INDEX

EX-NO: 01 **DATE: 17.03.2023**

GPU BASED VECTOR SUMMATION

AIM:

(i) To modify or set the execution configuration of block.x as 1023 & 1024 and compare the elapsed time obtained on Host and GPU.

(ii) To set the number of threads as 256 and obtain the elapsed time on Host and GPU.

PROCEDURE:

1. Initialize the device and set the device properties.

2. Allocate memory on the host for input and output arrays.

3. Initialize input arrays with random values on the host.

4. Allocate memory on the device for input and output arrays, and copy input data from host to device.

5. Launch a CUDA kernel to perform vector addition on the device.

6. Copy output data from the device to the host and verify the results against the host's sequential vector addition. Free memory on the host and the device.

PROGRAM:

Developed By: H.DHAYANITHA
Reg.No: 212220230010

1. Block.x=1023

```
#include "common.h"
#include <cuda_runtime.h>
#include <stdio.h>

void checkResult(float *hostRef, float *gpuRef, const int N)
{
   double epsilon = 1.0E-8;
   bool match = 1;

   for (int i = 0; i < N; i++)
   {
      if (abs(hostRef[i] - gpuRef[i]) > epsilon)
```

```
        {
            match = 0;
            printf("Arrays do not match!\n");
            printf("host %5.2f gpu %5.2f at current %d\n", hostRef[i],
                  gpuRef[i], i);
            break;
        }
    }

    if (match) printf("Arrays match.\n\n");

    return;
}

void initialData(float *ip, int size)
{
    // generate different seed for random number
    time_t t;
    srand((unsigned) time(&t));

    for (int i = 0; i < size; i++)
    {
        ip[i] = (float)( rand() & 0xFF ) / 10.0f;
    }

    return;
}

void sumArraysOnHost(float *A, float *B, float *C, const int N)
{
    for (int idx = 0; idx < N; idx++)
    {
        C[idx] = A[idx] + B[idx];
    }
}
__global__ void sumArraysOnGPU(float *A, float *B, float *C, const int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < N) C[i] = A[i] + B[i];
}
```

```
int main(int argc, char **argv)
{
    printf("%s Starting...\n", argv[0]);

    // set up device
    int dev = 0;
    cudaDeviceProp deviceProp;
    CHECK(cudaGetDeviceProperties(&deviceProp, dev));
    printf("Using Device %d: %s\n", dev, deviceProp.name);
    CHECK(cudaSetDevice(dev));

    // set up data size of vectors
    int nElem = 1 << 27;
    printf("Vector size %d\n", nElem);

    // malloc host memory
    size_t nBytes = nElem * sizeof(float);

    float *h_A, *h_B, *hostRef, *gpuRef;
    h_A     = (float *)malloc(nBytes);
    h_B     = (float *)malloc(nBytes);
    hostRef = (float *)malloc(nBytes);
    gpuRef  = (float *)malloc(nBytes);

    double iStart, iElaps;

    // initialize data at host side
    iStart = seconds();
    initialData(h_A, nElem);
    initialData(h_B, nElem);
    iElaps = seconds() - iStart;
    printf("initialData Time elapsed %f sec\n", iElaps);
    memset(hostRef, 0, nBytes);
    memset(gpuRef,  0, nBytes);

    // add vector at host side for result checks
    iStart = seconds();
    sumArraysOnHost(h_A, h_B, hostRef, nElem);
    iElaps = seconds() - iStart;
    printf("sumArraysOnHost Time elapsed %f sec\n", iElaps);

    // malloc device global memory
    float *d_A, *d_B, *d_C;
    CHECK(cudaMalloc((float**)&d_A, nBytes));
```

```
    CHECK(cudaMalloc((float**)&d_B, nBytes));
    CHECK(cudaMalloc((float**)&d_C, nBytes));

    // transfer data from host to device
    CHECK(cudaMemcpy(d_A, h_A, nBytes, cudaMemcpyHostToDevice));
    CHECK(cudaMemcpy(d_B, h_B, nBytes, cudaMemcpyHostToDevice));
    CHECK(cudaMemcpy(d_C, gpuRef, nBytes, cudaMemcpyHostToDevice));

    // invoke kernel at host side
    int iLen = 1023;
    dim3 block (iLen);
    dim3 grid  ((nElem + block.x - 1) / block.x);

    iStart = seconds();
    sumArraysOnGPU<<<grid, block>>>(d_A, d_B, d_C, nElem);
    CHECK(cudaDeviceSynchronize());
    iElaps = seconds() - iStart;
    printf("sumArraysOnGPU <<<  %d, %d  >>>  Time elapsed %f sec\n", grid.x,
          block.x, iElaps);

    // check kernel error
    CHECK(cudaGetLastError()) ;

    // copy kernel result back to host side
    CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));

    // check device results
    checkResult(hostRef, gpuRef, nElem);

    // free device global memory
    CHECK(cudaFree(d_A));
    CHECK(cudaFree(d_B));
    CHECK(cudaFree(d_C));

    // free host memory
    free(h_A);
    free(h_B);
    free(hostRef);
    free(gpuRef);

    return(0);
}
```

2. Block.x=1024

```
#include "common.h"
#include <cuda_runtime.h>
#include <stdio.h>

void checkResult(float *hostRef, float *gpuRef, const int N)
{
   double epsilon = 1.0E-8;
   bool match = 1;

   for (int i = 0; i < N; i++)
   {
      if (abs(hostRef[i] - gpuRef[i]) > epsilon)
      {
         match = 0;
         printf("Arrays do not match!\n");
         printf("host %5.2f gpu %5.2f at current %d\n", hostRef[i],
               gpuRef[i], i);
         break;
      }
   }

   if (match) printf("Arrays match.\n\n");

   return;
}

void initialData(float *ip, int size)
{
   // generate different seed for random number
   time_t t;
   srand((unsigned) time(&t));

   for (int i = 0; i < size; i++)
   {
      ip[i] = (float)( rand() & 0xFF ) / 10.0f;
   }

   return;
}

void sumArraysOnHost(float *A, float *B, float *C, const int N)
{
```

```
    for (int idx = 0; idx < N; idx++)
    {
        C[idx] = A[idx] + B[idx];
    }
}
__global__ void sumArraysOnGPU(float *A, float *B, float *C, const int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < N) C[i] = A[i] + B[i];
}

int main(int argc, char **argv)
{
    printf("%s Starting...\n", argv[0]);

    // set up device
    int dev = 0;
    cudaDeviceProp deviceProp;
    CHECK(cudaGetDeviceProperties(&deviceProp, dev));
    printf("Using Device %d: %s\n", dev, deviceProp.name);
    CHECK(cudaSetDevice(dev));

    // set up data size of vectors
    int nElem = 1 << 27;
    printf("Vector size %d\n", nElem);

    // malloc host memory
    size_t nBytes = nElem * sizeof(float);

    float *h_A, *h_B, *hostRef, *gpuRef;
    h_A     = (float *)malloc(nBytes);
    h_B     = (float *)malloc(nBytes);
    hostRef = (float *)malloc(nBytes);
    gpuRef  = (float *)malloc(nBytes);

    double iStart, iElaps;

    // initialize data at host side
    iStart = seconds();
    initialData(h_A, nElem);
    initialData(h_B, nElem);
    iElaps = seconds() - iStart;
    printf("initialData Time elapsed %f sec\n", iElaps);
```

```
memset(hostRef, 0, nBytes);
memset(gpuRef,  0, nBytes);

// add vector at host side for result checks
iStart = seconds();
sumArraysOnHost(h_A, h_B, hostRef, nElem);
iElaps = seconds() - iStart;
printf("sumArraysOnHost Time elapsed %f sec\n", iElaps);

// malloc device global memory
float *d_A, *d_B, *d_C;
CHECK(cudaMalloc((float**)&d_A, nBytes));
CHECK(cudaMalloc((float**)&d_B, nBytes));
CHECK(cudaMalloc((float**)&d_C, nBytes));

// transfer data from host to device
CHECK(cudaMemcpy(d_A, h_A, nBytes, cudaMemcpyHostToDevice));
CHECK(cudaMemcpy(d_B, h_B, nBytes, cudaMemcpyHostToDevice));
CHECK(cudaMemcpy(d_C, gpuRef, nBytes, cudaMemcpyHostToDevice));

// invoke kernel at host side
int iLen = 1024;
dim3 block (iLen);
dim3 grid  ((nElem + block.x - 1) / block.x);

iStart = seconds();
sumArraysOnGPU<<<grid, block>>>(d_A, d_B, d_C, nElem);
CHECK(cudaDeviceSynchronize());
iElaps = seconds() - iStart;
printf("sumArraysOnGPU <<<  %d, %d  >>>  Time elapsed %f sec\n", grid.x,
      block.x, iElaps);

// check kernel error
CHECK(cudaGetLastError()) ;

// copy kernel result back to host side
CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));

// check device results
checkResult(hostRef, gpuRef, nElem);

// free device global memory
CHECK(cudaFree(d_A));
CHECK(cudaFree(d_B));
```

```
    CHECK(cudaFree(d_C));

    // free host memory
    free(h_A);
    free(h_B);
    free(hostRef);
    free(gpuRef);

    return(0);
}
```

3. Block.x=256

```
#include "common.h"
#include <cuda_runtime.h>
#include <stdio.h>

void checkResult(float *hostRef, float *gpuRef, const int N)
{
    double epsilon = 1.0E-8;
    bool match = 1;

    for (int i = 0; i < N; i++)
    {
        if (abs(hostRef[i] - gpuRef[i]) > epsilon)
        {
            match = 0;
            printf("Arrays do not match!\n");
            printf("host %5.2f gpu %5.2f at current %d\n", hostRef[i],
                   gpuRef[i], i);
            break;
        }
    }

    if (match) printf("Arrays match.\n\n");

    return;
}

void initialData(float *ip, int size)
{
    // generate different seed for random number
    time_t t;
    srand((unsigned) time(&t));
```

```
    for (int i = 0; i < size; i++)
    {
        ip[i] = (float)( rand() & 0xFF ) / 10.0f;
    }

    return;
}

void sumArraysOnHost(float *A, float *B, float *C, const int N)
{
    for (int idx = 0; idx < N; idx++)
    {
        C[idx] = A[idx] + B[idx];
    }
}
__global__ void sumArraysOnGPU(float *A, float *B, float *C, const int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < N) C[i] = A[i] + B[i];
}

int main(int argc, char **argv)
{
    printf("%s Starting...\n", argv[0]);

    // set up device
    int dev = 0;
    cudaDeviceProp deviceProp;
    CHECK(cudaGetDeviceProperties(&deviceProp, dev));
    printf("Using Device %d: %s\n", dev, deviceProp.name);
    CHECK(cudaSetDevice(dev));

    // set up data size of vectors
    int nElem = 1 << 24;
    printf("Vector size %d\n", nElem);

    // malloc host memory
    size_t nBytes = nElem * sizeof(float);

    float *h_A, *h_B, *hostRef, *gpuRef;
    h_A     = (float *)malloc(nBytes);
    h_B     = (float *)malloc(nBytes);
```

```
hostRef = (float *)malloc(nBytes);
gpuRef  = (float *)malloc(nBytes);

double iStart, iElaps;

// initialize data at host side
iStart = seconds();
initialData(h_A, nElem);
initialData(h_B, nElem);
iElaps = seconds() - iStart;
printf("initialData Time elapsed %f sec\n", iElaps);
memset(hostRef, 0, nBytes);
memset(gpuRef,  0, nBytes);

// add vector at host side for result checks
iStart = seconds();
sumArraysOnHost(h_A, h_B, hostRef, nElem);
iElaps = seconds() - iStart;
printf("sumArraysOnHost Time elapsed %f sec\n", iElaps);

// malloc device global memory
float *d_A, *d_B, *d_C;
CHECK(cudaMalloc((float**)&d_A, nBytes));
CHECK(cudaMalloc((float**)&d_B, nBytes));
CHECK(cudaMalloc((float**)&d_C, nBytes));

// transfer data from host to device
CHECK(cudaMemcpy(d_A, h_A, nBytes, cudaMemcpyHostToDevice));
CHECK(cudaMemcpy(d_B, h_B, nBytes, cudaMemcpyHostToDevice));
CHECK(cudaMemcpy(d_C, gpuRef, nBytes, cudaMemcpyHostToDevice));

// invoke kernel at host side
int iLen = 256;
dim3 block (iLen);
dim3 grid  ((nElem + block.x - 1) / block.x);

iStart = seconds();
sumArraysOnGPU<<<grid, block>>>(d_A, d_B, d_C, nElem);
CHECK(cudaDeviceSynchronize());
iElaps = seconds() - iStart;
printf("sumArraysOnGPU <<<  %d, %d  >>>  Time elapsed %f sec\n", grid.x,
      block.x, iElaps);
```

```
    // check kernel error
    CHECK(cudaGetLastError()) ;

    // copy kernel result back to host side
    CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));

    // check device results
    checkResult(hostRef, gpuRef, nElem);

    // free device global memory
    CHECK(cudaFree(d_A));
    CHECK(cudaFree(d_B));
    CHECK(cudaFree(d_C));

    // free host memory
    free(h_A);
    free(h_B);
    free(hostRef);
    free(gpuRef);

    return(0);
}
```

OUTPUT:

1. Block.x=1023

```
(base) student@SAV-MLSystem:~/Downloads/CodeSamples/chapter02$ nvcc sumArraysOnGPU-timer.cu
(base) student@SAV-MLSystem:~/Downloads/CodeSamples/chapter02$ ./a.out
./a.out Starting...
Using Device 0: NVIDIA GeForce GT 710
Vector size 16777216
initialData Time elapsed 0.413154 sec
sumArraysOnHost Time elapsed 0.034583 sec
sumArraysOnGPU <<<  16401, 1023  >>>  Time elapsed 0.019774 sec
Arrays match.
```

2.
Block.x=1024

```
(base) student@SAV-MLSystem:~/Downloads/CodeSamples/chapter02$ nvcc sumArraysOnGPU-timer.cu
(base) student@SAV-MLSystem:~/Downloads/CodeSamples/chapter02$ ./a.out
./a.out Starting...
Using Device 0: NVIDIA GeForce GT 710
Vector size 16777216
initialData Time elapsed 0.416176 sec
sumArraysOnHost Time elapsed 0.033751 sec
sumArraysOnGPU <<<  16384, 1024  >>>  Time elapsed 0.022122 sec
Arrays match.

(base) student@SAV-MLSystem:~/Downloads/CodeSamples/chapter02$
```

3. Block.x=256

```
(base) student@SAV-MLSystem:~/Downloads/CodeSamples/chapter02$ nvcc sumArraysOnGPU-timer.cu
(base) student@SAV-MLSystem:~/Downloads/CodeSamples/chapter02$ ./a.out
./a.out Starting...
Using Device 0: NVIDIA GeForce GT 710
Vector size 16777216
initialData Time elapsed 0.415861 sec
sumArraysOnHost Time elapsed 0.034865 sec
sumArraysOnGPU <<<  65536, 256  >>>  Time elapsed 0.021174 sec
Arrays match.
```

RESULT:

The block size 1023 performs better in the GPU with an elapsed time of 0.0197 seconds, and the block size 1024 shows better results in the host with an elapsed time of 0.022 seconds. Using a block size of 256 and two threads simultaneously has provided the best results in the GPU with an elapsed time of 0.021 seconds. Thus, the differences between the execution configurations of GPU based vector summation has been explored successfully.

EX-N0: 02 **DATE: 31.03.2023**

MATRIX SUMMATION WITH A 2D GRID AND 2D BLOCKS

AIM:

To perform matrix summation with a 2D grid and 2D blocks and adapting it to integer matrix addition.

PROCEDURE:

1. Include the required files and library.

2. Declare a function sumMatrixOnHost , to perform matrix summation on the host side . Declare three matrix A , B , C . Store the resultant matrix in C.

3. Declare a function with __ global __ , which is a CUDA C keyword , to execute the function to perform matrix summation on GPU .

4. Declare Main method/function .

5. In the Main function Set up device and data size of matrix ,Allocate Host Memory and device global memory, Initialize data at host side and then add matrix at host side ,transfer data from host to device.

6. Invoke kernel at host side , check for kernel error and copy kernel result back to host side.

7. Finally Free device global memory, host memory and reset device.

8. Save and Run the Program.

PROGRAM:

Developed By: H.DHAYANITHA
Reg.No: 212220230010

```
#include "common.h"
#include <cuda_runtime.h>
#include <stdio.h>

void initialData(int *ip, const int size)
{
   int i;
   for(i = 0; i < size; i++)
   {
      ip[i] = (int)(rand() & 0xFF) / 10.0f;
```

```
    }
    return;
}

void sumMatrixOnHost(int *A, int *B, int *C, const int nx,const int ny)
{
    int *ia = A;
    int *ib = B;
    int *ic = C;

    for (int iy = 0; iy < ny; iy++)
    {
        for (int ix = 0; ix < nx; ix++)
        {
            ic[ix] = ia[ix] + ib[ix];

        }

        ia += nx;
        ib += nx;
        ic += nx;
    }

    return;
}


void checkResult(int *hostRef, int *gpuRef, const int N)
{
    double epsilon = 1.0E-8;
    bool match = 1;

    for (int i = 0; i < N; i++)
    {
        if (abs(hostRef[i] - gpuRef[i]) > epsilon)
        {
            match = 0;
            printf("host %d gpu %d\n", hostRef[i], gpuRef[i]);
            break;
        }
    }

    if (match)
        printf("Arrays match.\n\n");
```

```
    else
        printf("Arrays do not match.\n\n");
}
// grid 2D block 2D
__global__ void sumMatrixOnGPU2D(int *MatA, int *MatB, int *MatC, int nx,int ny)
{
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y;
    unsigned int idx = iy * nx + ix;

    if (ix < nx && iy < ny)
        MatC[idx] = MatA[idx] + MatB[idx];
}

int main(int argc, char **argv)
{
    printf("%s Starting...\n", argv[0]);

    // set up device
    int dev = 0;
    cudaDeviceProp deviceProp;
    CHECK(cudaGetDeviceProperties(&deviceProp, dev));
    printf("Using Device %d: %s\n", dev, deviceProp.name);
    CHECK(cudaSetDevice(dev));

    // set up data size of matrix
    int nx = 1 << 14;
    int ny = 1 << 14;

    int nxy = nx * ny;
    int nBytes = nxy * sizeof(int);
    printf("Matrix size: nx %d ny %d\n", nx, ny);

    // malloc host memory
    int *h_A, *h_B, *hostRef, *gpuRef;
    h_A = (int *)malloc(nBytes);
    h_B = (int *)malloc(nBytes);
    hostRef = (int *)malloc(nBytes);
    gpuRef = (int *)malloc(nBytes);

    // initialize data at host side
    double iStart = seconds();
    initialData(h_A, nxy);
    initialData(h_B, nxy);
```

```
double iElaps = seconds() - iStart;
printf("Matrix initialization elapsed %f sec\n", iElaps);

memset(hostRef, 0, nBytes);
memset(gpuRef, 0, nBytes);

// add matrix at host side for result checks
iStart = seconds();
sumMatrixOnHost(h_A, h_B, hostRef, nx, ny);
iElaps = seconds() - iStart;
printf("sumMatrixOnHost elapsed %f sec\n", iElaps);

// malloc device global memory
int *d_MatA, *d_MatB, *d_MatC;
CHECK(cudaMalloc((void **)&d_MatA, nBytes));
CHECK(cudaMalloc((void **)&d_MatB, nBytes));
CHECK(cudaMalloc((void **)&d_MatC, nBytes));

// transfer data from host to device
CHECK(cudaMemcpy(d_MatA, h_A, nBytes, cudaMemcpyHostToDevice));
CHECK(cudaMemcpy(d_MatB, h_B, nBytes, cudaMemcpyHostToDevice));

// invoke kernel at host side
int dimx = 32;
int dimy = 32;
dim3 block(dimx, dimy);
dim3 grid((nx + block.x - 1) / block.x, (ny + block.y - 1) / block.y);

iStart = seconds();
sumMatrixOnGPU2D<<<grid, block>>>(d_MatA, d_MatB, d_MatC, nx, ny);
CHECK(cudaDeviceSynchronize());
iElaps = seconds() - iStart;
printf("sumMatrixOnGPU2D <<<(%d,%d), (%d,%d)>>> elapsed %f sec\n", grid.x,
      grid.y,
      block.x, block.y, iElaps);
// check kernel error
CHECK(cudaGetLastError());

// copy kernel result back to host side
CHECK(cudaMemcpy(gpuRef, d_MatC, nBytes, cudaMemcpyDeviceToHost));

// check device results
checkResult(hostRef, gpuRef, nxy);
```

```
    // free device global memory
    CHECK(cudaFree(d_MatA));
    CHECK(cudaFree(d_MatB));
    CHECK(cudaFree(d_MatC));

    // free host memory
    free(h_A);
    free(h_B);
    free(hostRef);
    free(gpuRef);

    // reset device
    CHECK(cudaDeviceReset());

    return (0);
}
```

OUTPUT:

```
root@MidPC:/home/student/Desktop# nvcc first.cu
root@MidPC:/home/student/Desktop# ./a.out
./a.out Starting...
Using Device 0: NVIDIA GeForce GTX 1660 SUPER
Matrix size: nx 16384 ny 16384
Matrix initialization elapsed 6.338138 sec
sumMatrixOnHost elapsed 0.884061 sec
sumMatrixOnGPU2D <<<(512,512), (32,32)>>> elapsed 0.012146 sec
Arrays match.
```

Matrix initialization : 6.338138 sec.
Sum matrix on Host : 0.884061 sec.
Sum matrix on GPU2D : 0.012146 sec

RESULT:

The host took 0.884061 seconds to complete it's computation, while the GPU outperforms the host and completes the computation in 0.012146 seconds. Therefore, float variables in the GPU will result in the best possible result. Thus, matrix summation using 2D grids and 2D blocks has been performed successfully.

EX-NO: 03 **DATE: 21.04.2023**

SIMPLE WARP DIVERGENCE: SUM REDUCTION

AIM:

To implement the kernel reduceUnrolling16 and comapare the performance of kernal reduceUnrolling16 with kernal reduceUnrolling8 using proper metrics and events with nvprof.

PROCEDURE FOR UNROLLIN8:

1. Initialize the input size n and allocate host memory (h_idata and h_odata) for input and output data.

2. Initialize the input data on the host by assigning a value of 1 to each element in h_idata.

3. Allocate device memory (d_idata and d_odata) for input and output data on the GPU.

4. Copy the input data from the host to the device using cudaMemcpy.

5. Define the grid and block dimensions for the kernel launch. Each block will contain 256 threads, and the grid size will be calculated based on the input size n and block size.

6. Start the CPU timer to measure the CPU execution time.

7. Compute the sum of input data on the CPU using a for loop and store the result in sum_cpu.

---

8. Stop the CPU timer and calculate the elapsed CPU time.

9. Start the GPU timer to measure the GPU execution time.

10. Launch the reduceUnrolling8 kernel on the GPU with the specified grid and block dimensions.

11. Copy the result data from the device to the host using cudaMemcpy.

12. Compute the final sum on the GPU by adding up the elements in h_odata and store the result in sum_gpu.

13. Stop the GPU timer and calculate the elapsed GPU time.

14. Print the results: CPU sum, GPU sum, CPU elapsed time, and GPU elapsed time.

15. Free the allocated host and device memory using free and cudaFree.

16. Return from the main function.

PROGRAM:

Developed By : H.DHAYANITHA
Reg No : 212220230010

U8.cu

```
#include "common.h"
#include <cuda_runtime.h>
#include <stdio.h>

__global__ void reduceUnrolling8(int *g_idata, int *g_odata, unsigned int n)
{
   // Set thread ID
   unsigned int tid = threadIdx.x;
   unsigned int idx = blockIdx.x * blockDim.x * 8 + threadIdx.x;

   // Convert global data pointer to the local pointer of this block
   int *idata = g_idata + blockIdx.x * blockDim.x * 8;

   // Unrolling 8
   if (idx + 7 * blockDim.x < n)
   {
      int a1 = g_idata[idx];
      int a2 = g_idata[idx + blockDim.x];
      int a3 = g_idata[idx + 2 * blockDim.x];
      int a4 = g_idata[idx + 3 * blockDim.x];
      int b1 = g_idata[idx + 4 * blockDim.x];
      int b2 = g_idata[idx + 5 * blockDim.x];
      int b3 = g_idata[idx + 6 * blockDim.x];
      int b4 = g_idata[idx + 7 * blockDim.x];
      g_idata[idx] = a1 + a2 + a3 + a4 + b1 + b2 + b3 + b4;
   }

   __syncthreads();

   // In-place reduction in global memory
   for (int stride = blockDim.x / 2; stride > 0; stride >>= 1)
   {
      if (tid < stride)
      {
```

```
            idata[tid] += idata[tid + stride];
        }

        // Synchronize within threadblock
        __syncthreads();
    }

    // Write result for this block to global memory
    if (tid == 0)
    {
        g_odata[blockIdx.x] = idata[0];
    }
}



// Function to calculate elapsed time in milliseconds
double getElapsedTime(struct timeval start, struct timeval end)
{
    long seconds = end.tv_sec - start.tv_sec;
    long microseconds = end.tv_usec - start.tv_usec;
    double elapsed = seconds + microseconds / 1e6;
    return elapsed * 1000; // Convert to milliseconds
}

int main()
{
    // Input size and host memory allocation
    unsigned int n = 1 << 20; // 1 million elements
    size_t size = n * sizeof(int);
    int *h_idata = (int *)malloc(size);
    int *h_odata = (int *)malloc(size);

    // Initialize input data on the host
    for (unsigned int i = 0; i < n; i++)
    {
        h_idata[i] = 1;
    }

    // Device memory allocation
    int *d_idata, *d_odata;
    cudaMalloc((void **)&d_idata, size);
    cudaMalloc((void **)&d_odata, size);
```

```
// Copy input data from host to device
cudaMemcpy(d_idata, h_idata, size, cudaMemcpyHostToDevice);

// Define grid and block dimensions
dim3 blockSize(256); // 256 threads per block
dim3 gridSize((n + blockSize.x * 8 - 1) / (blockSize.x * 8));

// Start CPU timer
struct timeval start_cpu, end_cpu;
gettimeofday(&start_cpu, NULL);

// Compute the sum on the CPU
int sum_cpu = 0;
for (unsigned int i = 0; i < n; i++)
{
   sum_cpu += h_idata[i];
}

// Stop CPU timer
gettimeofday(&end_cpu, NULL);
double elapsedTime_cpu = getElapsedTime(start_cpu, end_cpu);

// Start GPU timer
struct timeval start_gpu, end_gpu;
gettimeofday(&start_gpu, NULL);

// Launch the reduction kernel
reduceUnrolling8<<<gridSize, blockSize>>>(d_idata, d_odata, n);

// Copy the result from device to host
cudaMemcpy(h_odata, d_odata, size, cudaMemcpyDeviceToHost);

// Compute the final sum on the GPU
int sum_gpu = 0;
for (unsigned int i = 0; i < gridSize.x; i++)
{
   sum_gpu += h_odata[i];
}

// Stop GPU timer
gettimeofday(&end_gpu, NULL);
double elapsedTime_gpu = getElapsedTime(start_gpu, end_gpu);

// Print the results and elapsed times
```

```
    printf("CPU Sum: %d\n", sum_cpu);
    printf("GPU Sum: %d\n", sum_gpu);
    printf("CPU Elapsed Time: %.2f ms\n", elapsedTime_cpu);
    printf("GPU Elapsed Time: %.2f ms\n", elapsedTime_gpu);

    // Free memory
    free(h_idata);
    free(h_odata);
    cudaFree(d_idata);
    cudaFree(d_odata);

    return 0;
}
```

OUTPUT:

```
root@MidPC:/home/student/Desktop/19AI407/EXPERIMENTED# nvcc U8.cu
root@MidPC:/home/student/Desktop/19AI407/EXPERIMENTED# ./a.out
CPU Sum: 1048576
GPU Sum: 1048576
CPU Elapsed Time: 1.80 ms
GPU Elapsed Time: 1.21 ms
root@MidPC:/home/student/Desktop/19AI407/EXPERIMENTED# nvprof ./a.out
==6409== NVPROF is profiling process 6409, command: ./a.out
CPU Sum: 1048576
GPU Sum: 1048576
CPU Elapsed Time: 1.82 ms
GPU Elapsed Time: 1.34 ms
==6409== Profiling application: ./a.out
==6409== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   60.94%  777.22us         1  777.22us  777.22us  777.22us  [CUDA memcpy DtoH]
                   37.61%  479.71us         1  479.71us  479.71us  479.71us  [CUDA memcpy HtoD]
                    1.45%  18.464us         1  18.464us  18.464us  18.464us  reduceUnrolling8(int*, int*, unsigned int)
      API calls:   98.26%  138.04ms         2  69.020ms  72.230us  137.97ms  cudaMalloc
                    1.32%  1.8489ms         2  924.46us  527.07us  1.3218ms  cudaMemcpy
                    0.15%  210.32us        97  2.1680us     170ns  90.690us  cuDeviceGetAttribute
                    0.14%  193.89us         1  193.89us  193.89us  193.89us  cuDeviceTotalMem
                    0.10%  135.21us         2  67.604us  47.790us  87.419us  cudaFree
                    0.02%  26.800us         1  26.800us  26.800us  26.800us  cuDeviceGetName
                    0.01%  18.600us         1  18.600us  18.600us  18.600us  cudaLaunchKernel
                    0.00%  4.8400us         1  4.8400us  4.8400us  4.8400us  cuDeviceGetPCIBusId
                    0.00%  1.3000us         3     433ns     190ns     870ns  cuDeviceGetCount
                    0.00%     790ns         2     395ns     180ns     610ns  cuDeviceGet
                    0.00%     240ns         1     240ns     240ns     240ns  cuDeviceGetUuid
root@MidPC:/home/student/Desktop/19AI407/EXPERIMENTED#
```

PROCEDURE FOR UNROLLING16:

1. Initialize the input size n and allocate host memory (h_idata and h_odata) for input and output data.

2. Initialize the input data on the host by assigning a value of 1 to each element in h_idata.

3. Allocate device memory (d_idata and d_odata) for input and output data on the GPU.

4. Copy the input data from the host to the device using cudaMemcpy.

5. Define the grid and block dimensions for the kernel launch. Each block will contain 256 threads, and the grid size will be calculated based on the input size n and block size.

6. Start the CPU timer to measure the CPU execution time.

7. Compute the sum of input data on the CPU using a for loop and store the result in sum_cpu. -----

8. Stop the CPU timer and calculate the elapsed CPU time.

9. Start the GPU timer to measure the GPU execution time.

10. Launch the reduceUnrolling16 kernel on the GPU with the specified grid and block dimensions.

11. Copy the result data from the device to the host using cudaMemcpy.

12. Compute the final sum on the GPU by adding up the elements in h_odata and store the result in sum_gpu.

13. Stop the GPU timer and calculate the elapsed GPU time.

14. Print the results: CPU sum, GPU sum, CPU elapsed time, and GPU elapsed time.

15. Free the allocated host and device memory using free and cudaFree.

16. Return from the main function..

PROGRAM:

U16.cu

```
#include "common.h"
#include <cuda_runtime.h>
#include <stdio.h>
#include <sys/time.h>
// Kernel function declaration
__global__ void reduceUnrolling16(int *g_idata, int *g_odata, unsigned int n);
// Function to calculate elapsed time in milliseconds
double getElapsedTime(struct timeval start, struct timeval end)
{
   long seconds = end.tv_sec - start.tv_sec;
   long microseconds = end.tv_usec - start.tv_usec;
   double elapsed = seconds + microseconds / 1e6;
```

```
    return elapsed * 1000; // Convert to milliseconds
}
int main()
{
    // Input size and host memory allocation
    unsigned int n = 1 << 20; // 1 million elements
    size_t size = n * sizeof(int);
    int *h_idata = (int *)malloc(size);
    int *h_odata = (int *)malloc(size);

    // Initialize input data on the host
    for (unsigned int i = 0; i < n; i++)
    {
        h_idata[i] = 1;
    }

    // Device memory allocation
    int *d_idata, *d_odata;
    cudaMalloc((void **)&d_idata, size);
    cudaMalloc((void **)&d_odata, size);

    // Copy input data from host to device
    cudaMemcpy(d_idata, h_idata, size, cudaMemcpyHostToDevice);

    // Define grid and block dimensions
    dim3 blockSize(256); // 256 threads per block
    dim3 gridSize((n + blockSize.x * 16 - 1) / (blockSize.x * 16));

    // Start CPU timer
    struct timeval start_cpu, end_cpu;
    gettimeofday(&start_cpu, NULL);

    // Compute the sum on the CPU
    int sum_cpu = 0;
    for (unsigned int i = 0; i < n; i++)
    {
        sum_cpu += h_idata[i];
    }

    // Stop CPU timer
    gettimeofday(&end_cpu, NULL);
    double elapsedTime_cpu = getElapsedTime(start_cpu, end_cpu);

    // Start GPU timer
```

```
    struct timeval start_gpu, end_gpu;
    gettimeofday(&start_gpu, NULL);

    // Launch the reduction kernel
    reduceUnrolling16<<<gridSize, blockSize>>>(d_idata, d_odata, n);

    // Copy the result from device to host
    cudaMemcpy(h_odata, d_odata, size, cudaMemcpyDeviceToHost);

    // Compute the final sum on the GPU
    int sum_gpu = 0;
    for (unsigned int i = 0; i < gridSize.x; i++)
    {
        sum_gpu += h_odata[i];
    }

    // Stop GPU timer
    gettimeofday(&end_gpu, NULL);
    double elapsedTime_gpu = getElapsedTime(start_gpu, end_gpu);

    // Print the results and elapsed times
    printf("CPU Sum: %d\n", sum_cpu);
    printf("GPU Sum: %d\n", sum_gpu);
    printf("CPU Elapsed Time: %.2f ms\n", elapsedTime_cpu);
    printf("GPU Elapsed Time: %.2f ms\n", elapsedTime_gpu);

    // Free memory
    free(h_idata);
    free(h_odata);
    cudaFree(d_idata);
    cudaFree(d_odata);

    return 0;
}

__global__ void reduceUnrolling16(int *g_idata, int *g_odata, unsigned int n)
{
    // Set thread ID
    unsigned int tid = threadIdx.x;
    unsigned int idx = blockIdx.x * blockDim.x * 16 + threadIdx.x;

    // Convert global data pointer to the local pointer of this block
    int *idata = g_idata + blockIdx.x * blockDim.x * 16;
```

```
    // Unrolling 16
    if (idx + 15 * blockDim.x < n)
    {
        int a1 = g_idata[idx];
        int a2 = g_idata[idx + blockDim.x];
        int a3 = g_idata[idx + 2 * blockDim.x];
        int a4 = g_idata[idx + 3 * blockDim.x];
        int a5 = g_idata[idx + 4 * blockDim.x];
        int a6 = g_idata[idx + 5 * blockDim.x];
        int a7 = g_idata[idx + 6 * blockDim.x];
        int a8 = g_idata[idx + 7 * blockDim.x];
        int b1 = g_idata[idx + 8 * blockDim.x];
        int b2 = g_idata[idx + 9 * blockDim.x];
        int b3 = g_idata[idx + 10 * blockDim.x];
        int b4 = g_idata[idx + 11 * blockDim.x];
        int b5 = g_idata[idx + 12 * blockDim.x];
        int b6 = g_idata[idx + 13 * blockDim.x];
        int b7 = g_idata[idx + 14 * blockDim.x];
        int b8 = g_idata[idx + 15 * blockDim.x];
        g_idata[idx] = a1 + a2 + a3 + a4 + a5 + a6 + a7 + a8 + b1 + b2 + b3 + b4 + b5 + b6 + b7
+ b8;
    }

    __syncthreads();

    // In-place reduction in global memory
    for (int stride = blockDim.x / 2; stride > 0; stride >>= 1)
    {
        if (tid < stride)
        {
            idata[tid] += idata[tid + stride];
        }

        // Synchronize within thread block
        __syncthreads();
    }

    // Write result for this block to global memory
    if (tid == 0)
    {
        g_odata[blockIdx.x] = idata[0];
    }
}
```

OUTPUT:

```
root@MidPC:/home/student/Desktop/19AI407/EXPERIMENTED# nvcc U16.cu
root@MidPC:/home/student/Desktop/19AI407/EXPERIMENTED# ./a.out
CPU Sum: 1048576
GPU Sum: 1048576
CPU Elapsed Time: 1.80 ms
GPU Elapsed Time: 1.49 ms
root@MidPC:/home/student/Desktop/19AI407/EXPERIMENTED# nvprof ./a.out
==6612== NVPROF is profiling process 6612, command: ./a.out
CPU Sum: 1048576
GPU Sum: 1048576
CPU Elapsed Time: 1.81 ms
GPU Elapsed Time: 1.48 ms
==6612== Profiling application: ./a.out
==6612== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   65.83%  862.43us         1  862.43us  862.43us  862.43us  [CUDA memcpy DtoH]
                   32.80%  429.66us         1  429.66us  429.66us  429.66us  [CUDA memcpy HtoD]
                    1.37%  17.920us         1  17.920us  17.920us  17.920us  reduceUnrolling16(int*, int*, unsigned int
)
      API calls:   98.32%  148.05ms         2  74.025ms  77.199us  147.97ms  cudaMalloc
                    1.27%  1.9105ms         2  955.23us  454.45us  1.4560ms  cudaMemcpy
                    0.15%  218.48us        97  2.2520us     180ns  95.719us  cuDeviceGetAttribute
                    0.12%  187.43us         1  187.43us  187.43us  187.43us  cuDeviceTotalMem
                    0.10%  145.05us         2  72.524us  48.520us  96.529us  cudaFree
                    0.02%  32.699us         1  32.699us  32.699us  32.699us  cuDeviceGetName
                    0.01%  19.960us         1  19.960us  19.960us  19.960us  cudaLaunchKernel
                    0.00%  5.1700us         1  5.1700us  5.1700us  5.1700us  cuDeviceGetPCIBusId
                    0.00%  2.5400us         3     846ns     190ns  1.7500us  cuDeviceGetCount
                    0.00%     940ns         2     470ns     210ns     730ns  cuDeviceGet
                    0.00%     250ns         1     250ns     250ns     250ns  cuDeviceGetUuid
root@MidPC:/home/student/Desktop/19AI407/EXPERIMENTED#
```

RESULT:

Thus the program has been executed by unrolling by 8 and unrolling by 16. It is observed that Unrolling by 8 has executed with less elapsed time than unrolling by 16 with blocks 16.

EX-NO: 04 **DATE: 29.04.2023**

MATRIX ADDITION WITH UNIFIED MEMORY

AIM:

To perform Matrix addition with unified memory and check its performance with nvprof.

PROCEDURE:

1. Include the required files and library.

2. Introduce a function named "initialData","sumMatrixOnHost","checkResult" to return the initialize the data , perform matrix summation on the host and then check the result.

3. Create a grid 2D block 2D global function to perform matrix on the gpu.

4. Declare the main function. In the main function set up the device & data size of matrix , perform memory allocation on host memory & initialize the data at host side then add matrix at host side for result checks followed by invoking kernel at host side. Then warm-up kernel,check the kernel error, and check device for results.Finally free the device global memory and reset device.

5. Execute the program and run the terminal . Check the performance using nvprof.

PROGRAM :

Developed By: H.DHAYANITHA
Reg.No: 212220230010

```
#include "common.h"
#include <cuda_runtime.h>
#include <stdio.h>

void initialData(float *ip, const int size)
{
   int i;
   for (i = 0; i < size; i++)
   {
      ip[i] = (float)( rand() & 0xFF ) / 10.0f;
   }
   return;
}
```

```
void sumMatrixOnHost(float *A, float *B, float *C, const int nx, const int ny)
{
   float *ia = A;
   float *ib = B;
   float *ic = C;

   for (int iy = 0; iy < ny; iy++)
   {
      for (int ix = 0; ix < nx; ix++)
      {
         ic[ix] = ia[ix] + ib[ix];
      }

      ia += nx;
      ib += nx;
      ic += nx;
   }

   return;
}

void checkResult(float *hostRef, float *gpuRef, const int N)
{
   double epsilon = 1.0E-8;
   bool match = 1;

   for (int i = 0; i < N; i++)
   {
      if (abs(hostRef[i] - gpuRef[i]) > epsilon)
      {
         match = 0;
         printf("host %f gpu %f\n", hostRef[i], gpuRef[i]);
         break;
      }
   }

   if (!match)
   {
      printf("Arrays do not match.\n\n");
   }
}

// grid 2D block 2D
```

```
__global__ void sumMatrixGPU(float *MatA, float *MatB, float *MatC, int nx,
                         int ny)
{
   unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
   unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y;
   unsigned int idx = iy * nx + ix;

   if (ix < nx && iy < ny)
   {
      MatC[idx] = MatA[idx] + MatB[idx];
   }
}

int main(int argc, char **argv)
{
   printf("%s Starting ", argv[0]);

   // set up device
   int dev = 0;
   cudaDeviceProp deviceProp;
   CHECK(cudaGetDeviceProperties(&deviceProp, dev));
   printf("using Device %d: %s\n", dev, deviceProp.name);
   CHECK(cudaSetDevice(dev));

   // set up data size of matrix
   int nx, ny;
   int ishift = 12;

   if  (argc > 1) ishift = atoi(argv[1]);

   nx = ny = 1 << ishift;

   int nxy = nx * ny;
   int nBytes = nxy * sizeof(float);
   printf("Matrix size: nx %d ny %d\n", nx, ny);

   // malloc host memory
   float *A, *B, *hostRef, *gpuRef;
   CHECK(cudaMallocManaged((void **)&A, nBytes));
   CHECK(cudaMallocManaged((void **)&B, nBytes));
   CHECK(cudaMallocManaged((void **)&gpuRef,  nBytes);  );
   CHECK(cudaMallocManaged((void **)&hostRef, nBytes););
```

```
// initialize data at host side
double iStart = seconds();
initialData(A, nxy);
initialData(B, nxy);
double iElaps = seconds() - iStart;
printf("initialization: \t %f sec\n", iElaps);

memset(hostRef, 0, nBytes);
memset(gpuRef, 0, nBytes);

// add matrix at host side for result checks
iStart = seconds();
sumMatrixOnHost(A, B, hostRef, nx, ny);
iElaps = seconds() - iStart;
printf("sumMatrix on host:\t %f sec\n", iElaps);

// invoke kernel at host side
int dimx = 32;
int dimy = 32;
dim3 block(dimx, dimy);
dim3 grid((nx + block.x - 1) / block.x, (ny + block.y - 1) / block.y);

// warm-up kernel, with unified memory all pages will migrate from host to
// device
sumMatrixGPU<<<grid, block>>>(A, B, gpuRef, 1, 1);

// after warm-up, time with unified memory
iStart = seconds();

sumMatrixGPU<<<grid, block>>>(A, B, gpuRef, nx, ny);

CHECK(cudaDeviceSynchronize());
iElaps = seconds() - iStart;
printf("sumMatrix on gpu :\t %f sec <<<(%d,%d), (%d,%d)>>> \n", iElaps,
       grid.x, grid.y, block.x, block.y);

// check kernel error
CHECK(cudaGetLastError());

// check device results
checkResult(hostRef, gpuRef, nxy);

// free device global memory
CHECK(cudaFree(A));
```

```
    CHECK(cudaFree(B));
    CHECK(cudaFree(hostRef));
    CHECK(cudaFree(gpuRef));

    // reset device
    CHECK(cudaDeviceReset());

    return (0);
}
```

Removing the memsets:

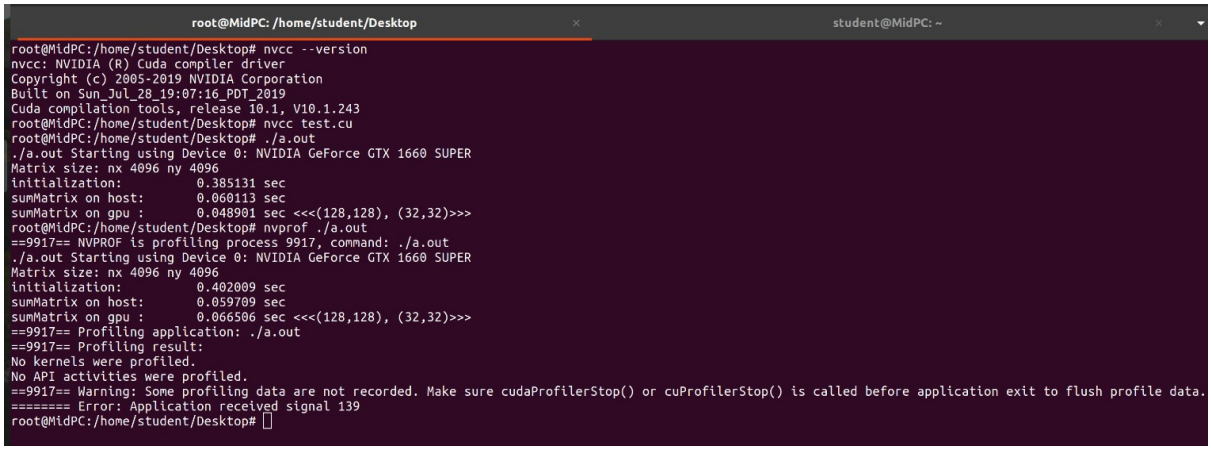
```
#include "common.h"
#include <cuda_runtime.h>
#include <stdio.h>

void initialData(float *ip, const int size)
{
    int i;

    for (i = 0; i < size; i++)
    {
        ip[i] = (float)( rand() & 0xFF ) / 10.0f;
    }

    return;
}

void sumMatrixOnHost(float *A, float *B, float *C, const int nx, const int ny)
{
    float *ia = A;
    float *ib = B;
    float *ic = C;

    for (int iy = 0; iy < ny; iy++)
    {
        for (int ix = 0; ix < nx; ix++)
        {
            ic[ix] = ia[ix] + ib[ix];
        }

        ia += nx;
        ib += nx;
        ic += nx;
```

```
    }

    return;
}

void checkResult(float *hostRef, float *gpuRef, const int N)
{
    double epsilon = 1.0E-8;
    bool match = 1;

    for (int i = 0; i < N; i++)
    {
        if (abs(hostRef[i] - gpuRef[i]) > epsilon)
        {
            match = 0;
            printf("host %f gpu %f\n", hostRef[i], gpuRef[i]);
            break;
        }
    }

    if (!match)
    {
        printf("Arrays do not match.\n\n");
    }
}

// grid 2D block 2D
__global__ void sumMatrixGPU(float *MatA, float *MatB, float *MatC, int nx,
                             int ny)
{
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y;
    unsigned int idx = iy * nx + ix;

    if (ix < nx && iy < ny)
    {
        MatC[idx] = MatA[idx] + MatB[idx];
    }
}

int main(int argc, char **argv)
{
    printf("%s Starting ", argv[0]);
```

```
// set up device
int dev = 0;
cudaDeviceProp deviceProp;
CHECK(cudaGetDeviceProperties(&deviceProp, dev));
printf("using Device %d: %s\n", dev, deviceProp.name);
CHECK(cudaSetDevice(dev));

// set up data size of matrix
int nx, ny;
int ishift = 12;

if  (argc > 1) ishift = atoi(argv[1]);

nx = ny = 1 << ishift;

int nxy = nx * ny;
int nBytes = nxy * sizeof(float);
printf("Matrix size: nx %d ny %d\n", nx, ny);

// malloc host memory
float *A, *B, *hostRef, *gpuRef;
CHECK(cudaMallocManaged((void **)&A, nBytes));
CHECK(cudaMallocManaged((void **)&B, nBytes));
CHECK(cudaMallocManaged((void **)&gpuRef,  nBytes);  );
CHECK(cudaMallocManaged((void **)&hostRef, nBytes););

// initialize data at host side
double iStart = seconds();
initialData(A, nxy);
initialData(B, nxy);
double iElaps = seconds() - iStart;
printf("initialization: \t %f sec\n", iElaps);

// add matrix at host side for result checks
iStart = seconds();
sumMatrixOnHost(A, B, hostRef, nx, ny);
iElaps = seconds() - iStart;
printf("sumMatrix on host:\t %f sec\n", iElaps);

// invoke kernel at host side
int dimx = 32;
int dimy = 32;
dim3 block(dimx, dimy);
dim3 grid((nx + block.x - 1) / block.x, (ny + block.y - 1) / block.y);
```

```
    // warm-up kernel, with unified memory all pages will migrate from host to
    // device
    sumMatrixGPU<<<grid, block>>>(A, B, gpuRef, 1, 1);

    // after warm-up, time with unified memory
    iStart = seconds();

    sumMatrixGPU<<<grid, block>>>(A, B, gpuRef, nx, ny);

    CHECK(cudaDeviceSynchronize());
    iElaps = seconds() - iStart;
    printf("sumMatrix on gpu :\t %f sec <<<(%d,%d), (%d,%d)>>> \n", iElaps,
           grid.x, grid.y, block.x, block.y);

    // check kernel error
    CHECK(cudaGetLastError());

    // check device results
    checkResult(hostRef, gpuRef, nxy);

    // free device global memory
    CHECK(cudaFree(A));
    CHECK(cudaFree(B));
    CHECK(cudaFree(hostRef));
    CHECK(cudaFree(gpuRef));

    // reset device
    CHECK(cudaDeviceReset());

    return (0);
}
```

OUTPUT:



RESULT:

The initialization process was completed in 0.418289seconds, and the matrix addition took 0.065890 seconds in the host, and 0.042262 seconds in the GPU and provides better performance between the host and GPU. Thus, matrix addition using CUDA programming with unified memory has been performed successfully.

EX-NO: 05 **DATE: 12.05.2023**

IMPLEMENT MATRIX MULTIPLICATION USING CUDA C

AIM:

To implement Matrix Multiplication using GPU.

PROGRAM:

1. Allocate memory for matrices h_a , h_b , and h_c on the host.
2. Initialize matrices h_a and h_b with random values between 0 and 1.
3. Allocate memory for matrices d_a , d_b , and d_c on the device.
4. Copy matrices h_a and h_b from the host to the device.
5. Launch the kernel matrixMulGPU with numBlocks blocks of threadsPerBlock threads.
6. Measure the time taken by the CPU and GPU implementations using CUDA events.
7. Print the elapsed time for each implementation.
8. Free the memory allocated on both the host and the device.

PROGRAM:

Developed By: H.DHAYANITHA
Reg.No: 212220230010

```
#include <stdio.h>
#include <sys/time.h>

#define SIZE 4
#define BLOCK_SIZE 2

// Kernel function to perform matrix multiplication
__global__ void matrixMultiply(int *a, int *b, int *c, int size)
{
   int row = blockIdx.y * blockDim.y + threadIdx.y;
   int col = blockIdx.x * blockDim.x + threadIdx.x;

   int sum = 0;
   for (int k = 0; k < size; ++k)
   {
      sum += a[row * size + k] * b[k * size + col];
   }
   c[row * size + col] = sum;
}
```

```
int main()
{
   int a[SIZE][SIZE], b[SIZE][SIZE], c[SIZE][SIZE];
   int *dev_a, *dev_b, *dev_c;
   int size = SIZE * SIZE * sizeof(int);

   // Initialize matrices 'a' and 'b'
   for (int i = 0; i < SIZE; ++i)
   {
      for (int j = 0; j < SIZE; ++j)
      {
         a[i][j] = i + j;
         b[i][j] = i - j;
      }
   }

   // Allocate memory on the device
   cudaMalloc((void**)&dev_a, size);
   cudaMalloc((void**)&dev_b, size);
   cudaMalloc((void**)&dev_c, size);

   // Copy input matrices from host to device memory
   cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
   cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

   // Set grid and block sizes
   dim3 dimGrid(SIZE / BLOCK_SIZE, SIZE / BLOCK_SIZE);
   dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);

   // Start timer
   struct timeval start, end;
   gettimeofday(&start, NULL);

   // Launch kernel
   matrixMultiply<<<dimGrid, dimBlock>>>(dev_a, dev_b, dev_c, SIZE);

   // Copy result matrix from device to host memory
   cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);

   // Stop timer
   gettimeofday(&end, NULL);
   double elapsed_time = (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) /
1000000.0;
```

```
    // Print the result matrix
    printf("Result Matrix:\n");
    for (int i = 0; i < SIZE; ++i)
    {
        for (int j = 0; j < SIZE; ++j)
        {
            printf("%d ", c[i][j]);
        }
        printf("\n");
    }

    // Print the elapsed time
    printf("Elapsed Time: %.6f seconds\n", elapsed_time);

    // Free device memory
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);

    return 0;
}
```

OUTPUT:

```
root@MidPC:/home/student/Desktop# nvcc first.cu
root@MidPC:/home/student/Desktop# ./a.out
Result Matrix:
14 8 2 -4
20 10 0 -10
26 12 -2 -16
32 14 -4 -22
Elapsed Time: 0.000023 seconds
root@MidPC:/home/student/Desktop# nvprof ./a.out
==18221== NVPROF is profiling process 18221, command: ./a.out
Result Matrix:
14 8 2 -4
20 10 0 -10
26 12 -2 -16
32 14 -4 -22
Elapsed Time: 0.000037 seconds
==18221== Profiling application: ./a.out
==18221== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   39.90%  2.5280us         1  2.5280us  2.5280us  2.5280us  matrixMultiply(int*, int*, int*, int)
                   38.89%  2.4640us         2  1.2320us     928ns  1.5360us  [CUDA memcpy HtoD]
                   21.21%  1.3440us         1  1.3440us  1.3440us  1.3440us  [CUDA memcpy DtoH]
      API calls:   99.38%  126.78ms         3  42.262ms  2.2600us  126.78ms  cudaMalloc
                    0.28%  356.84us         1  356.84us  356.84us  356.84us  cuDeviceTotalMem
                    0.20%  252.08us        97  2.5980us     210ns  107.52us  cuDeviceGetAttribute
                    0.07%  87.360us         3  29.120us  2.5700us  79.360us  cudaFree
                    0.03%  36.470us         1  36.470us  36.470us  36.470us  cuDeviceGetName
                    0.02%  29.180us         3  9.7260us  5.9900us  12.080us  cudaMemcpy
                    0.02%  23.180us         1  23.180us  23.180us  23.180us  cudaLaunchKernel
                    0.00%  4.5900us         1  4.5900us  4.5900us  4.5900us  cuDeviceGetPCIBusId
                    0.00%  2.4000us         3     800ns     250ns  1.8100us  cuDeviceGetCount
                    0.00%     930ns         2     465ns     210ns     720ns  cuDeviceGet
                    0.00%     310ns         1     310ns     310ns     310ns  cuDeviceGetUuid
root@MidPC:/home/student/Desktop# ^C
root@MidPC:/home/student/Desktop#
```

RESULT:

The implementation of Matrix Multiplication using GPU is done successfully.

EX-NO: 06 **DATE: 02.06.2023**

MATRIX TRANSPOSITION USING SHARED MEMORY

AIM:

To demonstrate the Matrix transposition on shared memory with grid (1,1) block (16,16).

PROCEDURE:

1. The code implements various matrix transposition techniques using shared memory in CUDA.

2. The different implementations include:

SetRowReadRow : Transpose matrix using row-major ordering for both read and write operations.

SetColReadCol : Transpose matrix using column-major ordering for both read and write operations.

SetColReadCol2 : Transpose matrix using column-major ordering for write operation and row-major ordering for read operation.

SetRowReadCol : Transpose matrix using row-major ordering for write operation and column-major ordering for read operation.

SetRowReadColDyn : Transpose matrix using dynamic shared memory and rowmajor ordering for write operation and column-major ordering for read operation.

SetRowReadColPad : Transpose matrix using row-major ordering for write operation and column-major ordering for read operation, with padding.

SetRowReadColDynPad : Transpose matrix using dynamic shared memory, rowmajor ordering for write operation, column-major ordering for read operation, with padding.

3. The code measures the execution time of each implementation using CUDA events.

4. The results of the matrix transposition are verified by comparing the output with the expected result.

5. The performance of each implementation is compared based on their execution times.

PROGRAM:

Developed By: H.DHAYANITHA

Reg.No: 212220230010

```
#include "common.h"
#include <cuda_runtime.h>
#include <stdio.h>

#define BDIMX 16
#define BDIMY 16
#define IPAD  2

void printData(char *msg, int *in,  const int size)
{
   printf("%s: ", msg);

   for (int i = 0; i < size; i++)
   {
      printf("%4d", in[i]);
      fflush(stdout);
   }

   printf("\n\n");
}

__global__ void setRowReadRow(int *out)
{
   // static shared memory
   __shared__ int tile[BDIMY][BDIMX];

   // mapping from thread index to global memory index
   unsigned int idx = threadIdx.y * blockDim.x + threadIdx.x;

   // shared memory store operation
   tile[threadIdx.y][threadIdx.x] = idx;

   // wait for all threads to complete
   __syncthreads();

   // shared memory load operation
   out[idx] = tile[threadIdx.y][threadIdx.x] ;
}
```

```
__global__ void setColReadCol(int *out)
{
    // static shared memory
    __shared__ int tile[BDIMX][BDIMY];

    // mapping from thread index to global memory index
    unsigned int idx = threadIdx.y * blockDim.x + threadIdx.x;

    // shared memory store operation
    tile[threadIdx.x][threadIdx.y] = idx;

    // wait for all threads to complete
    __syncthreads();

    // shared memory load operation
    out[idx] = tile[threadIdx.x][threadIdx.y];
}

__global__ void setColReadCol2(int *out)
{
    // static shared memory
    __shared__ int tile[BDIMY][BDIMX];

    // mapping from 2D thread index to linear memory
    unsigned int idx = threadIdx.y * blockDim.x + threadIdx.x;

    // convert idx to transposed coordinate (row, col)
    unsigned int irow = idx / blockDim.y;
    unsigned int icol = idx % blockDim.y;

    // shared memory store operation
    tile[icol][irow] = idx;

    // wait for all threads to complete
    __syncthreads();

    // shared memory load operation
    out[idx] = tile[icol][irow] ;
}

__global__ void setRowReadCol(int *out)
{
    // static shared memory
    __shared__ int tile[BDIMY][BDIMX];
```

```
    // mapping from 2D thread index to linear memory
    unsigned int idx = threadIdx.y * blockDim.x + threadIdx.x;

    // convert idx to transposed coordinate (row, col)
    unsigned int irow = idx / blockDim.y;
    unsigned int icol = idx % blockDim.y;

    // shared memory store operation
    tile[threadIdx.y][threadIdx.x] = idx;

    // wait for all threads to complete
    __syncthreads();

    // shared memory load operation
    out[idx] = tile[icol][irow];
}

__global__ void setRowReadColPad(int *out)
{
    // static shared memory
    __shared__ int tile[BDIMY][BDIMX + IPAD];

    // mapping from 2D thread index to linear memory
    unsigned int idx = threadIdx.y * blockDim.x + threadIdx.x;

    // convert idx to transposed (row, col)
    unsigned int irow = idx / blockDim.y;
    unsigned int icol = idx % blockDim.y;

    // shared memory store operation
    tile[threadIdx.y][threadIdx.x] = idx;

    // wait for all threads to complete
    __syncthreads();

    // shared memory load operation
    out[idx] = tile[icol][irow] ;
}

__global__ void setRowReadColDyn(int *out)
{
    // dynamic shared memory
    extern  __shared__ int tile[];
```

```
    // mapping from thread index to global memory index
    unsigned int idx = threadIdx.y * blockDim.x + threadIdx.x;

    // convert idx to transposed (row, col)
    unsigned int irow = idx / blockDim.y;
    unsigned int icol = idx % blockDim.y;

    // convert back to smem idx to access the transposed element
    unsigned int col_idx = icol * blockDim.x + irow;

    // shared memory store operation
    tile[idx] = idx;

    // wait for all threads to complete
    __syncthreads();

    // shared memory load operation
    out[idx] = tile[col_idx];
}

__global__ void setRowReadColDynPad(int *out)
{
    // dynamic shared memory
    extern  __shared__ int tile[];

    // mapping from thread index to global memory index
    unsigned int g_idx = threadIdx.y * blockDim.x + threadIdx.x;

    // convert idx to transposed (row, col)
    unsigned int irow = g_idx / blockDim.y;
    unsigned int icol = g_idx % blockDim.y;

    unsigned int row_idx = threadIdx.y * (blockDim.x + IPAD) + threadIdx.x;

    // convert back to smem idx to access the transposed element
    unsigned int col_idx = icol * (blockDim.x + IPAD) + irow;

    // shared memory store operation
    tile[row_idx] = g_idx;

    // wait for all threads to complete
    __syncthreads();
```

```
    // shared memory load operation
    out[g_idx] = tile[col_idx];
}

int main(int argc, char **argv)
{
    // set up device
    int dev = 0;
    cudaDeviceProp deviceProp;
    CHECK(cudaGetDeviceProperties(&deviceProp, dev));
    printf("%s at ", argv[0]);
    printf("device %d: %s ", dev, deviceProp.name);
    CHECK(cudaSetDevice(dev));

    cudaSharedMemConfig pConfig;
    CHECK(cudaDeviceGetSharedMemConfig ( &pConfig ));
    printf("with Bank Mode:%s ", pConfig == 1 ? "4-Byte" : "8-Byte");

    // set up array size
    int nx = BDIMX;
    int ny = BDIMY;

    bool iprintf = 0;

    if (argc > 1) iprintf = atoi(argv[1]);

    size_t nBytes = nx * ny * sizeof(int);

    // execution configuration
    dim3 block (BDIMX, BDIMY);
    dim3 grid  (1, 1);
    printf("<<< grid (%d,%d) block (%d,%d)>>>\n", grid.x, grid.y, block.x,
           block.y);

    // allocate device memory
    int *d_C;
    CHECK(cudaMalloc((int**)&d_C, nBytes));
    int *gpuRef  = (int *)malloc(nBytes);

    CHECK(cudaMemset(d_C, 0, nBytes));
    setRowReadRow<<<grid, block>>>(d_C);
    CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));

    if(iprintf)  printData("setRowReadRow       ", gpuRef, nx * ny);
```

```
CHECK(cudaMemset(d_C, 0, nBytes));
setColReadCol<<<grid, block>>>(d_C);
CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));

if(iprintf)  printData("setColReadCol       ", gpuRef, nx * ny);

CHECK(cudaMemset(d_C, 0, nBytes));
setColReadCol2<<<grid, block>>>(d_C);
CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));

if(iprintf)  printData("setColReadCol2      ", gpuRef, nx * ny);

CHECK(cudaMemset(d_C, 0, nBytes));
setRowReadCol<<<grid, block>>>(d_C);
CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));

if(iprintf)  printData("setRowReadCol       ", gpuRef, nx * ny);

CHECK(cudaMemset(d_C, 0, nBytes));
setRowReadColDyn<<<grid, block, BDIMX*BDIMY*sizeof(int)>>>(d_C);
CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));

if(iprintf)  printData("setRowReadColDyn    ", gpuRef, nx * ny);

CHECK(cudaMemset(d_C, 0, nBytes));
setRowReadColPad<<<grid, block>>>(d_C);
CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));

if(iprintf)  printData("setRowReadColPad    ", gpuRef, nx * ny);

CHECK(cudaMemset(d_C, 0, nBytes));
setRowReadColDynPad<<<grid, block, (BDIMX + IPAD)*BDIMY*sizeof(int)>>>(d_C);
CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));

if(iprintf)  printData("setRowReadColDynPad ", gpuRef, nx * ny);

// free host and device memory
CHECK(cudaFree(d_C));
free(gpuRef);

// reset device
CHECK(cudaDeviceReset());
return EXIT_SUCCESS;
```

}

OUTPUT:

```
root@MidPC:/home/student/Desktop# ./a.out
./a.out at device 0: NVIDIA GeForce GTX 1660 SUPER with Bank Mode:4-Byte <<< grid (1,1) block (16,16)>>>
root@MidPC:/home/student/Desktop# nvprof ./a.out
==14603== NVPROF is profiling process 14603, command: ./a.out
./a.out at device 0: NVIDIA GeForce GTX 1660 SUPER with Bank Mode:4-Byte <<< grid (1,1) block (16,16)>>>
==14603== Profiling application: ./a.out
==14603== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   35.10%  7.8410us         7  1.1200us  1.1200us  1.1210us  [CUDA memcpy DtoH]
                   32.52%  7.2640us         7  1.0370us     960ns  1.4720us  [CUDA memset]
                    4.73%  1.0560us         1  1.0560us  1.0560us  1.0560us  setRowReadCol(int*)
                    4.73%  1.0560us         1  1.0560us  1.0560us  1.0560us  setColReadCol2(int*)
                    4.73%  1.0560us         1  1.0560us  1.0560us  1.0560us  setRowReadColDyn(int*)
                    4.58%  1.0240us         1  1.0240us  1.0240us  1.0240us  setRowReadColDynPad(int*)
                    4.58%  1.0240us         1  1.0240us  1.0240us  1.0240us  setColReadCol(int*)
                    4.58%  1.0240us         1  1.0240us  1.0240us  1.0240us  setRowReadColPad(int*)
                    4.44%     992ns         1     992ns     992ns     992ns  setRowReadRow(int*)
      API calls:   74.98%  139.00ms         1  139.00ms  139.00ms  139.00ms  cudaDeviceGetSharedMemConfig
                   24.49%  45.404ms         1  45.404ms  45.404ms  45.404ms  cudaDeviceReset
                    0.12%  215.90us        97  2.2250us     170ns  93.829us  cuDeviceGetAttribute
                    0.11%  195.87us         1  195.87us  195.87us  195.87us  cuDeviceTotalMem
                    0.10%  186.03us         1  186.03us  186.03us  186.03us  cudaGetDeviceProperties
                    0.06%  110.17us         1  110.17us  110.17us  110.17us  cudaMalloc
                    0.04%  78.960us         1  78.960us  78.960us  78.960us  cudaFree
                    0.04%  73.669us         7  10.524us  9.3100us  14.940us  cudaMemcpy
                    0.03%  50.060us         7  7.1510us  4.2900us  23.140us  cudaLaunchKernel
                    0.02%  33.690us         1  33.690us  33.690us  33.690us  cuDeviceGetName
                    0.02%  29.020us         7  4.1450us  2.5200us  12.920us  cudaMemset
                    0.00%  4.6900us         1  4.6900us  4.6900us  4.6900us  cuDeviceGetPCIBusId
                    0.00%  2.6800us         1  2.6800us  2.6800us  2.6800us  cudaSetDevice
                    0.00%  2.1600us         3     720ns     180ns  1.7500us  cuDeviceGetCount
                    0.00%     750ns         2     375ns     170ns     580ns  cuDeviceGet
                    0.00%     240ns         1     240ns     240ns     240ns  cuDeviceGetUuid
root@MidPC:/home/student/Desktop#
```

RESULT:

The Matrix transposition on shared memory with grid (1,1) block (16,16) is demonstrated successfully.