The University of Washington Bothell
CSS-535: High Performance Computing, Winter Quarter 2021

Lab 03: CUDA GEMV

Afrooz Rahmati & Franz Anthony Varela
Instructor: Dr. Erika Parsons

19 February 2021

Purpose

The purpose of this lab is to further strengthen our understanding of the fundamentals of CUDA and the GPU hardware by implementing GEMV as a CUDA kernel. There are four main parts to this lab that involve slightly different variations of the naive algorithm, targeting specific aspects of the device (such as shared memory) to observe the possible benefits of utilizing them. We also explore the usage and benefits of profilers to gain intuition on proper kernel and execution configuration design.

Background

Not much new background can be said on this topic that hasn't already been covered in previous assignments. For a brief description of GEMV, refer back to the Program 1.A document; for a brief description on CUDA, refer back to the Lab 2 document.

Implementation Details

Hardware Specs

1. Tony
    - CPU: AMD Ryzen 9 3900x[*] (12 cores, 24 threads, 3.8 GHz Base)
    - GPU: NVIDIA GTX 1080ti[†] (Pascal, 11 GB VRAM, 11 Gbps, 1582 MHz); Compute Capability 6.1

[*]https://www.amd.com/en/products/cpu/amd-ryzen-9-3900x
[†]https://www.nvidia.com/en-sg/geforce/products/10series/geforce-gtx-1080-ti/

- OS: Windows 10 Pro (64-Bit)

2. Afrooz

    - CPU: Intel(R) Core(TM) i7-6700 (4 cores, 8 threads, 3.40GHz Base)
    - GPU: GeForce GTX 745(1.8 Gbps, 1582 MHz); Compute Capability 3.0
    - OS: CentOS Linux release 7.6.1810 (Core)

Tony is primarily running his experiments on his own machine, while Afrooz focuses on running her experiments through the Linux Labs provided by UWB.

Experiments Setup

This lab is divided into 4 parts: naive, global, shared, and register focused implementations. Across all of them, we are required to record the FLOPS, global hit rate (L1), shared memory, and number of registers used per thread (not all of the implementations will have every metric).
In this lab, although the instructions say to test different vector sizes, we primarily test each of these implementations and execution configurations with a vector of size 10000, for two main reasons:

1. With a square matrix involved, the memory requirements blow up quite quickly. When we test a vector of size 10000, that also requires a square matrix of $10000 * 10000$ elements, which is already almost a gigabyte of memory. As such, we cannot test all the way to extensively large vector sizes.

2. From Program 1.A, we were only required to test our implementations with a vector size of up to 10000 elements, and would be an appropriate maximum size to compare with this lab as well.

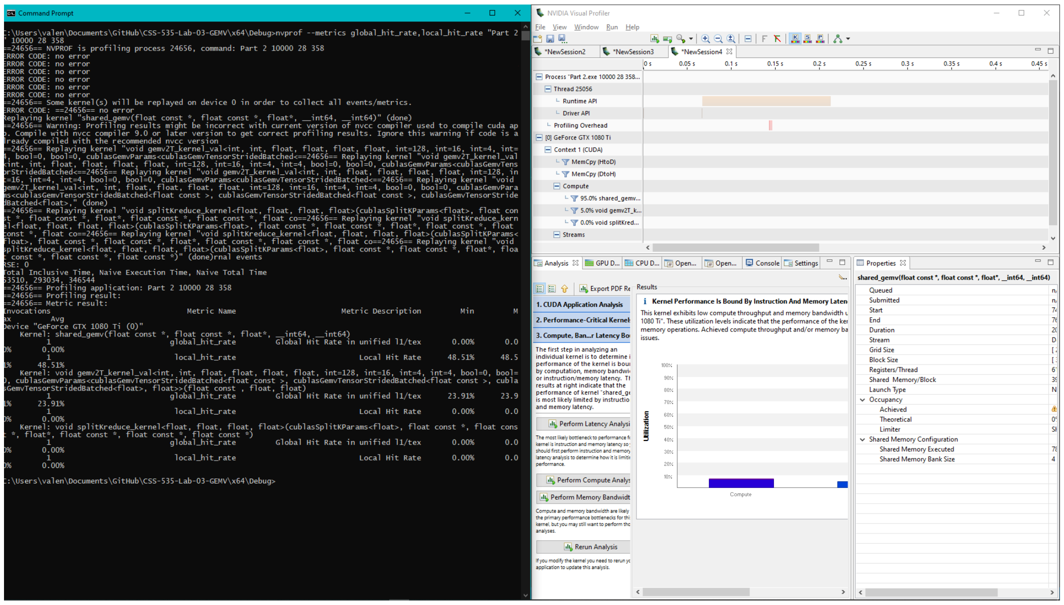
Another fundamentally different approach with this particular lab as opposed to the previous one is that we were encouraged to create kernels that did not operate on just one element[‡]; the kernel should be flexible in being able to run the full calculation regardless of the execution configuration. This ended up in an implementation that closely resembled our original threaded GEMV function from Program 1.A, where each thread in the grid is in charge of a consecutive set of vector elements, depending on the ratio between available threads and vector elements. Although this scheme is relatively simple, it suffers from giving imbalanced workloads across the threads (in particular, the last thread in a grid is prone to a larger remainder of the vector elements).

Profiling

In this assignment, we utilize profilers to not only give us the metrics required, but also to see where we can optimize our kernel and execution configuration design. Both of us used nvprof primarily as it was easy to use and could be run on just a command line (as Afrooz requires the Linux machine for CUDA applications, this is her only option). Tony tried to experiment with using Nsight Compute through Visual Studio, but the particular version he was using had already removed support for his particular GPU (Pascal); because of this,

[‡]Thanks Dylan!

he used the Visual Profiler application instead, which provided a lot of useful information about the metrics of the program, yet had a tendency to crash for execution configurations that provided less threads than vector elements.



```
:\Users\valen\Documents\GitHub\CSS-535-Lab-03-GEMV\x64\Debug>nvprof --metrics global_hit_rate,local_hit_rate "Part 2
' 10000 28 358
==24656== NVPROF is profiling process 24656, command: Part 2 10000 28 358
ERROR CODE: no error
ERROR CODE: no error
ERROR CODE: no error
ERROR CODE: no error
ERROR CODE: no error
ERROR CODE: no error
==24656== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
ERROR CODE: ==24656== no error
Replaying kernel "shared_gemv(float const *, float const *, float*, __int64, __int64)" (done)
==24656== Warning: Profiling results might be incorrect with current version of nvcc compiler used to compile cuda ap
p. Compile with nvcc compiler 9.0 or later version to get correct profiling results. Ignore this warning if code is a
lready compiled with the recommended nvcc version
==24656== Replaying kernel "void gemv2T_kernel_val<int, int, float, float, float, float, int=128, int=16, int=4, int=
4, bool=0, bool=0, cublasGemvParams<cublasGemvTensorStridedBatched<==24656== Replaying kernel "void gemv2T_kernel_val
<int, int, float, float, float, float, int=128, int=16, int=4, int=4, bool=0, bool=0, cublasGemvParams<cublasGemvTens
orStridedBatched<==24656== Replaying kernel "void gemv2T_kernel_val<int, int, float, float, float, float, int=128, in
t=16, int=4, int=4, bool=0, bool=0, cublasGemvParams<cublasGemvTensorStridedBatched<==24656== Replaying kernel "void
gemv2T_kernel_val<int, int, float, float, float, float, int=128, int=16, int=4, int=4, bool=0, bool=0, cublasGemvPara
ms<cublasGemvTensorStridedBatched<float const >, cublasGemvTensorStridedBatched<float const >, cublasGemvTensorStride
dBatched<float>," (done)
==24656== Replaying kernel "void splitKreduce_kernel<float, float, float, float>(cublasSplitKParams<float>, float con
st *, float const *, float*, float const *, float const *, float co==24656== Replaying kernel "void splitKreduce_kern
el<float, float, float, float>(cublasSplitKParams<float>, float const *, float const *, float*, float const *, float
const *, float co==24656== Replaying kernel "void splitKreduce_kernel<float, float, float, float>(cublasSplitKParams<
float>, float const *, float const *, float*, float const *, float const *, float co==24656== Replaying kernel "void
splitKreduce_kernel<float, float, float, float>(cublasSplitKParams<float>, float const *, float const *, float*, floa
t const *, float const *, float const *)" (done)rnal events
RSE: 0
Total Inclusive Time, Naive Execution Time, Naive Total Time
53510, 293034, 346544
==24656== Profiling application: Part 2 10000 28 358
==24656== Profiling result:
==24656== Metric result:
Invocations                               Metric Name                        Metric Description         Min         M
ax         Avg
Device "GeForce GTX 1080 Ti (0)"
    Kernel: shared_gemv(float const *, float const *, float*, __int64, __int64)
          1                           global_hit_rate         Global Hit Rate in unified l1/tex       0.00%       0.0
0%       0.00%
          1                            local_hit_rate                            Local Hit Rate      48.51%      48.5
1%      48.51%
    Kernel: void gemv2T_kernel_val<int, int, float, float, float, float, int=128, int=16, int=4, int=4, bool=0, bool=
0, cublasGemvParams<cublasGemvTensorStridedBatched<float const >, cublasGemvTensorStridedBatched<float const >, cubla
sGemvTensorStridedBatched<float>, float>>(float const , float, float)
          1                           global_hit_rate         Global Hit Rate in unified l1/tex      23.91%      23.9
1%      23.91%
          1                            local_hit_rate                            Local Hit Rate       0.00%       0.0
0%       0.00%
    Kernel: void splitKreduce_kernel<float, float, float, float>(cublasSplitKParams<float>, float const *, float cons
t *, float*, float const *, float const *, float const *)
          1                           global_hit_rate         Global Hit Rate in unified l1/tex       0.00%       0.0
0%       0.00%
          1                            local_hit_rate                            Local Hit Rate       0.00%       0.0
0%       0.00%

C:\Users\valen\Documents\GitHub\CSS-535-Lab-03-GEMV\x64\Debug>
```

Figure 1: A screenshot of the command line profiler nvprof (left) and the Visual Profiler (right). We can generate profile record files from nvprof and load them into the Visual Profiler, or directly profile the executable while it is running in either profiler.

Part 0: Naive GEMV

This part's implementation was relatively straightforward - what took the most time was deciding on the execution configurations to test.
As a starting point[§], Tony wanted to test the SM and SP count of his device (28 and 128, respectively) as the grid and block sizes, first at $n = 5000$; he later changes to $n = 10000$ and experiments with different multiples of his SM and SP count as well.

Config	Size	Blocks	Threads
1	5000	28	179
2	5000	40	128
3	10000	84	128
4	10000	40	256
5	10000	20	512

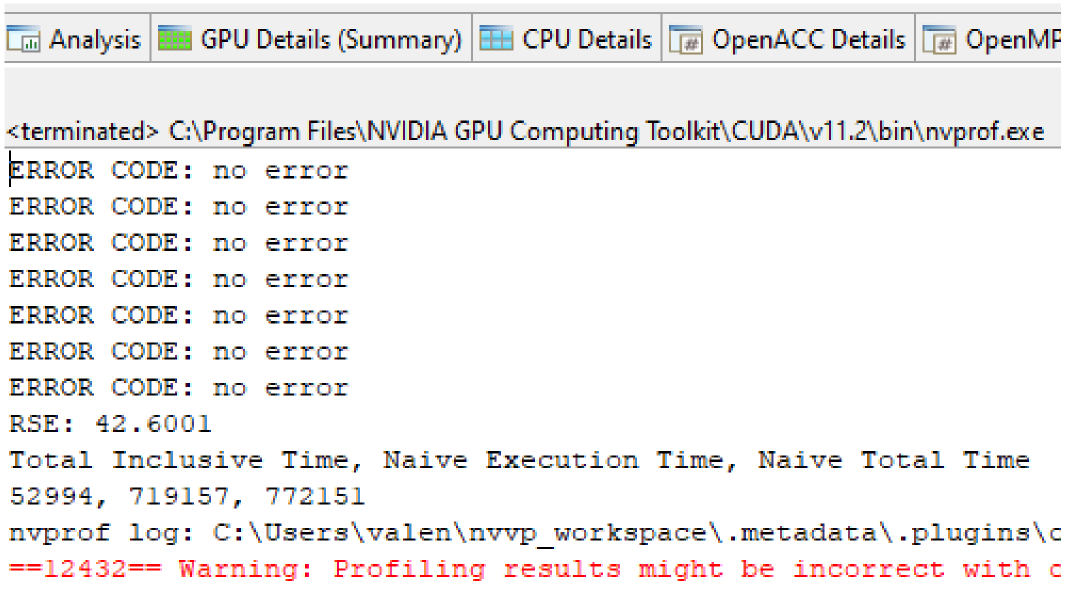
Below are the results from each of the configurations. Note that the Global and Local headers refer to the hit rates, Reg. refers to the number of registers used in the kernels, and Ach. Occ. is the achieved occupancy.

[§]Again, thanks to Dylan.

Config	Duration (ms)	GFLOPS	Global	Local	Shared	Reg.	Ach. Occ.
1	7.83	6.38	0.0%	0.0%	0	39	9.3%
2	7.57	6.61	0.0%	0.0%	0	39	8.8%
3	19.81	10.04	0.0%	0.0%	0	39	17.7%
4	26.33	7.60	0.0%	0.0%	0	39	19.1%
4	26.34	7.60	0.0%	0.0%	0	39	24.6%

Below is the data we got from Linux lab machine.

Config	Duration (ms)	GFLOPS	Global	Local	Shared	Reg.	Ach. Occ.
1	14.431	3.46	0.0%	0.0%	0	32	83%
2	13.83	3.615	0.0%	0.0%	0	32	78%
3	79.61	2.512	0.0%	0.0%	0	32	85%
4	81.06	2.46	0.0%	0.0%	0	32	86%
4	80.932	2.47	0.0%	0.0%	0	32	84%



```
<terminated> C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.2\bin\nvprof.exe
ERROR CODE: no error
ERROR CODE: no error
ERROR CODE: no error
ERROR CODE: no error
ERROR CODE: no error
ERROR CODE: no error
ERROR CODE: no error
RSE: 42.6001
Total Inclusive Time, Naive Execution Time, Naive Total Time
52994, 719157, 772151
nvprof log: C:\Users\valen\nvvp_workspace\.metadata\.plugins\c
==12432== Warning: Profiling results might be incorrect with c
```

Figure 2: A screenshot of our naive kernel running from the Visual Profiler console (10000 elements, 28 blocks, 512 threads); at every CUDA API call, we output the error code. At the end, we output the running sum error (RSE) between our implementation a cuBLAS', and some general metrics about the inclusive and execution times. We do not output the metrics from the program directly, since we have the profilers to record this data for us; every other implementation runs similarly to this one.

Part 1: Global Memory

For the global memory implementation, we do not need to change the naive GEMV kernel, but rather focus on the execution configuration; as stated in the ”Global Memory Considerations” slides from Dr. Parsons, we need to focus on thread management and data transfers when thinking about *just* global memory. More specifically, according to [1], global memory access is coalesced per thread warp (i.e. data required by each thread is loaded at the same time); ideally, to be as fast as possible, the data required per thread executing the same instruction in a warp are contiguous. Since warps are scheduled in thread groups of 32, we experiment with multiples and non-multiples of 32 and observe if there are any noticeable changes. However, an important thing to keep in mind is that as mentioned above, our naive implementation is strided, which means that a particular kernel thread can be in charge of one or more contiguous output vector elements; two sequential thread indexes might be working on output elements that are far apart depending on the amount of threads available, so we strive to keep the stride to at most 1 (i.e. have at least as many threads as elements). Furthermore, with the nature of matrix-vector multiplication and how our data is stored in row-major order, we already pay a penalty for threads in a warp trying to access global memory, since the relevant data for the current instruction they are executing could be far apart physically.

Config	Size	Blocks	Threads
1	10000	28	512
2	10000	28	400
3	10000	28	600
4	10000	168	64
5	10000	168	60

Config	Duration (ms)	GFLOPS	Global	Local	Shared	Reg.	Ach. Occ.
1	21.46	9.32	0.0%	0.0%	0	39	24.6%
2	23.03	8.68	0.0%	0.0%	0	39	19%
3	40.31	4.96	0.0%	0.0%	0	39	27.9%
4	19.98	10.01	0.0%	0.0%	0	39	16.8%
5	21.01	9.52	0.0%	0.0%	0	39	18.6%

The top three configurations revolve around testing a multiple of 32 (512), and then values below and above it. We see that in both cases, we have a penalty in runtime, presumably because of how the threads within the blocks cannot be evenly divided into warps of 32.
For the fourth configuration, Tony took a look at the Visual Profiler's suggestion for the number of blocks based on its theoretical occupation calculation; this came out to be about 6 blocks per SM, so Tony tried $6 * 28 = 168$ blocks. This appears to improve the runtime, however only appears to be 16.8% of achieve occupancy (furthermore, the reported theoretical occupancy was only 31%). A slight variant of the fourth configuration, that attempts to closely reduce the amount of wasted threads at 60 threads per block, is shown to run slightly a bit longer.

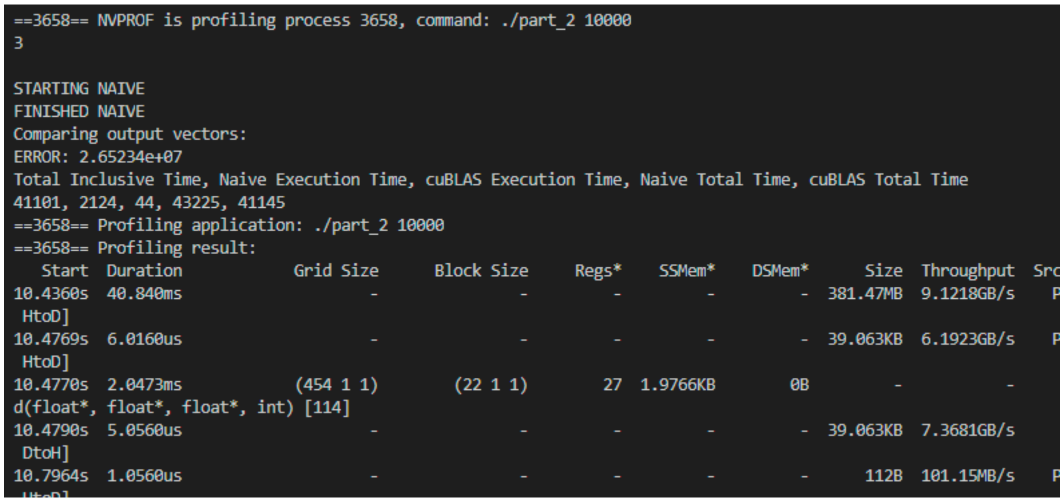
Part 2: Shared Memory Use

For the shared memory implementation, we used the tiled approach to take the advantage of locality and cache. In this approach the component of vector and matrix move to the shared memory for fast access and computation. The term __shared__ used to tell the compiler to put our data into multi processor's shared memory. This implementation is highly depends on our selection for the Block Size. for the optimum performance the size of the tiled (Block_Size) should match the size of device shared memory, otherwise there would be an overhead of processing data from Global Memory. The UWB lab machine has the compute capability of 3.0 , with 3 SM with maximum amount of 48KB per multiprocessor. We handled the elements which are falling outside the not-fully overlapping tiles with zero-ed. So we can handle the arbitrary size of matrix. The best performance achieved by block Size 22. A float takes 4 bytes, so we have requested 2*484×4 bytes 3k of shared memory. So with the 48KB,and compute capability of 3.0 , the multiprocessor can compute up to 16 number of blocks. So 16 * 3 =48 and exactly match the shared memory size. [2]

Config	Size	Blocks	Threads
1	5000	16	313
2	10000	16	626
3	10000	22	455
4	10000	28	358
5	10000	40	250
6	10000	20	500

We test lab machine shared memory implementation with above configuration. The lab machine has 16 resident block per multiprocessor.

Config	Duration (ms)	GFLOPS	Global	Local	Shared	Reg.	Ach. Occ.
1	0.0006	8.3	0.0%	0.0%	$1.06KB$	32	43%
2	3.5	57.14	0.0%	0.0%	$1.06KB$	32	47%
3	2.03	98.28	0.0%	0.0%	$1.97KB$	27	47%
4	2.14	93.38	0.0%	0.0%	$3.17KB$	32	27%
5	2.73	73.26	0.0%	0.0%	$6.40KB$	32	26%
6	2.65	75.47	0.0%	0.0%	$1.64KB$	32	45%



```
==3658== NVPROF is profiling process 3658, command: ./part_2 10000
3

STARTING NAIVE
FINISHED NAIVE
Comparing output vectors:
ERROR: 2.65234e+07
Total Inclusive Time, Naive Execution Time, cuBLAS Execution Time, Naive Total Time, cuBLAS Total Time
41101, 2124, 44, 43225, 41145
==3658== Profiling application: ./part_2 10000
==3658== Profiling result:
   Start  Duration            Grid Size      Block Size     Regs*    SSMem*    DSMem*      Size  Throughput  Src
10.4360s  40.840ms                    -               -         -         -         -  381.47MB  9.1218GB/s    P
 HtoD]
10.4769s  6.0160us                    -               -         -         -         -  39.063KB  6.1923GB/s    P
 HtoD]
10.4770s  2.0473ms            (454 1 1)        (22 1 1)        27  1.9766KB        0B         -           -
d(float*, float*, float*, int) [114]
10.4790s  5.0560us                    -               -         -         -         -  39.063KB  7.3681GB/s
 DtoH]
10.7964s  1.0560us                    -               -         -         -         -      112B  101.15MB/s    P
```

Figure 3: sample screenshot of our shared memory kernel running from Linux machine for the size=10000 , block=22 and threads=455 with profiling option.

While Afrooz worked on a tiled implementation, Tony tried his own variant of the shared memory implementation by focusing on fitting the input vector $\vec{x}$ within shared memory; this made the most sense to him because this vector is the only thing reused within matrix-vector multiplication[¶]. In this way, since each block can now have their own local copy of the input vector, we would imagine that we could get away with using less blocks and more threads per block. Note that we dynamically allocate the shared memory, instead of statically; since we only test 1000 elements, it fits nicely into the default shared memory max of 48KB[‖].

Config	Size	Blocks	Threads
1	10000	28	358
2	10000	78	128

Config	Duration (ms)	GFLOPS	Global	Local	Shared	Reg.	Ach. Occ.
1	21.55	9.28	0.0%	48.5%	$40KB$	61	18.6%
2	225.04	0.89	0.0%	48.6%	$40KB$	61	9.7%

As we can see, the configuration that had more blocks and less threads ran for a longer time; this is most likely due to how there will be more kernels copying the input vector into shared memory and syncing (since syncs happen across a thread block, more blocks will naturally lead to more synchronizations).

[¶]Note that our kernel implementations, even though they *could* compute multiple elements, they do so *fully*, i.e. do *not* spread the calculation of one output vector element across multiple threads.

[‖]While comparing our implementation with cuBLAS' in the profiler, we noticed that cuBLAS SGEMV uses a statically allocated block of shared memory, around 25.6KB.

Part 3: Register Use

Let us revisit configuration 3 from part 0; without manually specifying the register per thread limit, the each kernel is recorded to be using 39 registers. In this implementation, we try to unroll the dot product**. We first take a look at what happens when we try to only unroll the loop four and eight times:

Config	Size	Blocks	Threads	Unroll
1	10000	84	128	4
2	10000	84	128	8

Config	Duration (ms)	GFLOPS	Global	Local	Shared	Reg.	Ach. Occ.
1	21.04	9.50	0.0%	0.0%	0	39	17.7%
2	21.70	9.22	0.0%	0.0%	0	39	17.7%

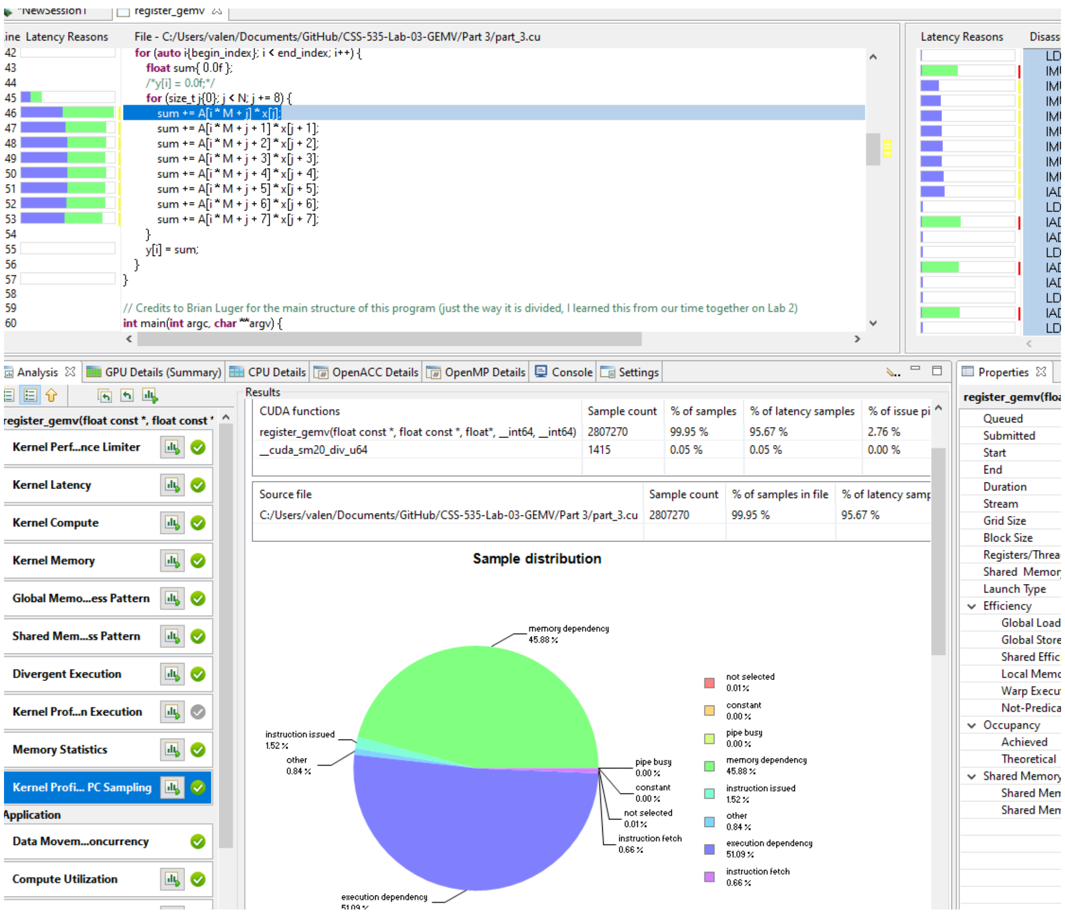
Not much changes apparently - only the duration/GFLOPS between the two unrolls change slightly. We further try testing this configuration by setting a limit to the maximum registers per thread to 16.

Config	Size	Blocks	Threads	Unroll
3	10000	84	128	8

Config	Duration (ms)	GFLOPS	Global	Local	Shared	Reg.	Ach. Occ.
3	27.03	7.40	0.0%	0.0%	0	16	17.6%

This time around, we actually see a decrease in performance; a closer look at the PC sampling from the Visual Profiler shows that our kernel begins to suffer from a larger memory dependency than before (when it used to be primarily dominated by just an execution dependency) - this is register spilling in action.

**We do not attempt to unroll the amount of vector elements we process, since the division would be different for every execution configuration.



File - C:/Users/valen/Documents/GitHub/CSS-535-Lab-03-GEMV/Part 3/part_3.cu

```
    for (auto i{begin_index}; i < end_index; i++) {
        float sum{ 0.0f };
        /*y[i] = 0.0f;*/
        for (size_t j{0}; j < N; j += 8) {
            sum += A[i * M + j] * x[j];
            sum += A[i * M + j + 1] * x[j + 1];
            sum += A[i * M + j + 2] * x[j + 2];
            sum += A[i * M + j + 3] * x[j + 3];
            sum += A[i * M + j + 4] * x[j + 4];
            sum += A[i * M + j + 5] * x[j + 5];
            sum += A[i * M + j + 6] * x[j + 6];
            sum += A[i * M + j + 7] * x[j + 7];
        }
        y[i] = sum;
    }
}

// Credits to Brian Luger for the main structure of this program (just the way it is divided, I learned this from our time together on Lab 2)
int main(int argc, char **argv) {
```

CUDA functions	Sample count	% of samples	% of latency samples	% of issue pi
register_gemv(float const *, float const *, float*, __int64, __int64)	2807270	99.95 %	95.67 %	2.76 %
__cuda_sm20_div_u64	1415	0.05 %	0.05 %	0.00 %

Source file	Sample count	% of samples in file	% of latency samp
C:/Users/valen/Documents/GitHub/CSS-535-Lab-03-GEMV/Part 3/part_3.cu	2807270	99.95 %	95.67 %

Figure 4: A screenshot of the PC Sampling from the Visual Profiler of our GEMV implementation focusing on register use. When manually limiting the amount of registers used by the kernel, we can see some register spilling happening, as there is not enough space for the relevant matrix/vector elements.

Concluding Thoughts

While in the last lab we saw, at a high level, the impact that the host/device memory transfer has, in this one we take a closer look at the intricacies of designing kernels and execution configurations with respect to the underlying hardware. By observing the profile data, we learned that the highest achieved occupancy is not the most important piece - other factors influence the limit of the theoretical occupancy, thus matching achieved occupancy with theoretical occupancy does not mean much if the theoretical occupancy was low to begin with. Depending on the particular strategy we were experimenting, we saw certain benefits/disadvantages with a particular execution configuration that would be oppositely effective in another implementation. Furthermore, the same execution configurations across different compute capabilities, such as between our devices, also have a significant impact on the kernel performance. By taking the strengths and weaknesses of each focused implementation, and comparing it with cuBLAS' implementation, we can say that it takes a thorough understanding and balance between the hardware's resources and particular problem design to maximize kernel execution efficiency.

Appendix

Running the Code

Each part was divided across separate .cu files, and can be built by either building the respective project file in the Visual Studio IDE *or* manually calling nvcc yourself (however, the code does require at least C++14/17 on most compilers, otherwise if it complains to you for syntax errors, you can still recompile using C++11 if you change the respective parts, it's just a legacy syntax issue); the only external library you need to link to is cublas, as it is used as a reference to the integrity of our solution. Following this section is a sample of the code we used in our implementation; please read the respective source files for further documentation on the structure of the code.

Naive/Global GEMV

```
void naive_gemv(const float *A, const float* x, float* y,
                const size_t M, const size_t N) {
    const size_t total_threads{gridDim.x*blockDim.x};
    const size_t tid{threadIdx.x+blockIdx.x*blockDim.x};
    size_t stride{M / total_threads};
    if (!stride) {
        if (tid >= M) return;
        stride = 1;
    }

    const size_t begin_index{tid*stride};
    size_t end_index{begin_index + stride};
    end_index += (tid == total_threads - 1) ?
                (M <= total_threads) ? 0 : M % total_threads) : 0;

    for (size_t i{begin_index}; i < end_index; i++) {
        float sum{0.0f};
        for (size_t j{0}; j < N; j++) sum+= A[i*M+j] *x[j];
        y[i] = sum;
    }
}
```

Shared GEMV (Tony)

This snippet will only cover the shared memory copying, as it is identical to the naive implementation following it.

```
void shared_gemv(const float *A, const float* x, float* y,
                 const size_t M, const size_t N) {
    const size_t total_threads{gridDim.x*blockDim.x};
    const size_t tid{threadIdx.x+blockIdx.x*blockDim.x};

    extern __shared__ float x_shared[];

    const size_t tpb{blockDim.x};
    const size_t tidx{threadIdx.x};
    size_t x_stride{N / tpb};
    if (!x_stride) {
        if (tidx >= N) return;
        x_stride = 1;
    }

    const size_t x_begin_index{tidx*x_stride};
    size_t x_end_index{x_begin_index + x_stride};
    x_end_index += (tidx == tpb - 1) ?
                 (N <= tpb) ? 0 : N % tpb) : 0;

    for (size_t i{x_begin_index}; i < x_end_index; i++) {
        x_shared[i] = x[i];
    }

    __syncthreads();

    // ... now execute naive gemv, but instead use x_shared !
}
```

Shared GEMV (Afrooz)

```
__global__ void mat_mul_tiled(float *vec, float *mat, float *res, const int N
){

    __shared__  float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__  float Bs[BLOCK_SIZE];

    int row = blockIdx.y * BLOCK_SIZE + threadIdx.y; //the row index of As and Bs
    int col = blockIdx.x * BLOCK_SIZE + threadIdx.x; //the column index of As and Bs
    float tmp = 0;
    int idx;


    for (int i = 0; i < gridDim.x; ++i) //initialize the ID index
    {
        idx = row * N + i * BLOCK_SIZE + threadIdx.x;
        //if N is not divisible by block width
        if (idx >= N*N)
        {
            As[threadIdx.y][threadIdx.x] = 0;
        }
        else{
            As[threadIdx.y][threadIdx.x] = mat[idx];
        }

        idx = col + i * BLOCK_SIZE;

        //if N is not divisible by block width
        if(idx >= N)
        {
            Bs[threadIdx.x] = 0;
        }
        else{
            Bs[threadIdx.x] = vec[idx];
        }

        //Matrix and vectors should be loaded completely before any further process
        __syncthreads();

        //multiply sub matrices
        for (int k = 0; k < BLOCK_SIZE; k++)
        {
            tmp += As[threadIdx.y][k] * Bs[k];

        }

        __syncthreads();
    }

    //write result back to global memory
    if(row < N && col < N)
    {
        res[row] = tmp;
    }
}
```

References

[1] M. Harris, “Using Shared Memory in CUDA C/C++,” Jan. 2013. [Online]. Available: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

[2] R. Hochberg, “Matrix multiplication with cuda — a basic introduction to the cuda programming model,” Aug. 2012. [Online]. Available: http://www.shodor.org/media/content//petascale/materials/UPModules/matrixMultiplication/moduleDocument.pdf

[3] E. Parsons, “CSS535 GPU PartII v3,” Jan. 2021.

[4] M. Harris, “How to Access Global Memory Efficiently in CUDA C/C++ Kernels,” Jan. 2013. [Online]. Available: https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/

[5] E. Parsons, “GPU Part1,” Jan. 2021.