15-418 Project Final Report

CUDA Program Debugger/Optimizer

Website: https://kflorendo.github.io/15418-final-project/

Application: https://github.com/kflorendo/cuda-debugger-optimizer

Scripts: https://github.com/kflorendo/cuda-debugger-optimizer-scripts

SUMMARY

Our project is a CUDA program debugger/optimizer. The debugger helps analyze the results of incorrect code and determine potential areas of error. For example, it will be able to output the resulting values for each thread, and help resolve issues such as overwriting shared memory. This tool would help users debug their CUDA code more efficiently, without needing to manually add streams of print statements. Further, the optimizer helps users select the optimal parameters of their program, such as CUDA grid configurations (ex. block size, grid size), to determine what will achieve the best performance. The optimizer also calculates time and memory metrics to identify bottlenecks and understand overall program performance. Visuals such as graphs will be generated based on the information found above as well.

BACKGROUND

We implemented a CUDA program debugging/optimizing tool to help students and developers understand and visualize the behavior of their program.

Primary Feature 1: Debugging

The output of each individual thread will be written to a file so they can be analyzed later. Additionally, the program will check for overwriting by tracking the number of times a variable has been modified before it is accessed. The index of the thread, as well as the new variable value, will also be recorded. For example, a thread can return an array of RGB values, but the rendered image values do not match. Asking the debugger to track the variable would print out its value each time it was accessed, as well as the thread that made each access. The thread with value corresponding to what is actually displayed in the mismatched areas would be the thread doing the overwriting. For each memory address recorded, the threads that access them are also sorted in chronological order. Our scripts support 1D, 2D, or 3D CUDA grids.

Primary Feature 2: Optimizing

We will write scripts that run the program multiple times with different CUDA grid configurations and output the performance of each to help the programmer decide on the optimal configuration. This can be done using bash scripts that modify the program files. First, users specify their block/grid dimensions through macros (i.e. `#define THREADS_PER_BLOCK 1024`). Then, the user will enter the command they wish to run (i.e. `./render rgb`). Our script will then look for these macros and modify them to test out different parameters. For different configurations of these macros, we'll run the command the user entered, and return a file as well as a graph of the time it took the program to run with respect to the parameter value that was changed. Our program will use profiling tools to time the code and keep track of the amount of memory used. Then, that data will be outputted and also expressed visually as a graph, so it would be easier for the programmer to find time and memory bottlenecks.

Extras:

Inspired by our course homeworks, we decided to implement some extra features. In order to check for memory leaks, there is an option of placing CUDA-MEMCHECK in various places in the code. A thread breakpoint feature was also added, which returns a text file containing a sorted list of thread indices that have reached that point in the code. This helps with checking previous conditional statements and ensuring that all available threads are used in the parallel calculations. Lastly, we included a speedup calculation feature, which finds the speedup between two given files by using timers to find their individual total runtime.

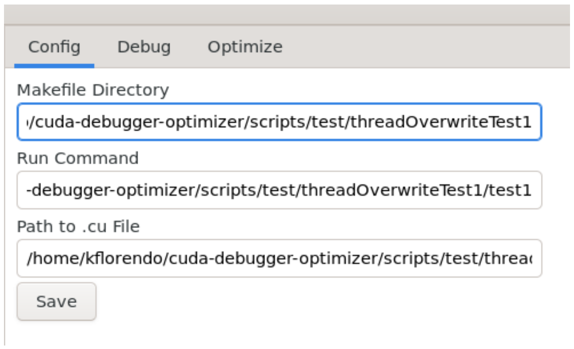
APPROACH

Our implementation uses shell scripts that target the CUDA interface. The features our program includes include the adjustment of CUDA parallel parameters such as the number of threads per block.

GUI Overview

The GUI was developed using GTK (gtkmm), a library for creating graphical user interfaces, and C++. The layout for the GUI was designed and created using glade. The layout is formatted as an XML hierarchy of the application, its components (like buttons and tree displays), and various properties (like position and spacing). The GUI program reads in the glade file, renders and obtains references to the different components, and specifies the behavior when the user interacts with the components, like click and text change signals. Custom functions were written to display data in different formats (tree and list views, tables, etc.) and simplify the process of taking in user input (for instance, the user can directly interact with the code file to set breakpoints, line numbers, and specify variable names).

The GUI consists of 3 pages: Config, Debug, and Optimize. In Config, the user specifies the directory containing the Makefile for their CUDA program, the command to run the executable, and the path to their .cu file. These allow us to modify the user's CUDA file by inserting lines of code, recompiling, and extracting/analyzing from the output. These settings get saved in a text file.



Makefile Directory: ı/cuda-debugger-optimizer/scripts/test/threadOverwriteTest1

Run Command: -debugger-optimizer/scripts/test/threadOverwriteTest1/test1

Path to .cu File: /home/kflorendo/cuda-debugger-optimizer/scripts/test/threa

The Debug and Optimize pages let users interact with all the scripts and features we implemented. The specifics for retrieving user input, preprocessing the input for Bash and Python scripts, post processing script outputs, and displaying the output GUI vary based on the feature implemented and will be explained in the later sections.

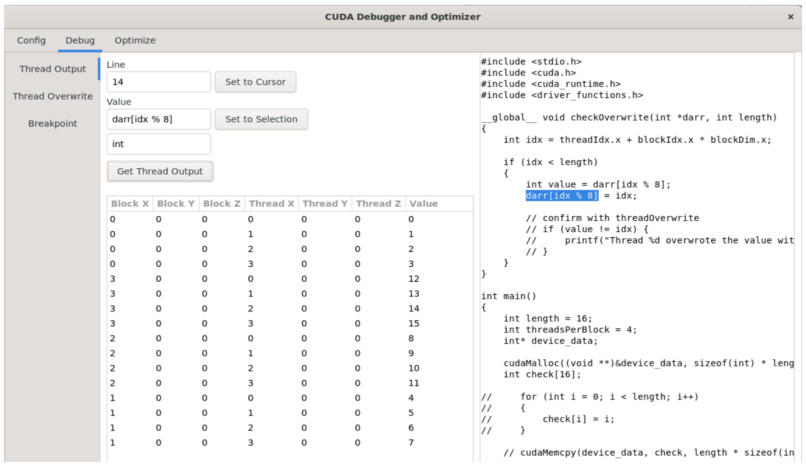
Thread Output Feature

GUI Input → Bash Script

This feature allows the user to inspect the value of a variable or expression observed by every thread. The UI consists of the CUDA file on the right, which allows the user to select the variable or expression by highlighting the code and clicking “Set to Selection” next to the Value entry input. To input the line number to inspect the value at, the user can place their cursor on the desired line and click “Set to Cursor”. Finally, the user inputs the type of the variable (currently int and float are supported). When the user clicks “Get Thread Output”, the data is extracted from these input fields and passed to a Bash file.

Bash Script → Text File → GUI

A temporary print statement is inserted into the given program, which writes the thread’s blockIdx, threadIdx, and the value of the user-inputted variable into an output text file. The GUI reads in the text file, and displays each line of the file as a row in a table, including the block and thread indices and the value observed for each thread.



Line: 14

Value: darr[idx % 8]

int

Block X	Block Y	Block Z	Thread X	Thread Y	Thread Z	Value
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	0	2	0	0	2
0	0	0	3	0	0	3
3	0	0	0	0	0	12
3	0	0	1	0	0	13
3	0	0	2	0	0	14
3	0	0	3	0	0	15
2	0	0	0	0	0	8
2	0	0	1	0	0	9
2	0	0	2	0	0	10
2	0	0	3	0	0	11
1	0	0	0	0	0	4
1	0	0	1	0	0	5
1	0	0	2	0	0	6
1	0	0	3	0	0	7

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <driver_functions.h>

__global__ void checkOverwrite(int *darr, int length)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;

    if (idx < length)
    {
        int value = darr[idx % 8];
        darr[idx % 8] = idx;

        // confirm with threadOverwrite
        // if (value != idx) {
        //     printf("Thread %d overwrote the value wit
        // }
    }
}

int main()
{
    int length = 16;
    int threadsPerBlock = 4;
    int* device_data;

    cudaMalloc((void **)&device_data, sizeof(int) * leng
    int check[16];

//     for (int i = 0; i < length; i++)
//     {
//         check[i] = i;
//     }

    // cudaMemcpy(device_data, check, length * sizeof(in
```

A screenshot of the output of an array element on line 14.
The user can see that each thread observes the correct value: the thread index.

Thread Overwrite Feature

GUI Input → Bash Script

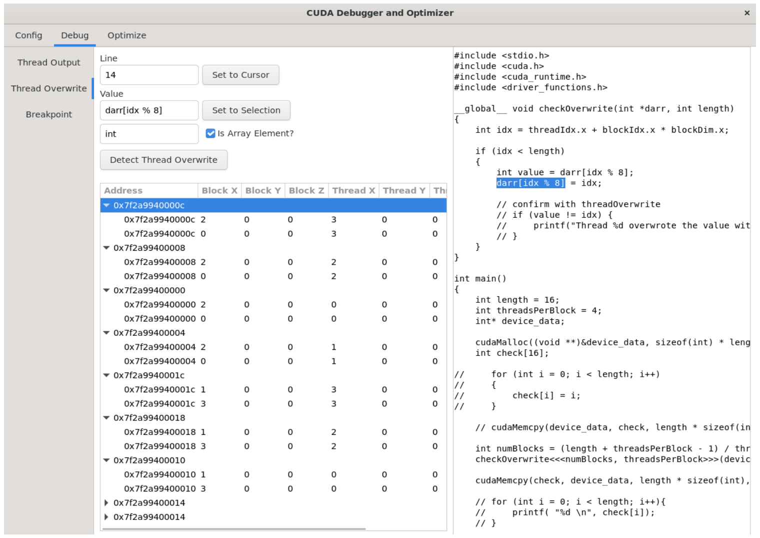
This feature helps the user detect if 2 threads are writing to the same memory address. In the GUI, the user specifies the variable that they wish to check overwriting for, the line number to detect overwriting at, and the type of the variable, similar to Thread Output. There is also a checkbox that allows the user to specify if the variable is an element of an array and if we should display the index of the array in the output. When the user clicks “Detect Thread Overwrite”, the input gets extracted and fed to a Bash file.

Bash Script → Text File

A print statement is temporarily inserted into the given code, which writes the following information about the specific thread to a line in the output file: blockIdx, threadIdx, address that the thread is writing to, the value that the thread wrote, and if applicable, the index of the array variable that was passed in by the user. Then, we make an associative array, and each line is added to the array by using the address written to by the thread as the key, and the rest of the information is stored in the corresponding value of the array. Lastly, once the entire file has been parsed, the address, along with all threads that have been stored in the corresponding value, will be printed. The threads in the value are ones that have written to the same address, which is represented by the key.

Text File → GUI Output

Then, the GUI reads in the results from the file and displays it in a tree table. Each address has its own row, its children correspond to the threads that wrote to that address, sorted in time order of the write. For each thread, the user can view the block and thread indices, the index it wrote to (if it was an array element), and the value that it wrote.



Line: 14

Value: darr[idx % 8]

int

[✓ Is Array Element?]

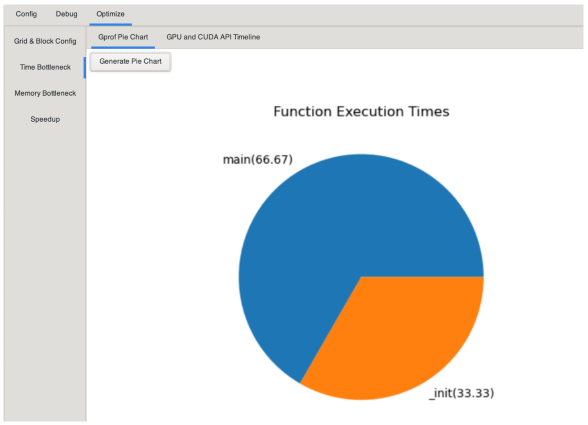
Address	Block X	Block Y	Block Z	Thread X	Thread Y	Th
▾ 0x7f2a9940000c						
0x7f2a9940000c	2	0	0	3	0	0
0x7f2a9940000c	0	0	0	3	0	0
▾ 0x7f2a99400008						
0x7f2a99400008	2	0	0	2	0	0
0x7f2a99400008	0	0	0	2	0	0
▾ 0x7f2a99400000						
0x7f2a99400000	2	0	0	0	0	0
0x7f2a99400000	0	0	0	0	0	0
▾ 0x7f2a99400004						
0x7f2a99400004	2	0	0	1	0	0
0x7f2a99400004	0	0	0	1	0	0
▾ 0x7f2a9940001c						
0x7f2a9940001c	1	0	0	3	0	0
0x7f2a9940001c	3	0	0	3	0	0
▾ 0x7f2a99400018						
0x7f2a99400018	1	0	0	2	0	0
0x7f2a99400018	3	0	0	2	0	0
▾ 0x7f2a99400010						
0x7f2a99400010	1	0	0	0	0	0
0x7f2a99400010	3	0	0	0	0	0
▸ 0x7f2a99400014						
▸ 0x7f2a99400014						

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <driver_functions.h>

__global__ void checkOverwrite(int *darr, int length)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;

    if (idx < length)
    {
        int value = darr[idx % 8];
        darr[idx % 8] = idx;

        // confirm with threadOverwrite
        // if (value != idx) {
        //     printf("Thread %d overwrote the value wit
        // }
    }
}

int main()
{
    int length = 16;
    int threadsPerBlock = 4;
    int* device_data;

    cudaMalloc((void **)&device_data, sizeof(int) * leng
    int check[16];

//     for (int i = 0; i < length; i++)
//     {
//         check[i] = i;
//     }

    // cudaMemcpy(device_data, check, length * sizeof(in

    int numBlocks = (length + threadsPerBlock - 1) / thr
    checkOverwrite<<<numBlocks, threadsPerBlock>>>(devic

    cudaMemcpy(check, device_data, length * sizeof(int),

    // for (int i = 0; i < length; i++){
    //     printf( "%d \n", check[i]);
    // }
```

A screenshot of the output of checking if an array element was overwritten.

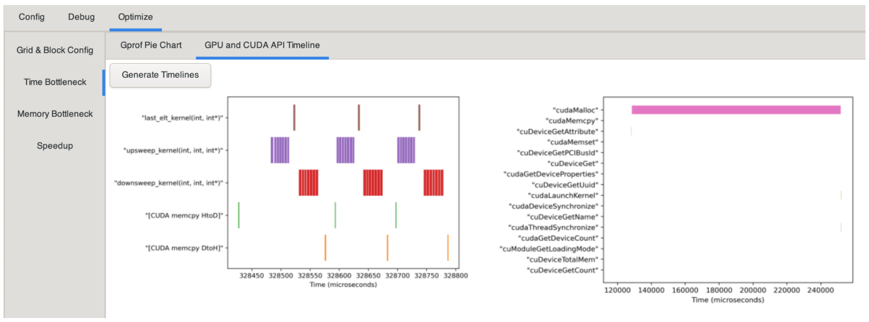
Time Bottleneck Pie Chart (GProf)

On the Optimize page, under the Time Bottleneck tab, the user can view a pie chart of the percentage of time different functions took when the program ran. The gprof command executes upon a gmon.out file that is created by a Makefile. gmon.out contains time profiling information about the program, such as the time each function took to execute during one run of the program as a percentage of the total runtime. The nonzero runtimes and corresponding name columns of the data file are preserved, and a pie chart is generated to visually represent the time percentages.



Time Bottleneck Timeline (GPU/API trace)

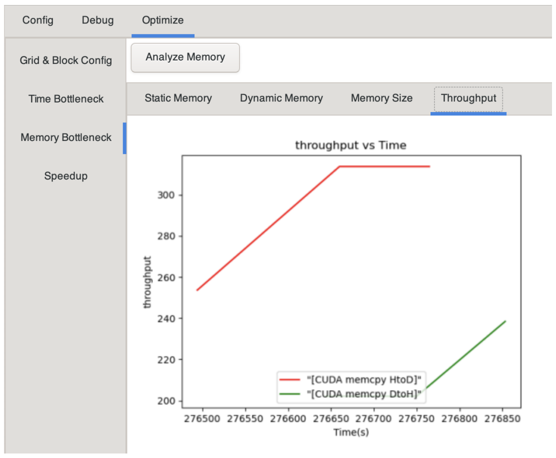
Also in the Time Bottleneck tab, the user can view timelines of GPU and API operations and their start and end times during the program execution.The text file containing the results of running nvprof CPU/API trace on a program is the input for this function. Each of the lines of the input is then parsed and the start time, duration, and corresponding name of the function called are extracted. Then, that data is processed into a PolyCollection data type variable, which can later be displayed into a timeline, where the x-axis contains the time in microseconds and the y-axis contains the function names.



GPU timeline on the left, CUDA API timeline on the right

Memory Bottleneck (nvprof)

In the Memory Bottleneck tab, the user can generate graphs of different memory metrics vs. time on the GPU. A Python script takes in a file generated by the nvprof GPU trace command, which contains data on the following fields for a given program: time, staticMem, dynamicMem, size, throughput, and function name. Each line in the file corresponds to a record of the characteristics of a function at a given time stamp. The columns of the file are placed into lists, then the data points are sorted into either the X or Y dictionary, which contains the x and y point values in the correlation. Then, points with the same function name are graphed as one continuous line, and this is repeated across all variable correlations against time for all functions.



Finding Optimal Configuration

GUI Input → Bash Script

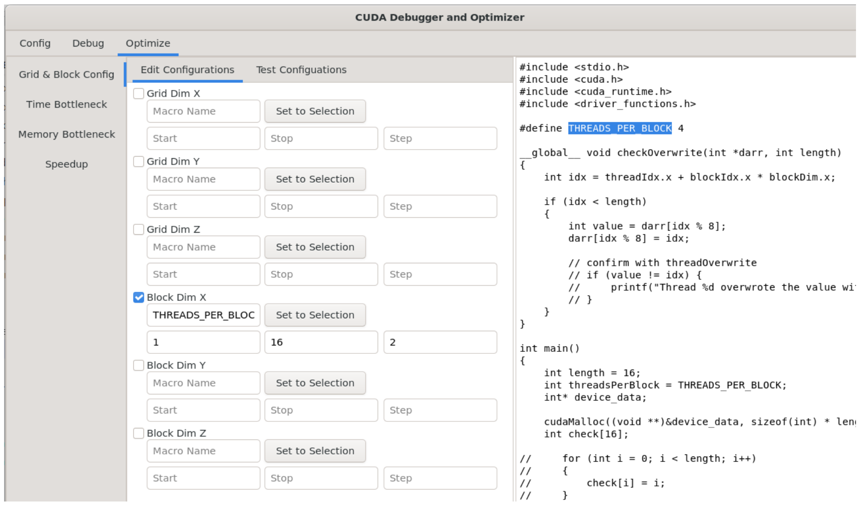
This feature lets the user try out multiple configurations of grid and block dimensions and determine which one is the fastest. In the GUI, the user selects checkboxes for the grid and block dimensions to vary (these must be defined as macros in their CUDA file). They can specify macro names directly from the CUDA file similar to Thread Output and Overwrite. The user also specifies a start, stop, and step value (multiplier) for possible values to test for this dimension. Then, all permutations of these possible values are calculated and inputted to a Bash script.

Bash → Text File

The Bash script then parses the permutations and alters the existing macros in the code, Each configuration is compiled then the program is executed 3 times, and for each run, the config and runtime in seconds are outputted to a text file.

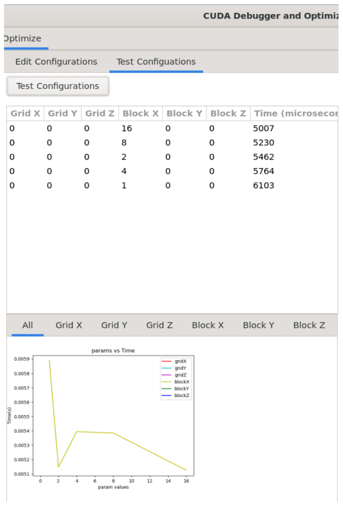
Text File and Graphs → GUI Output

The text file results are parsed in the GUI, and times for the same config are averaged. All the configs are sorted by time and displayed in a table with the fastest config on top.



[✓ Block Dim X]: THREADS_PER_BLOC, Start 1, Stop 16, Step 2

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <driver_functions.h>

#define THREADS_PER_BLOCK 4

__global__ void checkOverwrite(int *darr, int length)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;

    if (idx < length)
    {
        int value = darr[idx % 8];
        darr[idx % 8] = idx;

        // confirm with threadOverwrite
        // if (value != idx) {
        //     printf("Thread %d overwrote the value wi
        // }
    }
}

int main()
{
    int length = 16;
    int threadsPerBlock = THREADS_PER_BLOCK;
    int* device_data;

    cudaMalloc((void **)&device_data, sizeof(int) * len
    int check[16];

//     for (int i = 0; i < length; i++)
//     {
//         check[i] = i;
//     }
```



Grid X	Grid Y	Grid Z	Block X	Block Y	Block Z	Time (microsecor
0	0	0	16	0	0	5007
0	0	0	8	0	0	5230
0	0	0	2	0	0	5462
0	0	0	4	0	0	5764
0	0	0	1	0	0	6103

Parameter Graph Generation

A file containing various configurations and corresponding time measurements is passed in. Then, the data points will be appended to dictionaries where they will be grouped by value of a designated variable. The average of the times of the points with the same value is taken, and then the correlation will be graphed. There will be a total of 7 graphs generated representing each variable that was cased on, with the seventh as a combination of the other six graphs.

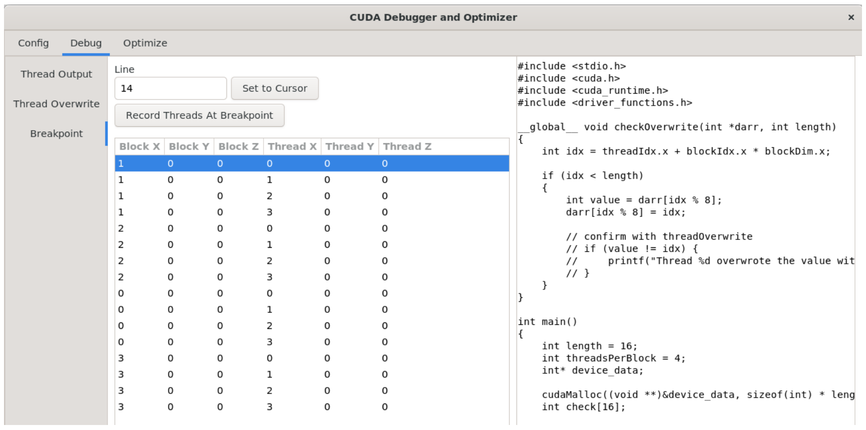
Thread Breakpoints Features

GUI Input → Bash Script

This feature lets the user place breakpoints in the code and observe which threads reach that portion of the code. This could be useful if the user thinks that a thread is entering an if statement or loop but it's not actually getting there. In the GUI, the user enters the line number by selecting a cursor position from the CUDA file on the right.

Bash Script → Text File → GUI Output

In the Bash script, a print statement is temporarily inserted at the user inputted line number, which will write to an output file. Each write contains the blockIdx and threadIdx values needed to be able to identify each individual thread. Then, the threads are sorted based on those index values, with blockIdx prioritized over threadIdx values, and the threads corresponding to missing index values within the sorted list would be those that did not reach this point in the code. The sorted threads are displayed in a table to the user.



Line: 14

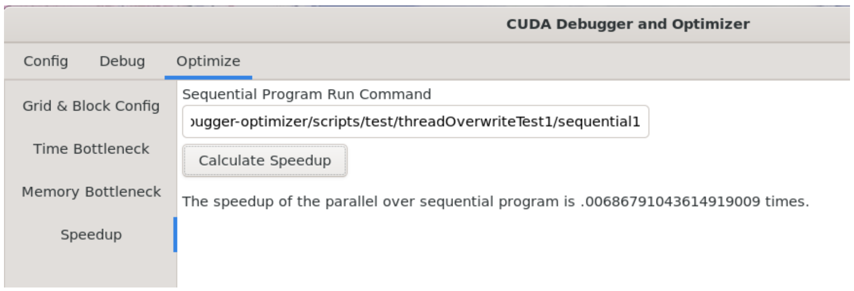
Block X	Block Y	Block Z	Thread X	Thread Y	Thread Z
1	0	0	0	0	0
1	0	0	1	0	0
1	0	0	2	0	0
1	0	0	3	0	0
2	0	0	0	0	0
2	0	0	1	0	0
2	0	0	2	0	0
2	0	0	3	0	0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	0	2	0	0
0	0	0	3	0	0
3	0	0	0	0	0
3	0	0	1	0	0
3	0	0	2	0	0
3	0	0	3	0	0

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <driver_functions.h>

__global__ void checkOverwrite(int *darr, int length)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;

    if (idx < length)
    {
        int value = darr[idx % 8];
        darr[idx % 8] = idx;

        // confirm with threadOverwrite
        // if (value != idx) {
        //     printf("Thread %d overwrote the value wit
        // }
    }
}

int main()
{
    int length = 16;
    int threadsPerBlock = 4;
    int* device_data;

    cudaMalloc((void **)&device_data, sizeof(int) * leng
    int check[16];
```

A screenshot of the threads that reached line 14 in the program

Speedup Calculation Feature

For each of the given programs, the sequential and parallel versions of the same algorithm, a timer is placed before the start of the code and its value is checked after the code terminates. The speedup is then calculated using the two measured times.



Sequential Program Run Command: ›ugger-optimizer/scripts/test/threadOverwriteTest1/sequential1

The speedup of the parallel over sequential program is .00686791043614919009 times.

RESULTS

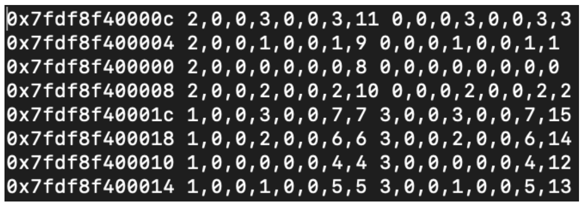
Test Scripts

In order to verify the correctness of our scripts, simple CUDA test files were implemented so it was evident that certain threads would write to specific indices in the given variable. Additionally, one of the test cases contained a sequential and a parallel version, so the speedup between the two can be calculated. The simplicity of the test code made it easy to identify the correct solution, but was also adaptable in that it could be used for all the features that were implemented in our program.

test1.cu

test1.cu is a short CUDA script that updates a device_data array in parallel. The array has length 16, and there are 4 threads per block. It is defined in the kernel that each thread will write its own thread index as the value of the index corresponding to its own thread index modded by 8. This way, it can be ensured that multiple threads will write at the same address in the array.

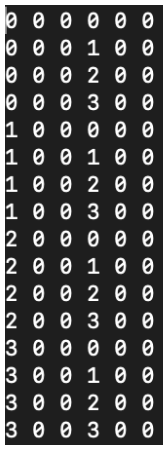
threadOverwrite output:



```
0x7fdf8f40000c 2,0,0,3,0,0,3,11 0,0,0,3,0,0,3,3
0x7fdf8f400004 2,0,0,1,0,0,1,9 0,0,0,1,0,0,1,1
0x7fdf8f400000 2,0,0,0,0,0,0,8 0,0,0,0,0,0,0,0
0x7fdf8f400008 2,0,0,2,0,0,2,10 0,0,0,2,0,0,2,2
0x7fdf8f40001c 1,0,0,3,0,0,7,7 3,0,0,3,0,0,7,15
0x7fdf8f400018 1,0,0,2,0,0,6,6 3,0,0,2,0,0,6,14
0x7fdf8f400010 1,0,0,0,0,0,4,4 3,0,0,0,0,0,4,12
0x7fdf8f400014 1,0,0,1,0,0,5,5 3,0,0,1,0,0,5,13
```

The data to the right represents what is returned in the output file. Each line is formatted to contain the address that is being written to, and all threads that wrote to that address separated by spaces. Each thread is represented by 8 numbers, with the first 3 corresponding to blockIdx values, the next 3 to threadIdx values, the second to last number representing the index of the array equivalent to the address, and the last number representing the value that was written to the address. In this example, each address has 2 threads, which is trivial, since there are a total of 16 writes, and 2 threads with the same % 8 value for their blockIdx and threadIdx representation would write to the same location. All possible blockIdx and threadIdx appear in the above output, and all values that written are distinct. The index included in threads also match across each line.

breakpoint.sh output:



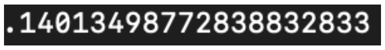
```
0 0 0 0 0 0
0 0 0 1 0 0
0 0 0 2 0 0
0 0 0 3 0 0
1 0 0 0 0 0
1 0 0 1 0 0
1 0 0 2 0 0
1 0 0 3 0 0
2 0 0 0 0 0
2 0 0 1 0 0
2 0 0 2 0 0
2 0 0 3 0 0
3 0 0 0 0 0
3 0 0 1 0 0
3 0 0 2 0 0
3 0 0 3 0 0
```

The data to the right represents the set of threads that were able to reach line 14 in the test1.cu file. Since none of the threads were excluded, all possible blockIdx and threadIdx combinations are included in the file, with 16 total lines. Regardless of the order each thread arrives at the line, they will be sorted based on their values so it is easier for the user to identify which threads, if any, were unable to make it to the breakpoint.

sequential1.cu

sequential1.cu is a sequential implementation of test1.cu. Instead of defining a kernel and threads per block, a for loop is used to sequentially iterate through the length of the array, with each individual iteration *i* setting the array's *i* mod 8th index to value *i*.

Speedup Achieved between *sequential1.cu* and *test1.cu*:



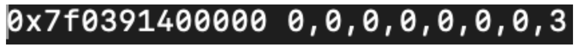
```
.14013498772838832833
```

The above value indicates that sequential1.cu was faster than test1.cu. This meets our expectations, because the number of iterations is small. test1.cu calls cudaMemcpy to transfer the information from the kernel to the main function, but that is not necessary for sequential1.cu. In this situation, the communication overhead for cudaMemcpy dominates over the speedup we get from parallelism.

test2.cu

test2.cu is almost identical to test1.cu, but instead of an array variable, there is an integer. This test case is used to check that the index value generated by the threadOverwrite.sh file initializes to 0 for all threads, since an integer should not have index values.

threadOverwrite output:



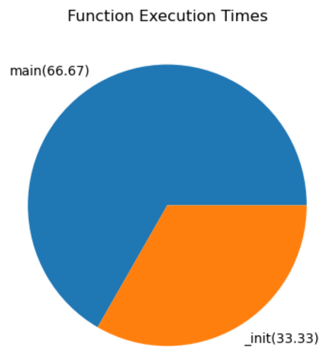
```
0x7f0391400000 0,0,0,0,0,0,0,3
```

The above output meets our expectations. The formatting of the file is mentioned above, in the threadOverwrite output section for test1.cu. Since there is only 1 variable being edited, only 1 thread is used. The value that the thread set is initialized to 3, so it can be differentiated with the other values. As expected, the index value is also set to 0, since the variable is not an array.

Case Study: 15-418 Assignment 2 - scan.cu

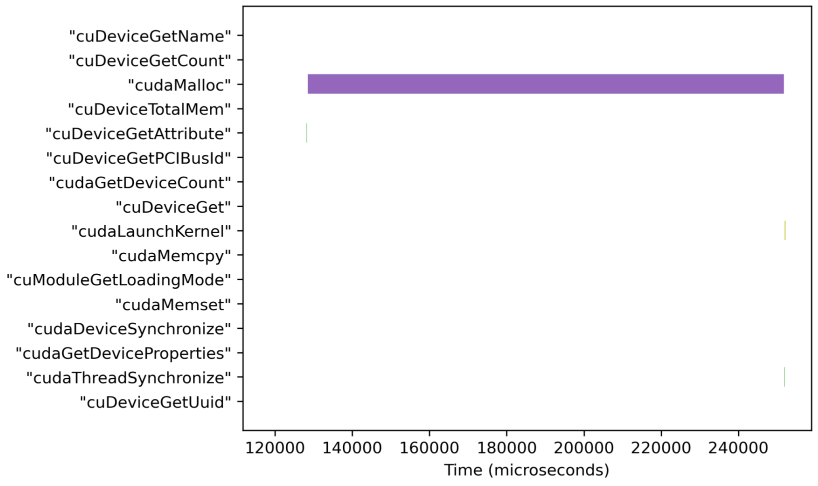
Command: cudaScan -m scan -i random -n 10000000

Time Bottleneck (GProf) Output:

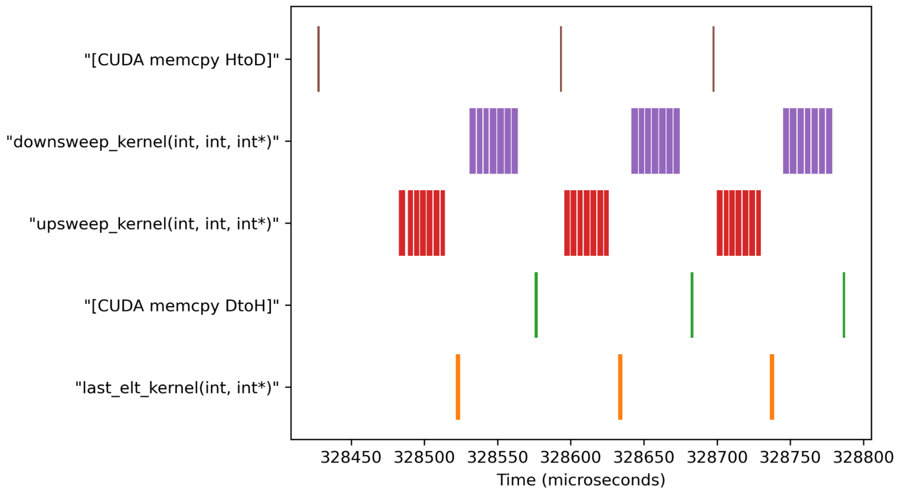


Time Bottleneck (GPU/API trace) Output:

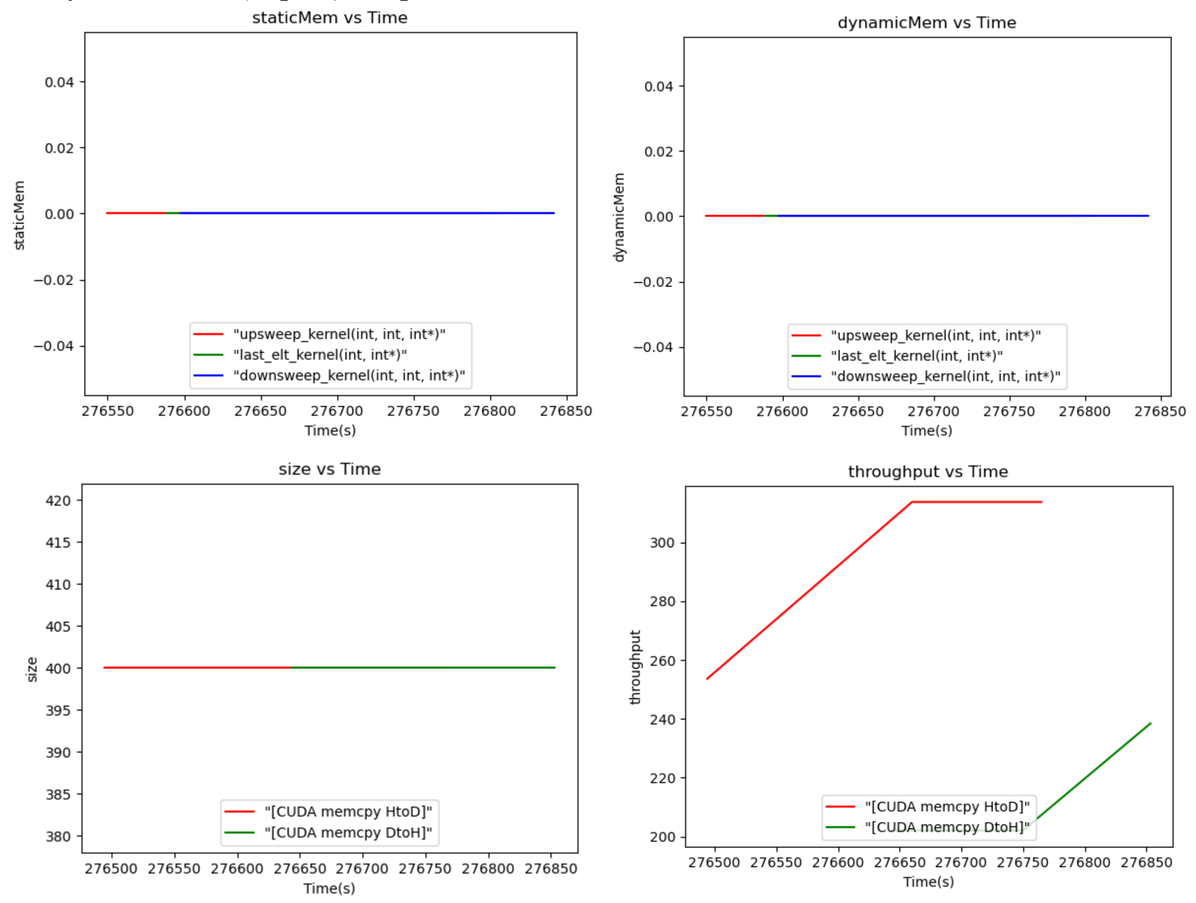
- API:



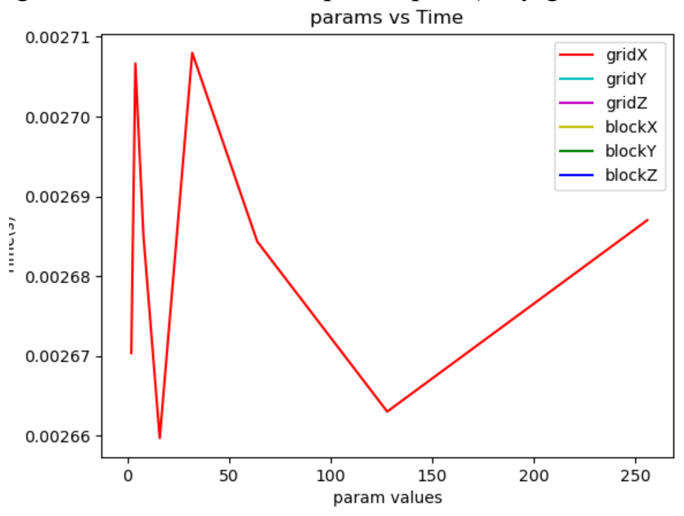
- GPU:



Memory Bottleneck (nvprof) Output:



Finding Optimal Configuration, Parameter Graph Output: (only gridX value adjusted)



REFERENCES

"Profiler User's Guide." *CUDA*, NVIDIA, 19 Apr. 2023, https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof.

"The GNU Profiler." *GNU GPROF*, 7 Nov. 1998, https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html.

"Timeline Bar Graph Using Python and Matplotlib." *Stack Overflow*, July 2018, https://stackoverflow.com/questions/51505291/timeline-bar-graph-using-python-and-matplotlib.

LIST OF WORK DONE BY EACH STUDENT

Task Name	Member
GUI: Config, Ddebug, Optimize Pages, Preprocess script input, Postprocess and display script output	Katrina Florendo
Thread Output Script	Katrina Florendo
Thread Overwrite Script	Huining Liang
Time Bottleneck GProf Script + Graphs	Huining Liang
Time Bottleneck GPU/API trace Script + Graphs	Katrina Florendo
Memory Bottleneck nvprof Graphs	Huining Liang
Finding Optimal Configuration Script	Katrina Florendo
Parameter Graph Generation	Huining Liang
GUI Linking	Katrina Florendo
Test Scripts	Huining Liang
Thread Breakpoint Script	Huining Liang
Speedup Calculation Script	Huining Liang

The distribution of total credit is around 55% for Katrina Florendo and 45% for Huining Liang.