EXPERIMENT 3 : Simple-warp-divergence---Implement-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

1. Initialize an input array of size 1024.
2. Launch the reduceUnrolling8 kernel, which performs reduction using 8 data blocks per thread.
3. Launch the reduceUnrolling16 kernel, which performs reduction using 16 data blocks per thread.
4. Compare the results obtained from both kernels.

PROGRAM

```
%%cu
#include <cuda_runtime.h>
#include <stdio.h>
#include <cuda.h>
#include <sys/time.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 mem
    if (tid == 0) g_odata[blockIdx.x] = idata[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 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];
        int c1 = g_idata[idx + 8 * blockDim.x];
        int c2 = g_idata[idx + 9 * blockDim.x];
        int c3 = g_idata[idx + 10 * blockDim.x];
        int c4 = g_idata[idx + 11 * blockDim.x];
        int d1 = g_idata[idx + 12 * blockDim.x];
        int d2 = g_idata[idx + 13 * blockDim.x];
        int d3 = g_idata[idx + 14 * blockDim.x];
        int d4 = g_idata[idx + 15 * blockDim.x];
        g_idata[idx] = a1 + a2 + a3 + a4 + b1 + b2 + b3 + b4 + c1 + c2 + c3 + c4
                       + d1 + d2 + d3 + d4;
    }

    __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 mem
    if (tid == 0) g_odata[blockIdx.x] = idata[0];
}

#ifndef _COMMON_H
#define _COMMON_H

#define CHECK(call)                                                            \
{                                                                              \
    const cudaError_t error = call;                                            \
    if (error != cudaSuccess)                                                  \
    {                                                                          \
        fprintf(stderr, "Error: %s:%d, ", __FILE__, __LINE__);                 \
        fprintf(stderr, "code: %d, reason: %s\n", error,                       \
                cudaGetErrorString(error));                                    \
        exit(1);                                                               \
    }                                                                          \
}

#define CHECK_CUBLAS(call)                                                     \
{                                                                              \
    cublasStatus_t err;                                                        \
    if ((err = (call)) != CUBLAS_STATUS_SUCCESS)                               \
    {                                                                          \
        fprintf(stderr, "Got CUBLAS error %d at %s:%d\n", err, __FILE__,       \
                __LINE__);                                                     \
        exit(1);                                                               \
    }                                                                          \
}

#define CHECK_CURAND(call)                                                     \
{                                                                              \
    curandStatus_t err;                                                        \
    if ((err = (call)) != CURAND_STATUS_SUCCESS)                               \
    {                                                                          \
        fprintf(stderr, "Got CURAND error %d at %s:%d\n", err, __FILE__,       \
                __LINE__);                                                     \
        exit(1);                                                               \
    }                                                                          \
}

#define CHECK_CUFFT(call)                                                      \
```

```
{                                                                                          \
    cufftResult err;                                                                       \
    if ( (err = (call)) != CUFFT_SUCCESS)                                                  \
    {                                                                                      \
        fprintf(stderr, "Got CUFFT error %d at %s:%d\n", err, __FILE__,                    \
                __LINE__);                                                                 \
        exit(1);                                                                           \
    }                                                                                      \
}

#define CHECK_CUSPARSE(call)                                                               \
{                                                                                          \
    cusparseStatus_t err;                                                                  \
    if ((err = (call)) != CUSPARSE_STATUS_SUCCESS)                                         \
    {                                                                                      \
        fprintf(stderr, "Got error %d at %s:%d\n", err, __FILE__, __LINE__);               \
        cudaError_t cuda_err = cudaGetLastError();                                         \
        if (cuda_err != cudaSuccess)                                                       \
        {                                                                                  \
            fprintf(stderr, "  CUDA error \"%s\" also detected\n",                         \
                    cudaGetErrorString(cuda_err));                                         \
        }                                                                                  \
        exit(1);                                                                           \
    }                                                                                      \
}

inline double seconds()
{
    struct timeval tp;
    struct timezone tzp;
    int i = gettimeofday(&tp, &tzp);
    return ((double)tp.tv_sec + (double)tp.tv_usec * 1.e-6);
}

#endif // _COMMON_H

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

    bool bResult = false;

    // initialization
    int size = 1 << 24; // total number of elements to reduce
    printf("    with array size %d  ", size);

    // execution configuration
```

```
    int blocksize = 512;   // initial block size

    if(argc > 1)
    {
        blocksize = atoi(argv[1]);   // block size from command line argument
    }

    dim3 block (blocksize, 1);
    dim3 grid  ((size + block.x - 1) / block.x, 1);
    printf("grid %d block %d\n", grid.x, block.x);

    // allocate host memory
    size_t bytes = size * sizeof(int);
    int *h_idata = (int *) malloc(bytes);
    int *h_odata = (int *) malloc(grid.x * sizeof(int));
    int *tmp     = (int *) malloc(bytes);

    // initialize the array
    for (int i = 0; i < size; i++)
    {
        // mask off high 2 bytes to force max number to 255
        h_idata[i] = (int)( rand() & 0xFF );
    }

    memcpy (tmp, h_idata, bytes);

    double iStart, iElapsunroll8,iElapsunroll16;
    int gpu_sum = 0;

    // allocate device memory
    int *d_idata = NULL;
    int *d_odata = NULL;
    CHECK(cudaMalloc((void **) &d_idata, bytes));
    CHECK(cudaMalloc((void **) &d_odata, grid.x * sizeof(int)));

    // kernel 1: reduceUnrolling8
    CHECK(cudaMemcpy(d_idata, h_idata, bytes, cudaMemcpyHostToDevice));
    CHECK(cudaDeviceSynchronize());
    iStart = seconds();
    reduceUnrolling8<<<grid.x / 8, block>>>(d_idata, d_odata, size);
    CHECK(cudaDeviceSynchronize());
    iElapsunroll8 = seconds() - iStart;
    CHECK(cudaMemcpy(h_odata, d_odata, grid.x / 8 * sizeof(int),
                     cudaMemcpyDeviceToHost));
    gpu_sum = 0;

    for (int i = 0; i < grid.x / 8; i++) gpu_sum += h_odata[i];

    printf("gpu Unrolling8  elapsed %f sec gpu_sum: %d <<<grid %d block "
           "%d>>>\n", iElapsunroll8, gpu_sum, grid.x / 8, block.x);

    for (int i = 0; i < grid.x / 16; i++) gpu_sum += h_odata[i];
// kernel 2: reduceUnrolling16
```

```
    CHECK(cudaMemcpy(d_idata, h_idata, bytes, cudaMemcpyHostToDevice));
    CHECK(cudaDeviceSynchronize());
    iStart = seconds();
    reduceUnrolling16<<<grid.x / 16, block>>>(d_idata, d_odata, size);
    CHECK(cudaDeviceSynchronize());
    iElapsunroll16 = seconds() - iStart;
    CHECK(cudaMemcpy(h_odata, d_odata, grid.x / 16 * sizeof(int),
                     cudaMemcpyDeviceToHost));
    gpu_sum = 0;

    for (int i = 0; i < grid.x / 16; i++) gpu_sum += h_odata[i];

    printf("gpu Unrolling16 elapsed %f sec gpu_sum: %d <<<grid %d block "
           "%d>>>\n", iElapsunroll16, gpu_sum, grid.x / 16, block.x);

    // Determine the kernel with the least execution time
    if (iElapsunroll8< iElapsunroll16)
    {
        printf("reduceUnrolling8 has the least execution time.\n");
    }
    else if (iElapsunroll16 < iElapsunroll8)
    {
        printf("reduceUnrolling16 has the least execution time.\n");
    }
    else
    {
        printf("reduceUnrolling8 and reduceUnrolling16 have the same execution time.
    }
    // free host memory
    free(h_idata);
    free(h_odata);
    // free device memory
    CHECK(cudaFree(d_idata));
    CHECK(cudaFree(d_odata));
    // reset device
    CHECK(cudaDeviceReset());
    return EXIT_SUCCESS;
}
```

OUTPUT

```
/tmp/tmplhk94aac/343257a3-dd1c-4f40-b103-93feae93952d.out starting reduction at device 0: Tesla T4     with array size 16777216  grid 32768 block 512
gpu Unrolling8  elapsed 0.000331 sec gpu_sum: 2139353471 <<<grid 4096 block 512>>>
gpu Unrolling16 elapsed 0.000295 sec gpu_sum: 2139353471 <<<grid 2048 block 512>>>
reduceUnrolling16 has the least execution time.
```

RESULT

Thus, the performance of two CUDA kernels, reduceUnrolling8 and reduceUnrolling16 has been compared successfully.