Sampled Dense-Dense Matrix Multiplication (SDDMM)

Yevhen Khavrona, Dumeni Manatschal, Konstantinos Stavratis, Woojin Ban, Dominic Wüst

18.12.2023

Overview

1. Setup
   - Components, testing, data generation, benchmarking, hardware specs, compiler settings
2. Algorithms
   - Naive, cuSPARSE, SM-L2
3. Experiments
   - Outline of all performed experiments
4. Data Analysis
   - Methods and plots
5. Profiling
   - Profiling of SM-L2 using NVidia NSight

Setup I: Benchmark Components and Testing

- CSR/COO/Dense
  - Our own implementations
  - Column- and Row- storage
  - Efficient *ToFile* and *FromFile* methods
- Multiple algorithm variations
- **Every** component tested with unit tests using UTEST

https://github.com/sheredom/utest.h

- Ensuring correctness: time-consuming but top priority

```
PS C:\repos\sddmm\build\x64-Release> .\test1.exe
[==========] Running 14 test cases.
[ RUN      ] Matrix.TestEquals
[       OK ] Matrix.TestEquals (46700ns)
[ RUN      ] Matrix.TestDenseMult
[       OK ] Matrix.TestDenseMult (37800ns)
[ RUN      ] Matrix.ToCSR
[       OK ] Matrix.ToCSR (75700ns)
[ RUN      ] Matrix.Hadamard
[       OK ] Matrix.Hadamard (399800ns)
[ RUN      ] Matrix.Flip
[       OK ] Matrix.Flip (11900ns)
[ RUN      ] Matrix.CSR_COO_Conversion
[       OK ] Matrix.CSR_COO_Conversion (112100ns)
[ RUN      ] Matrix.SDDMM_op
[       OK ] Matrix.SDDMM_op (288800ns)
[ RUN      ] Matrix.COO_equal
1 2 6 7
[       OK ] Matrix.COO_equal (257200ns)
[ RUN      ] Matrix.SDDMM_parallel
[       OK ] Matrix.SDDMM_parallel (160835000ns)
[ RUN      ] Matrix.Tiled_MM_Mult
[       OK ] Matrix.Tiled_MM_Mult (31600ns)
[ RUN      ] Matrix.SDDMM_tiled_op
[       OK ] Matrix.SDDMM_tiled_op (34600ns)
[ RUN      ] Matrix.COO_To_Dense
[       OK ] Matrix.COO_To_Dense (8200ns)
[ RUN      ] Matrix.Sparse_Mat_Gen
################################################################
Generate sparse row maj [100x200], sparsity: 0.100000
...Generate coords...
[100%]
[100%]
Sparse to dense
[       OK ] Matrix.Sparse_Mat_Gen (144148800ns)
[ RUN      ] Matrix.Sparse_Mat_Cuda_Gen
################################################################
Generate sparse col maj [14000x14000], sparsity: 0.999000
...Filter coords...
[100%]
...Split coords...
[100%]
Summary: generated 235197 random pairs, out of which at least
..Finished in [463.101000ms]
Sparse to dense
hello world 0.99900001 195998
[       OK ] Matrix.Sparse_Mat_Cuda_Gen (1938142500ns)
[==========] 14 test cases ran.
[  PASSED  ] 14 tests.
PS C:\repos\sddmm\build\x64-Release>
```

Setup II: Data Generation

- Three generators for three sets of problems

1. Synthetic SDDMM: *Dense* A *[NxK]*, B *[KxM], Sparse* S *[NxM]*
2. Real-world SDDMM: *Dense* A *[NxK]*, B *[KxM] fitting* existing Sparse S
3. Real-world SDDMM Companion: *Dense* A *[NxK]*, B *[KxM], Sparse* S *[NxM]*

Limitations: cuSPARSE requires sorted CSR format

Setup II: Data Generation

- Three generators for three sets of problems

1. Synthetic SDDMM: *Dense* A *[NxK],* B *[KxM], Sparse* S *[NxM]*

```
[==========] Running 1 test cases.
[ RUN      ] HugeFile.Huge_Generator
projected sizes:
x_mb  209999872
y_mb  209999872
s_mb  2102847298
total 2522847042
Generating
...dense X:  [102539 x 512], 200.271484MB
...dense Y:  [512 x 102539], 200.271484MB
...sparse S: [102539 x 102539] with sparsity 0.990000, approx 105142364.938331 nnz values, 2005.431460MB
total required size: 2405.974429MB
using K_row: 512
=====================================================================================================
Proceed? [y/n]
```

Setup II: Data Generation

- Three generators for three sets of problems

2. Real-world SDDMM: *Dense* A *[NxK],* B *[KxM] fitting* existing sparse

https://sparse.tamu.edu/Pajek?filterrific%5Bsorted_by%5D=rows_asc

Id	Name	Group	Rows	Cols	Nonzeros	Kind	Date	Download File
1513	patents_main	Pajek	240,547	240,547	560,943	Directed Weighted Graph	2001	MATLAB / Rutherford Boeing / Matrix Market
1504	IMDB	Pajek	428,440	896,308	3,782,463	Bipartite Graph	2006	MATLAB / Rutherford Boeing / Matrix Market
1514	patents	Pajek	3,774,768	3,774,768	14,970,767	Directed Graph	2001	MATLAB / Rutherford Boeing / Matrix Market

← 1 2 3 4 →

Display per page: 20

Setup II: Data Generation

- Three generators for three sets of problems

3. Real-world SDDMM Companion: *Dense* A *[NxK],* B *[KxM], Sparse* S *[NxM]*

```
C:\repos\sddmm\build\x64-Release>C:\repos\sddmm\build\x64-Release\data_gen_mat_market_companion.exe "C:/sddmm_data/data_sets/patents_companion/"
lol
C:/sddmm_data/data_sets/patents_companion/ K:256 N:3774768 M:3774768 sparsity:0.999998927116394 256
Generating
...dense X:  [3774768 x 256], 3686.296875MB
...dense Y:  [256 x 3774768], 3686.296875MB
...sparse S: [3774768 x 3774768] with sparsity 0.999999, approx 15287382.731964 nnz values, 291.583686MB
total required size: 7664.177436MB

===================================================================================================
Proceed? [y/n]
```

Setup II: Data Generation

- All used data is stored and can be used to reproduce the experiments

```
.\data_gen_mat_market.exe "C:/sddmm_data/data_sets/patents/" "C:/sddmm_data/data_sets/patents/mm_market/patents.mtx" 32
.\data_gen_mat_market.exe "C:/sddmm_data/data_sets/patents/" "C:/sddmm_data/data_sets/patents/mm_market/patents.mtx" 128
.\data_gen_mat_market.exe "C:/sddmm_data/data_sets/patents/" "C:/sddmm_data/data_sets/patents/mm_market/patents.mtx" 256

.\data_gen_mat_market.exe "C:/sddmm_data/data_sets/patents_main/" "C:/sddmm_data/data_sets/patents_main/mm_market/patents_main.mtx" 32
.\data_gen_mat_market.exe "C:/sddmm_data/data_sets/patents_main/" "C:/sddmm_data/data_sets/patents_main/mm_market/patents_main.mtx" 128
.\data_gen_mat_market.exe "C:/sddmm_data/data_sets/patents_main/" "C:/sddmm_data/data_sets/patents_main/mm_market/patents_main.mtx" 256

.\data_gen_mat_market.exe "C:/sddmm_data/data_sets/imdb/" "C:/sddmm_data/data_sets/imdb/mm_market/imdb.mtx" 32
.\data_gen_mat_market.exe "C:/sddmm_data/data_sets/imdb/" "C:/sddmm_data/data_sets/imdb/mm_market/imdb.mtx" 128
.\data_gen_mat_market.exe "C:/sddmm_data/data_sets/imdb/" "C:/sddmm_data/data_sets/imdb/mm_market/imdb.mtx" 256
```

Setup III: Benchmarking Process

- All used matrices are generated and stored as *bindat* binary files
- One file contains dense A, B and sparse S in COO format
- Experiment varying over 5 sparsities for fixed N, M, K includes 5 binmat files in the same folder
- Experiment loader loads all files in sequence, runs all algorithms and produces benchmark output files

Name	Type	Size
hadamard_S-102539x102539-0.99_X-102539x128-0_Y-128x102539-0__Wed_Dec_13_21-31-33_2023__17024994931749238.bindat	BINDAT File	2'156'101 KB
hadamard_S-102539x102539-0.995_X-102539x128-0_Y-128x102539-0__Wed_Dec_13_21-35-10_2023__17024997102399268.bindat	BINDAT File	1'129'321 KB
hadamard_S-102539x102539-0.999_X-102539x128-0_Y-128x102539-0__Wed_Dec_13_21-35-44_2023__17024997448199940.bindat	BINDAT File	307'893 KB
hadamard_S-102539x102539-0.9995_X-102539x128-0_Y-128x102539-0__Wed_Dec_13_21-36-01_2023__17024997616012556.bindat	BINDAT File	205'223 KB
hadamard_S-102539x102539-0.9999_X-102539x128-0_Y-128x102539-0__Wed_Dec_13_21-36-06_2023__17024997660200808.bindat	BINDAT File	123'079 KB

Setup IV: Hardware Specs

- Benchmarking Hardware
  - AMD Ryzen 7 5800X, 32GB RAM
  - Nvidia GeForce RTX 3080, 10GB RAM, 6MB L2-cache, 49KB shared memory
- Profiling Hardware
  - Intel Core i9-12900K, 64GB RAM
  - Nvidia GeForce RTX 3080Ti, 12GB RAM, 6MB L2-cache, 98KB shared memory
- Operating System
  - Windows 11
- C++ Compiler (run cl.exe without args)
  - Microsoft (R) C/C++ Optimizing Compiler Version 19.38.33133 for x64
  - /O2 /Ob2 /DNDEBUG /std:c++20
  - Visual Studio Community Edition 2022
- Cuda Compiler (output of nvcc --version)
  - Built on Fri_Nov__3_17:51:05_Pacific_Daylight_Time_2023
  - Cuda compilation tools, release 12.3, V12.3.103
  - Build cuda_12.3.r12.3/compiler.33492891_0

Algorithms I: Overview

- 3 algorithms: naive, SM-L2, cuSDDMM
- Our assumptions/decisions
    - A in row store, B in col store but not transposed
    - Preprocessing time for SM-L2 algo not measured
    - Time is measured only around the Cuda kernel using C++ chrono::high_resolution_clock from start of kernel to directly after cudaDeviceSynchronize();

Algorithms II: Naive

- Straightforward
- Direct translation of the CPU code to CUDA
- No tiling
- No efficient data handling

```
// Each CUDA thread is responsible for computing one entry
// of the output sparse matrix `out_d`.

int index = threadIdx.x;
int stride = blockDim.x;
int blockNum = blockIdx.x;

SDDMM::Types::vec_size_t access_ind = index + blockNum*stride;
SDDMM::Types::expmt_t val = A_sparse_values_d[access_ind];
SDDMM::Types::vec_size_t row = A_sparse_rows_d[access_ind];
SDDMM::Types::vec_size_t col = A_sparse_cols_d[access_ind];

SDDMM::Types::expmt_t inner_product = 0;

// the ind index has to be tiled later
// X == X_n x X_m
// Y == Y_n x Y_m
// ==> X_m == Y_n (if Y_n existed)
for(SDDMM::Types::vec_size_t ind=0; ind < X_m; ++ind){
    inner_product += X_dense_d[row * X_m + ind]*Y_dense_d[col * Y_n + ind];
}

out_values_d[access_ind] = val*inner_product;
out_row_d[access_ind] = row;
out_col_d[access_ind] = col;
```

Algorithms III: cuSPARSE

- NVIDIA's closed source SDDMM implementation

https://docs.nvidia.com/cuda/cusparse/

This function performs the multiplication of `matA` and `matB`, followed by an element-wise multiplication with the sparsity pattern of `matC`. Formally, it performs the following operation:

$$\boxed{\mathbf{C} = \alpha(op(\mathbf{A}) \cdot op(\mathbf{B})) \circ spy(\mathbf{C}) + \beta\mathbf{C}}$$

where

- `op(A)` is a dense matrix of size $m \times k$
- `op(B)` is a dense matrix of size $k \times n$
- `C` is a sparse matrix of size $m \times n$
- α and β are scalars
- $\circ$ denotes the Hadamard (entry-wise) matrix product, and $spy\left(\mathbf{C}\right)$ is the sparsity pattern matrix of `C` defined as:

$$spy(\mathbf{C})_{ij} = \begin{cases} 0 & \text{if } \mathbf{C}_{ij} = 0 \\ 1 & \text{otherwise} \end{cases}$$

Algorithms III: cuSPARSE

- NVIDIA's closed source SDDMM implementation

https://docs.nvidia.com/cuda/cusparse/

Performance notes: `cuspaseSDDMM()` for `CUSPARSE_FORMAT_CSR` provides the best performance when `matA` and `matB` satisfy:

- `matA`:
  - `matA` is in row-major order and `opA` is `CUSPARSE_OPERATION_NON_TRANSPOSE`, or
  - `matA` is in col-major order and `opA` is not `CUSPARSE_OPERATION_NON_TRANSPOSE`
- `matB`:
  - `matB` is in col-major order and `opB` is `CUSPARSE_OPERATION_NON_TRANSPOSE`, or
  - `matB` is in row-major order and `opB` is not `CUSPARSE_OPERATION_NON_TRANSPOSE`

Algorithms IV: SM-L2

- SM-L2: fast SDDMM for sample matrices with sparsity > 95%
- I. Nisa, A. Sukumaran-Rajam, S. E. Kurt, C. Hong and P. Sadayappan, **"Sampled Dense Matrix Multiplication for High-Performance Machine Learning"** 2018 IEEE 25th International Conference on High Performance Computing (HiPC), Bengaluru, India, 2018, pp. 32-41, doi: 10.1109/HiPC.2018.00013.
- Key points:
  - Loading only necessary ("active") rows of A into fast on-chip memory (shared memory)
  - Reuse of elements loaded into shared memory and L2 cache through tiling
- Main technics: (3-way) tiling, vectorisation, loop unrolling, virtual warps, warp shuffling, autotuning
- Kernel launch configuration: 1024 threads, roughly ½ of SM's shared memory reserved, 1 CUDA stream (sequential kernel execution)

Experiments I: Overview

- Varying sparsity with same dimensions N, M, K
  - All sparse input matrices have uniform density
- Varying K with same N, M and sparsity
  - One series using existing sparse matrix with **non-uniform** distribution and produced dense matrices
  - One companion series with all produces inputs with **uniform** distribution

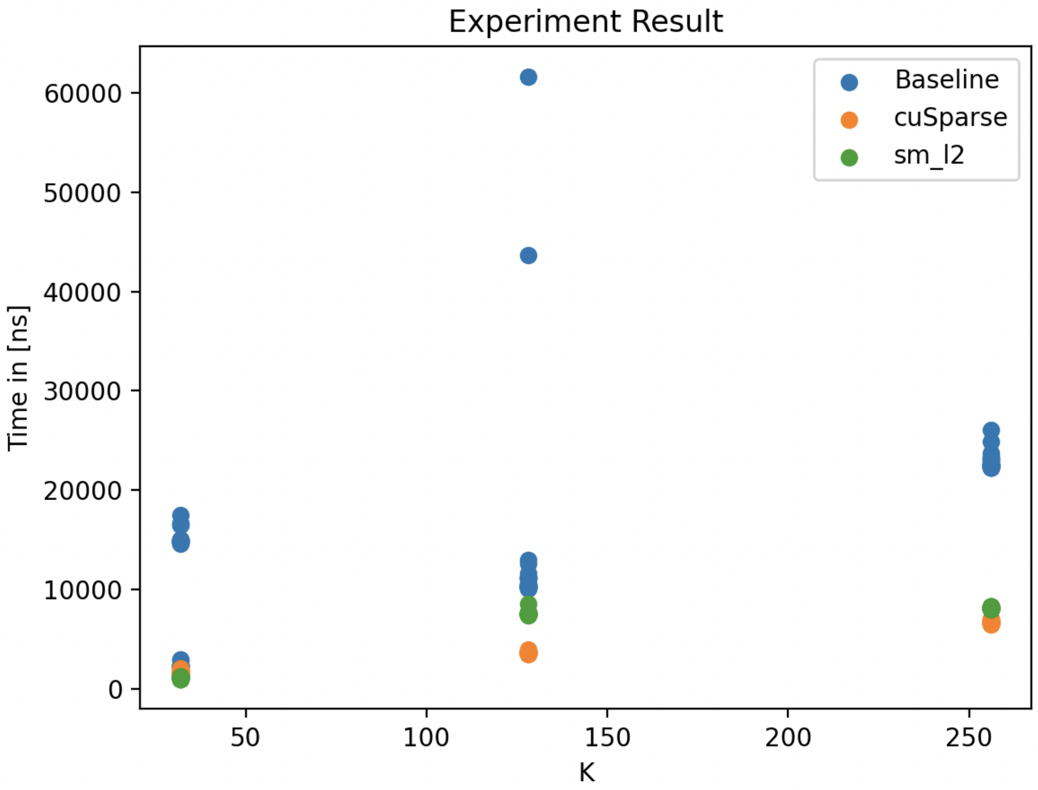
- Reproducibility
  - 40GB of stored matrix data
  - One bindat file per algorithm run
  - Code on GitHub

```
.\GPU_SDDMMBenchmarks.exe 1
.\GPU_SDDMMBenchmarks.exe 2
.\GPU_SDDMMBenchmarks.exe 3
.\GPU_SDDMMBenchmarks.exe 4
.\GPU_SDDMMBenchmarks.exe 5
.\GPU_SDDMMBenchmarks.exe 6
.\GPU_SDDMMBenchmarks.exe 7
.\GPU_SDDMMBenchmarks.exe 8
.\GPU_SDDMMBenchmarks.exe 9
.\GPU_SDDMMBenchmarks.exe 10
.\GPU_SDDMMBenchmarks.exe 11
.\GPU_SDDMMBenchmarks.exe 12
```

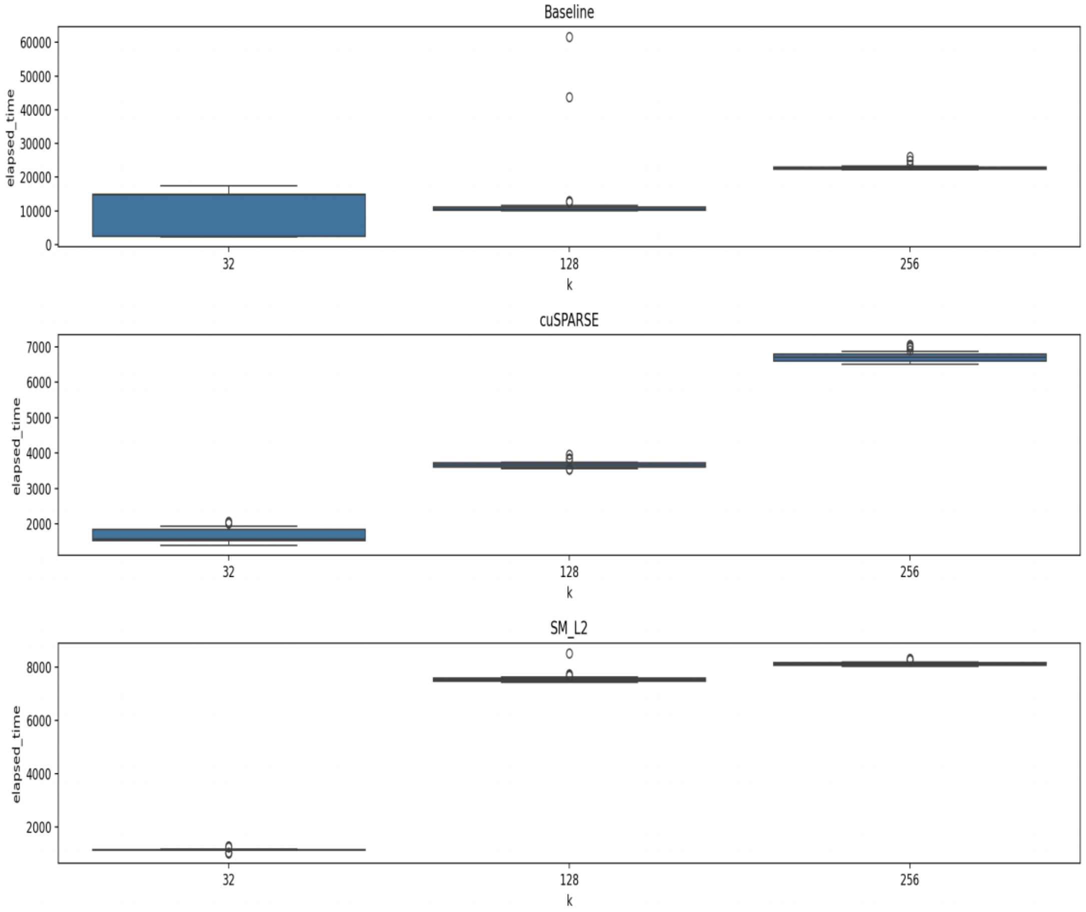
```
Test Nr: 1
############################################################################
============================================================================
Start measurements sparsity_large_K32: Compare matrices with K=32 for varyi
****************************************************************************
...loading data...
...stats:
......N:         102539
......M:         102539
......K:         32
......sparsity: 0.990000
...run experiment iterations...
Experiment: Baseline
..(1/3)..
[105 / 105]
Experiment: cuSPARSE
..(2/3)..
[105 / 105]
Experiment: sm_l2
..(3/3)..
...preparations...
   ...finished
[105 / 105]
Saving experiment data
```

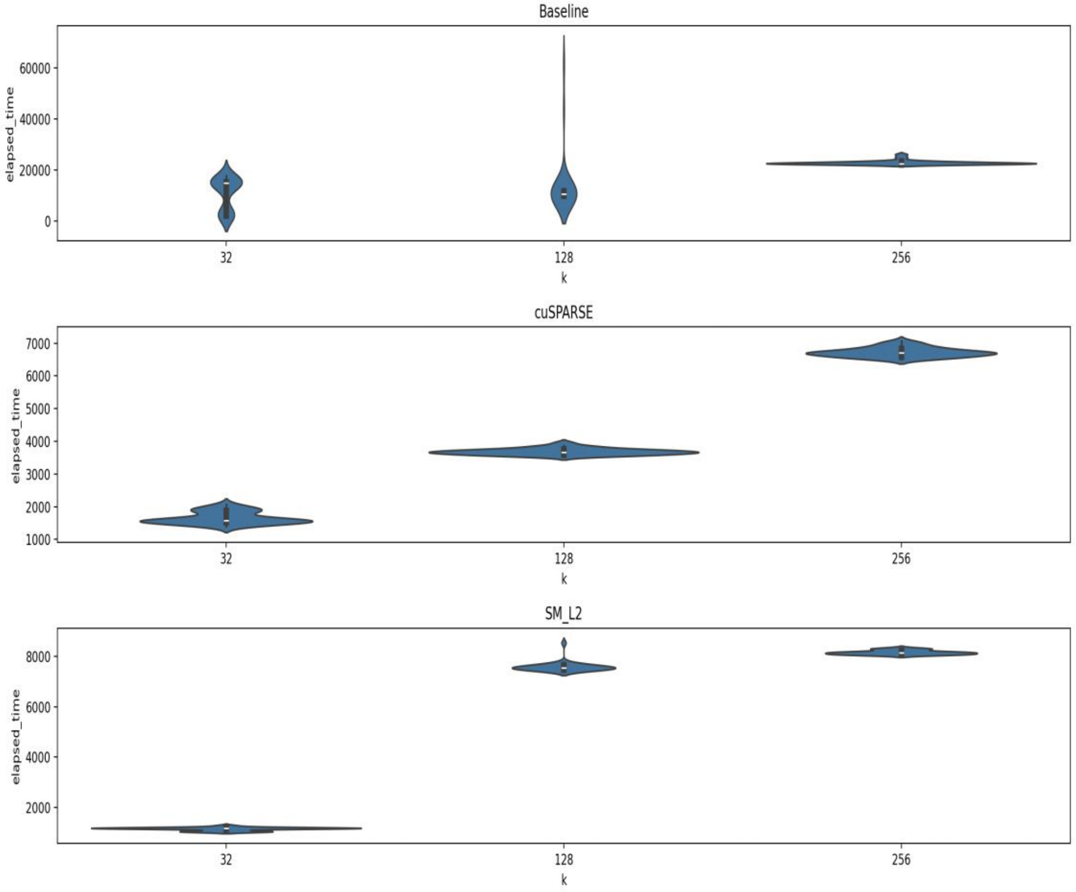
Experiments II: IMDB

- IMDB
  https://sparse.tamu.edu/Pajek?filterrific%5Bsorted_by%5D=rows_asc (p. 4, id: 1504)
- Rows = 428,400, cols = 896,308, #nnz = 3,782,463
- We measured the runtime with sparsity = 0.99999, K = 32, 128, 256 (#iterations = 30, #warm-up iterations = 5)



Experiments II: IMDB



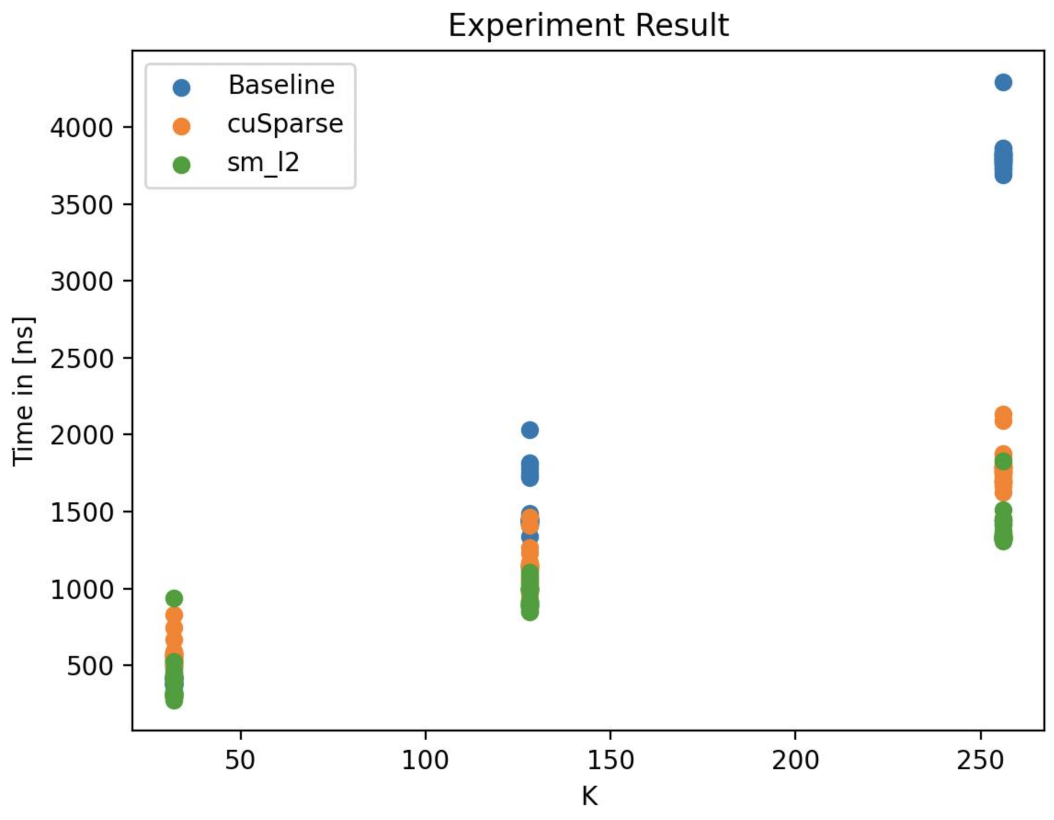


Experiments III: patents_main

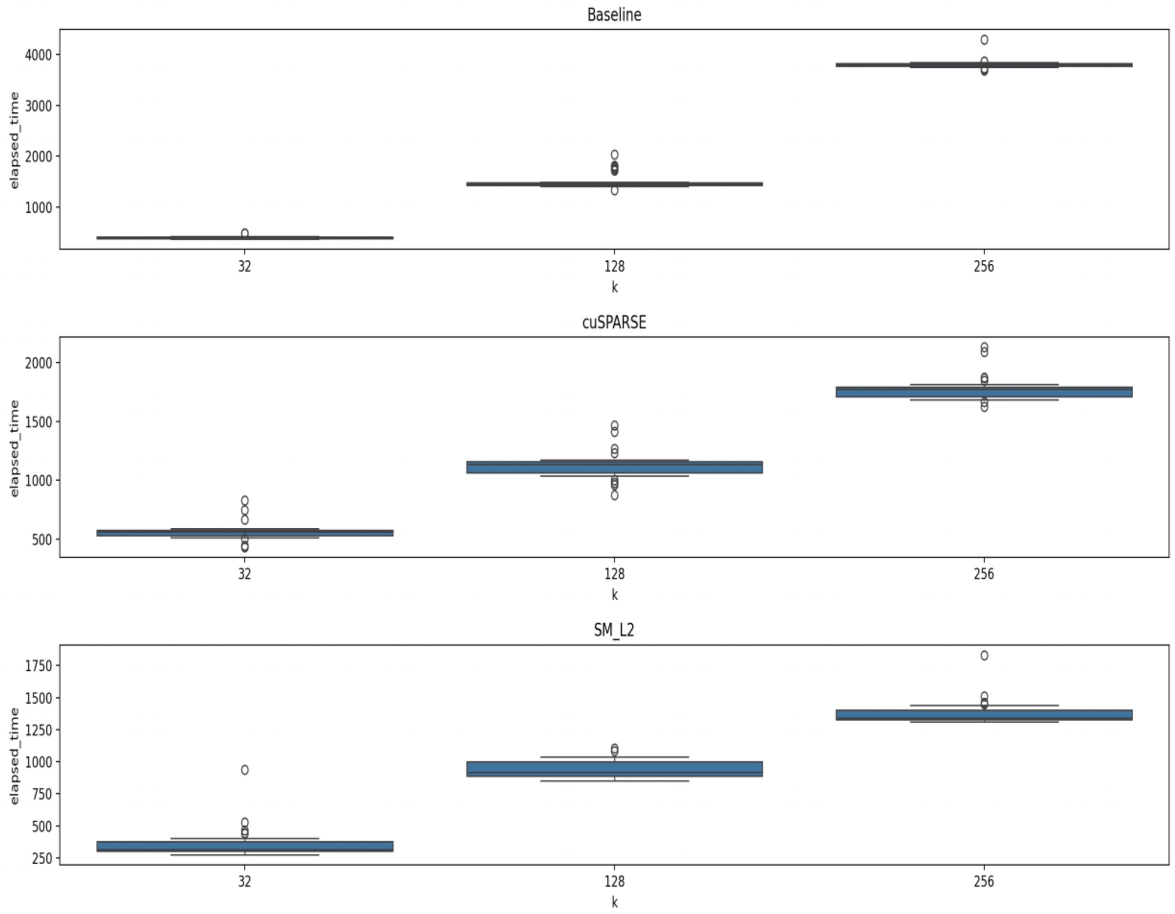
- Patents_main

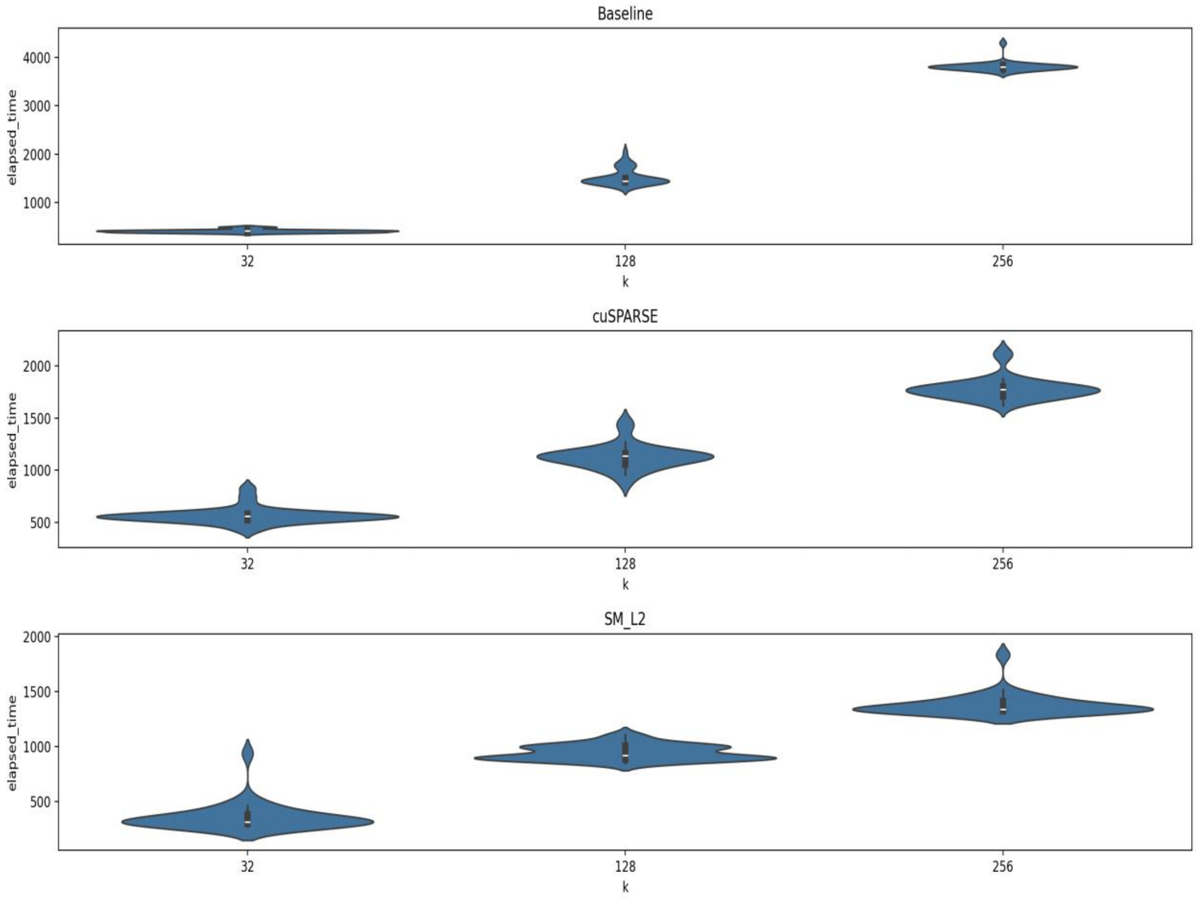
  https://sparse.tamu.edu/Pajek?filterrific%5Bsorted_by%5D=rows_asc (p. 4, id: 1513)

- Rows = 240,547, cols = 240,547, #nnz = 560,943
- Measurement with sparsity = 0.99999, K = 32, 128, 256 (#iterations = 30, #warm-up iterations = 5)



Experiments III: patents_main

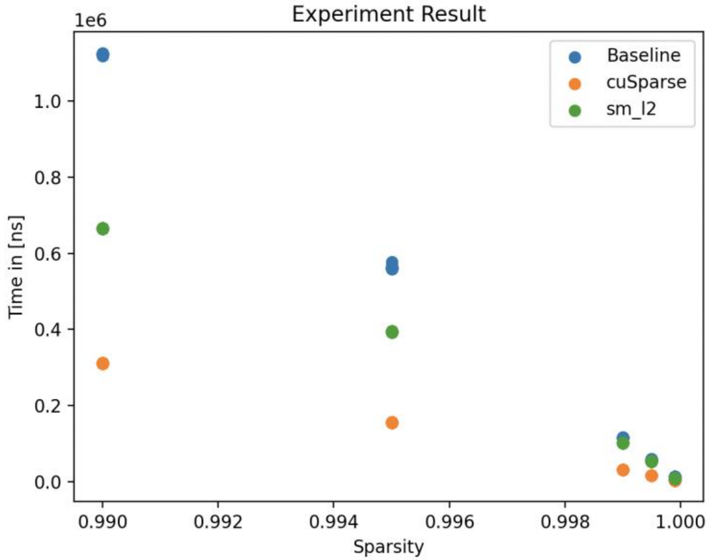


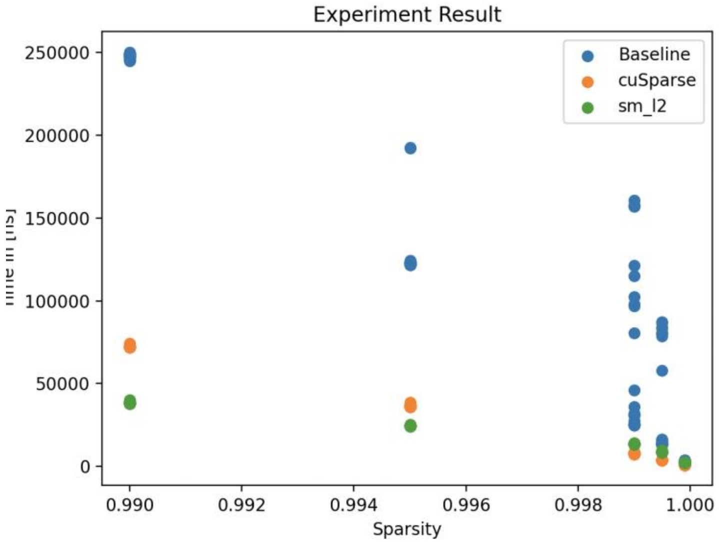


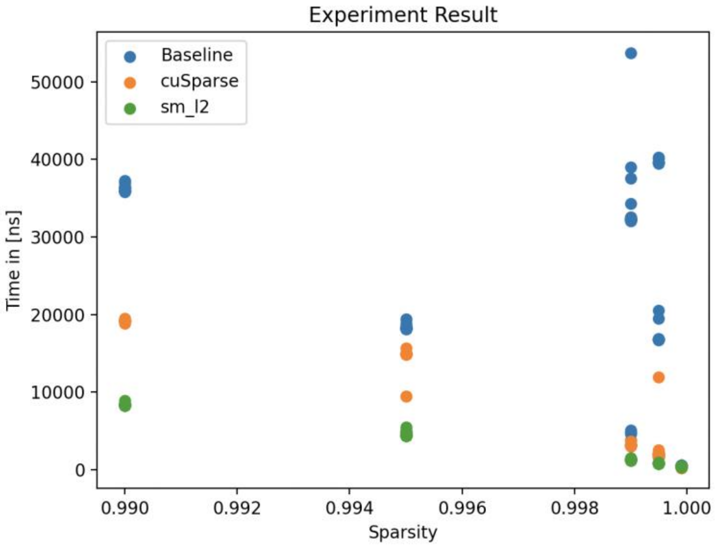
Experiments IV: sparsity_large

- N = 102539, M = 102539, K = 32, 128, 512
- Large in terms of the matrix size
- Varying sparsity 0.99, 0.995, 0.999, 0.9995, 0.9999
- #iteration = 30, #warm_up iteration = 5

Experiments IV: sparsity_large



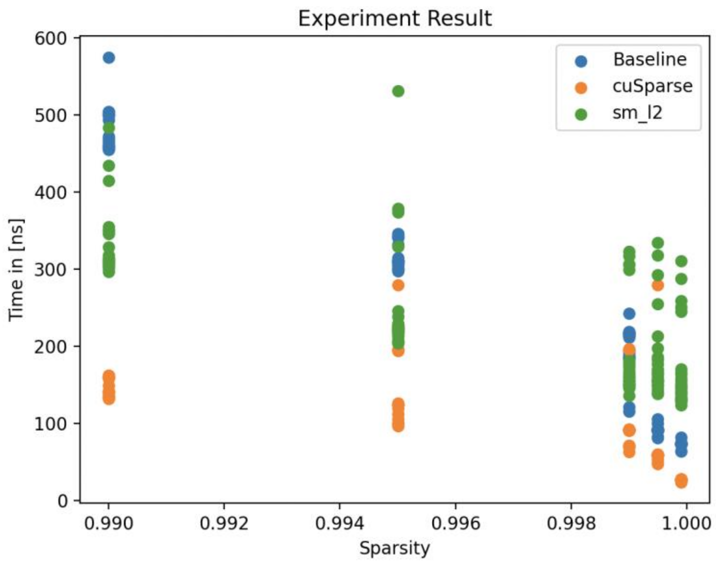


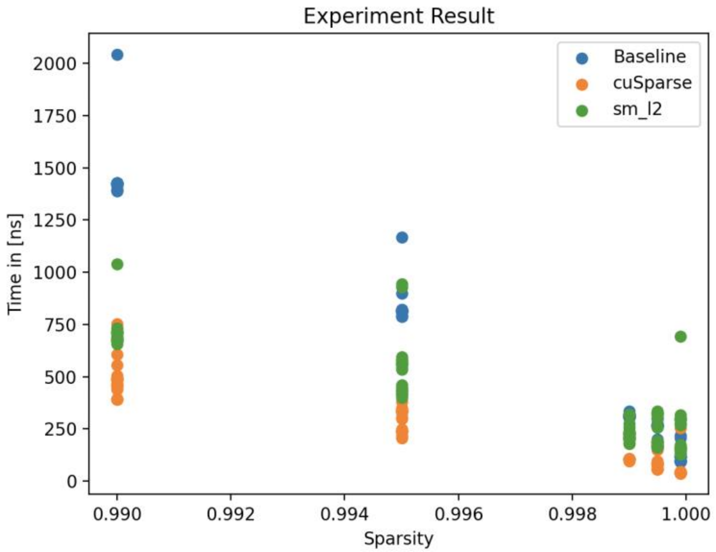


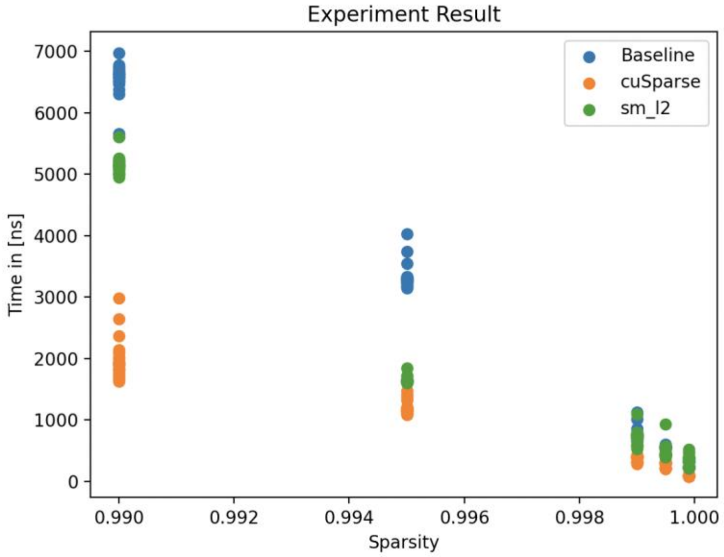
Experiments V: sparsity_small

- N = 10253, M = 10253, K = 32, 128, 256
- Small in terms of the matrix size
- Varying sparsity 0.99, 0.995, 0.999, 0.9995, 0.9999

Experiments V: sparsity_small







Experiments VI: Conclusion

- We measured baseline, cuSparse and sm_l2 on various datasets such as IMDB, IMDB_companion, patents, patents_companion, patents_main, patents_main_companion with different K and fixed matrix size with varying sparsity
- For K=32, sm_l2 is superior to baseline and cuSparse (in all datasets and with varying sparsity except for sparsity with small matrix size)
- From K=128, cuSparse is superior to sm_l2 and baseline

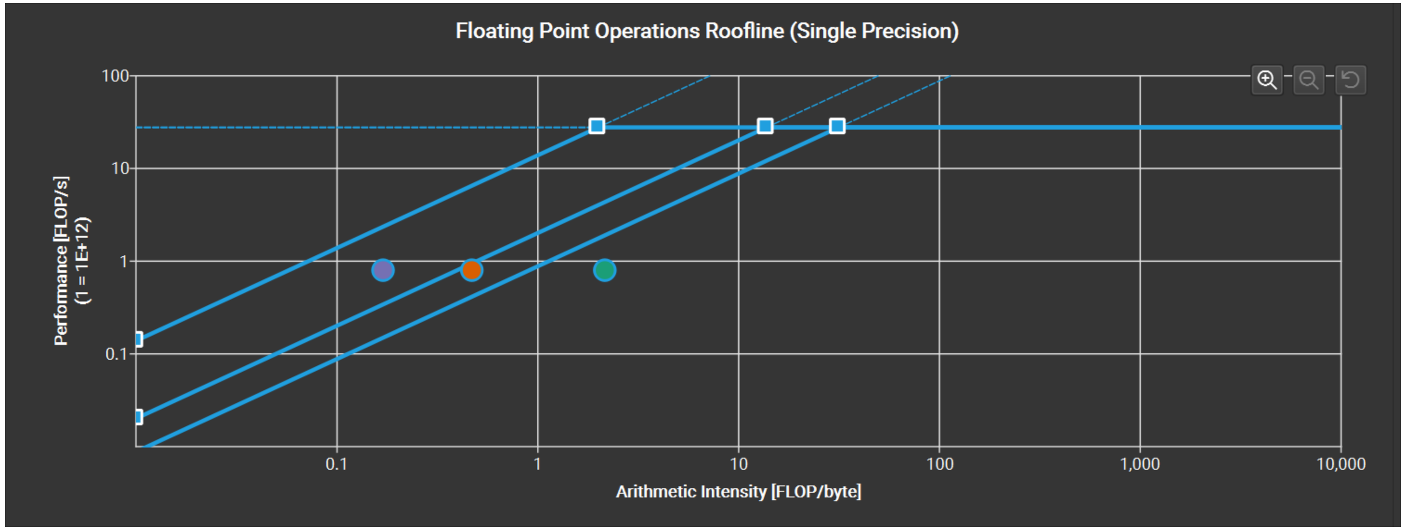
Profiling SM-L2 I: Overview

- NVIDIA Nsight Compute
- Input: N=M=100K, sparsity = 0.99
- Results in 13 kernel invocations, each launching 268 threadblocks

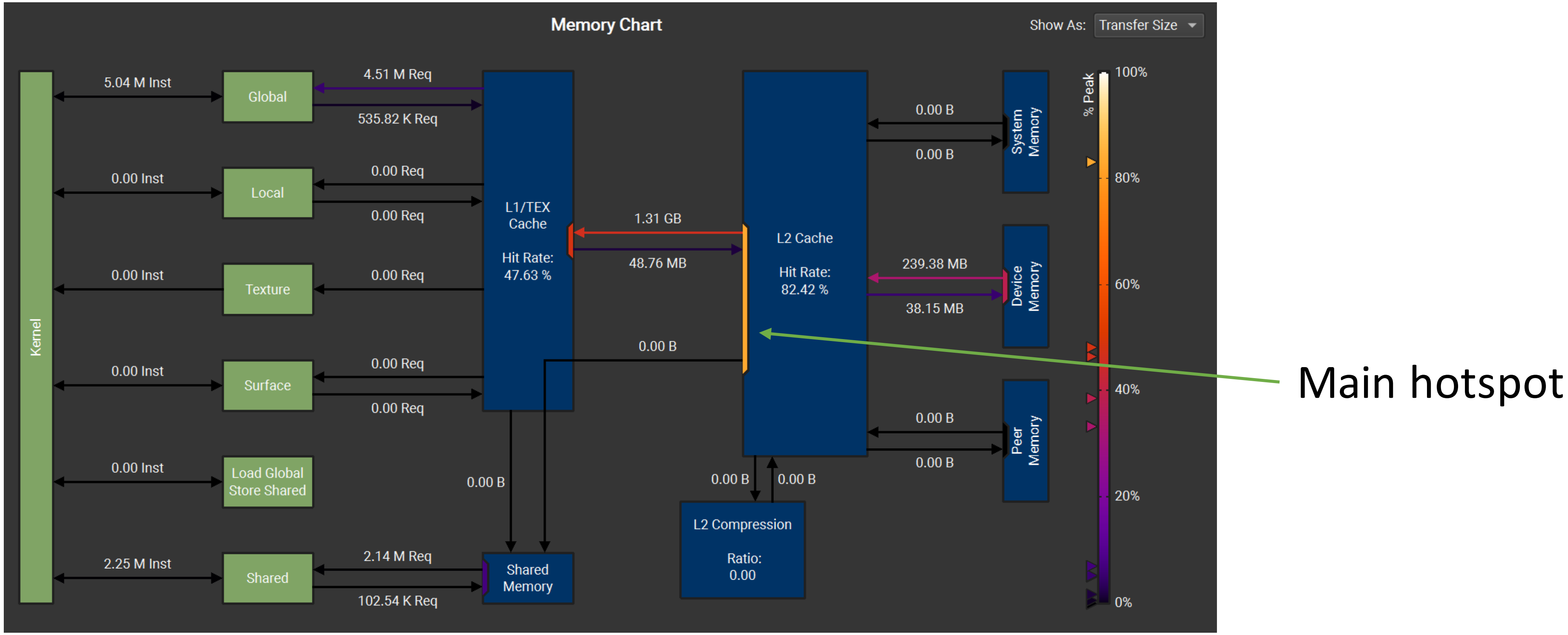
Compute Throughput	Memory Throughput
21.21	83.10
21.15	89.01
21.09	82.85
21.11	82.88
21.06	82.52
21.12	82.73
21.23	83.00
21.12	82.94
21.11	82.70
21.10	82.79
21.01	82.38
20.93	82.02
22.46	68.55

Profiling SM-L2 II: Observations

- Memory bound, L2 (red dot) saturated, low "compute" utilisation
- Uncoalesced global memory accesses
- Occupancy limited by the number of required registers
- High on-chip memory usage (register file and shared memory)



Profiling SM-L2 III: Memory



```
__global__ void thankYouKernel()
{
    printf("Thank you for your attention!\\n");
}
```