SGEMM Characterization on NVIDIA GPUs using Profiling Metrics for Autonomous Driving

***Abstract*—In this paper we discuss about better designing methods of SGEMM (single-precision general matrix multiplication) in terms of memory and thread-block hierarchy on NVIDIA GPUs. This paper is focused on extracting important metrics using command-line, light-weight, GUI-less profiler called Nvprof which were used to explain why different designs of SGEMM algorithm results in different GPU performance. NVIDIA Tesla T4 based on latest Turing architecture with Compute Unified Data Architecture (CUDA) version 11.1 and NVIDIA GeForce GTX 1050 Max-Q design with CUDA version 10.1 were used to test and compare different designing methods.**

***Index Terms*—CUDA; Shared Memory; Threads; SIMD; NVIDIA; Nvprof; cuBLAS;**

I. Introduction

The machine learning algorithms have robust generalisation performances and have been widely used in image recognition, bioinformatics and text processing. However, training those algorithms and using them for inference is still very computational intensive task.

The long training time has seriously hindered the development of machine learning and cannot deploy in a real-time scenario.

Using machine learning algorithms for Autonomous Driving is one of these tasks which needs to work in real-time scenario. Even though, Autonomous Driving is crucial to decrease number of crashes on the roads and enable the elderly, the disabled and physically limited people move much easier, it is currently risky to deploy them on the roads because of abovementioned constraints.

As Autonomous Driving includes images from real-life, it will use CNN (Convolutional Neural Network) - based object detection where large fraction of the arithmetic operations consists of matrix multiplications, in both convolution and fully connected layers.

Matrix multiplication is one of the most common and most important operations in linear algebra and has a wide range of applications in the field of machine learning. Therefore, improving the computing performance of matrix multiplication has great significance for machine learning.

The CUDA programming model targets application development for such platforms. Its execution model consists of a hierarchy of parallelization layers: grids, thread blocks, warps and threads. Each processor runs a number of threads. A thread block is a batch of threads running on one multiprocessor and therefore all threads in a thread block share some part of the shared memory. A grid consists of several thread blocks. Because there can be more blocks than multiprocessors, different thread blocks of a grid are scheduled for the multiprocessors dynamically. Threads within a block are grouped in warps, each running as an SIMD (Single Instruction Multiple Data) unit. The threads of a thread block are scheduled to occupy the multiprocessor exclusively each time.

CUDA provides a low entry level for learning many-core programming, but obtaining peak performance can be a very challenging task. A programmer must understand all the constraints of the hardware platforms and devise an algorithm that reaches a right balance for all the performance-affecting factors.

Along with hardware constraints, the programmer should be able to use analysis metrics properly in order to detect bottlenecks in the devised algorithm to further improve it. For this purpose, this paper will focus on retrieving important profiling metrics using profiling tools such as NVIDIA Nsight Compute and NVIDIA Visual Profiler.

Our paper makes the following specific contributions: In terms of memory and thread hierarchy with the help of profiling metrics,

- Characterize the effects of thread hierarchy optimizations (Threads per block vs computed values per thread)
- Characterize the effects of memory hierarchy optimization (Global vs Shared memory)
- Explain the picture behind above-mentioned optimizations using profiling metrics (stall_memory_dependency, flop_sp_efficiency achieved_occupancy, etc.) and identify bottlenecks

II. Motivation

Currently, there are many research papers that focused on SGEMM (single-precision general matrix multiplication) in NVIDIA GPUs. Each of these papers contributed a lot in understanding how GPUs perform SGEMM and how to design better SGEMM algorithms using opportunities given by each newly released architectures of NVIDIA GPUs.

One of the possible ways to decrease inference time of matrix-multiplication on GPUs is to use shared memory. First, matrices should be copied from host memory to device memory (global memory). Data that is stored in global memory is available by all the blocks. However, reading and writing speed of global memory is very small which is important for

SGEMM as each cell in the matrices will be accessed more than once. That is why, although access to the data will be limited within the block, shared memory was used to achieve faster reading and writing speed [1]. This speedup can be seen from Figure 1, where inference time was almost halved when shared memory was applied. That is the example of using local memory over off-chip memory.

Another way to improve SGEMM is to compute more than one value of resultant matrix per each thread thus increasing number of floating point operations in each clock cycle. This can be done by using faster memory units, registers, within each thread which will be used to decrease number of repeated accesses to shared memory [2].

However, almost all of the previous papers did not use any profiling tools and metrics to explain why the proposed SGEMM designs are better than predecessors. That is what this paper will be focused on.

The following GPU profiling metrics were chosen to evaluate different SGEMM algorithms [3]:

- ***Regs*** - Number of registers used per CUDA thread. This number includes registers used internally by the CUDA driver and/or tools and can be more than what the compiler shows.
- ***global_load*** - Number of executed load instructions where state space is specified as global, increments per warp on a multiprocessor.
- ***shared_load*** - Number of executed load instructions where state space is specified as shared, increments per warp on a multiprocessor
- ***shared_efficiency*** - Ratio of requested shared memory throughput to required shared memory throughput expressed as percentage
- ***achieved_occupancy*** - Ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor
- ***stall_memory_dependency*** - Percentage of stalls occurring because a memory operation cannot be performed due to the required resources not being available or fully utilized, or because too many requests of a given type are outstanding

Importance of profiling metrics:

- Using higher memory hierarchy reduces inference time (global -> shared -> registers) (see Figure 1) because of reduced access to global memory which can be described by global_load metric (See Figure 3)
- Increasing block size with shared memory implementation also gives speedup in time as it increases size of shared memory which in turn decreases number of accesses to global memory.
- Computing multiple values per thread also decreases inference time (see Figure 2) as this implementation takes advantage of registers within each thread and therefore decreases number of accesses to shared memory. This reduction in accesses to shared memory can be illustrated with inst_executed_shared_loads metric (See Figure 3) which uniquely can describe this implementation.
- The last chosen stall_memory_dependency metric help us to see relationship between these 2 optimizations of SGEMM algorithm (see Figure 3). As it was discussed earlier it is important to balance all the performance-affecting factors and this is a suitable profiling metric. The meaning of this metric and relationship between increasing block size and computing more values per thread will be thoroughly explained in the following sections.

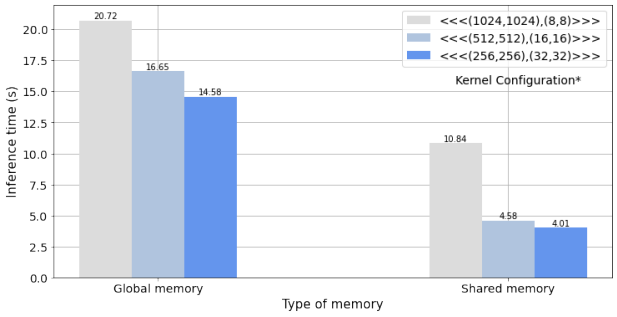

Fig. 1. Inference Time of SGEMM (matrix size 8192x8192) algorithm with different memory and thread/block optimizations
* - <<<Number of blocks per Grid on (x,y) axis, Number of threads per Block on (x,y) axis>>>

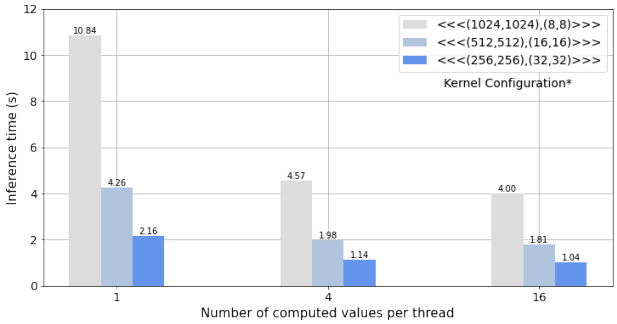

Fig. 2. Inference Time of SGEMM (matrix size 8192x8192) algorithm for different number of computed values per thread
* - <<<Number of blocks per Grid on (x,y) axis, Number of threads per Block on (x,y) axis>>>

However, we need to do experiments on more thread/block configurations to better investigate proposed assumptions.

III. Methodology

Matrix multiplication task is highly dependent on memory throughput of GPUs and memory optimization of kernel. This is due to low ratio of arithmetic operation per memory access. To illustrate, in order to multiply two matrices of size N by N, N^3 FMA operations needed and 2 read operations per each FMA. Since memory operations are much slower than arithmetic ones, and we need 2 memory read per 1 arithmetic execution, this task looks hard to exploit GPU's

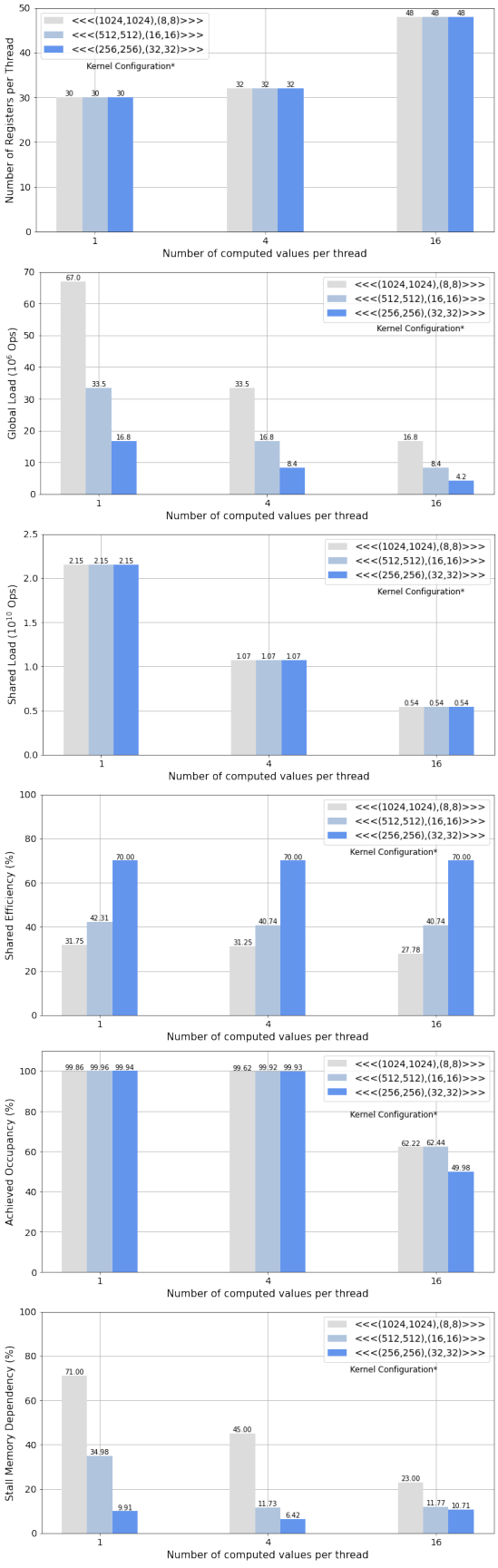

Fig. 3. Profiling metrics on SGEMM (matrix size 8192x8192) algorithm for different number of computed values per thread
* - <<<Number of blocks per Grid on (x,y) axis, Number of threads per Block on (x,y) axis>>>

hardware capabilities. However, each value of input matrices are gonna be used in arithmetic operations N times, and this is an opportunity to optimize kernel using memory hierarchy of GPUs.

A. Memory part

1) Exploiting 3 level memory hierarchy:
Opportunity to reuse data allows to exploit three level memory hierarchy of GPUs. Tiled approach to solve GEMM task is an example of memory optimization through temporarily saving data in memory with low latency to reuse it and reduce number of accesses to global memory which has high latency. Register is fastest memory and the same approach might be applied to reduce number of accesses to shared memory by saving data in registers and reusing it several times.

2) Reading from global memory:
In this sub-section, aspects of effective access to global memory will be discussed. Memory coalescing technique allows to effective usage of global memory bandwidth. To achieve such result, threads of one warp have to access consecutive addresses in global memory. Example of mapping threads to consecutive addresses is given in line 16 of source code.

3) Shared load and store:
Hardware implementation of shared memory consists of so called memory banks. Each bank is able to work (load/store) with only one data address at a time. For modern GPU's it is 32 banks which is equal to number of threads in one warp. Such design is implemented to give opportunity for all threads to work with separate banks at one time (Nvidia developer, 2013). If threads are accessing different addresses that are in the same bank, such access will be serialized. It is important to avoid such cases and exploit parallelism of shared load/store instructions. For instance, when threads are storing values into shared memory, consecutive addresses of shared array guarantees no bank conflicts. Line 16 of source code represents avoiding bank conflicts by mapping threads of one warp into consecutive destination addresses.

B. Thread part

In general, we are interested in increasing block work size because it reduces access to global memory. And increasing threads per block will increase block work size also, so why not just increase threads per block as maximum as possible? Limitations:

- SM's resources such as registers and shared memory are limited. By increasing threads per block we are reducing available resources per thread.
- In SGEMM kernel with tiled approach there are synchronizations. As massive our block, more likely that most of warps will stall waiting at synchronization part.
- Resources allocated by block can not be freed up for new block until block totally not finished. So if several warps

of a massive block are taking much time, new warps can not be created, reducing the number of available active warps.

Executing multiple outputs per thread is another use of memory hierarchy of GPUs. Global memory -> shared memory optimization allowed to diminish access to global memory using shared memory as temporary place for reusable values. Shared memory -> registers is next level of memory, temporarily saving in registers and reducing high latency (relatively) access to shared memory. More outputs per thread -> less number of access to shared memory and more effective of each read operation, yielding higher percentage of FMA operations in the main computational loop. Limitations:

- More outputs per thread -> more register allocation per thread. Number of registers per SM is limited and too much usage per thread will decrease active warps which leads to low occupancy
- Experiments on register blocking (trying to find balance between register blocking and occupancy)

IV. Experimental Results

Two main points derived from motivational example showed us two tendencies: 1) Increasing executed values per thread leads to faster kernel execution. 2) There is no clear correlation between block size and execution time of kernels with optimized memory hierarchy. To validate these tendencies in a more detailed and more numerous conditions, experiments on varying parameters of values per thread and block size is conducted. The main idea behind conducted experiments is data driven method. Then results were described using metrics and explained according to theory. Various combinations of parameters were tested while other parameters fixed, this allows to see tendency of that parameter and feel the importance from variation of inference time. Block size, register blocking and tile size parameters have been experimented.

1) Experiments on block size: The table below describes various block sizes experimented with constant multi-value per thread

Block size	8x8	16x8	16x16	32x16	32x32
Values per thread	1	1	1	1	1
Values per thread	4	4	4	4	4
Values per thread	16	16	16	16	16
Values per thread	64	64	64	64	64

2) Experiments on values per thread: The tables below describes various values per thread experimented with constant block size

V.p.t	1	4	16	36	64	100	144
B.s.	8x8	8x8	8x8	8x8	8x8	8x8	8x8
B.s.	16x16	16x16	16x16	16x16	16x16	16x16	16x16
B.s.	32x16	32x16	32x16	32x16	32x16	32x16	32x16

```
#define ms 8192
#define bs_x 16
#define bs_y 16
#define rb_x 8
#define rb_y 8
#define TILE_WIDTH_X 128
#define TILE_WIDTH_Y 128
#define TILE_COMMON 16

unsigned int grid_rows = ms/TILE_WIDTH_Y;
unsigned int grid_cols = ms/TILE_WIDTH_X;


dim3 dimGrid(grid_cols, grid_rows);
dim3 dimBlock(bs_x, bs_y);

gpu_matrix_mult<<<dimGrid, dimBlock>>>(d_a, d_b,
 d_c, m, n, k);
```

Listing 1. Kernel Configuration of Our_best SGEMM algorithm

```
__global__ void gpu_matrix_mult(float *a,float *b,
    float *c, int m, int n, int k){

    __shared__ float A[TILE_WIDTH_Y][TILE_COMMON];
    __shared__ float B[TILE_COMMON][TILE_WIDTH_X];

    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    float sum[rb_y*rb_x]={0};
    for (int t=0; t<n/TILE_COMMON; t++)
    {
      for (int i=0; i<TILE_COMMON/bs_y; i++){
          for (int j=0; j<TILE_WIDTH_X/bs_x; j++){
              B[ty+bs_y*i][tx+bs_x*j] = b[bx*
    TILE_WIDTH_X+t*TILE_COMMON*n+(ty+bs_y*i)*n+tx+
    bs_x*j];
          }
      }
      for (int i=0; i<TILE_COMMON/bs_x; i++){
           for (int j=0; j<TILE_WIDTH_Y/bs_y; j++){
            A[ty+j*bs_y][tx+i*bs_x] = a[by*
    TILE_WIDTH_Y*n+t*TILE_COMMON+(ty+j*bs_y)*n+tx+i*
    bs_x];
          }
      }

      __syncthreads();
      for (int i=0; i<TILE_COMMON; i++){

          float b_t[rb_x];
          float a_t[rb_y];
          for (int k=0; k<rb_x; k++){
              b_t[k]=B[i][bs_x*k+tx];
          }
          for (int k=0; k<rb_y; k++){
              a_t[k]=A[bs_y*k+ty][i];
```

```
            }
            for (int q=0; q<rb_y; q++){
                for (int w=0; w<rb_x; w++){
                    sum[rb_x*q+w]+=b_t[w]*a_t[q];
                }
            }

        }
        __syncthreads();
    }
    for (int j=0; j<rb_y; j++){
        for (int i=0; i<rb_x; i++){
            c[by*TILE_WIDTH_Y*n+bx*TILE_WIDTH_X+j*
    bs_y*n+bs_x*i+ty*n+tx]=sum[rb_x*j+i];
        }
    }
}
```

Listing 2. Source code of Our_best SGEMM algorithm

```
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <cublas_v2.h>
#include <curand.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>



int main(){
    int n = 8192;
    size_t bytes = n*n*sizeof(float);

    float *h_a, *h_b, *h_c;
    float *d_a, *d_b, *d_c;

    h_a = (float*)malloc(bytes);
    h_b = (float*)malloc(bytes);
    h_c = (float*)malloc(bytes);
    cudaMalloc(&d_a, bytes);
    cudaMalloc(&d_b, bytes);
    cudaMalloc(&d_c, bytes);

    cublasHandle_t handle;
    cublasCreate(&handle);

    float alpha = 1.0;
    float beta = 0.0;

    cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, n,
     n, n, &alpha, d_a, n, d_b, n, &beta, d_c, n);
    cudaDeviceSynchronize();


    cudaMemcpy(h_a, d_a, bytes,
    cudaMemcpyDeviceToHost);
    cudaMemcpy(h_b, d_b, bytes,
    cudaMemcpyDeviceToHost);
    cudaMemcpy(h_c, d_c, bytes,
    cudaMemcpyDeviceToHost);
    return 0;
}
```

Listing 3. Source code of CuBLAS SGEMM algorithm

A. Experimental Setup

Tools:
1) NVIDIA (R) Cuda compiler (nvcc) version 10.1.243
2) NVIDIA (R) Cuda command line profiler (nvprof) version 10.1.243

Platforms:
1) Google Colab (NVIDIA Tesla T4)
2) Laptop (NVIDIA GeForce GTX 1050 with Max-Q Design)

Measuring Metrics: using Nvprof

- Regs
- global_load
- shared_load
- shared_efficiency
- achieved_occupancy
- stall_memory_dependency

B. Results and Analysis

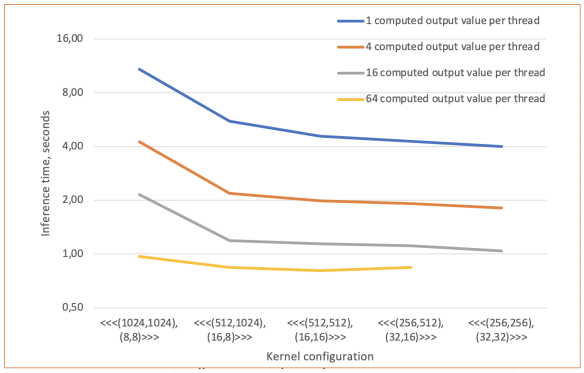

Fig. 4. Inference time vs thread block dimension (matrix size 8192x8192)

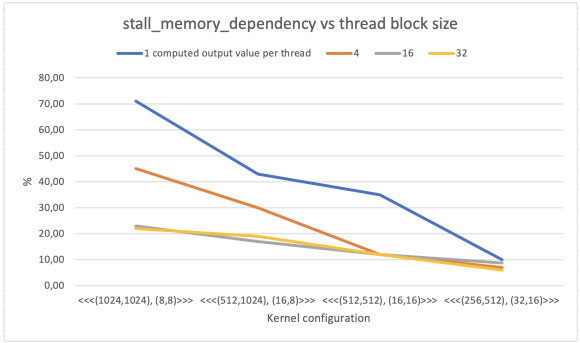

Fig. 5. Stall memory dependency vs thread block size (matrix size 8192x8192)

1) Observation in global-load: Global-load metric shows number of read memory instructions from global memory. As seen in figure 10, increasing both block size or computed value per thread reduces access to global memory. Thread part of methodology section explains why this happens.

2) Observation in stall-memory-dependency: Figure 5 shows that stall-memory-dependency level is decreasing with increased number of computed values per thread or increased block size. If we compare it with figure 9, there is clear correlation between global-load and stall-memory-dependency

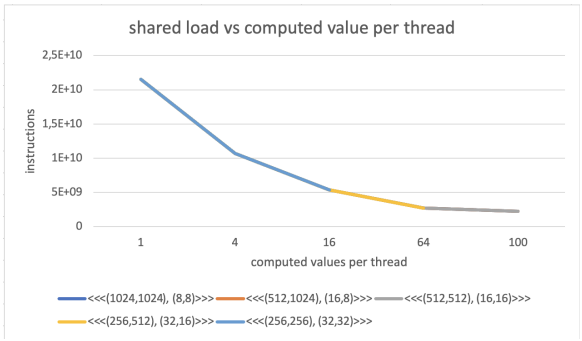

Fig. 6. shared load vs values per thread (matrix size 8192x8192)

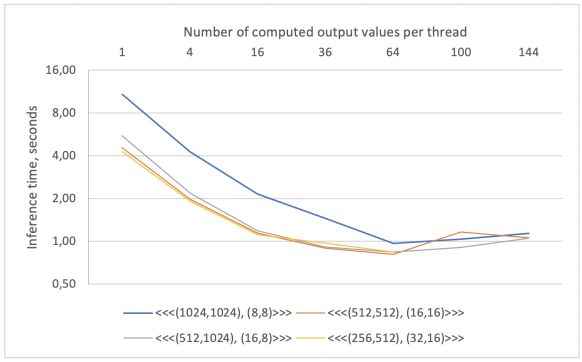

Fig. 7. Inference time vs Values per thread (matrix size 8192x8192)

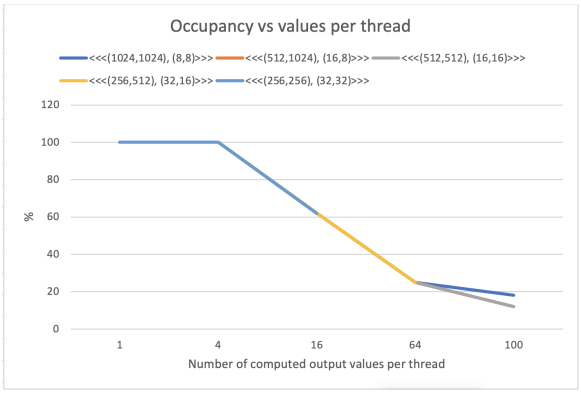

Fig. 8. Achieved occupancy vs values per thread for various block sizes (matrix size 8192x8192)

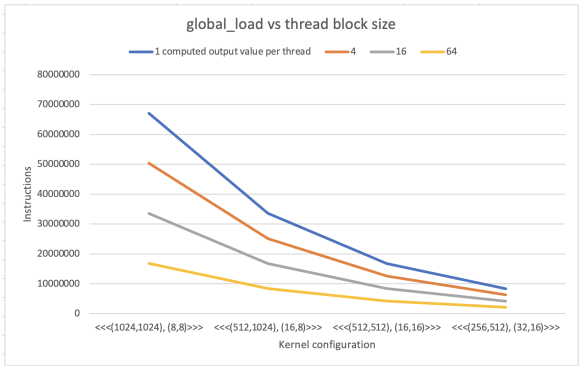

Fig. 9. Global load vs thread block size (matrix size 8192x8192)

metrics. Less access to global memory reduces pressure on memory read units and warps are less stall on waiting their inputs due to non-available memory units.

3) Observation in shared-load: Shared-load metric shows number of read memory instructions from shared memory. Figure 6 shows that shared-load depends only on computed values per thread, not on block size

4) Observation in shared-efficiency: Figure 10 shows that shared efficiency is independent from computed values per thread. For various block size dimensions, shared efficiency level varies, and there is no clear correlation with block size.

5) Observation in occupancy: Figure 8 shows that occupancy is reducing with increased computed values per thread. This is due to limited register resource. The best result in terms of execution time occurs with 64 values per thread, so high occupancy does not mean better performance. The same conclusion was made by Xiaqing L. et al (2016).

Experiment part 1 results showed that increasing block dimension (threads per block) with constant value per thread will result in faster execution time. The main reason of such result is in expanded block work size when we increased block dimension. The idea behind why increased block work size gives speed-up is discussed in the 'thread part' section of methodology part. Shortly, increased block size means less access to global memory by reusing each read from global more times. This explanation can be justified by metrics such as global-load, gld-inst-32bit, these metrics show how much kernel accessing global memory. Figure 9 shows that with increasing thread block size, number of accesses to global memory decreases. This should affect warps to less stall on waiting on memory operations. This might be justified by metric stall-memory-dependency.Stall-memory-dependency metric shows percentage of stall occurring because a memory operation cannot be performed due to required resources not being available or fully utilized. Kernels with more more block work sizes are utilizing each read more efficiently, thus warps are less in stall by waiting when it's input value will be available. Stall-memory-dependency, global-load and gld-inst-32bit metrics are decreasing by increased block work size which is what we expected.

Experiment part 2 results showed that increasing values per thread while constant thread block size gives faster execution time. It was found 2 significant reasons on such results. First one is the same as explanation of experiment part 1. Shortly, increasing values per thread also increases block work size which leads to less access to global memory. Second reason is also related to memory part, increasing values per thread optimizes shared memory accesses. This explanation can be justified by metric shared-load, which shows that increasing values per thread (or register blocking) results in less access to shared memory, reusing each access to shared memory more efficiently. At this moment it is important to understand that increasing block work size by increasing thread block size and

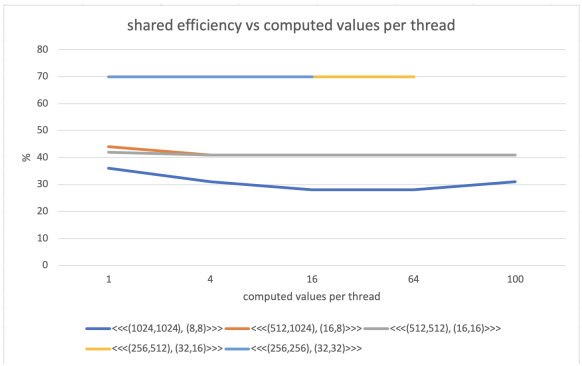

Fig. 10. shared efficiency vs thread block size (matrix size 8192x8192)

by values per thread is different. Increasing values per thread gives roughly all advantages of increasing thread block size, plus, at the same time optimizing shared memory loads (figure 6). So, if it is available in terms of resources to increase thread block size and values per thread, second one is preferable. But having too less thread block size is not efficient due to low occupancy. Figure 9 justifies this assumption, while occupancy is too low, increasing values per thread gives no boost-up. This is also reason on why we cannot increase values per thread as much as possible, more values per thread -> more registers occupied by thread -> less active warps -> less occupancy. For instance, in figures 7 and 8, it is shown that all implementations are slowing down when occupancy level drops down below 25%. From our observations, looking for best kernel configuration for SGEMM can be interpreted as finding balance between occupancy and values per thread. So, we discussed two tendencies, thread block size and values per thread, increasing both will give speed-up. However both actions will increase register usage, since register resource is limited it is not possible to optimize the kernel by just increasing both thread block size and values per thread. Above we discussed that increasing values per thread is preferable than thread block size. This can be justified by figure 9 and 6, where we see that increasing register blocking gives both reduced global accesses and reduced shared memory accesses. While increasing thread block size gives only reduced global memory accesses. So, if values per thread is preferable than thread block size, then why we don't just increase values per thread as much as possible by reducing thread block size? In that case we are gonna meet low occupancy issue. Figures 7 and 8 shows that increasing values per thread gives speed-up until occupancy drops below 25%. After observations it is assumed that finding best kernel configuration problem can be interpreted as finding balance between values per thread and occupancy.

The main differences between our best and cublas are in memory hierarchy, global-load and shared load numbers are high in our best implementation (see Table 1).

TABLE I
COMPARISON WITH CUBLAS:

		Our best (a)	Cublas(b)	Ratio a/b
	Execution time	795 ms	675 ms	1.17
Memory hierarchy	global-load	2.7E7	1.0E7	2.7
	shared-load	2.7E9	1.08E9	2.5
	stall-memory-dependency	11.6	34.5	0.34
	shared-efficiency	41	44	0.93
Thread hierarchy	occupancy	25	25	1
	registers per thread	128	122	1.05

V. CONCLUSION

The paper studied matrix multiplication problem on GPU and tried to observe what factors should be considered configuring on SGEMM kernel. The experiments and analysis showed that matrix multiplication problem is mostly optimizing memory hierarchy problem. The basic matrix multiplication implementation (using only global memory) has low arithmetic/memory operation ratio. Such design is inappropriate because memory latency is too high to provide GPU's sp cores with enough data and fully utilize GPU's execution cores. The solution is reusing accessed data to diminish pressure on LSUs. This was realized by exploiting 3 level memory hierarchy of GPUs, gloabal memory, shared memory and registers. The experiments and metrics showed that optimization ways of matrix multiplication kernels are mostly due to effective memory usage. Increasing values per thread resulted in both global memory accesses and shared memory accesses, while increasing thread block size gives only effective global memory access. Increasing both values per thread and thread block size consumes register usage which is restricted resource, so priority is in increasing values per thread due to reasons discussed earlier. However, it is important to keep thread block size in an appropriate level because low occupancy might be bottleneck if we are trying to increase values per thread by reducing thread block size.

VI. REFERENCES

[1] Huang, Z., Ma, N., Wang, S., Peng, Y. (2019). GPU computing performance analysis on matrix multiplication. The Journal of Engineering, 2019(23), 9043-9048. doi: 10.1049/joe.2018.9178

[2] Nugteren C. (2014). Tutorial: OpenCL SGEMM tuning for Kepler. https://cnugteren.github.io/tutorial/pages/page1.html

[3] Docs Nvidia 2015 Profiler User Programming Guide CUDA Toolkit documentation.

[4] Li, X., Zhang, G., Huang, H. H., Wang, Z., amp; Zheng, W. (2016). Performance analysis of GPU-based Convolutional Neural Networks. 2016 45th International Conference on Parallel Processing (ICPP). https://doi.org/10.1109/icpp.2016.15