HPC/4/matrixMul.cpp

```cpp
#include <cmath>
#include <cstdlib>
#include <iostream>

#define checkCudaErrors(call)                                                                 \
    do {                                                                                      \
        cudaError_t err = call;                                                               \
        if (err  cudaSuccess) {                                                               \
            printf("CUDA error at %s %d: %s\n",   FILE  ,   LINE  , cudaGetErrorString(err)); \
            exit(EXIT_FAILURE);                                                               \
        }                                                                                     \
    } while (0)

using namespace std;

// Matrix multiplication Cuda
 global   void matrixMultiplication(int *a, int *b,  int *c, int n) {
    int row = threadIdx.y + blockDim.y * blockIdx.y;
    int col = threadIdx.x + blockDim.x * blockIdx.x;
    int sum = 0;

    if (row < n    col < n)
        for (int j = 0; j < n; j  ) {
            sum = sum + a[row * n + j] * b[j * n + col];
        }

    c[n * row + col] = sum;
}

int main() {
    int *a, *b, *c;
    int *a_dev, *b_dev, *c_dev;
    int n = 10;

    a = new int[n * n];
    b = new int[n * n];
    c = new int[n * n];
    int *d = new int[n * n];
    int size = n * n * sizeof(int);
    checkCudaErrors(cudaMalloc(&a_dev, size));
    checkCudaErrors(cudaMalloc(&b_dev, size));
    checkCudaErrors(cudaMalloc(&c_dev, size));

    // Array initialization
    for (int i = 0; i < n * n; i  ) {
        a[i] = rand() % 10;
        b[i] = rand() % 10;
    }

    cout << "Given matrix A is   \n";
    for (int row = 0; row < n; row  ) {
        for (int col = 0; col < n; col  ) {
        cout << a[row * n + col] << " "; 54
        }
        cout << "\n";
    }
    cout << "\n";

    cout << "Given matrix B is   \n";
    for (int row = 0; row < n; row  ) {
        for (int col = 0; col < n; col  ) {
        cout << b[row * n + col] << " "; 63
        }
        cout << "\n";
    }
    cout << "\n";

    cudaEvent_t start, end;

    checkCudaErrors(cudaEventCreate(&start));
```

```
    checkCudaErrors(cudaEventCreate(&end));

    checkCudaErrors(cudaMemcpy(a_dev, a, size, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(b_dev, b, size, cudaMemcpyHostToDevice));

    dim3 threadsPerBlock(n, n);
    dim3 blocksPerGrid(1, 1);

    // GPU Multiplication
    checkCudaErrors(cudaEventRecord(start));
     matrixMultiplication<<<blocksPerGrid, threadsPerBlock>>>(a_dev, b_dev, c_dev, n);

    checkCudaErrors(cudaEventRecord(end));
    checkCudaErrors(cudaEventSynchronize(end));

    float time = 0.0;
    checkCudaErrors(cudaEventElapsedTime(&time, start, end));

    checkCudaErrors(cudaMemcpy(c, c_dev, size, cudaMemcpyDeviceToHost));

    // CPU matrix multiplication
    int sum = 0;
    for (int row = 0; row < n; row  ) {
        for (int col = 0; col < n; col  ) {
            sum = 0;
            for (int k = 0; k < n; k  ) sum = sum + a[row * n + k] * b[k * n + col];
            d[row * n + col] = sum;
        }
    }

    cout << "CPU product is  \n";
    for (int row = 0; row < n; row  ) {
        for (int col = 0; col < n; col  ) {
            cout << d[row * n + col] << " ";
        }
        cout << "\n";
    }
    cout << "\n";

    cout << "GPU product is  \n";
    for (int row = 0; row < n; row  ) {
        for (int col = 0; col < n; col  ) {
            cout << c[row * n + col] << " ";
        }
        cout << "\n";
    }
    cout << "\n";

    int error = 0;
    int _c, _d;
    for (int row = 0; row < n; row  ) {
        for (int col = 0; col < n; col  ) {
            _c = c[row * n + col];
            _d = d[row * n + col];
            error  _c - _d;
            if (O  (_c - _d)) {                    << ", " << col << ")    GPU: " << _c << ", CPU: " << _d
                cout << "Error at (" << row
                     << "\n";
            }
        }
    }
    cout << "\n";

    cout << "Error : " << error;
    cout << "\nTime Elapsed: " << time;

    return
    return 0;
}

/ *

O UTPUT:

 Given matrix A is
```

```
3 7 3 6 9 2 0 3 0 2
1 7 2 2 7 9 2 9 3 1
9 1 4 8 5 3 1 6 2 6
5 4 6 6 3 4 2 4 4 3
7 6 8 3 4 2 6 9 6 4
5 4 7 7 7 2 1 6 5 4
0 1 7 1 9 7 7 6 6 9
8 2 3 0 8 0 6 8 6 1
9 4 1 3 4 4 7 3 7 9
2 7 5 4 8 9 5 8 3 8

Given matrix B is
6 5 5 2 1 7 9 6 6 6
8 9 0 3 5 2 8 7 6 2
3 9 7 4 0 6 0 3 0 1
5 7 5 9 7 5 5 7 4 0
8 8 4 1 9 0 8 2 6 9
0 8 1 2 2 6 0 1 9 9
9 7 1 5 7 6 3 5 3 4
1 9 9 8 5 9 3 5 1 5
8 8 0 0 4 4 6 1 5 6
1 8 7 1 5 7 3 8 1 9

CPU product is
190 278 145 132 190 136 200 169 161 167
186 355 156 157 207 209 185 164 210 246
191 335 233 179 196 257 220 227 174 232
191 319 172 156 167 218 182 186 165 186
276 433 239 205 229 305 251 252 193 257
233 378 222 181 218 240 231 216 180 226
232 430 221 155 255 274 187 203 193 328
248 319 178 137 201 217 233 171 165 236
267 379 184 141 231 276 259 247 218 301
252 477 239 204 282 302 239 261 245 334

GPU product is
190 278 145 132 190 136 200 169 161 167
186 355 156 157 207 209 185 164 210 246
191 335 233 179 196 257 220 227 174 232
191 319 172 156 167 218 182 186 165 186
276 433 239 205 229 305 251 252 193 257
233 378 222 181 218 240 231 216 180 226
232 430 221 155 255 274 187 203 193 328
248 319 178 137 201 217 233 171 165 236
267 379 184 141 231 276 259 247 218 301
252 477 239 204 282 302 239 261 245 334


Error : 0
Time Elapsed: 0.018144

*/

```

HPC/4/matrixVectorMul.cpp

```
#include <time.h>

#include <cmath>
#include <cstdlib>
#include <iostream>

#define checkCudaErrors(call)                                                   \
    do {\
        cudaError_t err = call;                                                 \
        if (err   cudaSuccess) {                                                \
            printf("CUDA error at %s %d: %s\n",   FILE  ,   LINE  , cudaGetErrorString(err)); \
            exit(EXIT_FAILURE);                                                 \
        }                                                                       \
    } while (0)

using namespace std;

 global   void matrixVectorMultiplication(int *a, int *b,  int *c, int n) {
    int row = threadIdx.x + blockDim.x * blockIdx.x;
    int sum = 0;

    if (row < n)
        for (int j = 0; j < n; j  ) {
            sum = sum + a[row * n + j] * b[j];
        }

    c[row] = sum;
}

int main() {
    int *a, *b, *c;
    int *a_dev, *b_dev, *c_dev;
    int n = 10;

    a = new int[n * n];
    b = new int[n];
    c = new int[n];
    int *d = new int[n];
    int size = n * sizeof(int);
    checkCudaErrors(cudaMalloc(&a_dev, size * size));
    checkCudaErrors(cudaMalloc(&b_dev, size));
    checkCudaErrors(cudaMalloc(&c_dev, size));

    for (int i = 0; i < n; i  ) {
        for (int j = 0; j < n; j  ) {
            a[i * n + j] = rand() % 10;
        }
        b[i] = rand() % 10;
    }

    cout << "Given matrix is   \n";
    for (int row = 0; row < n; row  ) {
        for (int col = 0; col < n; col  ) {
        cout << a[row * n + col] << " ";
        }
        cout << "\n";
    }
    cout << "\n";

    cout << "Given vector is   \n";
    for (int i = 0; i < n; i  ) {
        cout << b[i] << ", ";
    }
    cout << "\n\n";

    cudaEvent_t start, end;

    checkCudaErrors(cudaEventCreate(&start));
    checkCudaErrors(cudaEventCreate(&end));

```

```
    checkCudaErrors(cudaMemcpy(a_dev, a, size * size, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(b_dev, b, size, cudaMemcpyHostToDevice));

    dim3 threadsPerBlock(n, n);
    dim3 blocksPerGrid(1, 1);

    checkCudaErrors(cudaEventRecord(start));
     matrixVectorMultiplication<<<blocksPerGrid, threadsPerBlock>>>(a_dev, b_dev, c_dev, n);

    checkCudaErrors(cudaEventRecord(end));
    checkCudaErrors(cudaEventSynchronize(end));

    float time = 0.0;
    checkCudaErrors(cudaEventElapsedTime(&time, start, end));

    checkCudaErrors(cudaMemcpy(c, c_dev, size, cudaMemcpyDeviceToHost));

    // CPU matrixVector multiplication
    int sum = 0;
    for (int row = 0; row < n; row  ) {
        sum = 0;
        for (int col = 0; col < n; col  ) {
            sum = sum + a[row * n + col] * b[col];
        }
        d[row] = sum;
    }

    cout << "CPU product is  \n";
    for (int i = 0; i < n; i  ) {
        cout << d[i] << ", ";
    }
    cout << "\n\n";

    cout << "GPU product is  \n";
    for (int i = 0; i < n; i  ) {
        cout << c[i] << ", ";
    }
    cout << "\n\n";

    int error = 0;
    for (int i = 0; i < n; i  ) {
        error    d[i] - c[i];
        if (0    (d[i] - c[i])) {
            cout << "Error at (" << i << ")    GPU: " << c[i] << ", CPU: " << d[i] << "\n";
        }
    }

    cout << "Error: " << error;
    cout << "\nTime Elapsed: " << time;

    return 0;
}

/*

OUTPUT:

Given matrix is
3 6 7 5 3 5 6 2 9 1
7 0 9 3 6 0 6 2 6 1
7 9 2 0 2 3 7 5 9 2
8 9 7 3 6 1 2 9 3 1
4 7 8 4 5 0 3 6 1 0
3 2 0 6 1 5 5 4 7 6
6 9 3 7 4 5 2 5 4 7
4 3 0 7 8 6 8 8 4 3
4 9 2 0 6 8 9 2 6 6
9 5 0 4 8 7 1 7 2 7

Given vector is
 2, 8, 2, 9, 6, 5, 4, 1, 4, 2,

CPU product is
 220, 147, 190, 201, 168, 171, 245, 235, 234, 210,
```

```
145
146 GPU product is
147  220, 147, 190, 201, 168, 171, 245, 235, 234, 210,
148
149  Error: 0
150  Time Elapsed: 0.014336
151
152 */
153
```

HPC/4/vectorAdd.cpp

```cpp
#include <cstdlib>
#include <iostream>

#define checkCudaErrors(call)                                                                 \
    do {                                                                                      \
        cudaError_t err = call;                                                               \
        if (err  cudaSuccess) {                                                               \
            printf("CUDA error at %s %d: %s\n",   FILE  ,   LINE  , cudaGetErrorString(err)); \
            exit(EXIT_FAILURE);                                                               \
        }                                                                                     \
    } while (0)

using namespace std;

// VectorAdd parallel function
 global   void vectorAdd(int*a, int*b, int*result,   int n) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < n) {
    result[tid] = a[tid] + b[tid]; 20
    }
}

int main() {
    int *a, *b, *c;
    int *a_dev, *b_dev, *c_dev;
    int n = 1 << 4;

    a = new int[n];
    b = new int[n];
    c = new int[n];
    int *d = new int[n];
    int size = n * sizeof(int);
    checkCudaErrors(cudaMalloc(&a_dev, size));
    checkCudaErrors(cudaMalloc(&b_dev, size));
    checkCudaErrors(cudaMalloc(&c_dev, size));

    // Array initialization..You can use Randon function to assign values
    for (int i = 0; i < n; i  ) {
        a[i] = rand() % 1000;
        b[i] = rand() % 1000;
        d[i] = a[i] + b[i];  // calculating serial addition
    }
    cout << "Given array A is   \n";
    for (int i = 0; i < n; i  ) {
        cout << a[i] << ", ";
    }
    cout << "\n\n";

    cout << "Given array B is   \n";
    for (int i = 0; i < n; i  ) {
        cout << b[i] << ", ";
    }
    cout << "\n\n";

    cudaEvent_t start, end;

    checkCudaErrors(cudaEventCreate(&start));
    checkCudaErrors(cudaEventCreate(&end));

    checkCudaErrors(cudaMemcpy(a_dev, a, size, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(b_dev, b, size, cudaMemcpyHostToDevice));
    int threads = 1024;
    int blocks = (n + threads - 1) / threads;
    checkCudaErrors(cudaEventRecord(start));

    // Parallel addition program
    vectorAdd<<<blocks, threads>>>(a_dev, b_dev, c_dev, n);

    checkCudaErrors(cudaEventRecord(end));
    checkCudaErrors(cudaEventSynchronize(end));
```

```
    float time = 0.0;
    checkCudaErrors(cudaEventElapsedTime(&time, start, end));

    checkCudaErrors(cudaMemcpy(c, c_dev, size, cudaMemcpyDeviceToHost));

    // Calculate the error term.

    cout << "CPU sum is  \n";
    for (int i = 0; i < n; i  ) {
        cout << d[i] << ", ";
    }
    cout << "\n\n";

    cout << "GPU sum is  \n";
    for (int i = 0; i < n; i  ) {
        cout << c[i] << ", ";
    }
    cout << "\n\n";

    int error = 0;
    for (int i = 0; i < n; i  ) {
        error    d[i] - c[i];
        if (0    (d[i] - c[i])) {
            cout << "Error at (" << i << ")    GPU: " << c[i] << ", CPU:  " << d[i] << "\n";
        }
    }

    cout << "\nError : " << error;
    cout << "\nTime Elapsed: " << time;

    return 0;
}

/*

OUTPUT:

Given array A is
383, 777, 793, 386, 649, 362, 690, 763, 540, 172, 211, 567, 782, 862, 67, 929,

Given array B is
886, 915, 335, 492, 421, 27, 59, 926, 426, 736, 368, 429, 530, 123, 135, 802,

CPU sum is
1269, 1692, 1128, 878, 1070, 389, 749, 1689, 966, 908, 579, 996, 1312, 985, 202, 1731,

GPU sum is
1269, 1692, 1128, 878, 1070, 389, 749, 1689, 966, 908, 579, 996, 1312, 985, 202, 1731,


Error : 0
Time Elapsed:  0.017408

*/
```