COA lab

Group 4

Team Members:

1. Jatin Yadav 21cs02007

2. Shlok Kumar Shaw 21cs02008

3. Aniket Roy 21cs01061

4. Girish Jain 21cs01016

5. Ajinkya Deshmukh 21cs01029

Installing and setting up CUDA

Version chosen

We began our project using the most recent iteration of the Linux operating system, specifically version 22.04. However, the utilization of the CUDA software was found to be disruptive, prompting the decision to downgrade the system to version 20.04.6.

Installing the accessory dependancies

1) The installation of the gcc compiler was performed by executing the command "sudo apt install gcc-7".

2) In a similar manner, the installation of g++ was executed by running the command "sudo apt install g++-7".

3) To clone a git repository, the git package was installed using the command "sudo apt install git".

4) In order to install Python-based commands, the command "sudo apt install pip" was utilized to install the pip package manager.

5) The installation of the different dependencies of GPGPU-Sim was performed by executing the command "sudo apt-get install build-essential xutils-dev bison zlib1g-dev flex libglu1-mesa-dev".

6) Next, we proceeded to install the necessary dependencies for the gpgpu-sim documentation by executing the command "sudo apt-get install doxygen graphviz."

7) Subsequently, the diverse dependencies of AerialVision were installed. We employed a series of five consecutive commands for this purpose: "pip install pmw," "pip install ply," "pip install numpy," "pip install matplotlib," and "sudo apt-get install libpng-dev."

8) Next, the dependencies for the CUDA SDK were installed. To accomplish this task, we employed the command "sudo apt-get install libxi-dev libxmu-dev freeglut3-dev."

9) Subsequently, we proceeded to replicate the git repository of gpgpusim instructions by executing the command: "git clone https://github.com/gpgpu-sim/gpgpu-sim_distribution."

10) Then, we verified that the CUDA_INSTALL_PATH variable is appropriately configured to correspond with the designated directory in which the CUDA Toolkit was installed, such as /usr/local/cuda. In this experiment, the command "export CUDA_INSTALL_PATH=/usr/local/cuda" was employed.

Installing the CUDA Toolkit

1) The selection of CUDA 11.1.0 was made due to its compatibility with the chosen version of wlinux and its optimal suitability for gpgpu-sim.

2) The relevant commands were obtained by accessing the official CUDA Toolkit webpage and navigating to the Linux section, specifically targeting the x86_64 architecture (64-bit Intel x86). Further specifications included selecting the Ubuntu distribution and version 20.04, followed by opting for the runfile (local) option, which refers to a self-contained local installer.

3) The CUDA toolkit was installed by executing the command "wget https://developer.download.nvidia.com/compute/cuda/11.1.0/local_installers/cuda_11.1.0_455.23.05_linux.run".

4) The CUDA version was executed by running the command "sudo sh cuda_11.1.0_455.23.05_linux.run". A window was opened, where the user selected the option "accept" and proceeded with the installation by excluding the driver component.

Building

1) To access the bash shell for executing subsequent commands, navigate to the root directory of the simulator and input the command "bash".

2) The command "source setup_environment" was employed to establish the environment accordingly. By default, the release mode would be utilized.

3) The make command was employed to compile various program components and generate a final executable file.

4) Once the construction process is completed, the simulator will be prepared for utilization. In order to remove any residual files and restore the build to its original state, the command "make clean" was executed.

5) In order to generate the doxygen documentation, the command "make docs" was executed.

6) In the concluding phase, the documents were subjected to a cleansing process through the execution of the "make cleandocs" command.

Running a simple code

Code used

Since CUDA is a completely new language for us, we used a already and ubiquitously available code from the internet. Specifically, the site we used was http://web.mit.edu/pocky/www/cudaworkshopMatrix/VectorAdd.cu. This is a simple code of vector addition.

The code we used was:

```
#include <stdio.h>

#include <cuda.h>

#include <stdlib.h>

#include <time.h>

#define N 4096          // size of array

__global__ void add(int *a,int *b, int *c) {

              int tid = blockIdx.x *  blockDim.x + threadIdx.x;

        if(tid < N){

          c[tid] = a[tid]+b[tid];

        }

}

int main(int argc, char *argv[])  {

              int T = 10, B = 1;            // threads per block and blocks per grid

              int a[N],b[N],c[N];

              int *dev_a, *dev_b, *dev_c;

              printf("Size of array = %d\n", N);

              do {

                            printf("Enter number of threads per block: ");

                            scanf("%d",&T);

                            printf("\nEnter nuumber of blocks per grid: ");

                            scanf("%d",&B);

                            if (T * B != N) printf("Error T x B != N, try again");

              } while (T * B != N);

              cudaEvent_t start, stop;     // using cuda events to measure time

              float elapsed_time_ms;       // which is applicable for asynchronous code also

              cudaMalloc((void**)&dev_a,N * sizeof(int));

              cudaMalloc((void**)&dev_b,N * sizeof(int));

              cudaMalloc((void**)&dev_c,N * sizeof(int));
```

```
	for(int i=0;i<N;i++) {    // load arrays with some numbers
		a[i] = i;
		b[i] = i*1;
	}
	cudaMemcpy(dev_a, a , N*sizeof(int),cudaMemcpyHostToDevice);
	cudaMemcpy(dev_b, b , N*sizeof(int),cudaMemcpyHostToDevice);
	cudaMemcpy(dev_c, c , N*sizeof(int),cudaMemcpyHostToDevice);
	cudaEventCreate( &start );     // instrument code to measure start time
	cudaEventCreate( &stop );
	cudaEventRecord( start, 0 );
	add<<<B,T>>>(dev_a,dev_b,dev_c);
	cudaMemcpy(c,dev_c,N*sizeof(int),cudaMemcpyDeviceToHost);
	cudaEventRecord( stop, 0 );     // instrument code to measue end time
	cudaEventSynchronize( stop );
	cudaEventElapsedTime( &elapsed_time_ms, start, stop );
	for(int i=0;i<N;i++) {
		printf("%d+%d=%d\n",a[i],b[i],c[i]);
	}
	printf("Time to calculate results: %f ms.\n", elapsed_time_ms);  // print out execution time
	// clean up
	cudaFree(dev_a);
	cudaFree(dev_b);
	cudaFree(dev_c);
	cudaEventDestroy(start);
	cudaEventDestroy(stop);
	return 0;
}
```

Downgrading gcc and g++

1) It was observed that the utilization of the latest iterations of gcc and g++ resulted in encountering a segmentation fault during the execution of the code. Consequently, it was necessary to revert to the previous iteration of version 7.0.

2) The installation of the previous version was executed by employing the command "sudo apt install gcc-7 g++-7."

3) The directory was modified to "bin" using the command "cd /bin".

4) The gcc and g++ versions 9.4 were relocated to a separate directory, while version 7 was designated as the default version. In this study, we employed a set of four command lines.The commands "sudo mv g++ g++-9.4", "sudo mv gcc gcc-9.4", and "sudo mv g++-7 g++" were executed. The command "sudo mv gcc-7 gcc" is used to move the directory named "gcc-7" to a new location with the name "gcc" using superuser privileges.

Setting the path

1) The "nano" text editor grants elevated permissions to open the ".bashrc" file, enabling the user to modify and personalize their shell environment. The command "sudo nano ~/.bashrc" was employed for this purpose.

2) Next, we proceed to designate the directory of the CUDA installation by exporting the CUDA_INSTALL_PATH variable as "/usr/local/cuda-11.1". Additionally, we update the PATH variable to include the "/usr/local/cuda-11.1/bin" directory. Finally, we source the ~/.bashrc file to ensure the changes take effect.

3) Subsequently, the terminal was reopened and the process recommenced within a fresh terminal.

Rebuilding

1) We navigated to the directory in which the gpgpu-sim was installed. The cd (change directory) command was utilized for this purpose.

2) The command "source setup_environment" was employed to establish the environment accordingly. By default, the release mode would be utilized.

3) The make command was employed to compile various program components and generate a final executable file.

4) Once the construction process is completed, the simulator will be prepared for utilization. In order to remove any residual files and restore the build to its original state, the command "make clean" was executed.

<u>Making the test file</u>

1) A new directory was created using the command "mkdir test".

2) We accessed the test directory by utilizing the "cd test" command.

3) A new file was created using the touch command. The file was given the name "add.cu" in accordance with its utilization of the CUDA programming language. The command executed was "touch add.cu."

4)   The code provided at the beginning of this section was replicated and inserted into the current file for the purpose of execution.

<u>Running the code</u>

1) The complete contents of the inherent SM75_RTX2060 architecture were replicated from the downloaded file for gpgpu-sim and stored in the root directory. In this experiment, the command "cp -r ~/Downloads/gpgpu-sim_distribution/configs/tested-cfgs/SM75_RTX2060/* ./" was employed.

2) The compilation process for a typical CUDA code involves the utilization of nvcc, an acronym for NVIDIA CUDA Compiler. The NVCC compiler is employed to compile CUDA code that is written in programming languages such as C, C++, or Fortran. This compilation process transforms the code into GPU machine code, enabling its execution on NVIDIA GPUs.

3) In order to compile the code using gpgpu-sim, the lcudart library was utilized. lcudart, an acronym for "CUDA Runtime Library," is a dynamically linked library that offers the necessary CUDA runtime functionality for the execution of CUDA applications on the GPU. The command employed in the execution of the task was nvcc -lcudart test.cu.

4) We verified the linkage of the libcudart.so library by executing the command "ldd a.out". In the absence of a

connection, we would have expected to encounter errors; however, no errors were observed.

5) Ultimately, the executable file "a.out" was acquired and subsequently executed via the command "./a.out". This provides the outcomes of the vector addition based on the implemented code.

Problems faced and solving them

Problem 1:

In our experimentation with the latest iteration of Ubuntu, specifically version 22.04, we progressed through the installation process until reaching the installation of CUDA. Nevertheless, during the process of constructing the file, the execution of the make command began to generate insurmountable version discrepancies.

Solution:- In order to address this issue, it was necessary to downgrade the Ubuntu operating system to version 20.04. Due to the non-linearity of the process, it was necessary to uninstall version 22.04 by removing the GRUB and subsequently reinstalling it.

Problem 2:

At the outset, we installed the latest iterations of both gcc and g++. The version in question was 11.4. During the execution of codes in gpgpu-sim, a segmentation error occurred, leading to an infinite loop.

Solution:- In order to address this matter, we opted to downgrade both gcc and g++ to version 7.0. The previous iteration was retained, while designating the current iteration as the default.