GPGPU-SIM INSTALLATION

COA LAB

Team members:
1. Ishan Mardani (21CS02023)
2.Akshit Dudeja (21CS01026)
3. Tushar Joshi(21CS01078)
4. Vishnu Tirth Bysani (21CS01077)
5. Joshua Dias Barreto (21CS01075)

Contents

What is a GPU?

GPU stands for **Graphics Processing Unit**. GPUs are also known as video cards or graphics cards. In order to display pictures, videos, and 2D or 3D animations, each device uses a GPU. A GPU performs fast calculations of arithmetic and frees up the CPU to do different things. A GPU has lots of smaller cores made for multi-tasking, while a CPU makes use of some cores primarily based on sequential serial processing. In the world of computing, graphics processing technology has advanced to offer specific benefits. The modern GPUs enables new possibilities in **content creation, machine learning, gaming**, etc.

GPU vs CPU

S.NO	CPU	GPU
1.	CPU stands for Central Processing Unit.	While GPU stands for Graphics Processing Unit.
2.	CPU consumes or needs more memory than GPU.	While it consumes or requires less memory than CPU.
3.	The speed of CPU is less than GPU's speed.	While GPU is faster than CPU's speed.
4.	CPU contain minute powerful cores.	While it contain more weak cores.
5.	CPU is suitable for serial instruction processing.	While GPU is not suitable for serial instruction processing.
6.	CPU is not suitable for parallel instruction processing.	While GPU is suitable for parallel instruction processing.
7.	CPU emphasis on low latency.	While GPU emphasis on high throughput.

Uses of GPU

1. Graphics Rendering: GPUs are primarily used for rendering graphics and images in applications like gaming, multimedia, and visual effects in movies and animations.

2. Machine Learning and Deep Learning: Modern GPUs are essential for accelerating machine learning tasks due to their ability to handle large amounts of data and complex computations.

3. Video Editing: GPUs accelerate video encoding and decoding, reducing the time required for video processing tasks.

4. Cryptocurrency Mining: GPUs are used to mine cryptocurrencies like Bitcoin by performing complex calculations to add new blocks to the blockchain.

5. Virtual Reality (VR) and Augmented Reality (AR): GPUs play a crucial role in rendering immersive VR and AR experiences, creating realistic simulations.

6. Computational Photography: GPUs improve image processing techniques in digital cameras and smartphones, enabling real-time enhancements and advanced features.

Why do we require GPGPU-SIM?

GPGPU-Sim (General-Purpose Graphics Processing Unit Simulator) is a valuable tool for modeling and evaluating GPU performance in general-purpose computing applications. GPUs were originally designed for graphics processing, but researchers discovered their potential for parallel computing in various non-graphics tasks, including scientific simulations, data processing, and machine learning.

GPGPU-Sim is essential for:

1. Architecture Design and Evaluation: It helps GPU architects and engineers explore new GPU architectures and assess their advantages and limitations to make informed design decisions.

2. Performance Analysis: Researchers use GPGPU-Sim to evaluate GPU performance under different architectures and configurations, including execution time, throughput, and resource utilization.

3. Algorithm and Code Optimization: Developers utilize GPGPU-Sim to identify performance bottlenecks in GPU-accelerated code and make necessary adjustments to enhance overall performance.

4. Workload Characterization: GPGPU-Sim allows researchers to study how different application types behave on GPUs concerning memory access patterns, data sizes, and computational complexities.

Dual Booting the system

The process of dual booting entails the installation and execution of two distinct operating systems on a single computer, affording the user the ability to select which operating system to utilize upon each instance of system startup.

Steps followed:

1) Prepartion of installation media: Download the installation media (ISO files) for Linux operating system. Create bootable USB drives from the downloaded ISO files.

2) Partitioning of hard drive: Prior to installing an additional operating system, it is required to establish a distinct partition on the hard drive specifically allocated for this purpose. This task can be accomplished by utilizing disk management tools or partitioning software. We allocated a space of 50-100 GB.

3) Installation: Insert the installation media for Linux, and proceeded to restart our computer. Follow the instructions displayed on the screen in order to proceed with the installation of the operating system. During the installation process, designate the previously established partition as

the preferred installation destination for the secondary operating system.

4) Installing the boot loader: When Linux is being installed as a secondary operating system, it typically includes the installation of its own boot loader, such as GRUB, as part of the installation process. Upon startup, the boot loader will identify the presence of multiple operating systems and present the user with the option to select the desired operating system for booting.

5) Boot selection: Upon initiating the computer, a boot menu will be displayed, providing the option to select between the two operating systems that have been installed.

Version chosen

We began our project using the most recent version of the Linux operating system, specifically version 22.04. However, the CUDA software was not compatible for this version, so we had to downgrade the system to version 20.04.6.

Installing and setting up CUDA

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) To install the dependencies of AerialVision. We executed the commands: "pip install pmw ply

numpy matplotlib," and "sudo apt-get install libpng-dev."

8) Next, the dependencies for the CUDA SDK were installed. For this task, we executed 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. So we used the command "export CUDA_INSTALL_PATH=/usr/local/cuda".

Installing the CUDA Toolkit

1) The selection of CUDA 11.1.0 was made due to its compatibility with the chosen version of Linux and GPGPU-sim.

2) The commands were obtained from the official CUDA Toolkit webpage and downloaded the x86_64 version. We opted 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 used to establish the environment. 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

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

Code used:

```
#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 using the latest versions of gcc and g++ resulted in a segmentation fault during the execution of the code. Consequently, it was necessary to revert to the previous 7.0 version.

2) The installation of the previous version was executed by running 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. To make this change, we employed a set of four command lines. The following commands were used:

- "sudo mv g++ g++-9.4",
- "sudo mv gcc gcc-9.4",
- "sudo mv g++-7 g++"
- "sudo mv gcc-7 gcc"

Setting the path

1) The "nano" text editor grants permissions to open the ".bashrc" file, enabling the user to modify and personalize their shell environment. The command "sudo nano ~/.bashrc" was used 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) Reopen the terminal for changes to take effect.

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.

Making the test file

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 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.

Running the code

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. To do this, we used "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. 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 used. 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 used was “nvcc -lcudart test.cu”.

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

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.

Why the code of vector addition?

We conducted an experiment in which we executed the code without utilizing the gpgpu-sim framework. The process involved compiling the source code file "add.cu" using the nvcc compiler, followed by executing the resulting executable file "a.out". However, this approach did not yield the expected results; instead, the additions produced highly inaccurate outcomes.

However, if we had utilized a simple code such as "Hello World," it would be easier for such an error to go unnoticed.

Problem 1:

In our experimentation with the latest version 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:

The utilization of CUDA 12.0, specifically the latest version, resulted in encountering multiple errors due to its incompatibility with UBUNTU 20.04. The construction process encountered a failure in this location as well.

Solution:- In order to resolve the matter at hand, we opted to downgrade CUDA to version 11.1, which was released prior to the current version. This version is fully compatible with Ubuntu 20.04.

Problem 3:

At the outset, we installed the latest versions 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 version was retained, while designating the current version as the default.

Problem 4:

During the execution of the code, we encountered an error message stating "overriding embedded ptx with ptx file". The execution of the executable file was unsuccessful, and in certain systems, it was not generated utilizing the lcudart command.

Solution:- The resolution to this intricate problem was rather straightforward. It was observed that when employing a file name consisting of multiple words, the inclusion of spaces in the name results in the file becoming unreadable. Consequently, the file name was modified in order to address the problem.

Experience and conclusion

Suggestions to the team

1) Several versions issues were encountered during the course of the entire process. It was necessary to perform a version downgrade of Ubuntu, CUDA, as well as gcc and g++. Efforts should be made to address these challenges in order to enhance the overall user experience.

2) The documentation on GitHub exhibited ambiguity in several instances, accompanied by numerous errors. The execution of the command "sudo apt-get install python-pmw python-ply python-numpy libpng12-dev python-matplotlib" resulted in multiple errors, necessitating its division into five smaller components. Furthermore, it should be noted that the command "sudo apt-get install libxi-dev libxmu-dev libglut3-dev" was incorrect in this context. It was necessary to utilize the package "freeglut" instead of "libglut". It is imperative to address and resolve such matters.

Conclusion

In summary, the incorporation of GPGPU-Sim and the CUDA library within our computer architecture laboratory represents a noteworthy achievement in the progression of our comprehension and investigation of parallel computing and GPU acceleration. By actively engaging with GPGPU-Sim, we acquired significant knowledge regarding the performance attributes of contemporary GPUs and their intricate architectural intricacies. This enabled us to undertake comprehensive examinations of diverse workloads, thereby yielding valuable insights. Furthermore, the CUDA library has proven to be a robust framework for the seamless development and optimization of GPU-accelerated applications, enabling us to fully exploit the capabilities of parallel computing in our research endeavors and projects.