

Nithin S  
221IT085

## IT301 Lab Assignment 8

**Q1. To know details of the device. Run the program and explain the result.**

### Code

```
[1] !nvcc --version
!pip install nvcc4jupyter
%load_ext nvcc4jupyter

nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Tue Aug 15 22:02:13 PDT 2023
Cuda compilation tools, release 12.2, V12.2.140
Build cuda_12.2.r12.2/compiler.33191640_0
Collecting nvcc4jupyter
  Downloading nvcc4jupyter-1.2.1-py3-none-any.whl.metadata (5.1 kB)
  Downloading nvcc4jupyter-1.2.1-py3-none-any.whl (10 kB)
Installing collected packages: nvcc4jupyter
Successfully installed nvcc4jupyter-1.2.1
Detected platform "Colab". Running its setup...
Source files will be saved in "/tmp/tmpztzs2az4_".
```

```

%%cuda
#include<stdio.h>
int main()
{
    int devcount;
    cudaGetDeviceCount(&devcount);
    printf("Device count:%d\n",devcount);
    printf("hello");
    for (int i = 0; i < devcount; ++i)
    {
        // Get device properties
        printf("\nCUDA Device #%d\n", i);
        cudaDeviceProp devProp;
        cudaGetDeviceProperties(&devProp, i);
        printf("Name:%s\n", devProp.name);
        printf("Compute capability: %d.%d\n",devProp.major ,devProp.minor);
        printf("Warp Size %d\n",devProp.warpSize);
        printf("Total global memory:%u bytes\n",devProp.totalGlobalMem);
        printf("Total shared memory per block: %u bytes\n", devProp.sharedMemPerBlock);
        printf("Total registers per block : %d\n",devProp.regsPerBlock);
        printf("Clock rate: %d khz\n",devProp.clockRate);
        printf("Maximum threads per block:%d\n", devProp.maxThreadsPerBlock);
        for (int i = 0; i < 3; ++i)
            printf("Maximum dimension %d of block: %d\n", i, devProp.maxThreadsDim[i]);
        for (int i = 0; i <= 2; ++i)
            printf("Maximum dimension %d of grid: %d\n", i, devProp.maxGridSize[i]);
        printf("Number of multiprocessors:%d\n", devProp.multiProcessorCount);
        printf("max threads per multiprocessor: %d\n",devProp.maxThreadsPerMultiProcessor);
    }
    return 0;
}

```

```

Device count:1
hello
CUDA Device #0
Name:Tesla T4
Compute capability: 7.5
Warp Size 32
Total global memory:2950758400 bytes
Total shared memory per block: 49152 bytes
Total registers per block : 65536
Clock rate: 1590000 khz
Maximum threads per block:1024
Maximum dimension 0 of block: 1024
Maximum dimension 1 of block: 1024
Maximum dimension 2 of block: 64
Maximum dimension 0 of grid: 2147483647
Maximum dimension 1 of grid: 65535
Maximum dimension 2 of grid: 65535
Number of multiprocessors:40
max threads per multiprocessor: 1024

```

## Analysis

This CUDA C program detects and prints information about available GPUs, revealing properties essential for optimizing CUDA applications.

First, it retrieves the number of CUDA-compatible devices.

For each device, it fetches and displays details like the device name, compute capability, warp size, global memory, shared memory per block, registers per block, clock rate, and thread limits.

For example, on a Tesla T4, the output shows a compute capability of 7.5, a warp size of 32 threads, around 2.95 GB of global memory, and 49152 bytes of shared memory per block.


It also includes details on maximum dimensions for blocks and grids, as well as the number of streaming multiprocessors (40) and maximum threads per multiprocessor (1024).

These metrics allow developers to fine-tune parallel processing by understanding the GPU's architectural constraints and capabilities, optimizing memory usage and thread configuration for better performance.

## Q2. Hello world program. Record the result and write the observation.

### Code and Output

```
[2] %%cuda
#include<stdio.h>
#include<cuda.h>
__global__ void helloworld(void)
{
    printf("Hello World from GPU\n");
}
int main() {
    helloworld<<<1,10>>>();
    printf("Hello World\n");
    return 0;
}
```

 Hello World

## Analysis

This CUDA C code demonstrates a simple "Hello World" program that runs on both the CPU and the GPU. It defines a kernel function, `helloworld`, marked by `__global__`, which allows it to be executed on the GPU.

Inside this function, a message "Hello World from GPU" is printed, and when launched from the main function, it is called with a kernel launch syntax `helloworld<<<1,10>>>()`. Here, `<<<1,10>>>` specifies one block with 10 threads, meaning 10 instances of the kernel will run in parallel on the GPU, each printing the message.

Additionally, a "Hello World" message is printed from the CPU to illustrate the capability of running code on both the CPU and GPU. While simple, this code is useful for verifying that the GPU is accessible for computation and that basic parallel execution works.

However, since `printf` calls from the GPU are executed independently by each thread, the "Hello World from GPU" message is expected to print multiple times, once for each thread in the block.

**Q3: Program to perform  $c[i] = a[i] + b[i]$ ; Here,  $c[i]$  is calculated for all  $i$ . But results are displayed only for few  $c[i]$ . Explain your observation.**

**Run the program for following and note down the time.**

**a) `vecAdd<<<1,100>>>(d_a, d_b, d_c, n);`**

## Code

```

%%cuda
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>

__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    // Get global thread
    int id = blockIdx.x*blockDim.x+threadIdx.x;
    // Do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}

int main(int argc, char* argv[] )
{
    // Size of vectors
    int n = 100;
    //time variables
    struct timeval t1, t2;
    // Host input vectors
    double *h_a, *h_b;
    //Host output vector
    double *h_c;
    // Device input vectors
    double *d_a, *d_b;
    //Device output vector
    double *d_c;
    // Size, in bytes, of each vector
    size_t bytes = n*sizeof(double);
    // Allocate memory for each vector on host
    h_a = (double*)malloc(bytes);
    h_b = (double*)malloc(bytes);
    h_c = (double*)malloc(bytes);
    // Allocate memory for each vector on GPU
    cudaMalloc(&d_a, bytes);
    cudaMalloc(&d_b, bytes);
    cudaMalloc(&d_c, bytes);
    int i;
    // Initialize vectors on host
    for( i = 0; i < n; i++ ) {
        h_a[i] = i+1;
        h_b[i] = i+1;
    }
    // Copy host vectors to device
    cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);

    gettimeofday(&t1, 0);

    // Execute the kernel
    vecAdd<<<1,100>>>>(d_a, d_b, d_c, n);
    cudaDeviceSynchronize();
}

```

```

gettimeofday(&t2, 0);

// Copy array back to host
cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );

for(i=0; i<n; i=i+10)
printf("c[%d]=%f\n",i,h_c[i]);
double time = (1000000.0*(t2.tv_sec-t1.tv_sec) + t2.tv_usec-t1.tv_usec)/1000.0;

printf("Time to generate: %3.10f ms \n", time);

// Release device memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
// Release host memory
free(h_a);
free(h_b);
free(h_c);
return 0;
}

```

## Output

```

c[0]=2.000000
c[10]=22.000000
c[20]=42.000000
c[30]=62.000000
c[40]=82.000000
c[50]=102.000000
c[60]=122.000000
c[70]=142.000000
c[80]=162.000000
c[90]=182.000000
Time to generate: 36.2510000000 ms

```

## Analysis

This CUDA program performs vector addition, where each element  $c[i]$  in the output vector  $c$  is calculated as the sum of corresponding elements from input vectors  $a$  and  $b$ . The program uses a GPU kernel function, `vecAdd`, to calculate  $c[i]=a[i]+b[i]$  for all elements  $i$  in parallel. It launches the kernel with a single block of 100 threads (`vecAdd<<<1,100>>>(d_a, d_b, d_c, n);`). Each thread computes a unique  $c[i]$  value by using its thread ID to index into the vectors.

The program only prints some of the `c[i]` values at intervals of 10 (`i = 0, 10, 20, ...`). This selective display reduces the amount of output, making it easier to view the results while verifying that the calculations are correct.

The execution time for the vector addition is measured using the `gettimeofday` function around the kernel launch. The total time taken includes memory transfers between the host and device as well as kernel execution and synchronization.

When running with `vecAdd<<<1,100>>>(d_a, d_b, d_c, n);`, the kernel utilizes 100 threads in a single block, which is suitable for vector operations where each thread can independently handle one addition operation. The program should execute very quickly, especially since the GPU can process many threads in parallel. The execution time may vary based on GPU architecture, but it should be significantly faster than a CPU implementation, especially for larger vectors.

**b) `vecAdd<<<1,50>>>(d_a, d_b, d_c, n);`**

## Code

```

%%cuda
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>

__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    // Get global thread
    int id = blockIdx.x*blockDim.x+threadIdx.x;
    // Do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}

int main(int argc, char* argv[] )
{
    // Size of vectors
    int n = 100;
    //time variables
    struct timeval t1, t2;
    // Host input vectors
    double *h_a, *h_b;
    //Host output vector
    double *h_c;
    // Device input vectors
    double *d_a, *d_b;
    //Device output vector
    double *d_c;
    // Size, in bytes, of each vector
    size_t bytes = n*sizeof(double);
    // Allocate memory for each vector on host
    h_a = (double*)malloc(bytes);
    h_b = (double*)malloc(bytes);
    h_c = (double*)malloc(bytes);
    // Allocate memory for each vector on GPU
    cudaMalloc(&d_a, bytes);
    cudaMalloc(&d_b, bytes);
    cudaMalloc(&d_c, bytes);
    int i;
    // Initialize vectors on host
    for( i = 0; i < n; i++ ) {
        h_a[i] = i+1;
        h_b[i] = i+1;
    }
    // Copy host vectors to device
    cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);

    gettimeofday(&t1, 0);

    // Execute the kernel
    vecAdd<<<1,50>>>>(d_a, d_b, d_c, n);
    cudaDeviceSynchronize();
}

```



```

gettimeofday(&t2, 0);

// Copy array back to host
cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );

for(i=0; i<n; i=i+10)
printf("c[%d]=%f\n",i,h_c[i]);
double time = (1000000.0*(t2.tv_sec-t1.tv_sec) + t2.tv_usec-t1.tv_usec)/1000.0;

printf("Time to generate: %3.10f ms \n", time);

// Release device memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
// Release host memory
free(h_a);
free(h_b);
free(h_c);
return 0;
}

```

## Output

```

c[0]=2.000000
c[10]=22.000000
c[20]=42.000000
c[30]=62.000000
c[40]=82.000000
c[50]=0.000000
c[60]=0.000000
c[70]=0.000000
c[80]=0.000000
c[90]=0.000000
Time to generate: 0.2110000000 ms

```

## Analysis

This CUDA program performs vector addition on the GPU, where each element  $c[i] = a[i] + b[i]$  is computed in parallel. The kernel `vecAdd` calculates  $c[i]$  for each element by assigning one thread to each index  $i$ , and each thread independently computes a corresponding  $c[i]$  value. The kernel is launched with a single block of only 50 threads (`vecAdd<<<1,50>>>(d_a, d_b, d_c, n);`),

meaning fewer threads than elements in the vector (100). Since there are only 50 threads and each is responsible for computing one  $c[i]$ , only the first 50 elements in  $c$  will be calculated; the remaining elements will remain uncomputed.

Since  $n$  (the vector size) is 100 and only 50 threads are launched, this setup leads to a situation where only the first half of  $c[i]$  values (i.e.,  $c[0]$  to  $c[49]$ ) will be computed. Values from  $c[50]$  to  $c[99]$  will not be initialized by the kernel and may contain uninitialized data, likely resulting in garbage values.

The time taken to execute the kernel is measured around the kernel launch and synchronization, accounting for the kernel execution, memory transfers, and host-GPU synchronization. Since the program only uses 50 threads, it may result in shorter kernel execution time due to the smaller number of active threads, but at the cost of incomplete results.

Only every 10th value of  $c[i]$  is printed ( $i = 0, 10, 20, \dots$ ), which should illustrate the computed values from  $c[0]$  to  $c[49]$ . For indices  $i \geq 50$ , the printed values could display random data or zeroes, depending on memory initialization.

Running `vecAdd<<<1,50>>>(d_a, d_b, d_c, n);` for  $n=100$  will lead to partial computation since only 50 threads are active. This situation demonstrates that the kernel launch configuration (block and grid size) must match or exceed  $n$  to ensure complete computation across all elements of  $c$ .

c)vecAdd<<<2,50>>>(d\_a, d\_b, d\_c, n);

## Code

```
%%cuda
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>

__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    // Get global thread
    int id = blockIdx.x*blockDim.x+threadIdx.x;
    // Do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}

int main(int argc, char* argv[] )
{
    // Size of vectors
    int n = 100;
    //time variables
    struct timeval t1, t2;
    // Host input vectors
    double *h_a, *h_b;
    //Host output vector
    double *h_c;
    // Device input vectors
    double *d_a, *d_b;
    //Device output vector
    double *d_c;
    // Size, in bytes, of each vector
    size_t bytes = n*sizeof(double);
    // Allocate memory for each vector on host
    h_a = (double*)malloc(bytes);
    h_b = (double*)malloc(bytes);
    h_c = (double*)malloc(bytes);
    // Allocate memory for each vector on GPU
    cudaMalloc(&d_a, bytes);
    cudaMalloc(&d_b, bytes);
    cudaMalloc(&d_c, bytes);
    int i;
    // Initialize vectors on host
    for( i = 0; i < n; i++ ) {
        h_a[i] = i+1;
        h_b[i] = i+1;
    }
    // Copy host vectors to device
    cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);

    gettimeofday(&t1, 0);

    // Execute the kernel
    vecAdd<<<2,50>>>(d_a, d_b, d_c, n);
    cudaDeviceSynchronize();
}
```

```

cudaDeviceSynchronize();
gettimeofday(&t2, 0);

// Copy array back to host
cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );

for(i=0; i<n; i=i+10)
printf("c[%d]=%f\n",i,h_c[i]);
double time = (1000000.0*(t2.tv_sec-t1.tv_sec) + t2.tv_usec-t1.tv_usec)/1000.0;

printf("Time to generate: %3.10f ms \n", time);

// Release device memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
// Release host memory
free(h_a);
free(h_b);
free(h_c);
return 0;
}

```

## Output

```

c[0]=2.000000
c[10]=22.000000
c[20]=42.000000
c[30]=62.000000
c[40]=82.000000
c[50]=102.000000
c[60]=122.000000
c[70]=142.000000
c[80]=162.000000
c[90]=182.000000
Time to generate: 0.2480000000 ms

```

## Analysis

This CUDA program calculates the element-wise sum of two vectors, storing the result in a third vector, using GPU parallelism. The vecAdd kernel is launched with `<<<2,50>>>`, meaning it uses 2 blocks, each with 50 threads, for a total of 100 threads. This is equal to the vector length  $n = 100$ , so each thread can handle one element of the vectors a and b

Unlike previous configurations, this launch configuration matches the vector size  $n$ , with a total of 100 threads covering all indices from 0 to 99. Each thread

computes one element of  $c[i]$ , resulting in complete and correct calculation across the entire vector. Thus, each element in  $c$  should be correctly computed as  $c[i] = a[i] + b[i]$

The program calculates the time taken for the kernel execution using `gettimeofday`, which measures the time from kernel launch to synchronization. The use of multiple blocks allows the GPU to more effectively manage thread resources and can improve efficiency compared to launching all threads in a single block, especially on devices with multiple multiprocessors.

Only every 10th value of  $c[i]$  is printed ( $i = 0, 10, 20, \dots$ ), making it easier to confirm that the computation is correct without printing all values. For a vector initialized with  $a[i] = i + 1$  and  $b[i] = i + 1$ ,  $c[i]$  should equal  $2*(i + 1)$  at each index.

The kernel configuration `vecAdd<<<2,50>>>(d_a, d_b, d_c, n);` efficiently maps the threads to the vector size. By dividing the work across multiple blocks, the program leverages parallelism while ensuring full computation across all elements. This setup likely provides better load balancing and resource utilization, especially for GPUs with more than one streaming multiprocessor, making the execution both complete and potentially faster.

In summary :

`vecAdd<<<1,100>>>` and `vecAdd<<<2,50>>>` configurations cover the entire vector, providing correct results, while `vecAdd<<<1,50>>>` fails to cover the vector fully.

`vecAdd<<<2,50>>>` divides the workload across multiple blocks, which is more efficient on GPUs with multiple multiprocessors, compared to `vecAdd<<<1,100>>>` which runs all threads in a single block.

The `vecAdd<<<2,50>>>` configuration scales better with larger vectors, as the GPU can handle more blocks effectively, while single-block configurations become a bottleneck as vector size increases