index, we have to do complete calculation of that element for present output vector(h_t) and partial calculation of that element of next output vector(h_{t+1}) in one step. This is the main reason due to which time reduction is not obtained.
- In Version1 and Version2, there is a need to use barrier functions and atomic functions. Use of this functions also make our implementation slow. In version3 and version4, only barrier function is used. Due to this, time taken by them is less than that in version1 and version2, but not better than that of conventional method.

4.4.2 Reduction in global transactions for most cases :-

In each version, We have obtained the reduction in the global transactions. In version4, we have obtained better results because the elements (or chunks) of present output vector(h_t)

and next output vector(h_{t+1}) are stored in shared memory. The reasons for this observations are discussed below :-

- These global transactions give the measure of memory accesses in the GPU(or CUDA) programs. The main reason for the reduction in the global transactions is the use of weight reuse method. In each time step, we are using only about half of the elements of output weight matrices(W_h). In first time step(When cond = 0), we are accessing only lower diagonal and diagonal elements. In second time step(When cond = 1), we are accessing only upper diagonal element of the output weight matrices(W_h).
- In many cases, we have obtained more than 50% reduction in global transactions. And, in some cases, it is about 65%-68%. This is possible due to the use of shared memory. It is faster memory in each thread-block of kernel function. In our implementation, it is used to store temporary results. Most of the computations are done through them, thus saving repeated access of global memory.

4.4.3 No or less reduction in global transactions for higher block sizes :-

In each version, for higher block sizes of 32 and 64, We have obtained no reduction in amount of global transactions for lower value of hidden units of 64 and 128, but the amount of global transactions are increased. And, for higher value of hidden units, there is less reduction in global transactions as compared to the result of element-wise approach and lower block sizes. The reason for this thing is discussed below :-

- In each thread-block, number of registers are limited.i.e.(65536). Each register has limited per thread local memory (made up of these registers). In our implementation, the threads have to store the blocks or elements of output weight matrices (W_h) into these per-thread local memory. When, the size of per-thread local memory get full, then remaining data is stored in global memory through proper mapping. This condition is called register spilling.
- Register spilling happens in the case of higher block sizes of 32 and 64. For smaller value of hidden units of 64 and 128, it lead to increase in the amount of global transactions. On the other hand, for higher value of hidden unit, due to this, there is less reduction in global transactions as compared to the result of element-wise approach and lower block sizes.

Chapter 5

Conclusion

5.1 Analysis of the Weight Reuse Method

The main objective of the thesis was to reduce time taken by LSTM algorithm by reducing the global transactions of the LSTM network. For this, we have used Weight reuse method. We have implemented four versions of this method. In each version, we are able to reduce global transactions, but are not able to reduce time taken in comparison to conventional method. In this section, brief analysis of all the versions is done below :-

5.1.1 Analysis about time taken :-

The analysis about time taken by all the versions is given below :-

- The Weight Reuse method has sequential dependency that more than one element of present output vector cannot be computed simultaneously (discussed in 4.4.1).
- In Version1 and Version2, for hidden unit of 512, the least multiplicative increase in time taken is about 19-20 times (very high). This is due to the large use of barrier functions and atomic functions.
- In Version3 and Version4, for hidden unit of 512, the least multiplicative increase in time taken is about 4-5 times. This is an improvement in time performance as compared to that of version1 and version2. Here, only barrier functions are used. And, atomic functions which causes too much overheads, are not used in these implementations.
- On increasing time steps further, the improvement in time performance can be seen.

5.1.2 Analysis about Global transactions :-

The analysis about global transactions by all the versions is given below :-

- As, the size of block size increases, the amount of reduction in global transactions decreases. It is due to register spilling. So, the larger block sizes should not be used.
- For Version4, as the elements (or chunks) of previous output vector(h_{t-1}) and present output vector(h_t) are stored in shared memory, there is a better reduction of global transactions.
- On increasing time steps further, the improvement in the reduction of global transactions can be seen.

5.1.3 Analysis about the parameters of Block size and Maximum value of Hidden units

The analysis about block size and maximum value of hidden units are given below :-

- For block-wise approach in all the versions, each thread is accessing the blocks of output weight matrices W_h. For higher block sizes, this causes register spilling, which donot give the desired reduction of global transactions. So, we should not use higher block sizes of 32 and 64 for our implementation.
- In the Version1 and Version2, the maximum value of hidden units for element-wise approach is equal to maximum number of threads in a thread-block i.e.1024.
- In Version3, the maximum value of hidden units depend upon the size of shared memory(i.e.48KB) and the datatype that is used. In case of double datatype, the maximum value of hidden unit is 768.
- In Version4, the maximum value of hidden units depend upon the size of shared memory, the datatype that is used and block size of the LSTM algorithm.

5.2 Summary

In this thesis, the concepts of LSTM algorithm, GPU and CUDA programming are explored. The features of CUDA programming like kernel function, barrier functions, atomic function, heterogeneous computing etc are also mentioned. Then, we have introduced the Weight Reuse method. By this method, we are trying to reduce the time taken by reducing the global memory accesses (global transactions) of the LSTM network. For this method, we have implemented four versions. All these four versions are described in the previous chapters. Out of the four versions, Version3 and Version4 give best time performance, but not better than that of conventional method. And, Version4 give best performance on the basis of reduction in global transactions. We have discussed about the results obtained. Though, we are not able to reduce time taken, but we are able to reduce global transactions. While exploring various papers, I have found that power and energy consumption[6] is reduced on reducing the global memory accesses (global transactions) of the LSTM network. So, In future, our work can be used in the reduction of power and energy consumption.

5.3 Future Works

We can do a lot of things to make this work better and useful. Some of them are given below :-

- As the global transaction are reducing by the Weight reuse method, So, it can be used to decrease the power consumption and energy consumption of the LSTM network.

- In our implementation, each thread of kernel function is accessing the blocks of weight matrices. Here, dynamic parallelism can be used. In this, a child kernel function can be called from parent kernel function and threads of child kernel will access different element of blocks of weight matrices
- We have done all the processing related to output weight matrices W_h in a single thread-block. For using multiple thread-block, we have to apply global synchronization or use global barrier.
- There are various CUDA concepts like streams and events that can be used.

Bibliography

[1] *CUDA C++ programming guide*. URL: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[2] *CUDA profiler user guide*. URL: `https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview`.

[3] Yijin Guan. "FPGA-based Accelerator for Long Short-Term Memory Recurrent Neural Networks". In: (2017). URL: `https://ieeexplore.ieee.org/document/7858394`.

[4] Andrey Ignatov. "AI Benchmark: All About Deep Learning on Smartphones in 2019". In: (2019). URL: `https://arxiv.org/abs/1910.06663`.

[5] Phil Blunsom Jeremy Appleyard. "Optimizing Performance of Recurrent Neural Networks on GPUs". In: (2016). URL: `https://arxiv.org/abs/1604.01946`.

[6] Naebeom Park. "Time-Step Interleaved Weight Reuse for LSTM Neural Network Computing". In: (2020). URL: `https://www.researchgate.net/publication/343556419_Time-step_interleaved_weight_reuse_for_LSTM_neural_network_computing`.

[7] Aruna Balasubramanian Qingqing Cao Niranjan Balasubramanian. "MobiRNN: Efficient Recurrent Neural Network Execution on Mobile GPU". In: (2017). URL: `https://www.researchgate.net/publication/317356279_MobiRNN_Efficient_Recurrent_Neural_Network_Execution_on_Mobile_GPU`.

[8] Anshul Kumar Saurabh Tewari Kolin Paul. "Split And Combine Approach to Reduce the Off-chip Memory Accesses of LSTM Accelerators". In: ().

[9] Jing Wang Xingyao Zhang Chenhao Xie. "Towards Memory Friendly Long-Short Term Memory Networks (LSTMs) on Mobile GPUs". In: (2018). URL: `https://ieeexplore.ieee.org/document/8574539`.