LP-V

HPC Practical Assignment

Practical No-4(A and B)

Title: Write a CUDA Program for :1. Addition of two large vectors 2.Matrix Multiplication using CUDA C

Objectives: To understand the concept of CUDA programming

To understand the concept of addition of two large vector

To understand the concept of Matrix multiplication

Environment:

Operating system:64-bit OS Linux/windows

Language: Object Oriented Language/Java/Python

Theory:

CUDA stands for Compute Unified Device Architecture. It is an extension of C/C++ programming. CUDA is a programming language that uses the Graphical Processing Unit (GPU). It is a parallel computing platform and an API (Application Programming Interface) model, Compute Unified Device Architecture was developed by Nvidia. This allows computations to be performed in parallel while providing well-formed speed. Using CUDA, one can harness the power of the Nvidia GPU to perform common computing tasks, such as processing matrices and other linear algebra operations, rather than simply performing graphical calculations.

How CUDA Works:

- GPUs run one kernel (a group of tasks) at a time.
- Each kernel consists of blocks, which are independent groups of ALUs.
- Each block contains threads, which are levels of computation.
- The threads in each block typically work together to calculate a value.
- Threads in the same block can share memory.
- In CUDA, sending information from the CPU to the GPU is often the most typical part of the computation.
- For each thread, local memory is the fastest, followed by shared memory, global, static, and texture memory the slowest.

Typical CUDA Program flow:

1. Load data into CPU memory
2. Copy data from CPU to GPU memory – e.g., cudaMemcpy(…, cudaMemcpyHostToDevice)
3. Call GPU kernel using device variable – e.g., kernel<<<>>> (gpuVar)
4. Copy results from GPU to CPU memory – e.g., cudaMemcpy(.., cudaMemcpyDeviceToHost)
5. Use results on CPU

Matrix Multiplication:

2D matrices can be stored in the computer memory using two layouts − **row-major** and **column-major**. Most of the modern languages,

including C (and CUDA) use the row-major layout. Here is a visual representation of the same of both the layouts −

M0,0	M0,1	M0,2	M0,3
M1,0	M1,1	M1,2	M1,3
M2,0	M2,1	M2,2	M2,3
M3,0	M3,1	M3,2	M3,3

Matrix to be stored

M0,0	M0,1	M0,2	M0,3	M1,0	M1,1	M1,2	M1,3	M2,0	M2,1	M2,2	M

Row-major layout

Actual organization in memory −

M0,0	M1,0	M2,0	M3,0	M0,1	M1,1	M2,1	M3,1	M0,2	M1,2	M2,2	M

Let us now understand the above kernel with an example −

Let d_M be −

2	4	1
8	7	4
7	4	9

The above matrix will be stored as −

2	4	1	8	7	4	7	4	9

And let d_N be −

4	8	9

1	7	0
2	5	4

The above matrix will be stored as −

4	8	9	1	7	0	2	5	4

Since d_P will be a 3x3 matrix, we will be launching 9 threads, each of which will compute one element of d_P.

d_P matrix

(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)
(2,0)	(2,1)	(2,2)

Program for CUDA Multiplication

```
#include<stdio.h>
#include<cuda.h>
#define row1 2 /* Number of rows of first matrix */
#define col1 3 /* Number of columns of first matrix */
#define row2 3 /* Number of rows of second matrix */
#define col2 2 /* Number of columns of second matrix */

__global__ void matproduct(int *l,int *m, int *n)
{
   int x=blockIdx.x;
   int y=blockIdx.y;
   int k;

n[col2*y+x]=0;
```

```
for(k=0;k<col1;k++)
  {
   n[col2*y+x]=n[col2*y+x]+l[col1*y+k]*m[col2*k+x];
  }
}

int main()
{
   int a[row1][col1];
   int b[row2][col2];
   int c[row1][col2];
   int *d,*e,*f;
   int i,j;

   printf("\n Enter elements of first matrix of size 2*3\n");
   for(i=0;i<row1;i++)
   {
      for(j=0;j<col1;j++)
         {
            scanf("%d",&a[i][j]);
         }
   }
   printf("\n Enter elements of second matrix of size 3*2\n");
      for(i=0;i<row2;i++)
      {
         for(j=0;j<col2;j++)
            {
               scanf("%d",&b[i][j]);
            }
      }

   cudaMalloc((void **)&d,row1*col1*sizeof(int));
   cudaMalloc((void **)&e,row2*col2*sizeof(int));
   cudaMalloc((void **)&f,row1*col2*sizeof(int));
```

```
 cudaMemcpy(d,a,row1*col1*sizeof(int),cudaMemcpyHostToDevice
);
 cudaMemcpy(e,b,row2*col2*sizeof(int),cudaMemcpyHostToDevice
);

dim3 grid(col2,row1);
/* Here we are defining two dimensional Grid(collection of blocks)
structure. Syntax is dim3 grid(no. of columns,no. of rows) */

    matproduct<<<grid,1>>>(d,e,f);

 cudaMemcpy(c,f,row1*col2*sizeof(int),cudaMemcpyDeviceToHost)
;
    printf("\nProduct of two matrices:\n ");
    for(i=0;i<row1;i++)
    {
        for(j=0;j<col2;j++)
        {
             printf("%d\t",c[i][j]);
        }
        printf("\n");
    }

    cudaFree(d);
    cudaFree(e);
    cudaFree(f);

    return 0;
}
```

Output

```
Enter elements of first matrix of size 2*3
1 2 3 4 5 6

 Enter elements of second matrix of size 3*2
```

7 8 9 10 11 12

Product of two matrices:
 58   64
139   154

Program 2:CUDA Addition

```
#include<stdio.h>

#include<cuda.h>


__global__ void arradd(int *x,int *y, int *z)    //kernel definition

{

  int id=blockIdx.x;

/* blockIdx.x gives the respective block id which starts from 0 */

  z[id]=x[id]+y[id];

}


int main()

{

    int a[6];
```

```
    int b[6];
    int c[6];
    int *d,*e,*f;
    int i;
    printf("\n Enter six elements of first array\n");
    for(i=0;i<6;i++)
    {
       scanf("%d",&a[i]);
    }
    printf("\n Enter six elements of second array\n");
       for(i=0;i<6;i++)
       {
          scanf("%d",&b[i]);
       }

/* cudaMalloc() allocates memory from Global memory on GPU */
    cudaMalloc((void **)&d,6*sizeof(int));
    cudaMalloc((void **)&e,6*sizeof(int));
    cudaMalloc((void **)&f,6*sizeof(int));
```

```
/* cudaMemcpy() copies the contents from destination to source. Here destination is GPU(d,e) and source is CPU(a,b) */
 cudaMemcpy(d,a,6*sizeof(int),cudaMemcpyHostToDevice);
 cudaMemcpy(e,b,6*sizeof(int),cudaMemcpyHostToDevice);

/* call to kernel. Here 6 is number of blocks, 1 is the number of threads per block and d,e,f are the arguments */
arradd<<<6,1>>>(d,e,f);

/* Here we are copying content from GPU(Device) to CPU(Host) */
 cudaMemcpy(c,f,6*sizeof(int),cudaMemcpyDeviceToHost);

printf("\nSum of two arrays:\n ");
    for(i=0;i<6;i++)
    {
        printf("%d\t",c[i]);
    }

/* Free the memory allocated to pointers d,e,f */
    cudaFree(d);
```

```
    cudaFree(e);
    cudaFree(f);

    return 0;
}
```

Program: Steps to run CUDA program on google collab

Step1:

==========

!apt update -qq;

!wget https://developer.nvidia.com/compute/cuda/8.0/Prod2/local_installers/cuda-repo-ubuntu1604-8-0-local-ga2_8.0.61-1_amd64-deb;

!dpkg -i cuda-repo-ubuntu1604-8-0-local-ga2_8.0.61-1_amd64-deb;

!apt-key add /var/cuda-repo-8-0-local-ga2/7fa2af80.pub;

!apt-get update -qq;

!apt-get install cuda gcc-5 g++-5 -y -qq;

!ln -s /usr/bin/gcc-5 /usr/local/cuda/bin/gcc;

!ln -s /usr/bin/g++-5 /usr/local/cuda/bin/g++;

!apt install cuda-8.0;

==========

==========

Step2:check the version

!/usr/local/cuda/bin/nvcc --version

==========

========

Step3:

!pip install git+git://github.com/andreinechaev/nvcc4jupyter.git

==========

==

Step4:

%load_ext nvcc_plugin

==========

Don't forget to write %%cu before writing cuda program

Conclusion: After successfully completion of this experiment we understand the concept of CUDA vector addition and matrix multiplication.