SANSKAAR PATNI
CSE C 23 180905134
PP LAB 5

1. Vector Addition
a. Using N blocks with 1 thread each

CODE

```
%%cu
#include <stdio.h>
#include <stdlib.h>
__global__ void vecAddKernel(int *A, int *B, int *C) {
int id=blockIdx.x;
C[id]=A[id]+B[id];
}
int main() {
// host copies of variables A, B & C
int N=5;
int A[N]={1,2,3,4,5};
int B[N]={6,7,8,9,10};
int C[N]={0,0,0,0,0};
// device copies of variables A, B & C
int *d_A, *d_B, *d_C;
// Allocate space for device copies of A, B, C
int size = N*sizeof(int);
cudaMalloc((void **)&d_A, size);
cudaMalloc((void **)&d_B, size);
cudaMalloc((void **)&d_C, size);

// Copy inputs to device
cudaMemcpy(d_A, &A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, &B, size, cudaMemcpyHostToDevice);
// Launch add() kernel on GPU
vecAddKernel<<<N,1>>>(d_A, d_B, d_C);
// Copy result back to host
cudaError err = cudaMemcpy(&C, d_C, size, cudaMemcpyDeviceToHost);
if(err!=cudaSuccess) {
printf("CUDA error copying to Host: %s\n", cudaGetErrorString(err));
}
printf("1a. Vector Addition using N= %d blocks with 1 thread each\n",N);
printf("Resultant Matrix C is\n");
for(int k=0;k<N;k++){
printf("%d ",C[k]);
}
// Cleanup
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
return 0;
}
```

SCREENSHOT

```
1a. Vector Addition using N= 5 blocks with 1 thread each
Resultant Matrix C is
7 9 11 13 15
```

```
%%cu
```

b. N threads in 1 block

CODE

```
%%cu
#include <stdio.h>
#include <stdlib.h>
__global__ void vecAddKernel(int *A, int *B, int *C) {
int id=threadIdx.x;
C[id]=A[id]+B[id];
}
int main() {
// host copies of variables A, B & C
int N=5;
int A[N]={1,2,3,4,5};
int B[N]={6,7,8,9,10};
int C[N]={0,0,0,0,0};
// device copies of variables A, B & C
int *d_A, *d_B, *d_C;
// Allocate space for device copies of A, B, C
int size = N*sizeof(int);
cudaMalloc((void **)&d_A, size);
cudaMalloc((void **)&d_B, size);
cudaMalloc((void **)&d_C, size);

// Copy inputs to device
cudaMemcpy(d_A, &A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, &B, size, cudaMemcpyHostToDevice);
// Launch add() kernel on GPU
vecAddKernel<<<1,N>>>(d_A, d_B, d_C);
// Copy result back to host
cudaError err = cudaMemcpy(&C, d_C, size, cudaMemcpyDeviceToHost);
if(err!=cudaSuccess) {
printf("CUDA error copying to Host: %s\n", cudaGetErrorString(err));
}
printf("1b. Vector Addition using N= %d threads within a block\n",N);
printf("Resultant Matrix C is\n");
for(int k=0;k<N;k++){
printf("%d ",C[k]);
}
// Cleanup
cudaFree(d_A);
```

```
cudaFree(d_B);
cudaFree(d_C);
return 0;
}
```

SCREENSHOT

```
1b. Vector Addition using N= 5 threads within a block
Resultant Matrix C is
7 9 11 13 15
```

c.Vector Addition using 256 threads per block and vary the number of blocks

CODE

```
%%cu
#include <stdio.h>
#include <stdlib.h>
__global__ void vecAddKernel(int *A, int *B, int *C,int N) {
int id=blockIdx.x * blockDim.x + threadIdx.x;
if(id < N)C[id]=A[id]+B[id];
}
int main() {
// host copies of variables A, B & C
int N=5;
int A[N]={1,2,3,4,5};
int B[N]={6,7,8,9,10};
int C[N]={0,0,0,0,0};
// device copies of variables A, B & C
int *d_A, *d_B, *d_C;
// Allocate space for device copies of A, B, C
int size = N*sizeof(int);
cudaMalloc((void **)&d_A, size);
cudaMalloc((void **)&d_B, size);
cudaMalloc((void **)&d_C, size);

// Copy inputs to device
cudaMemcpy(d_A, &A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, &B, size, cudaMemcpyHostToDevice);
// Launch add() kernel on GPU
vecAddKernel<<<ceil(N/256.0),256>>>(d_A, d_B, d_C, N);
// Copy result back to host
cudaError err = cudaMemcpy(&C, d_C, size, cudaMemcpyDeviceToHost);
if(err!=cudaSuccess) {
printf("CUDA error copying to Host: %s\n", cudaGetErrorString(err));
}
printf("1c. Vector Addition using 256 threads per block and vary the number
of blocks\n");
printf("Resultant Matrix C is\n");
for(int k=0;k<N;k++){
printf("%d ",C[k]);
}
```

```
// Cleanup
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
return 0;
}
```

SCREENSHOT

```
1c. Vector Addition using 256 threads per block and vary the number of blocks
Resultant Matrix C is
7 9 11 13 15
```

In the next two questions I've taken number of threads in each block as 2 and varied number of blocks (ceil(N/2.0)):

2. Parallel Selection Sort and store the result in another array

CODE

```
%%cu
#include <stdio.h>
#include <stdlib.h>
__global__ void parallelSort(int* a, int* b, int N){
int i = blockIdx.x * blockDim.x + threadIdx.x;
int pos = 0;
int data=a[i];
if(i<N){
for (int j = 0; j < N;j++)
if (a[j] < data || (a[j] == data && j < i))
pos++;
b[pos] = data;
}
}
int main() {
// host copies of variables A, B
int N=5;
int A[N]={2,5,4,3,1};
int B[N];
// device copies of variables A, B
int *d_A, *d_B;
// Allocate space for device copies of A, B
int size = N*sizeof(int);
cudaMalloc((void **)&d_A, size);
cudaMalloc((void **)&d_B, size);

// Copy inputs to device
cudaMemcpy(d_A, &A, size, cudaMemcpyHostToDevice);
// Launch add() kernel on GPU
parallelSort<<<ceil(N/2.0),2>>>(d_A, d_B, N);
// Copy result back to host
cudaError err = cudaMemcpy(&B, d_B, size, cudaMemcpyDeviceToHost);
```

```
if(err!=cudaSuccess) {
printf("CUDA error copying to Host: %s\n", cudaGetErrorString(err));
}
printf("2. Parallel Selection Sort\n");
printf("Initial array A elements:\n");
for(int k=0;k<N;k++){
printf("%d ",A[k]);
}
printf("\nResultant sorted array B elements:\n");
for(int k=0;k<N;k++){
printf("%d ",B[k]);
}
// Cleanup
cudaFree(d_A);
cudaFree(d_B);
return 0;
}
```

SCREENSHOT

```
2. Parallel Selection Sort
Initial array A elements:
2 5 4 3 1
Resultant sorted array B elements:
1 2 3 4 5
```

3.Odd-Even Transposition Sorting

CODE

```
%%cu
#include <stdio.h>
#include <stdlib.h>
__global__ void oddEven(int* a, int n)
{
int i = blockIdx.x * blockDim.x + threadIdx.x;
if(i<n){
if (i % 2 == 1 && i < n-1)
{
if (a[i] >= a[i + 1])
{
int t = a[i];
a[i] = a[i + 1];
a[i + 1] = t;
}
}
}
}
```

```
__global__ void evenOdd(int* a, int n)
{
int i = blockIdx.x * blockDim.x + threadIdx.x;
if(i<n){
if (i % 2 == 0 && i < n-1)
{
if (a[i] >= a[i + 1])
{
int t = a[i];
a[i] = a[i + 1];
a[i + 1] = t;
}
}
}
}

int main()
{
int N=5;
// host copies of variables A, B
int A[N]={2,5,4,3,1};
printf("3. Sort using Odd Even Transposition Sorting:\n");
printf("Initial Array A elements:\n");
for(int k=0;k<N;k++){
printf("%d ",A[k]);
}
int* d_A;
int size = N * sizeof(int);
cudaMalloc((void**)&d_A, size);
cudaMemcpy(d_A, &A, size, cudaMemcpyHostToDevice);
for(int i=0;i<N/2;i++){
oddEven <<<ceil(N/2.0), 2 >>> (d_A,N);
evenOdd <<<ceil(N/2.0), 2 >>> (d_A,N);
}
cudaMemcpy(&A, d_A, size, cudaMemcpyDeviceToHost);
printf("\nResultant sorted Array A:\n");
for (int i = 0;i < N;i++)
printf("%d ", A[i]);
return 0;
}
```

SCREENSHOT

```
3. Sort using Odd Even Transposition Sorting:
Initial Array A elements:
2 5 4 3 1
Resultant sorted Array A:
1 2 3 4 5
```