THESIS

MAX-PLUS MATRIX MULTIPLICATION LIBRARY FOR GPUS - MPMML

Submitted by

Prerana Prakash Ghalsasi

Department of Electrical and Computer Engineering

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2019

ABSTRACT

MAX-PLUS MATRIX MULTIPLICATION LIBRARY FOR GPUS - MPMML

Max-Plus algebra finds its applications in discrete event simulations, dynamic programming, biological sequence comparisons etc. Although there exist highly tuned libraries like CUDA Linear Algebra Subprograms (CuBLAS) [1] for matrix operations, they implement the standard matrix-multiplication (multiply-add) for floating points. We found no standard library for Max-Plus-Matrix-Multiplication (MPMM) on integers. Hence,we developed a highly tuned parallelized MPMM library kernel. We chose GPUs as hardware platform for this work because of their significantly more parallelism and arithmetic functional units as compared to CPUs. We designed this kernel to be portable across three successive Nvidia GPU architectures and it achieves performance in the range 3065 GOPs/S - 3631 GOPs/S on all of these architectures. We closely followed the benchmarking approach described by Volkov et al. [2] when they contributed to cuBLAS. This MPMM kernel can be part of a max-plus algebra library for GPUs and can help speed up Biological Sequence comparison applications like BPMax.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Sanjay Rajopadhye for his wise and invaluable guidance throughout my Master's degree and the development of this thesis. I am also grateful to him for having the patience to let me take the time to get this right.

I would also like to thank Melange team at CSU. Our many conversations were essential in making this work a reality. Whenever I was stuck they helped me get going again and whenever I thought I had it all figured out they found a way to trip me up.

Finally, I owe a great debt of gratitude to my support network outside of the lab. My parents Prakash Ghalsasi and Vrushali Ghalsasi, my siblings Preshita Ghalsasi and Pranav Ghalsasi And my friend Muthu Kumaran Sampath always believed in me and that confidence was what allowed me to push through the most difficult times.

DEDICATION

I would like to dedicate this thesis to my parents, siblings and Friends.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Tropical algebra is a relatively new field in mathematics. The term 'tropical' is said to be first used by French mathematicians [3] in respect of Brazilian colleague Imre Simon [4] who is considered as pioneer of tropical algebra. Tropical algebra was originally developed in the context of discrete mathematics and optimization but later on scientists realized the power of its applications in different fields like computational algebra, discrete event simulation [5], operating systems [6], dynamic programming, modeling communication networks, biological sequence comparisons [7], social network analysis, petri net, hidden Markov models, and other related applications. Tropical algebra can be used to represent and solve real world problems [8]. Tropical algebra consists of max-plus algebra and min-plus algebra. In max-plus algebra, the tropical sum of two numbers is their maximum and the tropical product of two numbers is their sum [10]. Equations 1.1 and 1.2 depict these operations.

$$x \bigoplus y = max(x, y) \tag{1.1}$$

$$x \bigodot y = x + y \tag{1.2}$$

Basic matrix multiplication can be expressed as :

$$C_{M\times N} = A_{M\times P} \cdot B_{P\times N} = \sum_{i,j,k=0}^{M,N,P} c_{ij} = a_{ik} \cdot b_{kj} \tag{1.3}$$

This Matrix-multiply operation is translated in max-plus algebra as:

$$c_{ij} = c_{ij} \bigoplus a_{ik} \bigodot b_{kj} = max(c_{ij}, a_{ik} + b_{kj}) \tag{1.4}$$

Researchers from different scientific areas solve their problems using max-plus algebra. However, there is no standardized highly optimized library for such operations.

In recent years there has a been a growing interest in the use of Graphics Processing Units (GPUs) to accelerate non-graphics applications, a phenomenon known as general-purpose GPU (GPGPU) computing [9] [10]. As GPUs are ubiquitous in modern systems, their use is important in getting maximum performance out of these systems. Relative to CPUs, GPUs provide significantly more parallelism and arithmetic functional units. This makes them ideal for accelerating highly-parallel and computationally intensive applications.

Most of the extensively used compute-intensive applications have matrix operations at their core. Basic Linear Algebra Subprograms (BLAS) are collection of optimized subroutines for executing basic vector and matrix operations on CPUs. CUDA Basic Linear Algebra Subprograms (CuBLAS) [1], was then developed by Nvidia with contributions from Vasily Volkov [2], Davide Barbieri and from the University of Tennessee [1].

In 2008, Volkov et al. [2] presented the fastest (at time) performing implementations of dense LU, QR and Cholesky factorizations running on a single or double NVIDIA GPUs. This achieved by using optimizations to match the capabilities of the hardware along with some other algorithm specific optimizations. The optimizations that could best exploit the hardware were selected based on results from detailed benchmarks that exercised GPU memory system, kernel start-up costs, and arithmetic throughput to expose the limits of performance of the hardware. This approach is particularly interesting trying to understand performance limitations of recent GPU architectures.

We took up a study similar to Volkov et al.to understand performance limitations of recent Nvidia GPU architectures for tropical semiring matrix operations as a starting point to develop the Max-Plus Matrix Multiplication benchmark described in this thesis document.

1.1 Background

1.1.1 GPU Architecture

GPUs are highly parallel platforms consisting of many simple cores. There are a few major GPU vendors, each with their own GPU architectures. In this work we have used GPU machines of successive generations of GPU architecture by Nvidia; namely GeForce GTX 980 and GeForce GTX TITAN X from the Maxwell architecture family (GM204) and GeForce GTX 1080 Ti from the Pascal architecture family (GP104)

In Maxwell GM204 architecture and the Pascal GP104 architecture the CUDA cores are clustered into groups known as Streaming Multiprocessors (SMMs). GeForce GTX 980 has 16 SMMs and GeForce GTX TITAN X has 24 SMMs. GeForce GTX 1080 Ti is equipped with 28 SMMs.

Each Maxwell and/or Pascal SMM consists of 128 CUDA cores. To keep the CUDA cores fully utilized, CUDA threads are launched in groups, known as warps. Thus multiple cores within an SMM may execute the same instruction simultaneously on possibly different data. Specifically, each SMM in Maxwell GM204 and GP104 can dispatch four warps per cycle with two instructions being issued for each warp (8 different instructions total per cycle). Each warp consists of 32 threads and each SMM can support a maximum of 64 warps simultaneously (a maximum of 2048 threads). To achieve good performance on these platforms, as with most GPU platforms, it is essential that applications expose sufficient parallelism to allow a high utilization of the platform's many cores.

GPU Memory Hierarchy

GPUs have a multi-tiered memory hierarchy whose characteristics have important performance implications. Below is a discussion of the various levels of this memory hierarchy. Figure 1.1 shows the simplest overview of this hierarchy for the Maxwell and Pascal architectures.

Global Memory

Global memory is the largest, off-chip, DRAM memory of the GPU. This memory is very slow compared to other memories in the hierarchy. One of the most important factors affecting performance is how data is loaded from global memory. If multiple consecutive threads all access a contiguous chunk of memory of a particular size these accesses are coalesced into a single memory transaction. Coalescing results in significantly fewer memory transactions, hence reducing memory access time.

Table 1.1: Global Memory Sizes available on GPU machines under consideration

NVidia GPU Machine	**Global Memory size**
GeForce GTX 980 (GM204)	4043 MBytes
GeForce GTX Titan X (GM204)	12212 MBytes
GeForce GTX 1080 Ti (GP104)	11178 MBytes

Texture Memory

Texture memory is a hardware-managed cache with multidimensional locality that is used to store certain data types. Data that makes use of this cache is initially stored in global memory but is cached in the texture cache. This cache can exploit locality in non-innermost dimensions of array data.

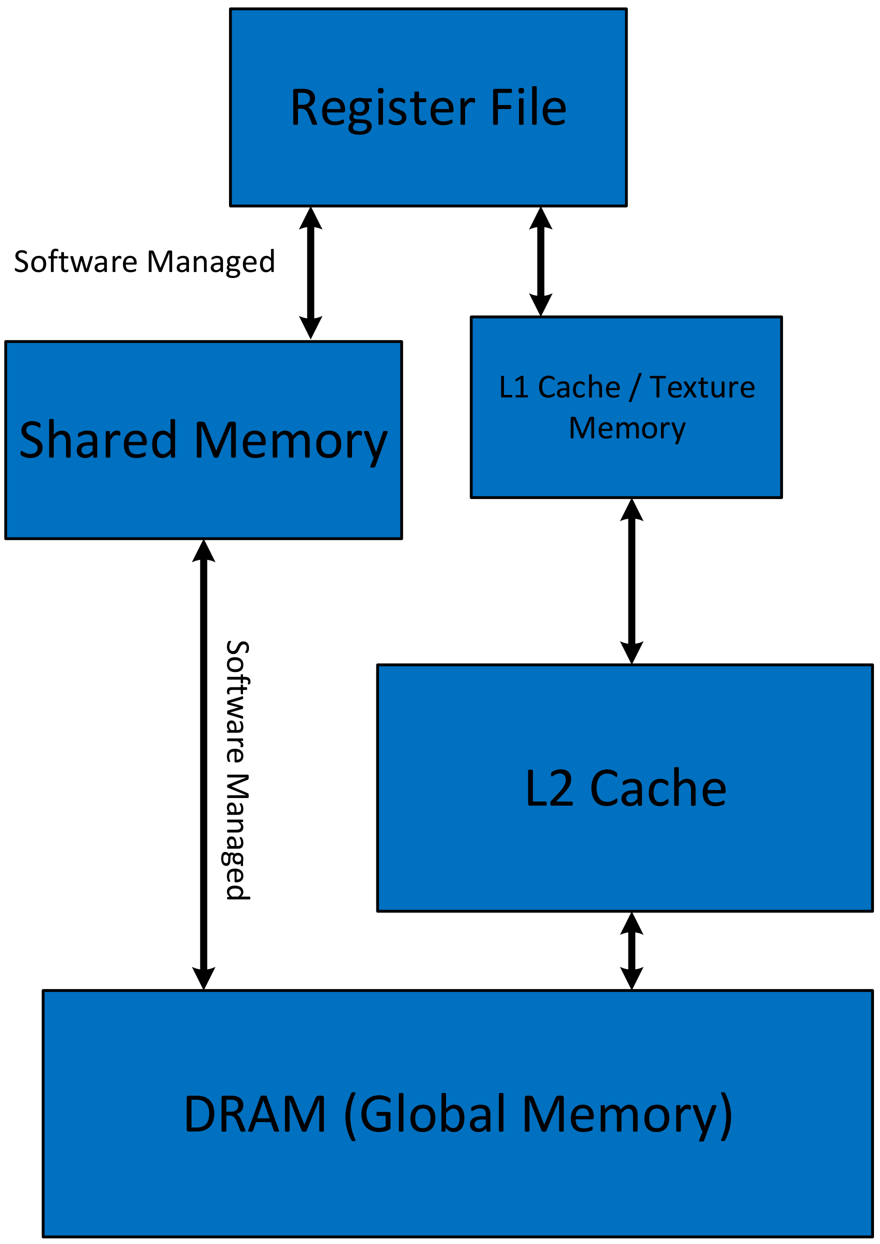

Figure 1.1: Overview of Memory Hierarchy of NVidia Maxwell/Pascal (GM204/GP104) - Adapted from GeForce GTX 980 Whitepapers [11] .

Shared Memory

Shared memory is a fast, programmable on-chip memory, and is one of the key architectural innovations of the first generation CUDA architecture present at the SM level. Since each SM has its own shared memory, it facilitates inter-thread communication. Thus, allows CUDA cores of an SM to work on data cooperatively and enables a broad range of applications to run efficiently on the GPU. However, the shared memory of one SM cannot be accessed by threads running on another SM. The GM204 and GP104 architectures have 96KB Shared memory per SM but only 48KB of it is available per thread block running on that SM.

Registers

The fastest level of the memory hierarchy are the registers. In both GM204 and GP104 architectures, each SM has 65536 32-bit registers. Despite the fact that the registers are physically present at the SM level, they are thread-specific and cannot be shared across threads within the same thread block.

The amount of shared memory used by each thread and the number of registers used per thread in an application impacts the GPU occupancy during the execution of the application.

1.1.2 CUDA programming model

CUDA® is a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs developed and maintained by NVIDIA. CUDA comes with a software environment that allows developers to use C as a high-level programming language. It also supports other languages, application programming interfaces, or directives-based approaches such as FORTRAN, DirectCompute, OpenACC.

CUDA provides a heterogeneous programming model where it assumes that a CUDA kernel executes separately on GPU Device which acts as co-processor to the CPU Host which runs sequential C program.

The GPU device kernel is launched as `devicekernel«<dimGrid, dimBlock»>` where,

- `dimGrid(x, y, z)` specifies the size of grid of Thread Blocks launched
- `dimBlock(x, y, z)` specifies the size of A Thread Block launched.

A thread block is a batch of threads that can cooperate with each other by:

- Sharing data through shared memory
- Synchronizing their execution

However, Threads from different blocks cannot cooperate with each other. For the ease of programming, Threads and Thread Blocks have 1D, 2D or 3D IDs. The Block IDs in a grid can be accessed using `BlockIdx.x/y/z` and the Thread IDs in a Thread Block can be accessed using `ThreadIdx.x/y/z`. This simplifies memory addressing when working with multidimensional data.

CUDA Memory Hierarchy

The CUDA Memory Hierarchy described in [12] suggest that A thread in a thread block can:

– Read/write per-thread registers.

Each thread can read and/or write to the registers specific to the thread but can not read/or write registers used by any other threads.

– Read/write per-block shared memory

All the threads in a Thread Block can Read and/or Write the shared memory available per Thread Block but, can not read/or write shared memory used by any other Thread Block.

– Read/write per-grid global memory

Any thread in a grid can Read and/or Write the global memory. However, it is advisable to introduce the `syncThreads()` after a read or write instruction before proceeding to next instructions that use the same data. This is to make sure there are no conflicts with threads accessing the same memory location.

1.2 Contribution

In this work:

- We present the microbenchmarking approach to model the behavior of the GPU hardware for Max-plus algebra operations. This study also provides guidelines for parametric design of MPMML.
- We then explain the detailed design steps to develop a parametric Max-plus matrix multiplication library. By adjusting these parameters, we make it portable across different GPU architectures.
- As a result, we provide a MPMML kernel that achieves performance in the range 3065 GOPs/S - 3631 GOPs/S on NVIDIA's GeForce GTX 980, GeForce GTX Titan X and GeForce GTX 1080 Ti for large matrices.

1.3 Thesis structure

Chapter 2 explains the motivation and approach of micro-benchmarking method used to exploit underlying hardware and find performance limits. Chapter 3 describes the detailed implementation of the Max-plus Matrix Multiplication library. In chapter 4 we provide experimental evaluation results. Chapter 5 presents conclusion and directs towards future expansion of this work.

Table 1.2: Technical Specifications of GPU platforms under consideration

	GeForce GTX 980	GeForce GTX Titan X	GeForce GTX 1080Ti
Compute Capability	5.2		6.1
SM Version	sm_52		sm_61
SMs	16	24	28
CUDA Cores/SM	128		128
GPU Max clock rate	1216MHz	1076MHZ	1582MHz
Operating Frequency with Turboboost	1.22 GHz	1.08GHZ	1.58GHz
Threads / Warp	32		32
Warps/Multiprocessor	64		64
Threads/Multiprocessor	2048		2048
Thread Blocks/Multi-processor	32		32
Shared Memory/Mul-tiprocessor (bytes)	98304		98304
Max Shared Memo-ry/Block (bytes)	49152		49152
Register File Size/-Multiprocessor (32-bit registers)	65536		65536
Max Registers/Block	65536		65536
Register Allocation Unit Size	256		256
Register Allocation Granularity	warp		warp
Max Registers/Thread	255		255
Shared Memory Allo-cation Unit Size	256		256
Warp Allocation Granularity	4		4
Max Thread Block Size	1024		1024

Chapter 2

Microbenchmarking

Inspired by Volkov et al. [2], we started with micro benchmarking the recent NVidia GPUs to reveal underlying hardware limitations for tropical semiring operations. The main purpose of this micro benchmarking was to find performance ceilings for the max-plus operation.

2.1 Register-Register access microbenchmark

The largest and fastest level of the on-chip memory hierarchy is the register file. We developed a micro benchmark that implements the max-plus algebra operation in Equation 2.1.

$$c = max(c, a + b) \tag{2.1}$$

Algorithm 1 Reg - Reg Access microbenchmark

```
1:  for $n \leftarrow 0$ to $N$ do
2:      for $i \leftarrow 0$ to $X$ do
3:          for $j \leftarrow 0$ to $Y$ do
4:              c[i,j] = maxc[i,j] , a[i] + b[j]
5:          end for
6:          a[i] = maxa[i] , a[i] + Const
7:      end for
8:      for $j \leftarrow 0$ to $Y$ do
9:          b[j] = maxb[j] , b[j] + Const
10:     end for
11: end for
12: return c[X×Y]
```

This microbenchmark computes an array c[X$\times$Y] using arrays a[X] and b[Y]; all read/written to/from the on-chip register file. This computation is performed multiple times in an aggressively unrolled loop. We make sure that the compiler does not optimize the operations by updating the values of a[] and b[] within the registers with similar max-plus operations in each iteration as shown in Listing 1

This Microbenchmark consists of one *addition* and one *max operation*. The *max* operation can be implemented in two ways:

A) By defining a compile time macro as **MAX = a>b? a:b**

B) By using the CUDA arithmetic instruction **max(a,b)**

We experimented with both above listed implementations of the *max* operation. A low-level parallel thread execution file (*.ptx file) generated for both the implementations translates these two into same PTX instruction which in turn is translated into the machine level instruction (INMNMAX(R1,—,Rn)) as seen in the objdump file. Hence we decide to use any of the implementation alternatively knowing that it would not affect the performance.

Operation Counter

To ensure we count the total number of operations done by this microbenchmark correctly, we introduced a device variable called *d_opctr* which is incremented by 2 after each max-plus operation. The *.ptx file generated for Register-Register access microbenchmark kernel shows that the total number of operations done by a single thread is $2 \times (X \times Y + X + Y)$ We use this method to count the operations done by a single thread for all of the other microbenchmarks in this work as well as the MPMML.

2.1.1 Experimentation

To measure the performance of Register-Register access microbenchmark, we launch the kernel on GPU for:

1. Different values of X and Y
2. Different values of number of threads in a 2D CUDA block defined as BS_i $\times$ BS_j where:
   (a) BS_i : #Rows of threads in a CUDA block
   (b) BS_j : #Columns of threads in a CUDA block
3. Different number of CUDA blocks in a 2D CUDA grid defined as GS_i $\times$ GS_j where:
   (a) GS_i : #Rows of CUDA blocks in a grid
   (b) BS_j : #Columns of CUDA blocks in a grid

We time the execution of only the GPU kernel launch on the host and calculate the performance in terms of GOPs/S.

We use NVCC compiler with CUDA driver Version 9.2 to compile the CUDA code along with GCC version 6.4.0. The CUDA Runtime is Version 7.5. We generate *.ptx file to reveal number of registers per thread. We then use this to calculate the GPU occupancy using [13] to predict the impact of varying Block Size to choose appropriate values of BS_i and BS_j for experimentation.

Resource	X=4 Y=4	X=4 Y=6
Registers/Thread	32	40

(a) Register allocation data obtained from *.ptx.

GPU Occupancy Data :	
Thread per block	1024
Registers per thread	32
Active Threads per Multiprocessor	2048
Active Warps per Multiprocessor	64
Active Thread Blocks per Multiprocessor	2
Occupancy of each Multiprocessor	100%

(b) GPU Occupancy data Registers/Thread = 32.

GPU Occupancy Data :	
Thread per block	1024
Registers per thread	40
Active Threads per Multiprocessor	1024
Active Warps per Multiprocessor	32
Active Thread Blocks per Multiprocessor	1
Occupancy of each Multiprocessor	50%

(c) GPU Occupancy data Registers/Thread = 40.

Table 2.1: GPU occupancy data Reg-Reg Access microbenchmark GeForce GTX 980.

The detailed plots of performance results for various values of all of the parameters listed above are in Appendix A

2.2 Shared memory access microbenchmark

To further study the effect of GPU memory hierarchy, we modified microbenchmark described in section 2.1 to include data transfers from shared memory. We implemented two versions of this modified microbenchmark.

A) One operand in shared memory: In this version, the operand b is stored in registers and updated in each iteration with max-plus operation similar as in the register-register access microbenchmark but, the operand a is stored and read from shared memory.

B) Both operands in shared memory: As the name suggests, in this version both the operands a and b are stored in and read from shared memory

We introduce a new outer loop k which encases the core computation loop in such a way that changing the upper bound on K will change frequency of shared memory read access per X*Y computations.

The size of shared memory available per block is 48KB. This restricts value of N which in turn restricts the number of times the inner loops iterate. This reduces the measurable time to of the order of microseconds making the measurements unreliable. To overcome this, we introduce another loop m with upper bound M which is a compile time parameter. Also, M = #Columns of shared_a and/or #Rows of shared_b. Within the computation loop, shared_a is read with m as it's column index. The drawback of this access pattern of shared_a is that it may introduce shared memory bank conflicts when it's value is power of 2.

Algorithm 2 Shared Memory Access microbenchmark : One operand in Shared Memory

```
1:  for $n \leftarrow 0$ to $N$ do
2:      for $m \leftarrow 0$ to $M$ do
3:          for $k \leftarrow 0$ to $K$ do
4:              for $i \leftarrow 0$ to $X$ do
5:                  for $j \leftarrow 0$ to $Y$ do
6:                      c[i,j] = max(c[i,j] , shared_a[f(TheadIdx.y,i)][m] + b[j])
7:                  end for
8:              end for
9:              for $j \leftarrow 0$ to $Y$ do
10:                 b[j] = maxb[j] , b[j] + Const
11:             end for
12:         end for
13:     end for
14: end for
15: return c[X×Y]
```

Algorithm 3 Shared Memory Access microbenchmark : Both operands in Shared Memory

```
1: for $n \leftarrow 0$ to $N$ do
2:     for $k \leftarrow 0$ to $K$ do
3:         for $i \leftarrow 0$ to $X$ do
4:             for $j \leftarrow 0$ to $Y$ do
5:                 c[i,j] = max(c[i,j] , shared_a[f(TheadIdx.x,i)] + shared_b[g(TheadIdx.y,j)]
6:             end for
7:         end for
8:     end for
9: end for
10: return c[X×Y]
```

2.2.1 Experimentation

To measure the performance of Shared Memory access microbenchmark, we launch the kernel on GPU for:

1. Different values of X and Y
2. K = 10, 5, 1
3. Different values of number of threads in a 2D CUDA block defined as BS_i × BS_j where:
   (a) BS_i : #Rows of threads in a CUDA block
   (b) BS_j : #Columns of threads in a CUDA block
4. Different number of CUDA blocks in a 2D CUDA grid defined as GS_i × GS_j where:
   (a) GS_i : #Rows of CUDA blocks in a grid
   (b) BS_j : #Columns of CUDA blocks in a grid

We time the execution of only the GPU kernel launch on the host and calculate the performance in terms of GOPs/S.

The detailed plots of performance results for various values of all of the above listed parameters are in Appendix A

2.3 Moral of the Story

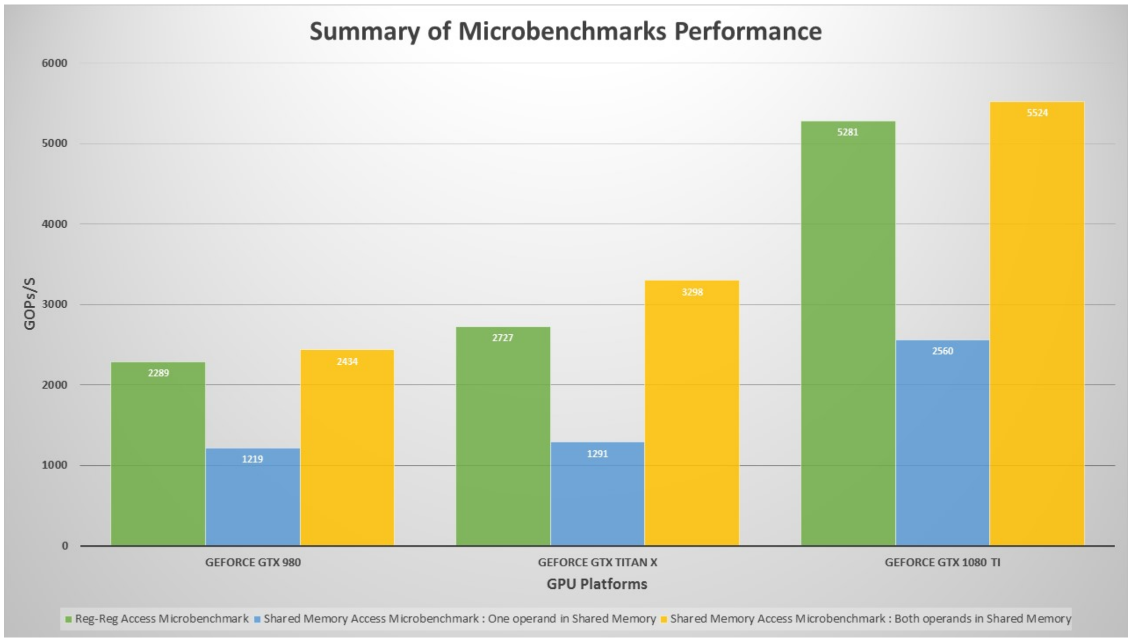

Figure 2.1: Microbenchmarks Performance Summary

Figure 2.1 shows the summary of the experimentation done to find out the performance bottlenecks of the GPU platforms under consideration. This approach is very similar to that of Volkov et al. [2] which involved saturating the underlying hardware to best utilize the hardware resources such as register file, shared memory, shared memory bandwidth, CUDA cores and in turn the SMs.

Intuitively, we predicted that the **Reg-Reg Access Microbenchmark** will achieve close to the TMP on all of the GPU platforms. While it's performance is not at the machine peak, it is the highest across all three GPU platforms under test compared to the other two microbenchmarks whose performance is discussed subsequently.

As we moved to implement **Shared Memory Access microbenchmark** for **One Operand**, we expected a drop in performance as we are now accessing a slower memory in the GPU memory hierarchy. We observed the a substantial drop in the performance when only one operand (operand A) is read from shared memory per two operations in the inner most loop of the computation. The performance of this microbenchmark is almost 50% of that of the Reg-Reg access microbenchmark across all three GPU platforms.

We extended this to **Shared Memory Access microbenchmark** for **Both Operands**. Here, we hypothesized to see a fruther drop in the performance as now now two operands are read from shared memory per two operations in the inner most loop of the computation. However, the observed performance rules out this performance. This Shared Memory access microbenchmark with both operands in shared memory, achieves GOPS/s very close to that of the Reg-Reg access microbenchmark. This performance result also rules out the possible effect of bank conflicts. This is a very interesting observation which suggests the evolution of the underlying hardware to better hide the shared memory latency. We take these insights from microbenchmarking effort and proceed to building a full implementation of Max-Plus-Matrix-Multiplication library.

Chapter 3

Max-Plus Matrix Multiplication Library

We now introduce our implementation of a Max-Plus Matrix multiplication Library. It provides a parameterized GPU-platform-portable function for Matrix-multiply operations in max-plus algebra as in 1.2. allowing programmers to use this function with parameter tuning to suit their applications. Our MPMML kernels support 32 bit integer data type. The optimization steps applied to achieve the maximum performance closely follow those steps followed for optimization of GEMM. A naive implementation of Equation 1.2 is represented by three nested loops as:

Algorithm 4 Naive MPMM

```
1: for $i \leftarrow 0$ to $M$ do
2:     for $j \leftarrow 0$ to $N$ do
3:         for $k \leftarrow 0$ to $K$ do
4:             C[i][j] = maxC[i][j] , A[i][k] + B[k][j]
5:         end for
6:     end for
7: end for
8: return C[M×N]
```

The i,j-th element of C is calculated using K elements in i-th row of A and j-th column of B O(N2) data is used for O(N3) math operations making the compute intensity of this implementation of the order of N for square matrices where M=N=K. But, the implementation is not compute bound because the reusable data cannot fit on fastest memory as for larger matrices.

We can apply loop permutation to above implementation such that the K dimension becomes the outermost loop. This permutation allows to read a column of A and a row of B only once. The *addition* and *max operation* is performed in the similar way partial accumulation of outer product in GEMM. This implementation still needs the entire C_{MxN} matrix to be available in the fastest memory so that the results of max operations can be stored simultaneously. This is a classic locality problem which can be solved by applying iteration space tiling such that each tile of C fits in registers of the target processor. This is a proven way of performance improvement. Furthermore, instead of exclusively performing max+ operations of vectors,we can perform it on blocked matrices and assigning each block to separate CUDA thread block. This gives additional opportunities to exploit both locality and parallelism.

3.1 MPMML Memory Hierarchy

We implement MPMM efficiently for GPUs by decomposing the computation into a hierarchy of **Thread Block tiles**, **Thread tiles** and **sub-patch tiles**. This hierarchy closely follows the NVIDIA CUDA programming model [12]. To experiment with different memory access patterns, we introduce two compile time parameters called MEMSSEL_A and MEMSEL_B so that MPMML can be compiled for four cases as:

Table 3.1: MPMML cases based on memory access pattern

Case	**MEMSEL_A**	**MEMSEL_B**	**Configuration**
00	0	0	Both A and B are loaded from shared memory to Registers
01	0	1	A is loaded from shared memory to registers and B is load from global memory to registers
10	1	0	B is loaded from shared memory to registers and A is load from global memory to registers
11	1	1	Both A and B are loaded from global memory to Registers

It is predictable that the performance of case IV would be the least of all. Hence, we implement it but do not experiment with it.

Figure 3.1, 3.2 and 3.3 show the data movement from global memory to shared memory (matrix to thread block tile), from shared memory to the register file (thread block tile to thread tile), and from the register file to the CUDA cores for computation for Case 00 , Case 01 and Case 10 respectively.

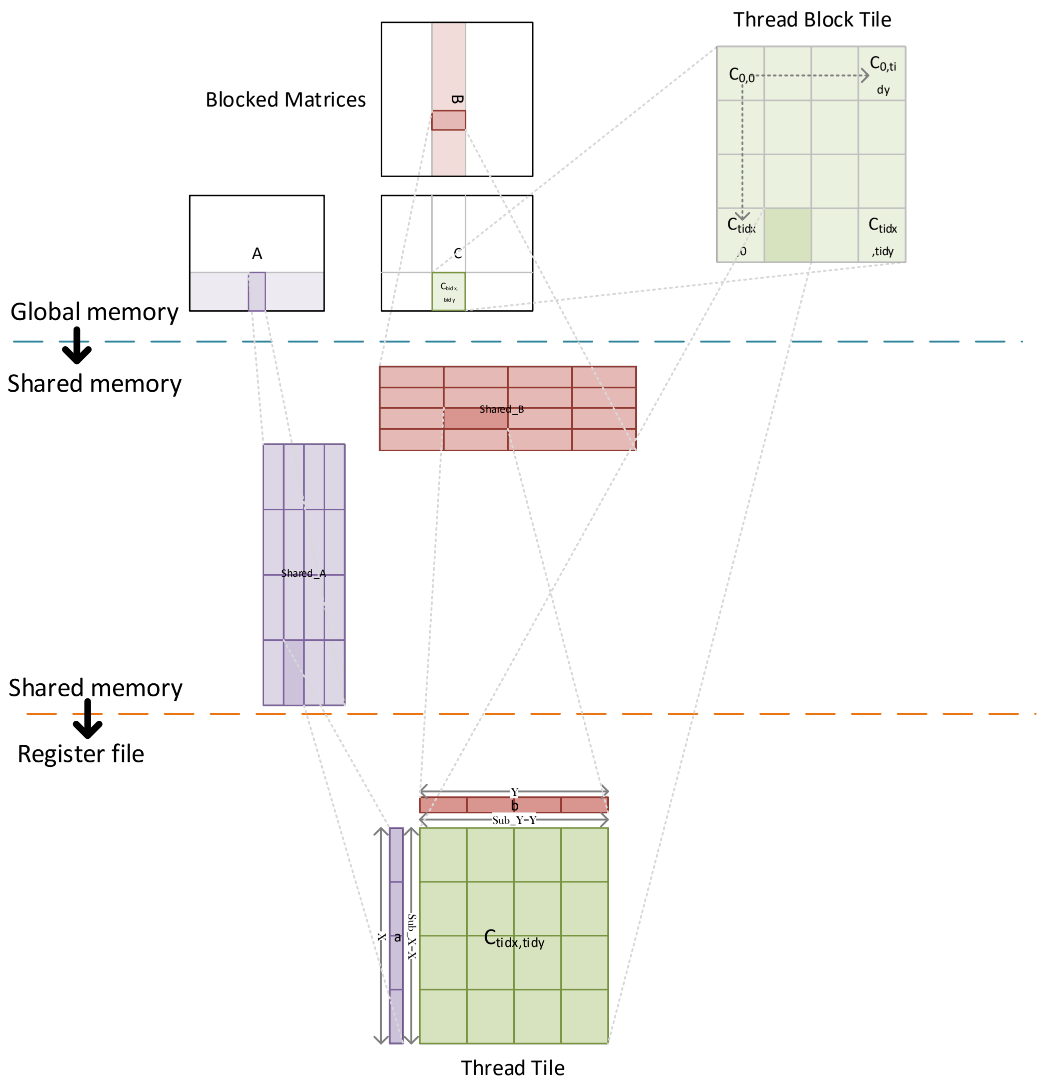

Figure 3.1: Case 00 : Both A and B operands in shared memory model.

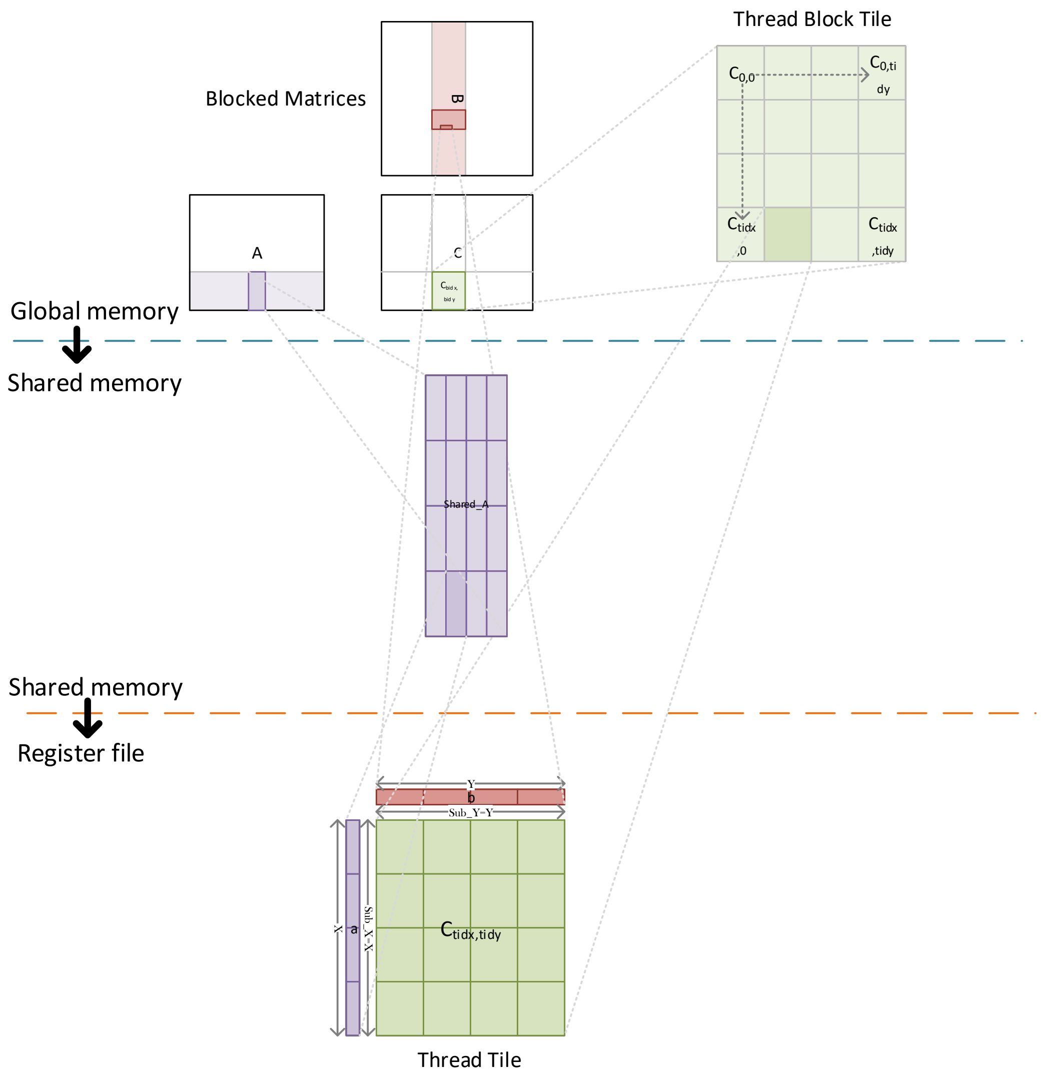

Figure 3.2: Case 01 : ’A’ operand in shared memory model.

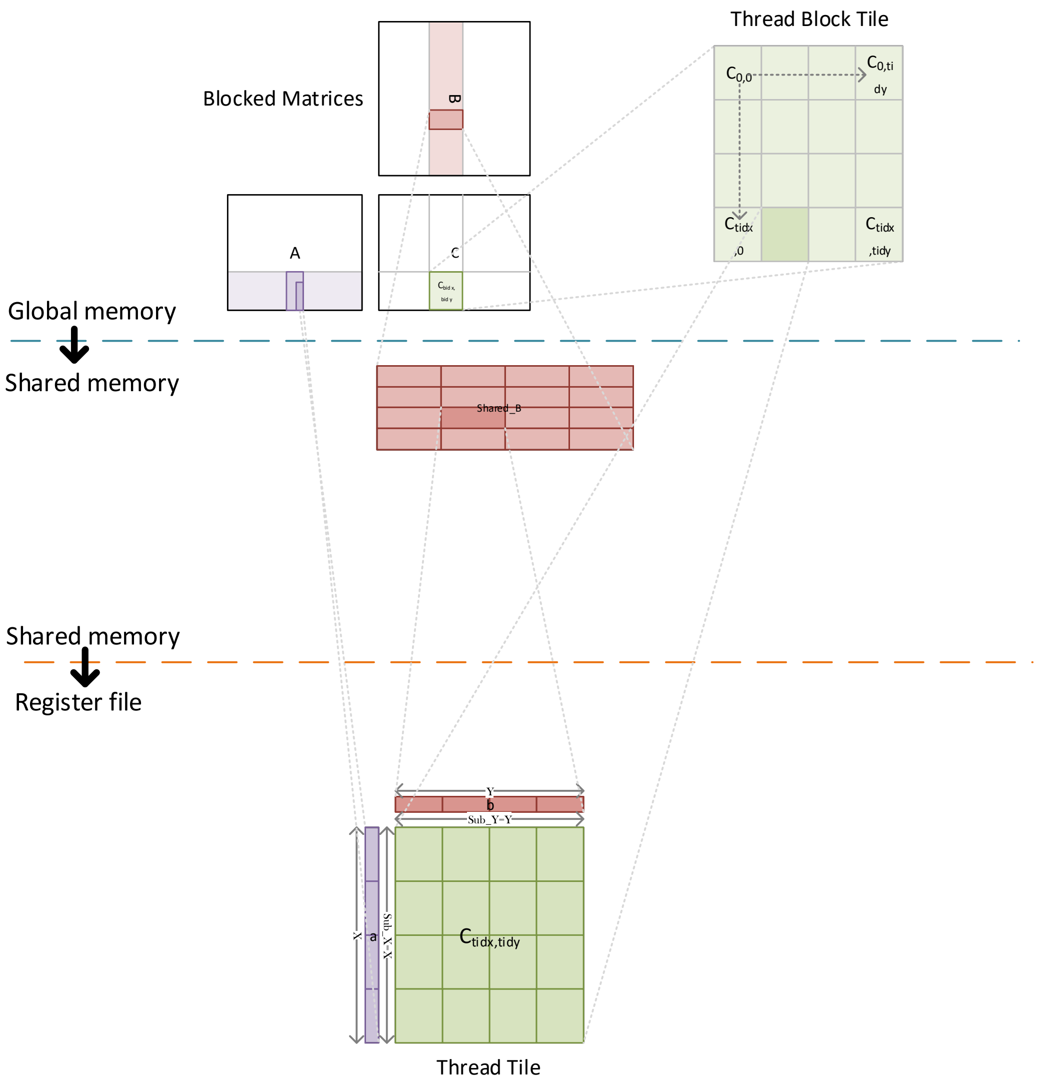

Figure 3.3: Case 10 : 'B' operand in shared memory model.

For the ease of explanation, we will describe hierarchy of case I in detail below.

3.2 Thread Block Tile

Each CUDA thread Block computes its part of the output MPMM by iteratively loading blocks of matrix data from the input matrices and computing an output matrix (C = max(C, A + B)).

Figure 3.4 shows the computation performed by a single thread block and highlights the blocks of data used in one iteration of its main loop.

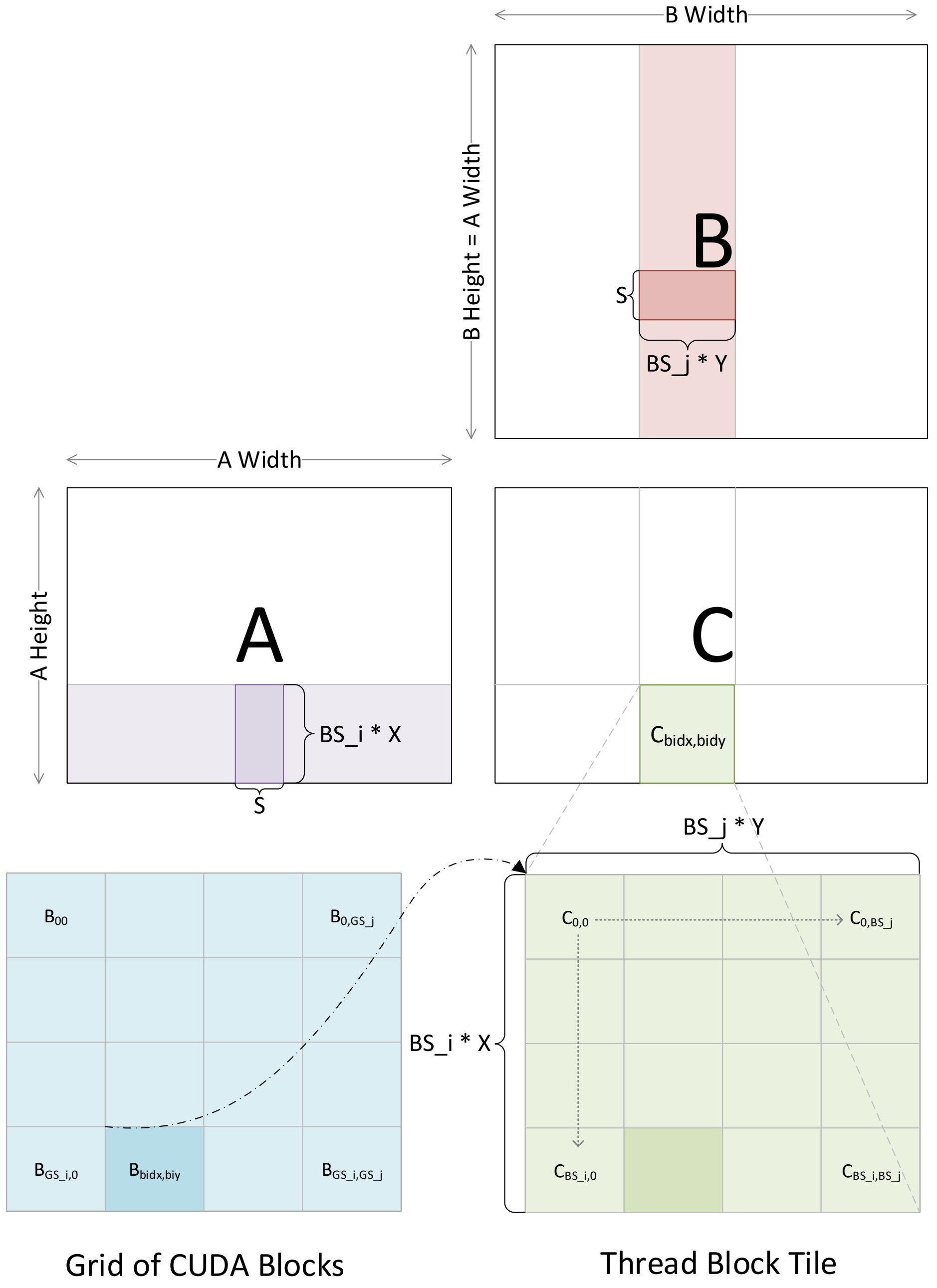

Figure 3.4: Detailed structure of Thread Block Tile.

A Height, *A Width* and *B Width* are command line inputs from the user. These input parameters are used to calculate the size of output matrix C as well as the size of Grid of CUDA Thread Blocks; GS_i (#Rows of thread blocks in a grid) and GS_j (#Columns of thread blocks in a grid). The submatrix of C shown in green is computed by the max-plus-matrix-product of a sub-matrix tile of A and a sub-matrix tile of B. This is performed by looping over the A and B, partitioned into tiles, and performing max operation on the results of max-plus-matrix-products of each tile.

Figure 3.5 shows that the Tiles of A and B are loaded from global memory and stored into shared memory accessible by all threads in a block. This load-store operation is done cooperatively by all threads and the threads then synchronize before the start of computation to make sure that all the required data is available in shared memory before it is loaded into registers for computation. In Case II this is done only for the tile of A and in Case III only for the tile of B. In case II, required elements of B are read directly from the global memory during computation and in case III , required elements of A are read directly from the global memory during computation.

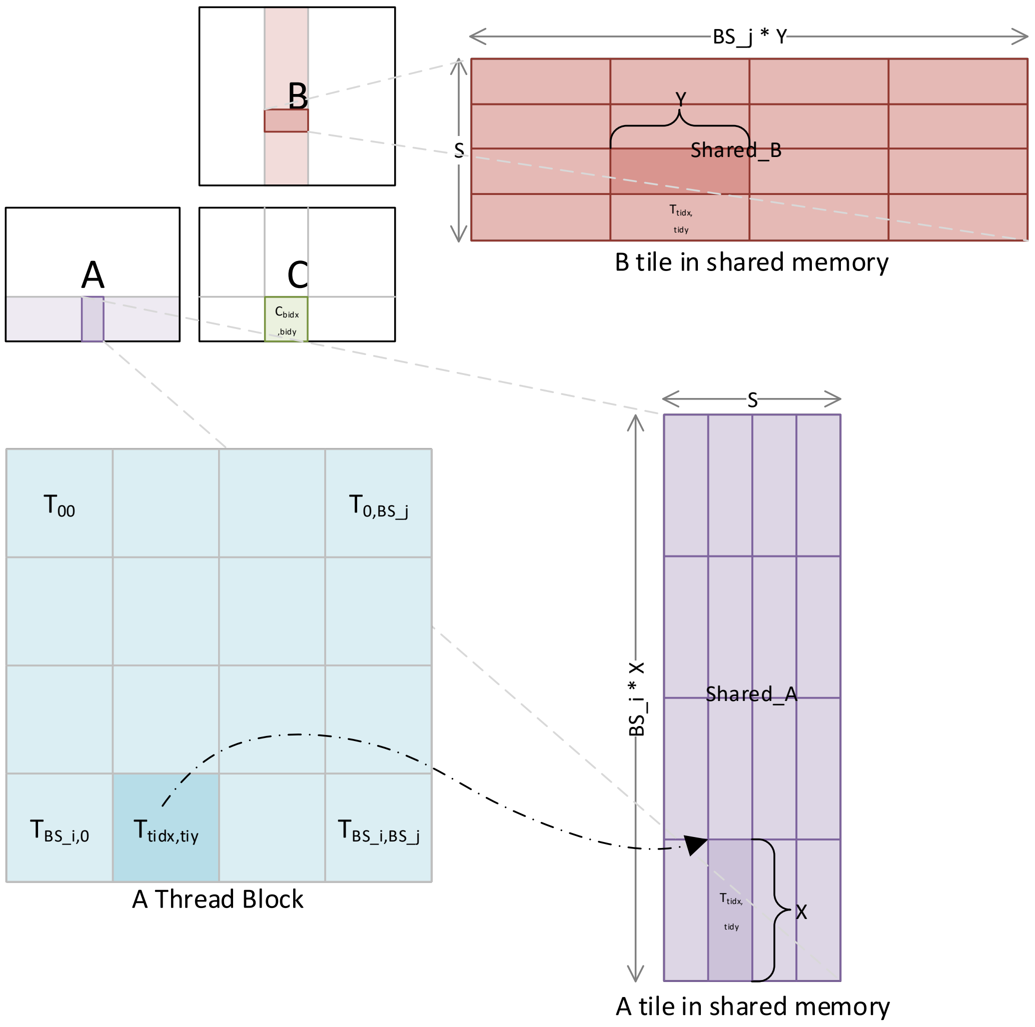

Figure 3.5: Tiles of A and B in Shared Memory.

A thread block's output tile is further partitioned across threads as Figure 3.4 shows. We define following compile time parameters that a programmer can specify to tune the MPMM computation for the target GPU platform and aspect ratio of a Thread Tile:

1. X = Rows of C tile computed by a thread
2. Y = Columns of C tile computed by a thread
3. BS_i = Rows of threads in a thread block

4. BS_j = Columns of threads in a thread block

Each of this partition is updated once per max-plus operation, so it needs to reside in the fastest memory in the SM: the register file as Figure 3.6 shows. Each thread computes this partition of C of size X*Y. Each thread first copies fragments of A_{X*1} and B_{1*Y} to registers from appropriate memory and the computes the max-plus product in outer-product fashion (product operation replaced by addition).

3.2.1 Thread tile

A thread computes a patch of C (X*Y elements of C) as shown in 3.6 using a partial column of A and a partial row of B. This is exactly the same as the computation in Reg-Reg access microbenchmark as described in 2.1.

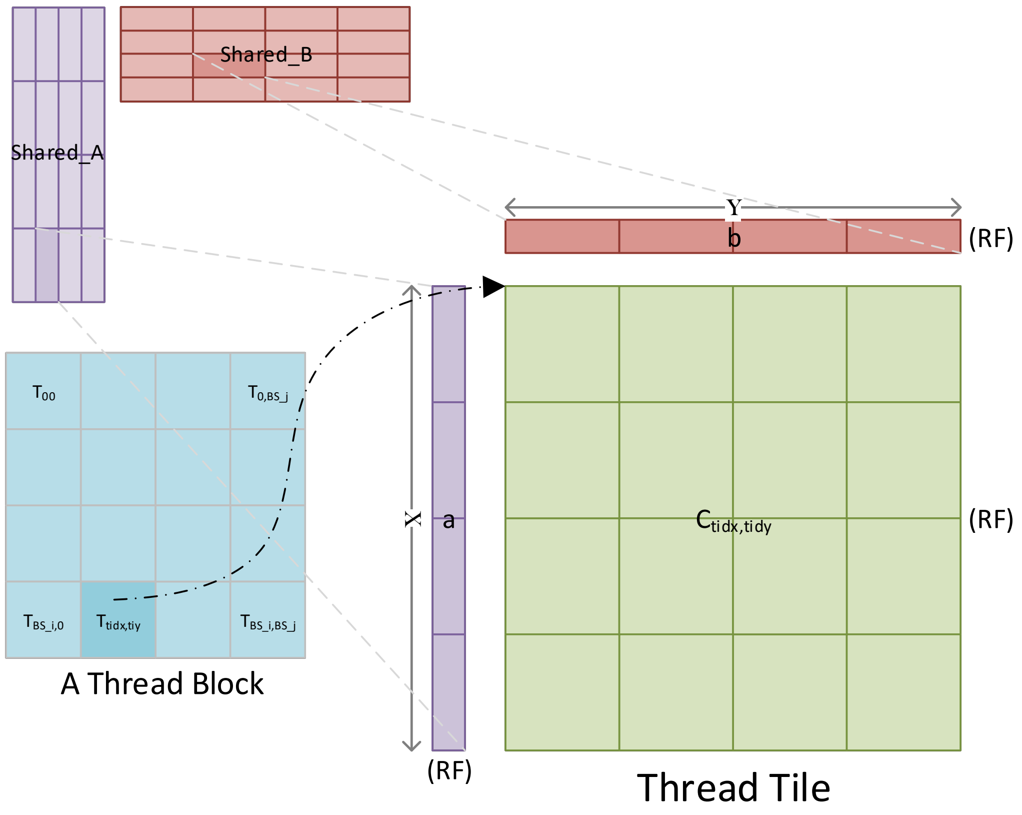

Figure 3.6: Detailed structure of a Thread Tile.

Each of this thread tile can be computed in three different computation patterns as described subsequently. To implement these three patterns, we introduce two more compile time parameters **Sub_X** and **Sub_Y**.

Block Computation Pattern

Sub_X = X , Sub_Y = Y The Block computation pattern is shown in Figure 3.7. Here, all X*Y elements of the C tile computed by a thread are a contiguous block in the final C matrix.

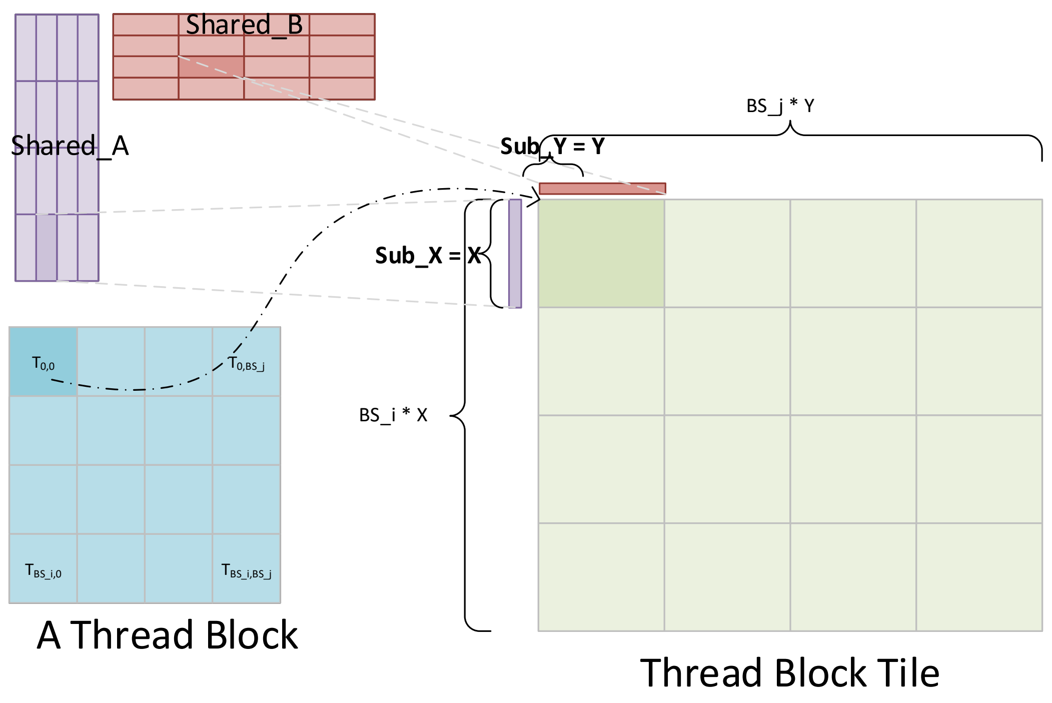

Figure 3.7: Detailed structure of a Thread Tile with Block Computation Pattern.

Block-Cyclic Computation Pattern

1 < Sub_X < X , 1 < Sub_Y < Y In this computation pattern, X*Y elements of C tile are computed as smaller **sub-patches** of size Sub_X * Sub_Y in the final C matrix as shown in Figure 3.8.

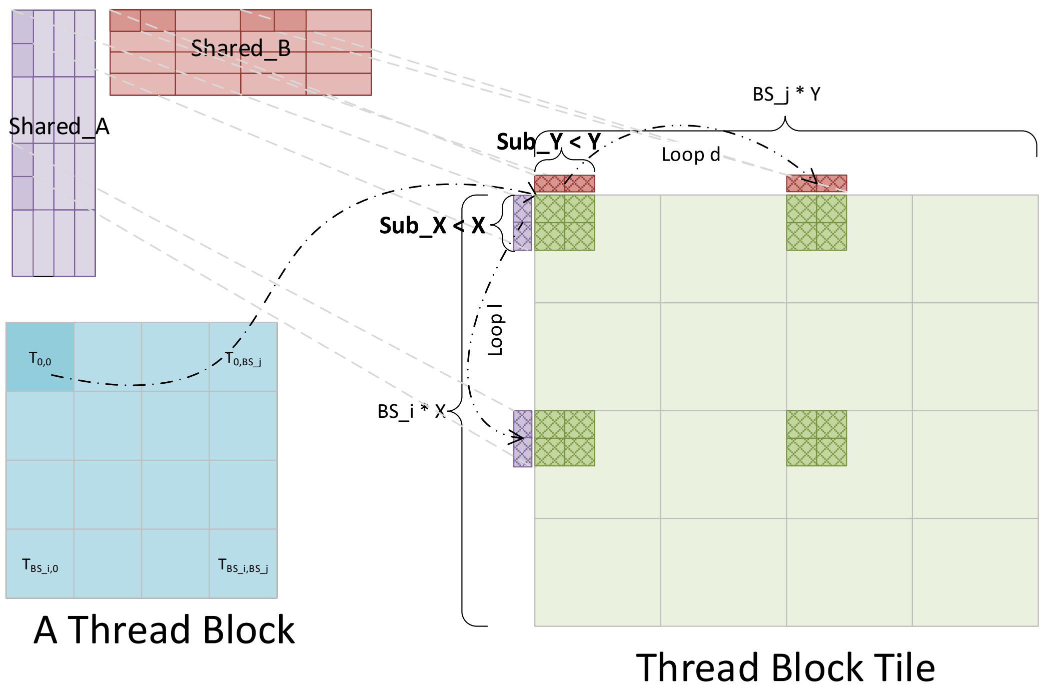

Figure 3.8: Detailed structure of a Thread Tile with Block Cyclic Computation Pattern.

Cyclic Computation Pattern

Sub_X = 1 , Sub_Y = 1 As Figure 3.9 shows by changing the size of the sub-patch to 1, X*Y elements of C tile are computed cyclically as one element at a stride of number of threads in a thread block.

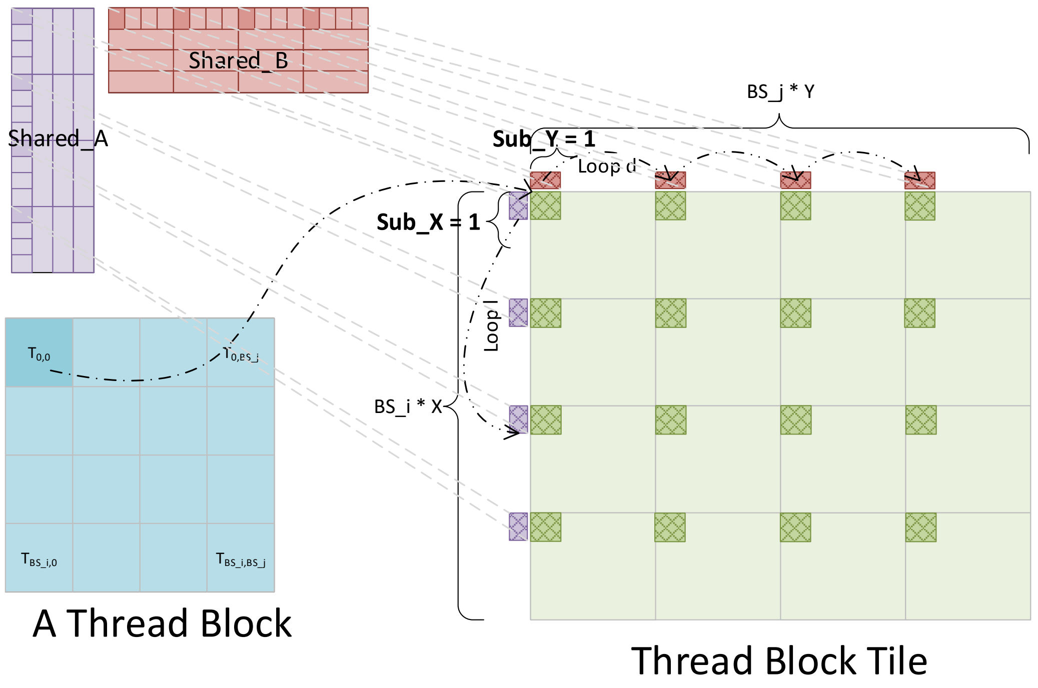

Figure 3.9: Detailed structure of a Thread Tile with Cyclic Computation Pattern.

3.3 Transposing Matrix A

To experiment with memory access pattern, we introduce another compile time parameter called Transpose. This parameter, if set, transposes the input matrix A as shown in Figure (transpose). This transformation in layout makes the access pattern of A exactly same as that of B.

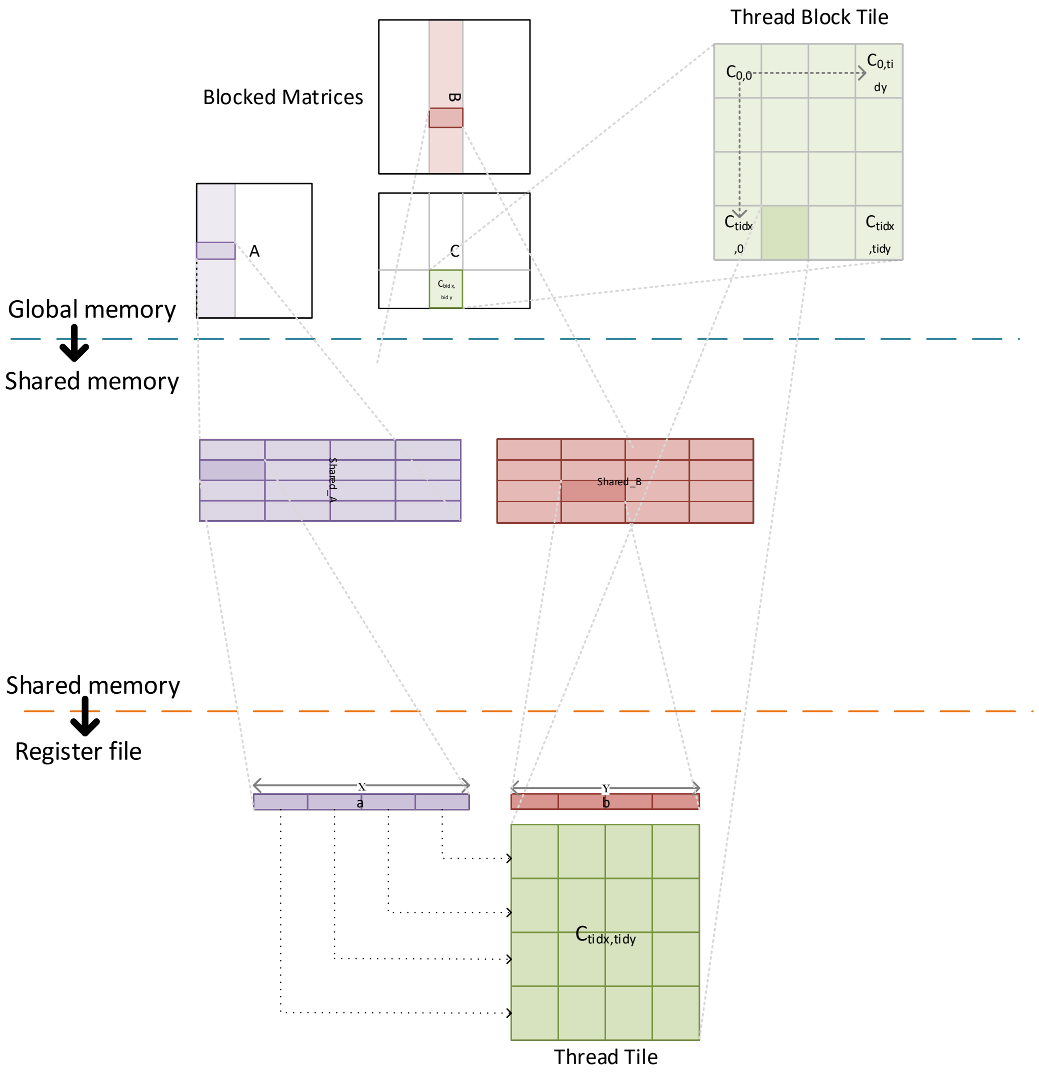

Figure 3.10: MPMML Memory Hierarchy with A Transpose.

3.4 Design for multiple instances of smaller input matrices

One of the immediately foreseen application of MPMML is BPMax. BPMax is RNA-RNA-Interaction (RRI) algorithm that operates in max-plus algebra. This algorithm performs the MPMM on multiple smaller input matrices. (e.g. of size 1024 x 1024). We include a feature in our MPMML to work with such set of input matrices. For this, we introduce a command line input parameter called **Ninst**. We use value of Ninst as the third dimension of the CUDA grid launched on the GPU (GS_k = Ninst). This way, MPMML computes *Ninst* instances of max-plus-matrix-products.

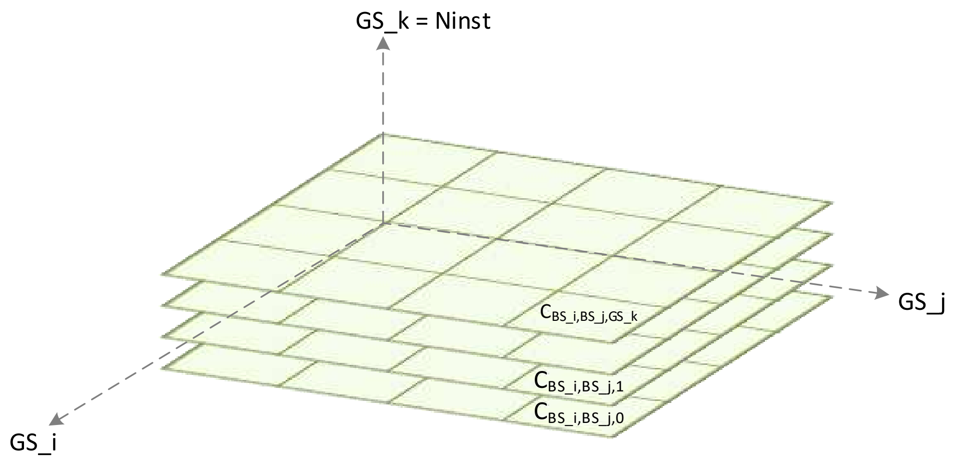

Figure 3.11: 3D grid of CUDA Blocks with smaller matrix sizes.

The final implementation of the MPMML GPU Kernel is shown below:

Algorithm 5 MPMML Kernel Computation

```
 1: for (int k=0; k<S; k++) do
 2:     for (int i=0; i<X/Sub_X; i++) do
 3:         for (int j=0; j<Y/Sub_Y; j++) do
 4:             for (int l=0; l<Sub_X; l++) do
 5:                 a[i] = shared_A[f(i,l)][k];
 6:                 for (int d=0; d<Sub_Y; d++) do
 7:                     b[j] = shared_B[k][f(j,d)];
 8:                     c[f(i, j, l, d)] = max(c[f(i, j, l, d)], (a[i]+b[j]));
 9:                 end for
10:             end for
11:         end for
12:     end for
13: end for
```

Chapter 4

Experimental results

We now describe the experimenation methods and performance results of our MPMM library

4.1 Experimental setup

We use the latest available NVCC version from the CUDA package 9.2.88_fc27 and GCC version 6.4.0 to compile the MPMML on all three GPU platforms (GeForce GTX 980, GeFOrce GTX Titan X, and GeForce GTX 1080Ti).

To verify the correctness of the library, we generate the correct output matrix on the host by computing the naive max-plus-matrix-multiplication and compare the output matrix copied back from the GPU device to the correct output matrix calculated on host. We initialize the input matrices to dummy data using the random number generator function with different seeds to the rand() function. This confirms the correctness of the library over a vast range of input values.

The performance of the MPMML is measured by calculating the total number of arithmetic operations (addition and max operation) and measuring the execution time of the kernel on GPU. This gives us the performance in terms of GOPs/S.

4.2 Memory access pattern case wise performance

We then experiment by changing the values of MEMSEL_A and MEMSEL_B to change the memory access patterns. We run these different case-wise versions on all the three GPU platforms with different input matrix sizes. Case 01, Case 10 and Case 11 perform very poorly on all the three GPU platforms. Case 00 is clearly the highest performing memory access pattern. Figure 4.1 shows the performance data for these different cases. For all the later experiments, we choose only the Case 00.

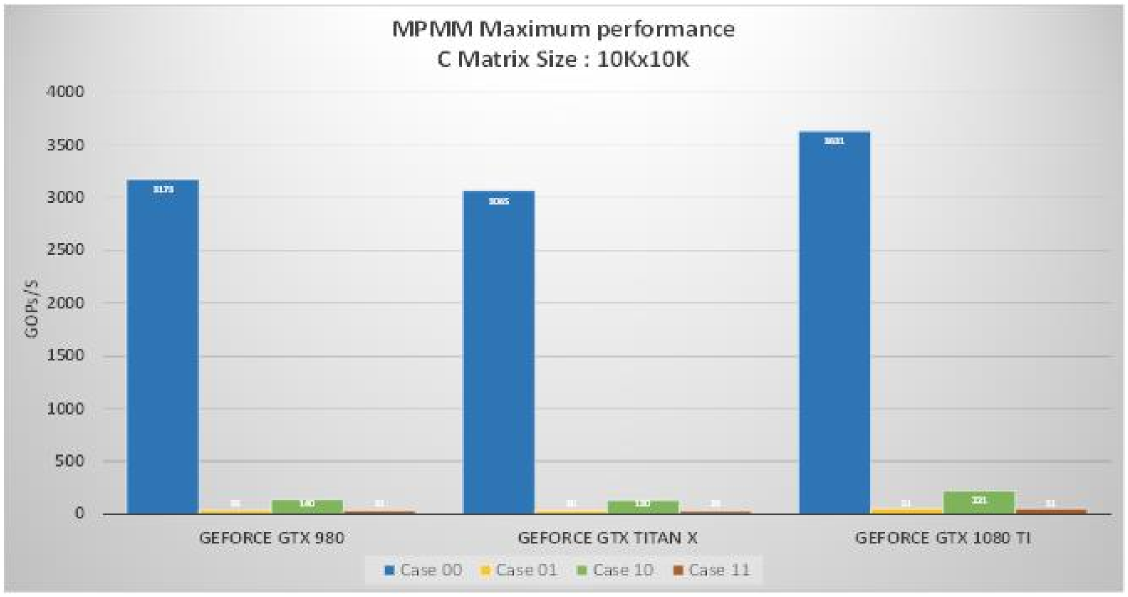

Figure 4.1: Case wise performance of MPMML

Computation pattern wise performance

We then experiment with the different computation patterns as described in 3.2.1 for different input matrix sizes. As Figures 4.2 , 4.3 and 4.3 show that:

1. The *Block*, *Block-Cyclic* and *Cyclic* patterns achieve performance very close to each other for square input matrices with sizes larger than 1Kx1K.

2. For input matrix size 1Kx1K:

    (a) On GeForce GTX 980, *Block* and *Block-Cyclic Cyclic* patterns achieve performance very close to each other

    (b) On GeForce GTX Titan X, *Block* achieves the lowest performance while *Block-Cyclic* and *Cyclic* patterns achieve performance close to each other

    (c) On GeForce GTX 1080 Ti, *Block* achieves the highest performance while *Block-Cyclic* and *Cyclic* patterns achieve performance close to each other

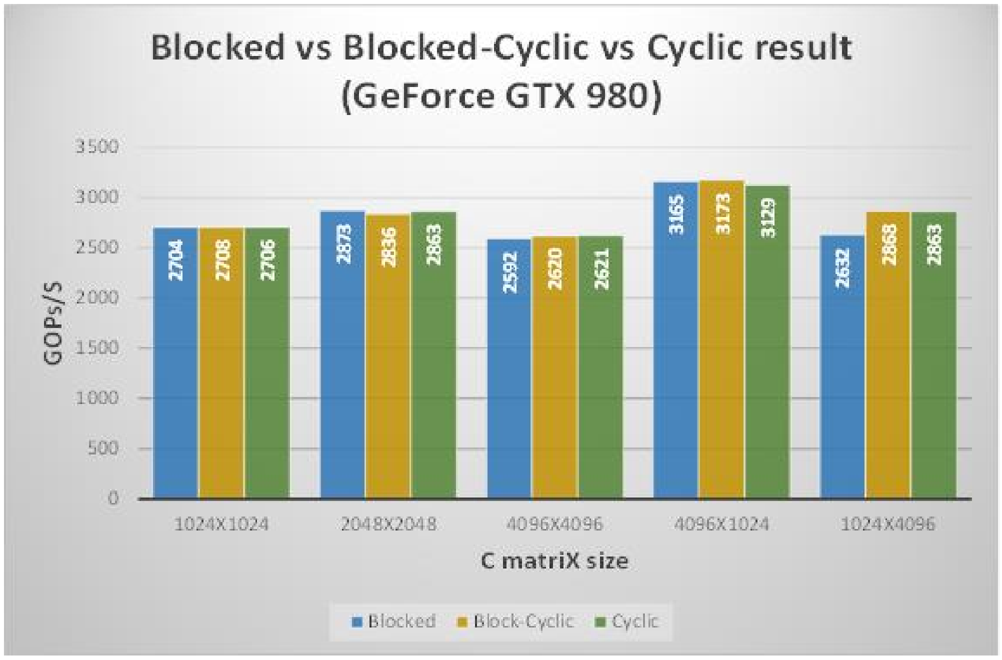

Figure 4.2: Performance of different computation patterns : GeForce GTX 980

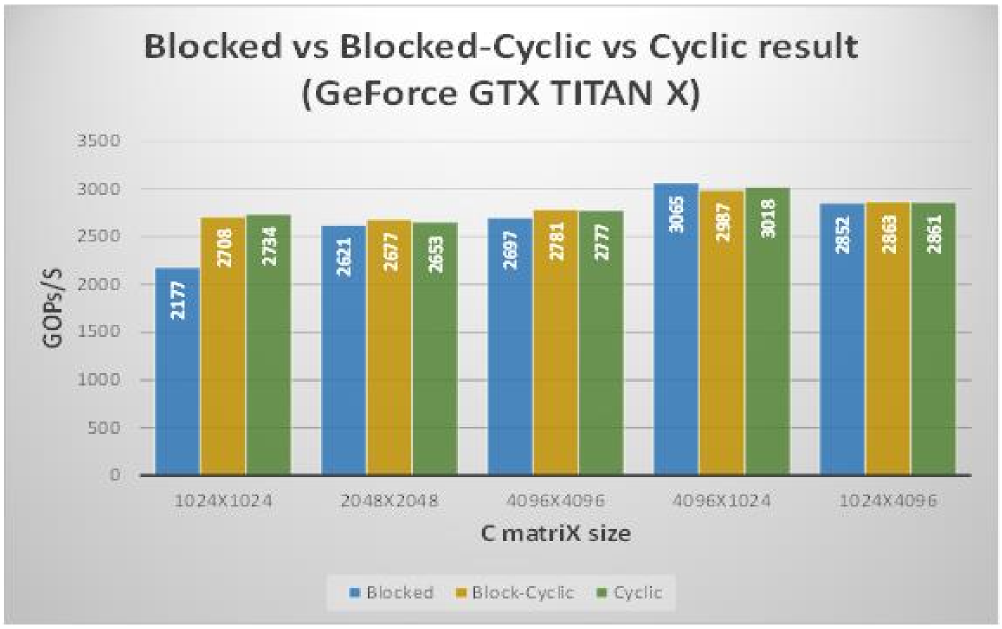

Figure 4.3: Performance of different computation patterns : GeForce GTX Titan X

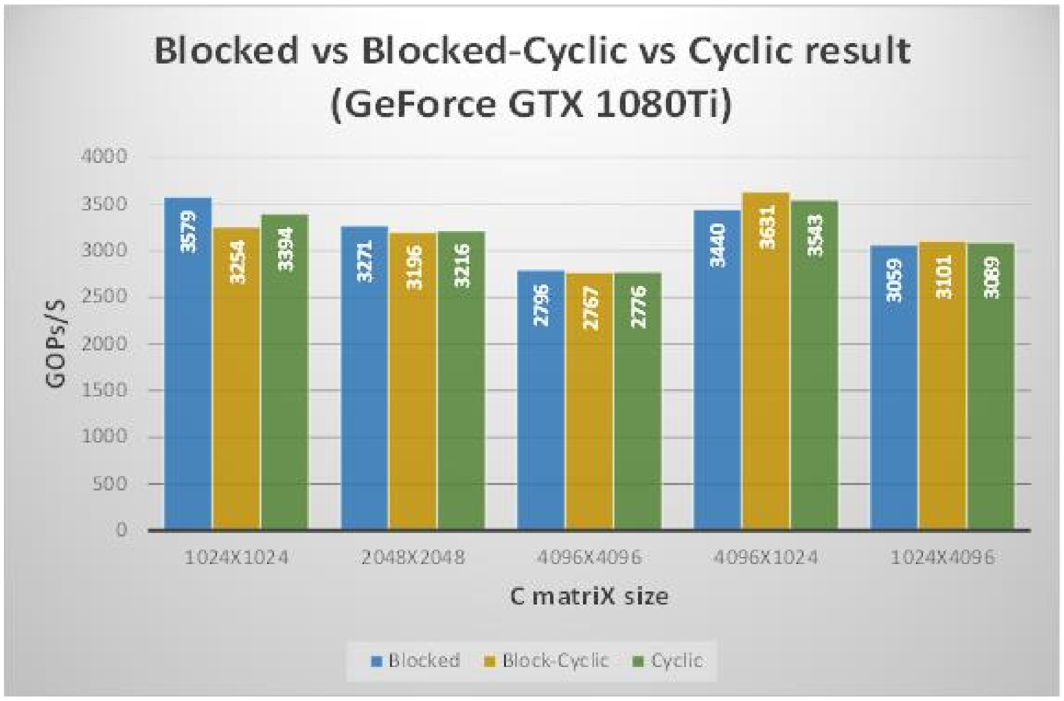

Figure 4.4: Performance of different computation patterns : GeForce GTX 1080 Ti

4.3 Performance of MPMML with A transpose

We also experimented with transposing matrix A as described in Section 3.3. We experiment with different input matrix sizes but for the simplification of analysis, we display only the results of input matrix size 10Kx10K. As Figure 4.5 shows, the performance of MPMML decreases significantly when A matrix is transposed. We hypothesized transposing matrix A will provide better coalescing for access pattern to matrix A improving the performance. But, the performance results indicate otherwise.

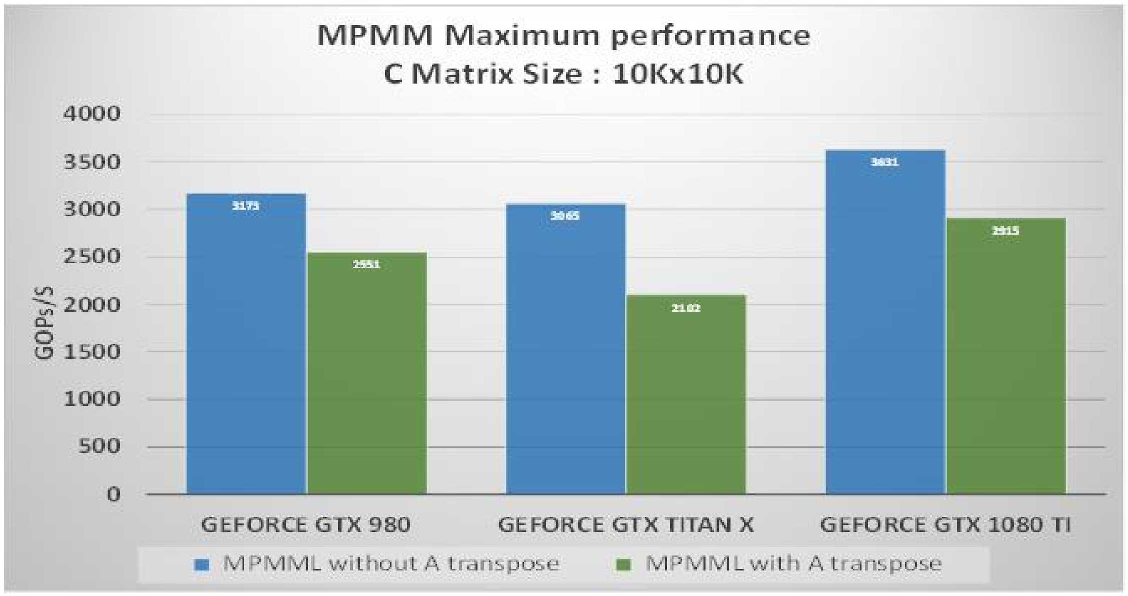

Figure 4.5: Performance comparison : With and Without A transpose

4.4 Performance of MPMML with multiple instances of smaller size matrices

As described in 3.4, we use the command line input *Ninst* to launch the GPU kernel on a 3D grid of CUDA Thread Blocks. We experiment with multiple instances on 1Kx1K matrix for values of Ninst ranging from 1 to 20. We choose the version of MPMML with *Block* computation pattern because of very little variance amongst the different computation patterns for the larger input matrix sizes. Figure 4.6 shows that MPMML for 20 instances of input matrix of size 1Kx1K achieves 2175GOPs/S - 2798 GOPs/S on Maxwell architecture and 3090 GOPs/S on Pascal architecture. All of these performance numbers are close to that of single instance of large matrix.

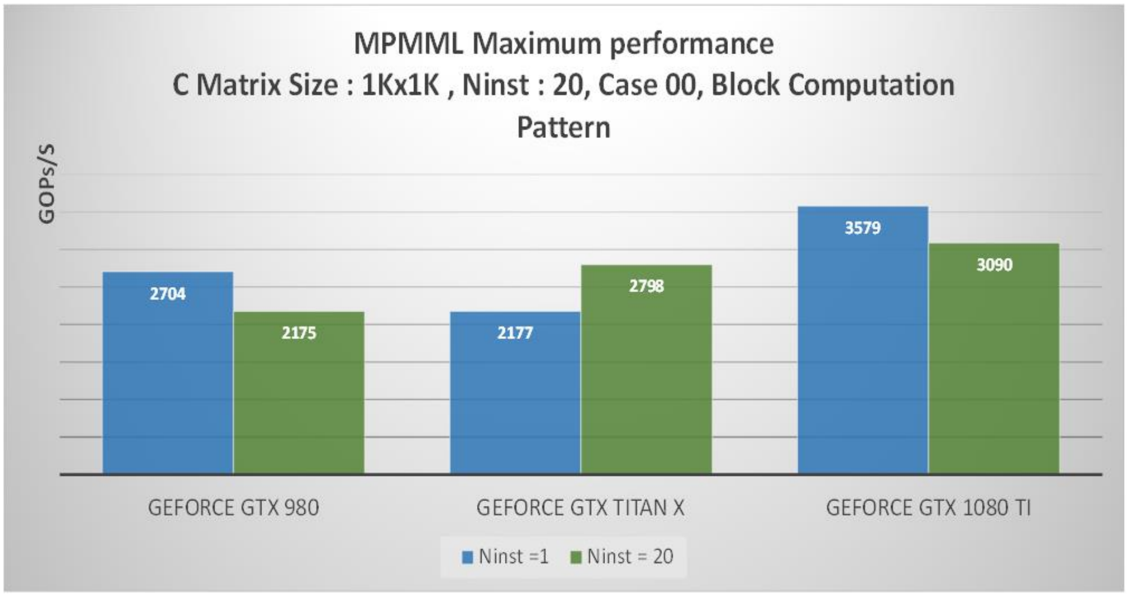

Figure 4.6: Performance comparison : With and Without A transpose

Chapter 5

Conclusion

As discussed in earlier chapter, Tropical algebra (also known as Max-Plus algebra) is increasingly proving pivotal in fields other than just discrete mathematics like, dynamic programming, biological sequence comparisons, network modeling etc. The highly optimized Basic Linear Algebra libraries like CuBLAS [1] implement the standard matrix operations in linear algebra for floating point data type. We came across the need for a tuned library that implements Max-Plus Matrix Multiplication (MPMM) however, we did not find a standard library for MPMM on integers. Consequently we set off on an effort to develop a parametrically tuned parallelized Max Plus Matrix Multiplication library kernel.

To start with, we chose GPUs as the hardware platform for this work because of their significantly more parallelism as compared to CPUs. We then implemented detailed microbenchmarks of the GPU memory system and arithmetic throughput. These micobenchmarks are important to understanding the limits of performance of our MPMML algorithm. This approach revealed the behavior of underlying hardware when so-called hardware bottlenecks are saturated. It also pointed us to guidelines for our parametric implementation of MPMML.

Finally, we present the parametric design of MPMML GPU kernel to be portable across three successive Nvidia GPU architectures. This MPMML attains performance in the range 3065 GOPs/S - 3631 GOPs/S on all of these architectures for large matrices and 2175 GOPs/S - 3090 GOPs/S on the same GPU machines for multiple instances of smaller matrices.

Future scope of this work includes tuning the parameters of MPMML to port it to newer generations of GPU architectures. Also, as we have identified earlier, MPMML can be directly applied to speed up one of the latest RRI tool BPMax [**?**]. This may however, need feature inclusions in this MPMML to perform MPMM on triangular matrices.

Bibliography

[1] NVidia. cublas library. https://developer.nvidia.com/cublas.

[2] Vasily Volkov and James W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press.

[3] Jean-Eric Pin, John M. Taylor, and Michael Atiyah. *Tropical semirings*, page 50?69. Publications of the Newton Institute. Cambridge University Press, 1998.

[4] Imre Simon. Recognizable sets with multiplicities in the tropical semiring. In Michal P. Chytil, Václav Koubek, and Ladislav Janiga, editors, *Mathematical Foundations of Computer Science 1988*, pages 107–120, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[5] Brett A. Becker and Alexey Lastovetsky. Max-plus algebra and discrete event simulation on parallel hierarchical heterogeneous platforms. In Mario R. Guarracino, Frédéric Vivien, Jesper Larsson Träff, Mario Cannatoro, Marco Danelutto, Anders Hast, Francesca Perla, Andreas Knüpfer, Beniamino Di Martino, and Michael Alexander, editors, *Euro-Par 2010 Parallel Processing Workshops*, pages 63–70, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[6] Guy Cohen, Stéphane Gaubert, and Jean-Pierre Quadrat. Max-plus algebra and system theory: Where we are and where to go now. *Annual Reviews in Control*, 23:207 – 219, 1999.

[7] L. Pachter and B. Sturmfels. *Algebraic Statistics for Computational Biology*. Cambridge University Press, New York, NY, USA, 2005.

[8] Bernd Heidergott, Geert Jan Olsder, and Jacob van der Woude. *Max Plus at Work: Modeling and Analysis of Synchronized Systems: A Course on Max-Plus Algebra and Its Applications*. Princeton University Press, 2006.

[9] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113.

[10] Mark Harris. A brief history of gpgpu. http://cs.unc.edu/xcms/wpfiles/50th-symp/Harris.pdf, 2003.

[11] Geforce gtx 980 gm204 whitepaper.

[12] NVidia. Cuda programming guide v10.0.130. https://docs.nvidia.com/cuda/cuda-c-programming-guide.

[13] NVidia. Cuda occupancy calculator. https://developer.download.nvidia.com/.../cuda/CUDA_Occupancy_calculator.xls.

Appendix A

Performance Results of Microbenchmarks

The Following pages include performance results plots of the microbenchmarks described in 2

(Note :-This page is intentionally left empty)

Performance Results plots of Reg-Reg Access Microbenchmark

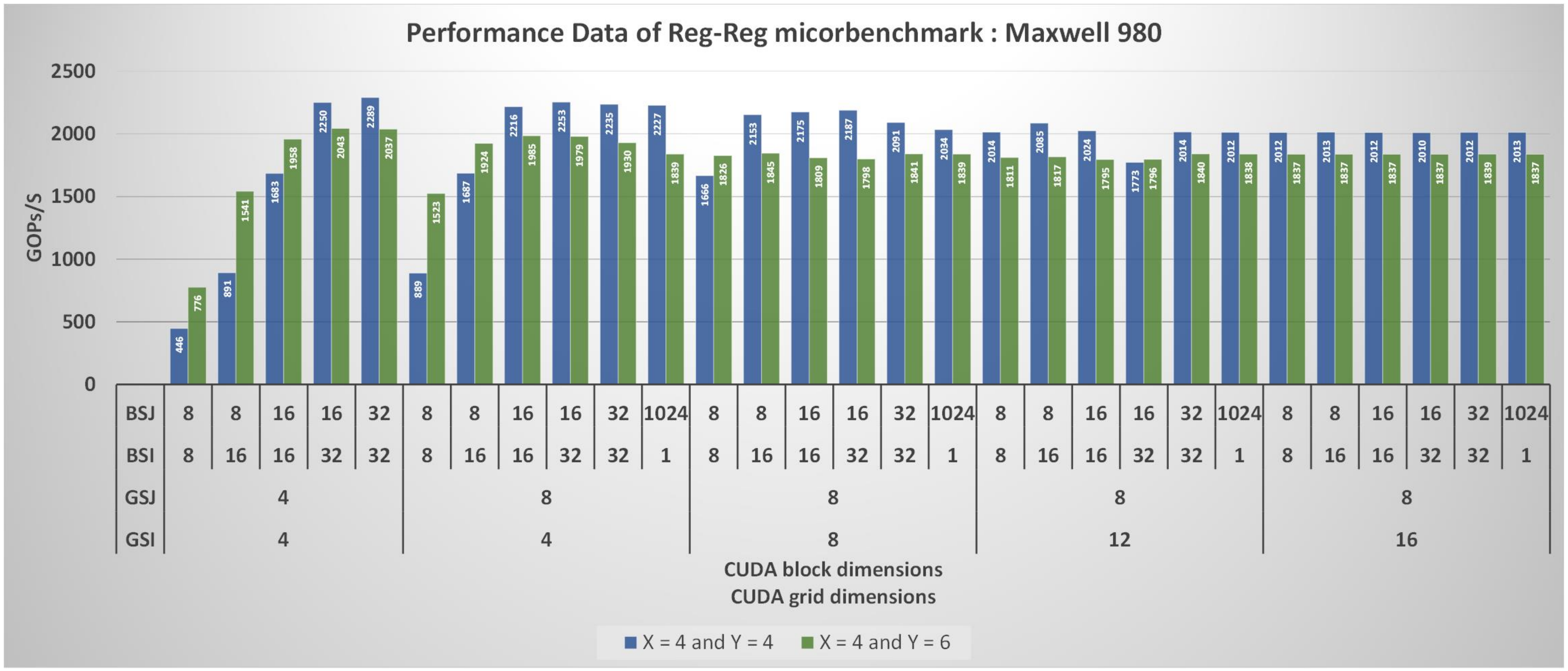

Figure A.1: Performance: Reg-Reg microbenchmark GeForce GTX 980.

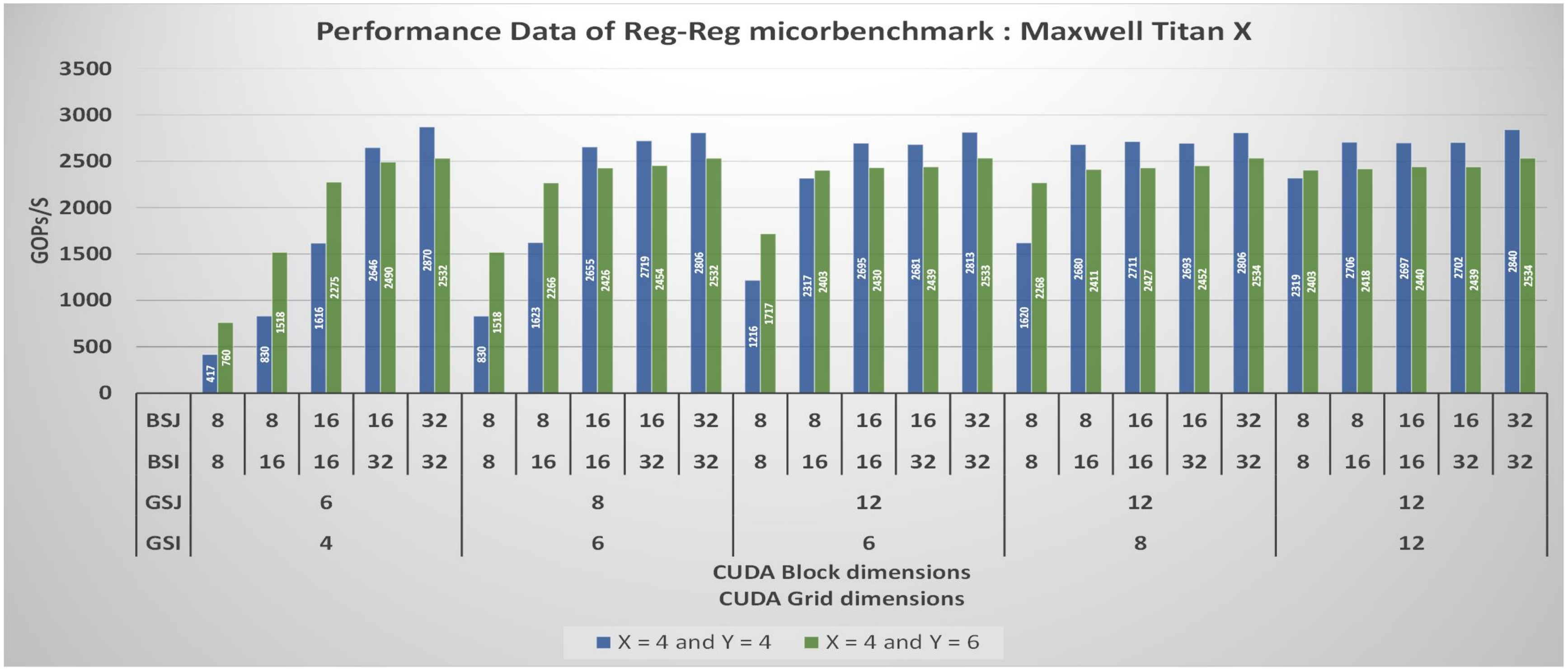

Figure A.2: Performance: Reg-Reg microbenchmark GeForce GTX TitanX.

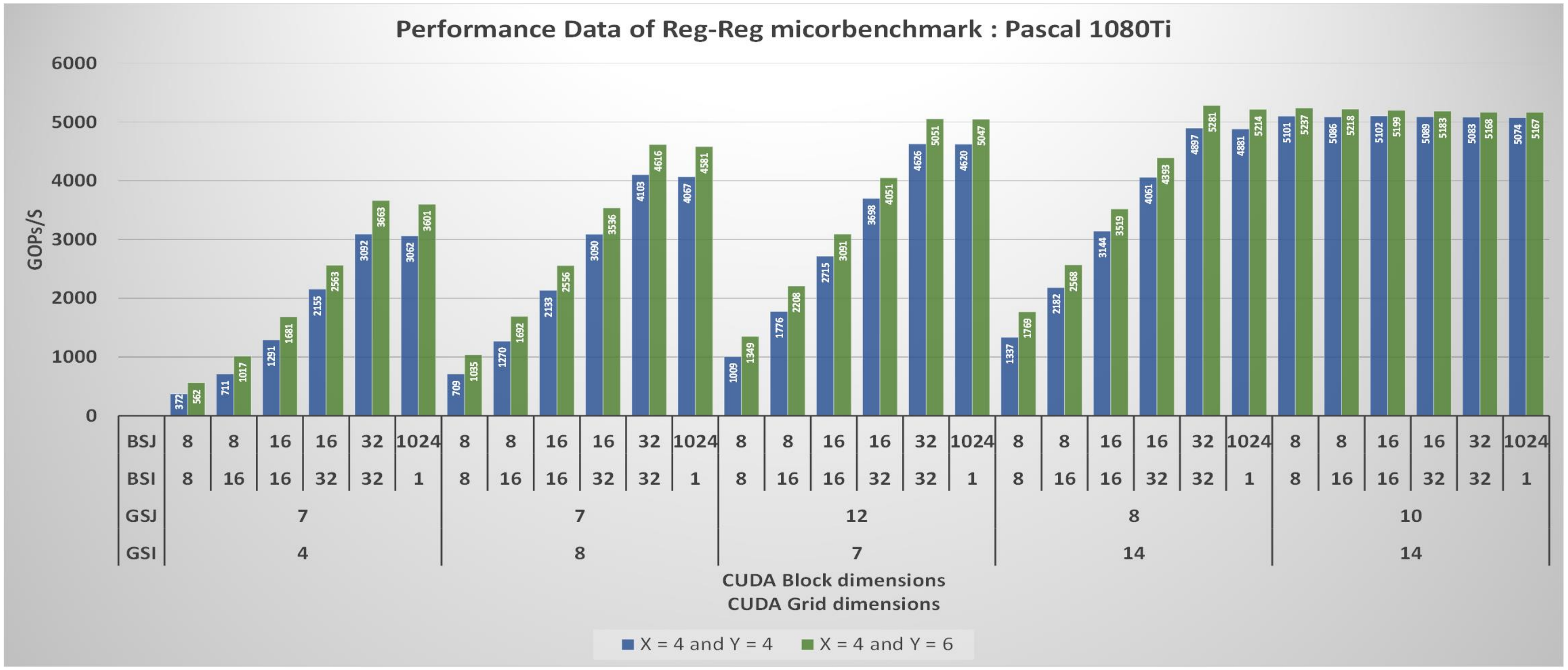

Figure A.3: Performance: Reg-Reg microbenchmark GeForce GTX 1080 Ti.

Performance Results plots of Shared Memory Access Microbenchmark

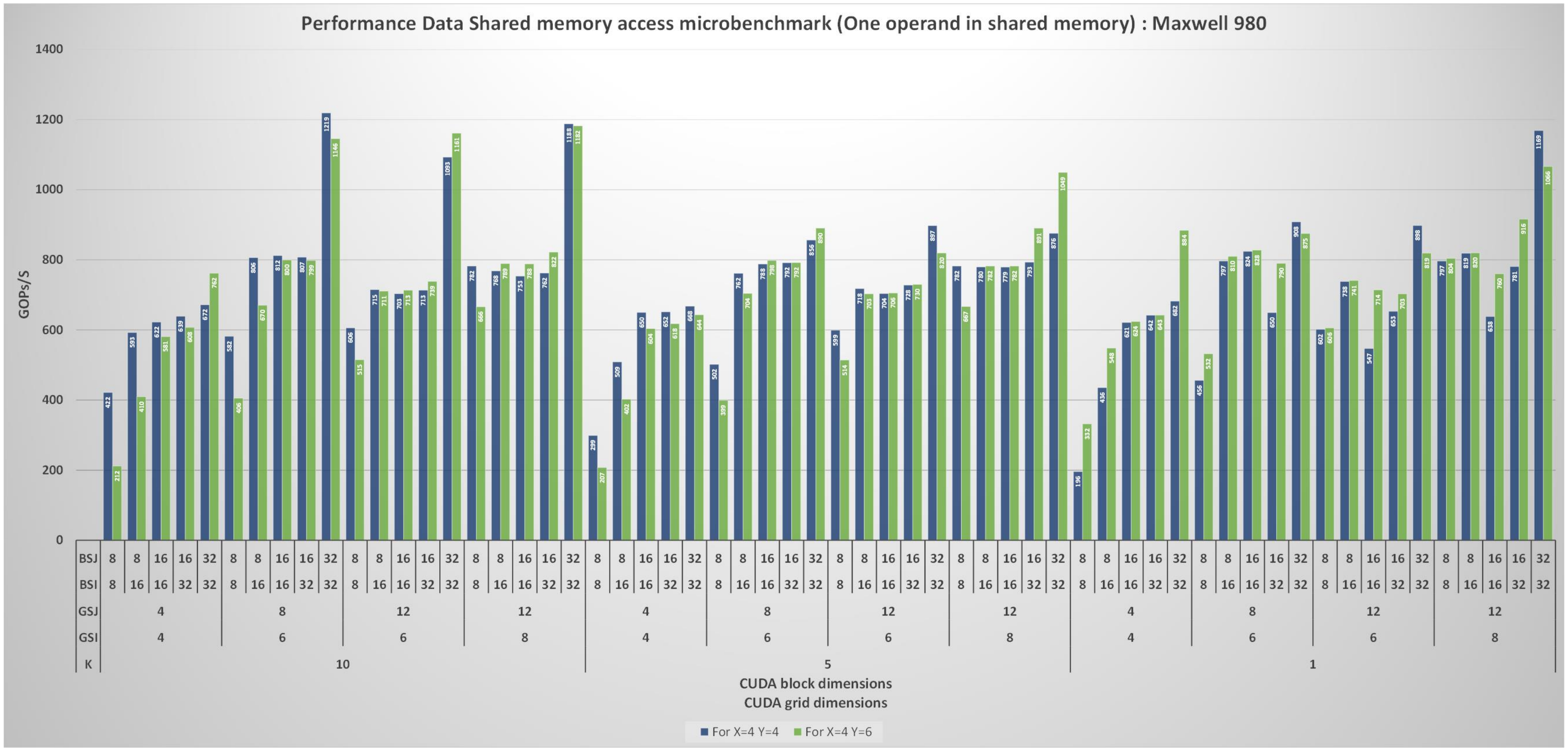

Figure A.4: Performance: on Maxwell GeForce GTX 980 machine : One operand from Shared Memory.

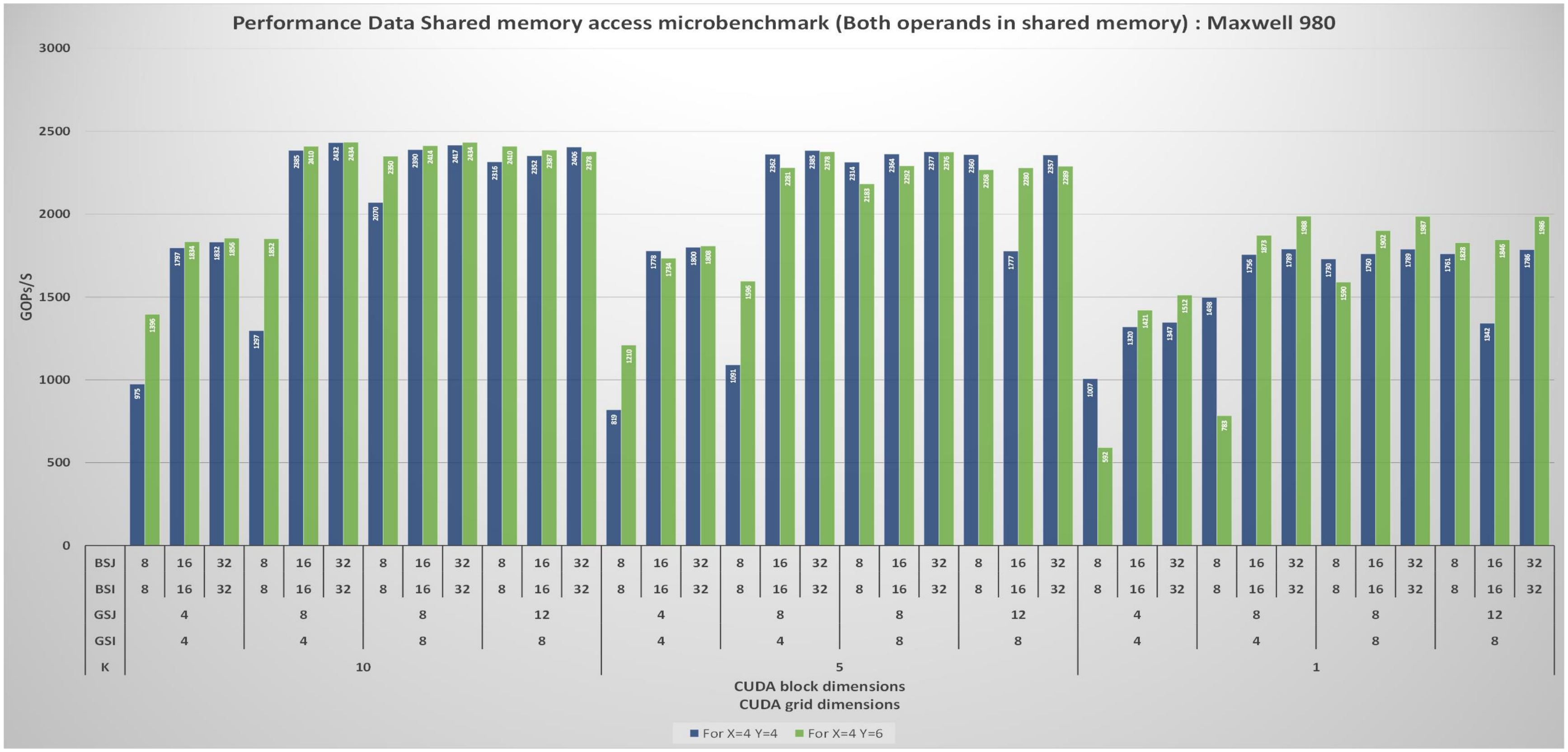

Figure A.5: Performance: on Maxwell GeForce GTX 980 machine : Both operand from Shared Memory.

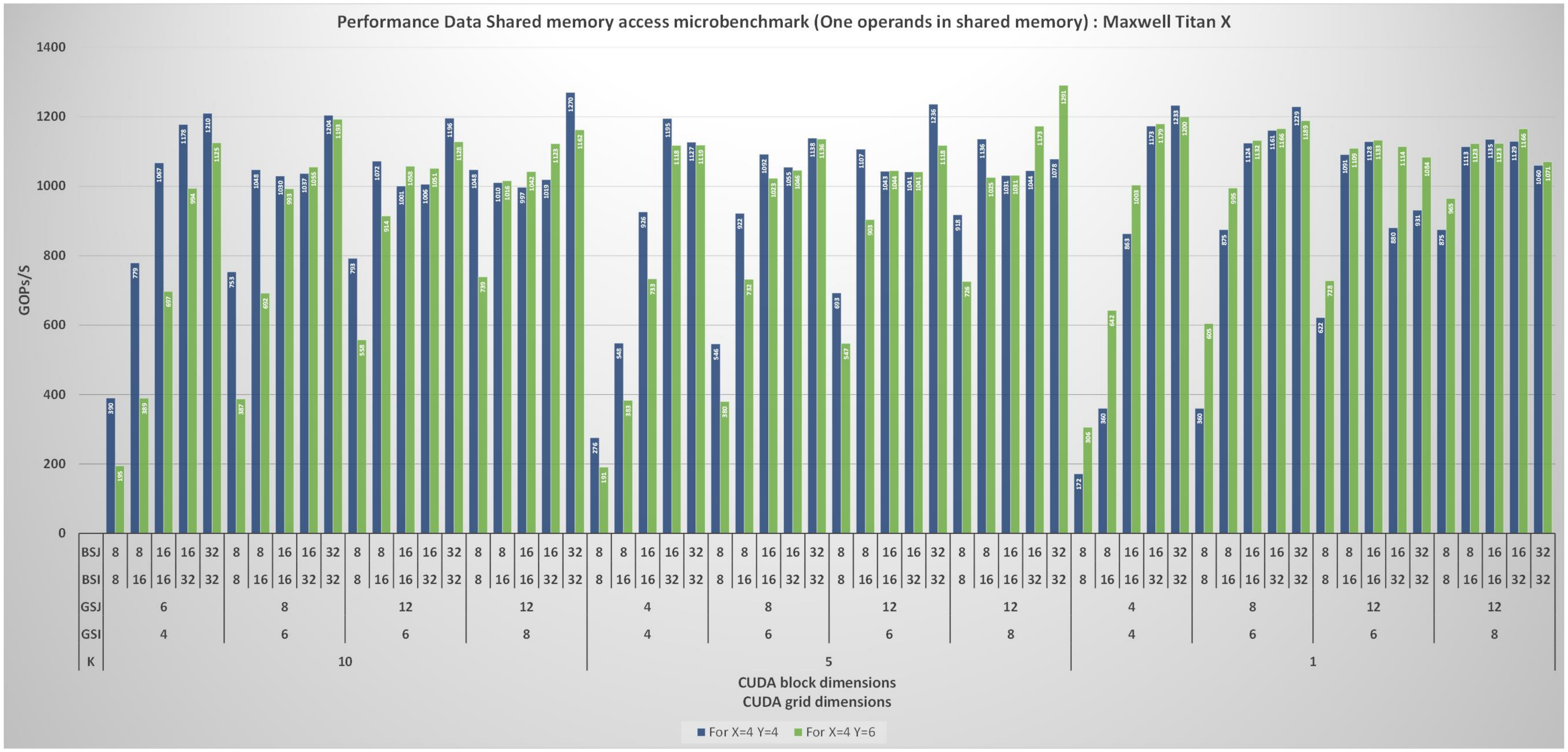

Figure A.6: Performance: on GeForce GTX TitanX machine : One operand from Shared Memory.

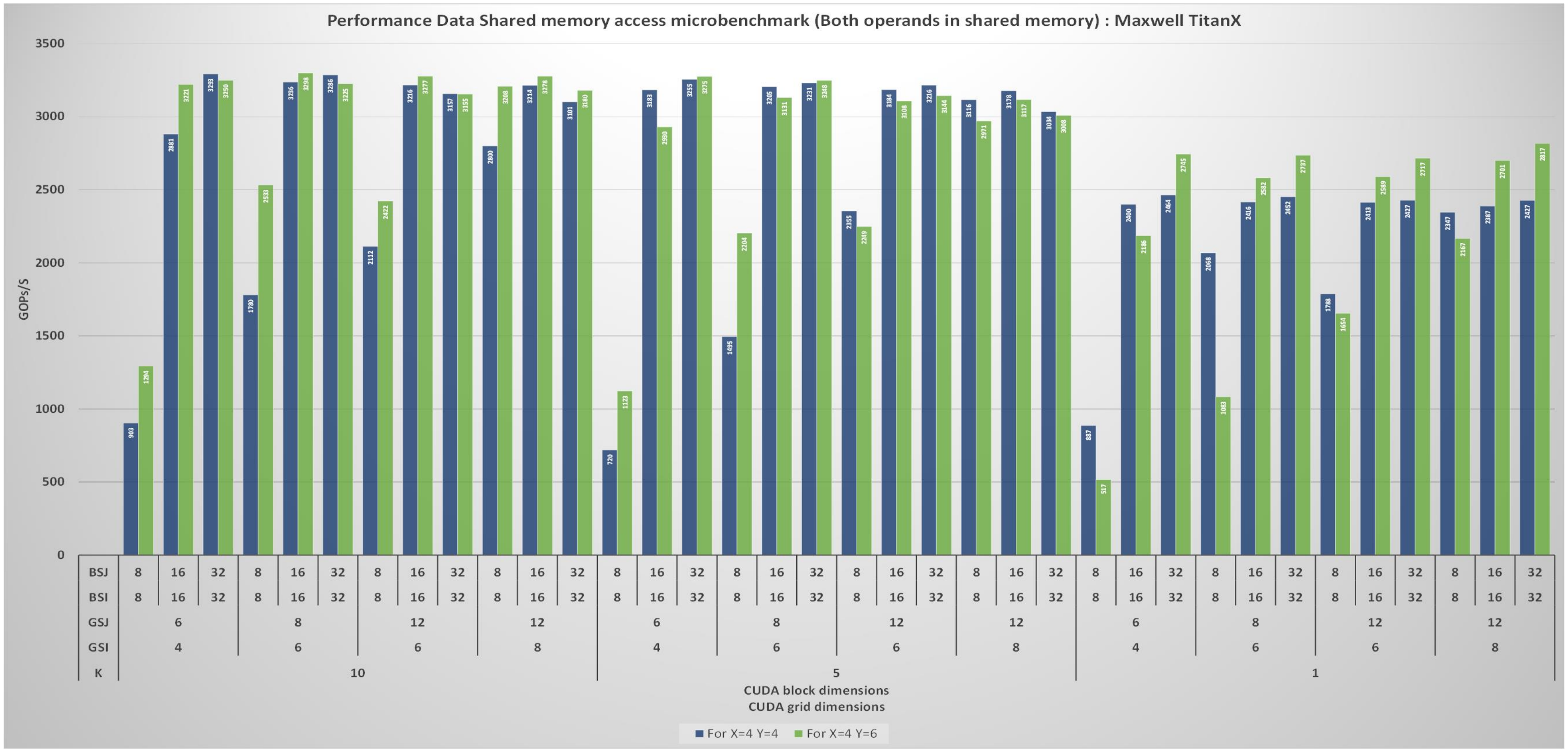

Figure A.7: Performance: on GeForce GTX TitanX machine : Both operand from Shared Memory.

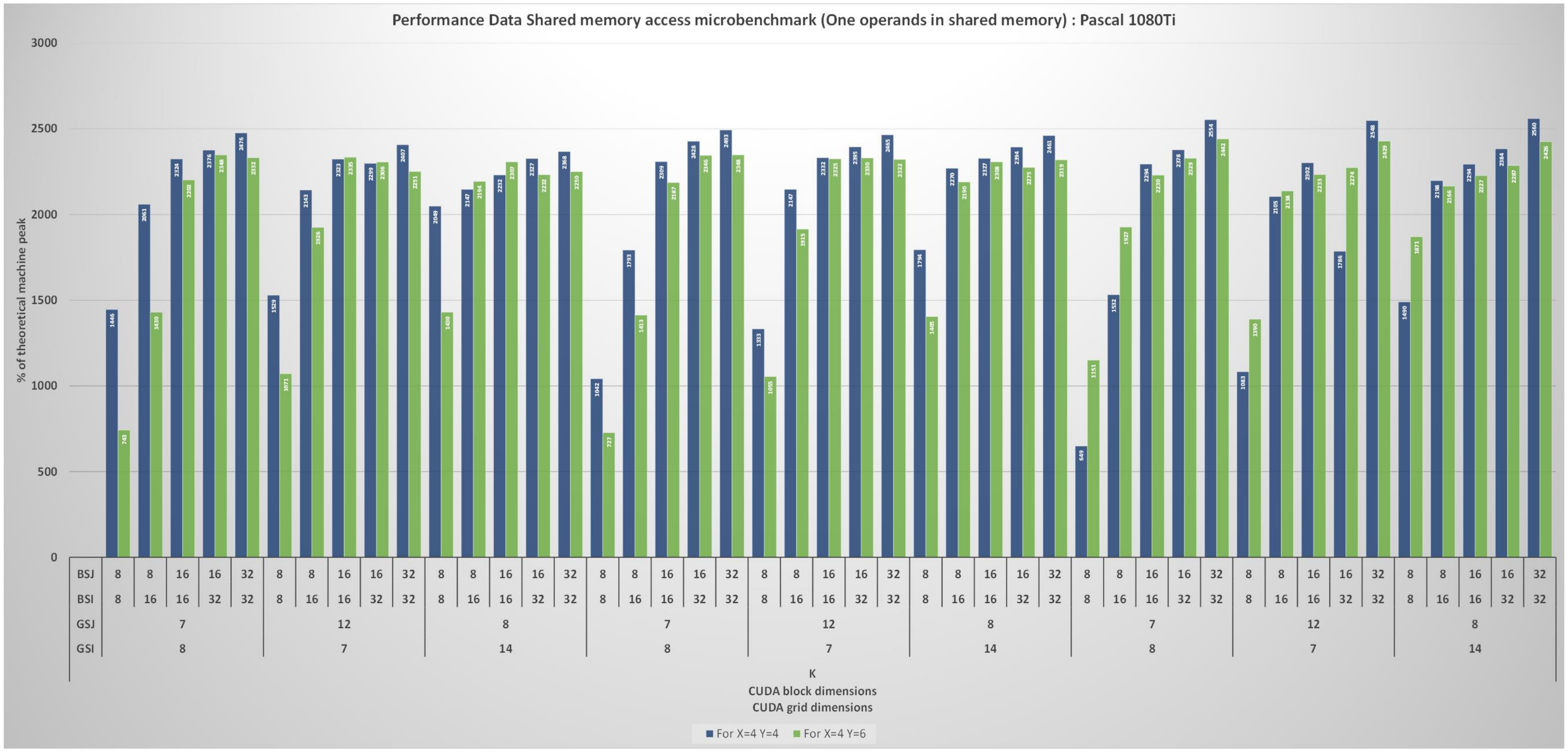

Figure A.8: Performance: GeForce GTX 1080 Ti machine : One operand from Shared Memory.

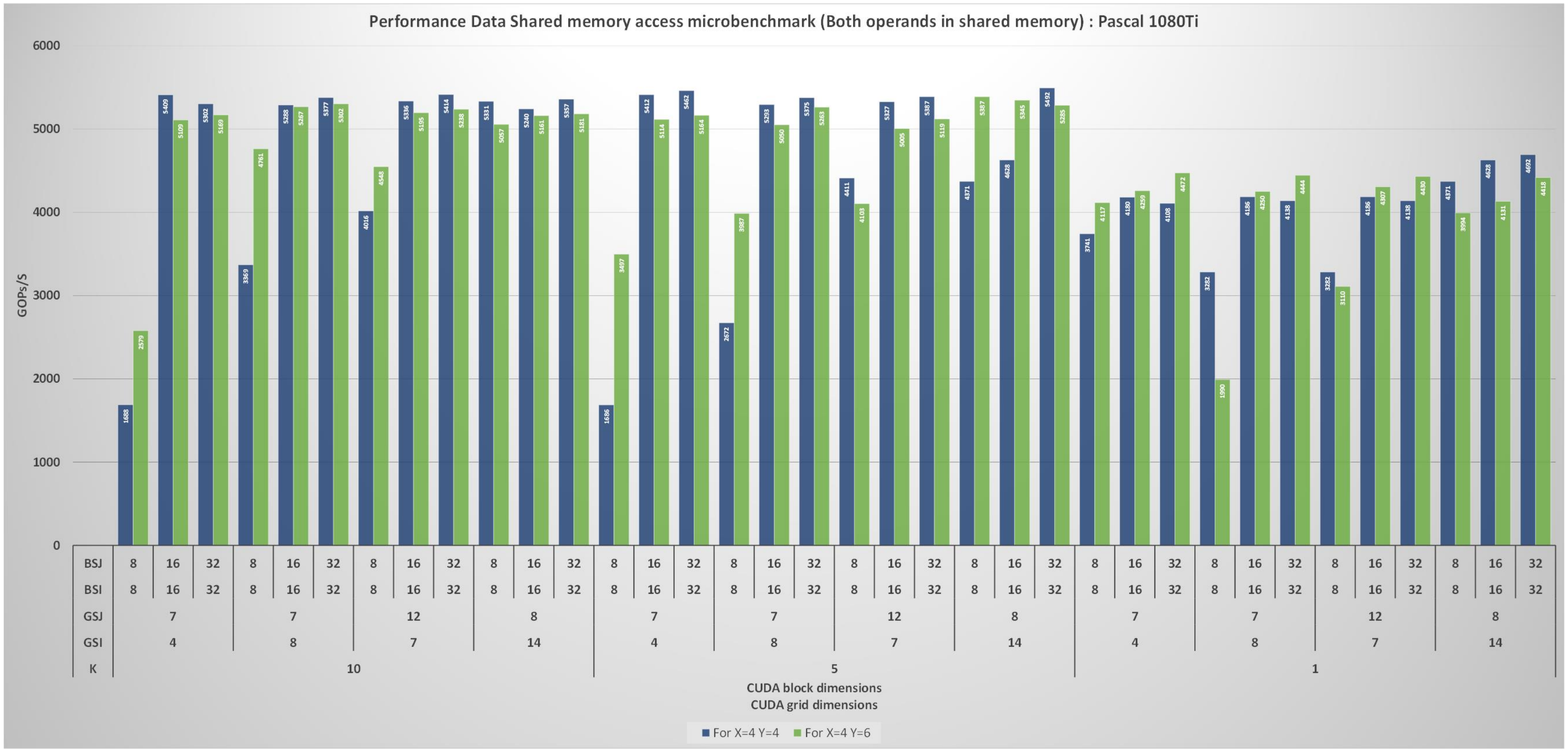

Figure A.9: Performance: GeForce GTX 1080 Ti machine : Both operand from Shared Memory.