GPU Sparse Matrix Multiplication and Power Profiling

Overview

This program performs matrix multiplication (with a set sparsity) using the NVIDIA CUDA framework and cuSPARSE library on the GPU. It measures the execution time for the multiplication, and logs the GPU's power consumption throughout the process, providing insights into the performance and energy efficiency.

This project was inspired by the high power demand that has grown alongside AI/ML model training. I wanted to get an insight and benchmark how much power a GPU would be consuming for a given operation, and profile the curve that might form. This project was also my first experience installing and using the NVidia CUDA Toolkit, which was insightful on its own.

In terms of code structure, all written code is contained in "matrix_multiplication.cu" and relies on the libraries and header files given in the NVidia CUDA Toolkit.

Features

There are several key elements to this program that are outlined in the sections below. These elements describe the features, goals, and data collection methods of this program.

Sparse Matrix Multiplication

Matrix multiplication is implemented for sparse matrices in Compressed Sparse Row (CSR) format. This utilizes NVidia's cuSPARSE library for optimized sparse operations, and uses a user-defined density (0-1 float) to generate a random sparse matrix. Memory is allocated dynamically using the "std::vector" function.

GPU Power Profiling

Profiling Power consumption was a key part of this project. GPU power (measured in Watts) is continuously monitored using "nvidia-smi", and is logged every second to "power_log.txt" for analysis, until completion. After every operation, the results in "power_log.txt" are averaged to develop an average power value for a given matrix multiplication operation.

Execution Time Measurement

The program accurately times the matrix multiplication computation the GPU using "std::chrono" and outputs it to the console upon completion.

Customizable Parameters

Square matrix dimensions (N) and density (sparsity) control the type of matrices being used to benchmark performance. It is important to get a wide range of matrix sizing and density, which will be shown in the results.

Usage

Compiling and running this program, including the NVidia CUDA Toolkit was tricky to set up, especially concerning PATH variables in the system control panel (on Windows OS). This program was run on my computer, containing an RTX 3070-Ti GPU, and built/compiled within Visual Studio 2022.

Prerequisites

This program will require an NVidia GPU with CUDA support, the NVidia CUDA Toolkit installed (including cuSPARSE), and “nvidia-smi” available in the system PATH.

The NVidia CUDA Toolkit (which includes required libraries and header files) can be found at https://developer.nvidia.com/cuda-toolkit

In my case, I am using an NVidia RT3070-Ti GPU (idle power around 86 W), paired with an Intel-12700k processor.

Compiling & Running

Compiling this program will require saving it as “matrix_multiplication.cu” and then compiling it using “nvcc.” For example, in bash you could write “nvcc matrix_multiplication.cu -o matrix_multiplication -lcusparse”, including the lcusparse flag to handle the cuSPARSE libraries. Alternatively, you could setup Visual Studio 2022 to build this program for you (build from the top panel), and then run without debugging. This will require setting up additional compile flags in the project manager.

The project can then be run via ./matrix_mutliplication, or by running without debugging in VS 2022.

Customizing Parameters

Matrix size (N) and density can be modified in the main() function.

```
const int N = 16384;    // Matrix size
const float density = 0.1f; // Sparsity level (10% non-zero elements)
```

Results

All results and data points can be seen with this spreadsheet, found at https://docs.google.com/spreadsheets/d/14KxecxRB6gYDuMkN5Rx_uXCn5TI7JLpx0YgtoI7FEGA/edit?usp=sharing

For each density (0.1, 0.5, 0.9) and matrix size (N=1024, 2048, 4096, 8192, 16384), a test was run to collect execution time (ms) and average GPU Power (W). Using these values, we can compute milliwatt-hours (mWh), and Power Efficiency (GFLOPs/W). Milliwatt-hours can be found by multiplying the time and power.

Power Efficiency is found by taking the number of non-zero elements,

$$nnz = density \times N^2$$

And multiplying it by 2N to get FLOPS,

$$FLOPs = 2 \times N \times nnz$$

And then computing power efficiency,

$$Power\ Efficiency = \frac{FLOPs}{Execution\ Time \times Average\ Power}$$

10% non-zero elements

For the 10% density operations, the results for each square matrix level (from 1024x1024 to 16384x16348 can be seen below.

Table 1: 10% Density Data

Density 0.1	// 10% non-zero elements				
Matrix Size	Execution Time (ms)	Average GPU Power (W)	Power Efficiency (GFLOPs/W)	Watt-hours	Milliwatt-Hours
1024x1024	6.6813	108.59	0.295991	0.00020153	0.20153
2048x2048	7.7294	110.5	2.011462	0.00023725	0.23725
4096x4096	21.2042	107.76	6.014927	0.0006347	0.6347
8192x8192	142.577	105.745	7.292736	0.004188	4.188
16384x16384	1867.26	118.6066667	3.971695	0.06152	61.52

50% non-zero elements

For the 50% density operations, the results for each square matrix level (from 1024x1024 to 16384x16348 can be seen below.

Table 2: 50% Density Data

Density 0.5	// 50% non-zero elements				
Matrix Size	Execution Time (ms)	Average GPU Power (W)	Power Efficiency (GFLOPs/W)	Watt-hours	Milliwatt-Hours
1024x1024	7.4478	105.91	1.361241	0.0002191	0.2191
2048x2048	15.2025	108.7	5.198108	0.000459	0.459
4096x4096	90.015	111.395	6.853292	0.0027853	2.7853
8192x8192	697.239	109.744	7.184679	0.021255	21.255
16384x16384	9427.26	143.8496	3.24314	0.3767	376.7

90% non-zero elements

For the 90% density operations, the results for each square matrix level (from 1024x1024 to 16384x16348 can be seen below.

Table 3: 90% Density Data

Density 0.9	// 90% non-zero elements				
Matrix Size	Execution Time (ms)	Average GPU Power (W)	Power Efficiency (GFLOPs/W)	Watt-hours	Milliwatt-Hours
1024x1024	7.2508	106.23	2.509223	0.00021396	0.21396
2048x2048	23.3848	105.2	6.285111	0.0006834	0.6834
4096x4096	156.147	105.285	7.524059	0.004567	4.567
8192x8192	1247.58	112.0942857	7.076043	0.03885	38.85
16384x16384	18722.1	154.0353659	2.745095	0.801	801

Density Comparison

All graphs from the previous three data tables are included in the graphs folder. To showcase the overall general trends, the three graphs have been overlaid on top of each other.

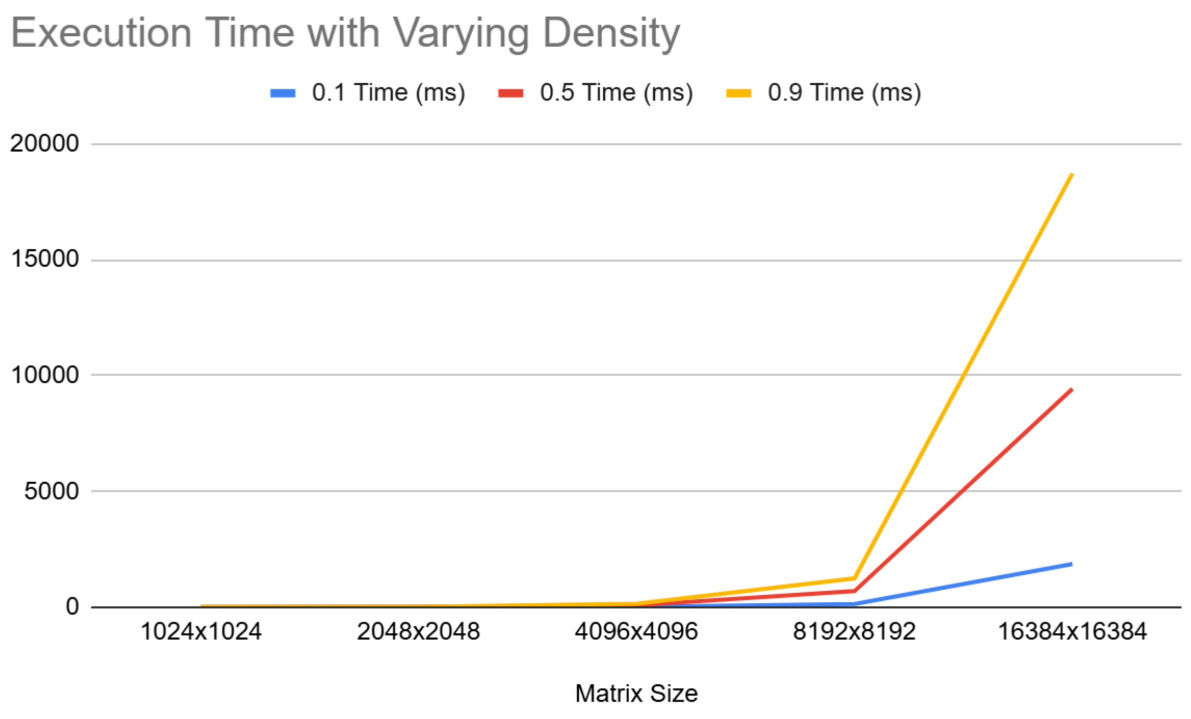

Figure 1: Execution time with varying density and dimension

For execution time, we can see that as the matrix size increases, the time to complete increases. This trend also grows faster with an increased matrix density, as expected. This is due to an increased number of computations needed to complete each operation.

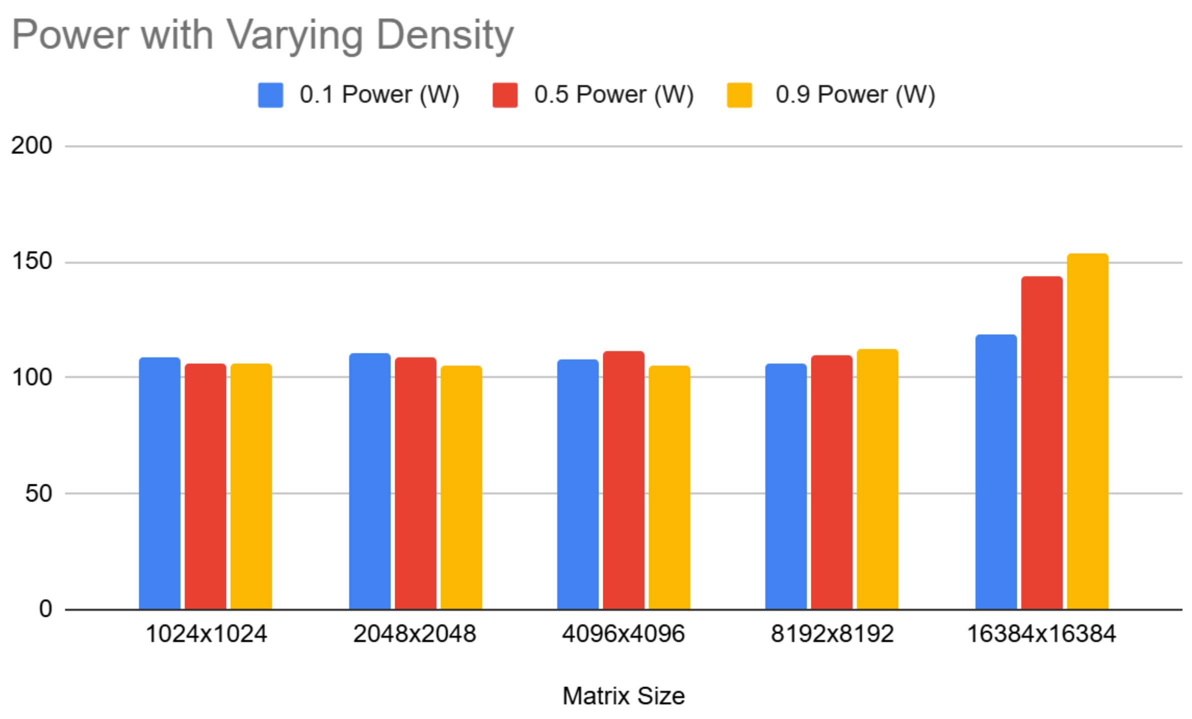

Figure 2: Power with varying density and dimension

The power with varying density shows that the power draw does not increase much from the 1024-8192 matrices. They are all generally around the same power (110 watts). The increased power demand from the 16,384 matrices is enough to show an increased spike in power, and each power level increases with density as well.

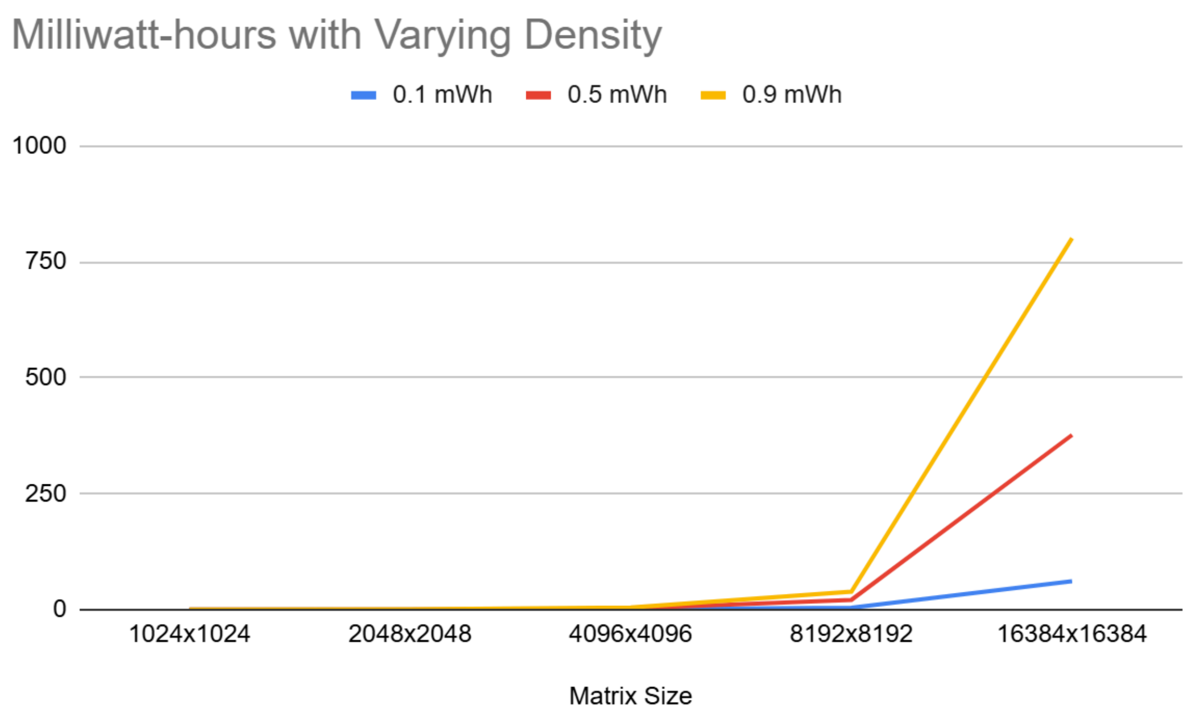

Figure 3: mWh with varying density and dimension

Following the previous two graphs, the Milliwatt-hours with varying density shows a similar trend where there isn't much change for the smaller matrices, but the larger matrices show a trend where the dense matrices demand more power. This is again due to complexity in calculation and amount of data being moved.

Power Draw Curve

I noticed that for the smaller matrices, the power didn't have a significant change. This is likely due to the operation being relatively short, and only one or a few wattage readings were taken. For the large 16,384 matrices, the average completion time was much higher, and thus we got many wattage data points (41 in total). Plotting these data points shows an interesting phenomenon.

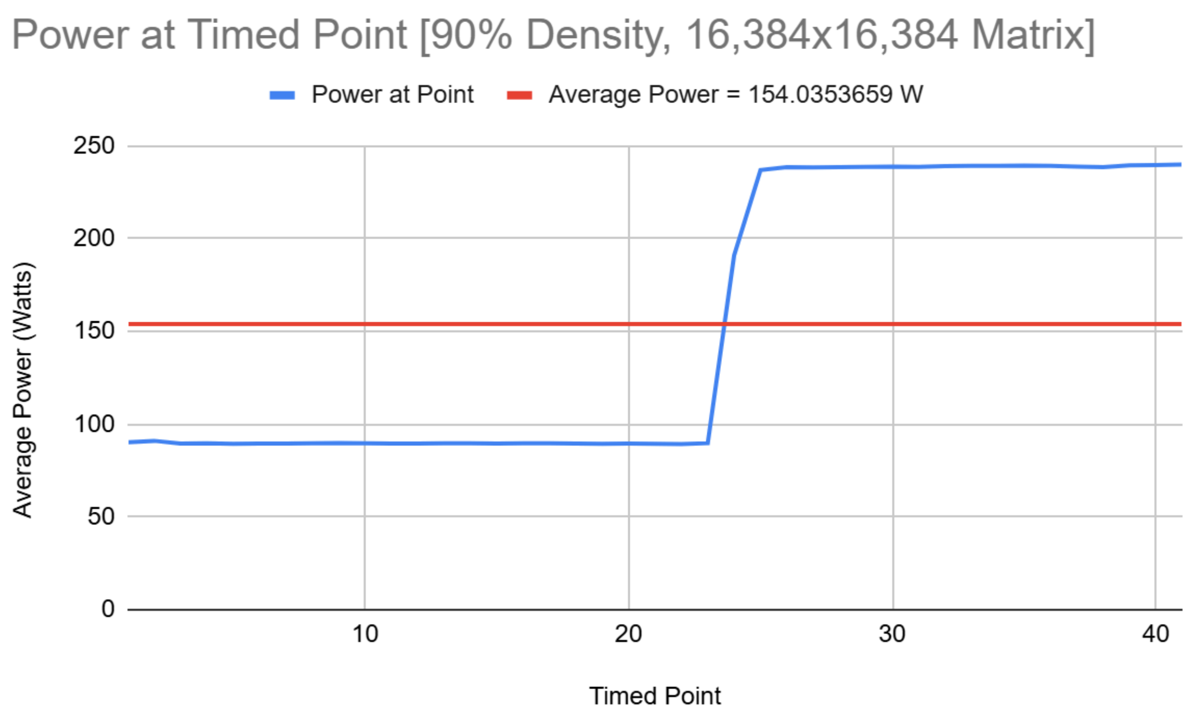

Figure 4: Power progression for 90% density, 16,384 matrix

I noticed that for the smaller matrices, the power didn't have a significant change. This is likely due to the operation being relatively short, and only one or a few wattage readings were taken. For the large 16,384 matrices, the average completion time was much higher, and thus we got many wattage data points (41 in total). Plotting these data points shows an interesting phenomenon. As about halfway through the operation (data point 24/41), the power greatly spikes from around 90 watts to 240 watts. These larger matrices may have demanded enough compute to warrant a greater power draw.

Final Thoughts

This project was a great way to interact with the CUDA toolkit and learn about GPU power draw. There are some potential flaws in this method, such as GPU power being used in background operations/running my monitor. However, enough data was collected to show a general trend. Larger matrices require more power draw, and at the same dimension, more dense matrices will also demand more power to compute.