

./swap_vectors.cu

Mon Oct 07 22:17:56 2024

1

```

1: #include <stdio.h>
2:
3:
4: const int DSIZE = 40960;
5: const int block_size = 256;
6: const int grid_size = DSIZE/block_size;
7:
8:
9: __global__ void vector_swap(float *A, float *B, float *C, int DSIZE) {
10:
11:     //FIXME:
12:     // Express the vector index in terms of threads and blocks
13:     int idx = threadIdx.x + blockIdx.x * blockDim.x;
14:     // Swap the vector elements - make sure you are not out of range
15:     if (idx < DSIZE) {
16:         C[idx] = A[idx];
17:         A[idx] = B[idx];
18:         B[idx] = C[idx];
19:     }
20: }
21:
22:
23: int main() {
24:
25:
26:     float *h_A, *h_B, *h_C, *d_A, *d_B, *d_C;
27:     int print_num = 10;
28:     h_A = new float[DSIZE];
29:     h_B = new float[DSIZE];
30:     h_C = new float[DSIZE];
31:
32:
33:     for (int i = 0; i < DSIZE; i++) {
34:         h_A[i] = rand() / (float)RAND_MAX;
35:         h_B[i] = rand() / (float)RAND_MAX;
36:         h_C[i] = 0;
37:     }
38:
39:     printf("Old A = [");
40:     for (int i = 0; i < print_num; i++) {
41:         printf("%f, ", h_A[i]);
42:     }
43:     printf("]\n");
44:
45:     printf("Old B = [");
46:     for (int i = 0; i < print_num; i++) {
47:         printf("%f, ", h_B[i]);
48:     }
49:     printf("]\n");
50:
51:     // Allocate memory for host and device pointers
52:     cudaMalloc(&d_A, DSIZE*sizeof(float));
53:     cudaMalloc(&d_B, DSIZE*sizeof(float));
54:     cudaMalloc(&d_C, DSIZE*sizeof(float));
55:
56:     // Copy from host to device
57:     cudaMemcpy(d_A, h_A, DSIZE*sizeof(float), cudaMemcpyHostToDevice);
58:     cudaMemcpy(d_B, h_B, DSIZE*sizeof(float), cudaMemcpyHostToDevice);
59:     cudaMemcpy(d_C, h_C, DSIZE*sizeof(float), cudaMemcpyHostToDevice);
60:
61:     // Launch the kernel
62:     vector_swap<<<grid_size, block_size>>>>(d_A, d_B, d_C, DSIZE);
63:
64:     // Copy back to host
65:     cudaMemcpy(h_A, d_A, DSIZE*sizeof(float), cudaMemcpyDeviceToHost);
66:     cudaMemcpy(h_B, d_B, DSIZE*sizeof(float), cudaMemcpyDeviceToHost);
67:
68:     // Print and check some elements to make sure swapping was successfull
69:     printf("New A = [");
70:     for (int i = 0; i < print_num; i++) {
71:         printf("%f, ", h_A[i]);
72:     }
73:     printf("]\n");
74:
75:     printf("New B = [");
76:     for (int i = 0; i < print_num; i++) {
77:         printf("%f, ", h_B[i]);
78:     }
79:     printf("]\n");
80:     // Free the memory
81:
82:     cudaFree(d_A);
83:     cudaFree(d_B);

```

```
84:     cudaFree(d_C);  
85:  
86:     free(h_A);  
87:     free(h_B);  
88:     free(h_C);  
89:     return 0;  
90: }
```

```
1: #include <stdio.h>
2:
3:
4: const int DSIZE_X = 256;
5: const int DSIZE_Y = 256;
6: const int block_size = 32;
7:
8: __global__ void add_matrix(const float *A, const float *B, float *C, int DSIZE_X, int DSIZE_Y)
9: {
10:     //FIXME:
11:     // Express in terms of threads and blocks
12:     int idx = threadIdx.x + blockDim.x * blockIdx.x;
13:     int idy = threadIdx.y + blockDim.y * blockIdx.y;
14:     // Add the two matrices - make sure you are not out of range
15:     if (idx < DSIZE_X && idy < DSIZE_Y)
16:         C[idy*DSIZE_Y + idx] = A[idy*DSIZE_Y + idx] + B[idy*DSIZE_Y + idx];
17: }
18:
19: int main()
20: {
21:     float *h_A, *h_B, *h_C, *d_A, *d_B, *d_C;
22:     int print_num = 3;
23:     // Create and allocate memory for host and device pointers
24:     h_A = new float[DSIZE_X * DSIZE_Y];
25:     h_B = new float[DSIZE_X * DSIZE_Y];
26:     h_C = new float[DSIZE_X * DSIZE_Y];
27:
28:     cudaMalloc(&d_A, DSIZE_X * DSIZE_Y*sizeof(float));
29:     cudaMalloc(&d_B, DSIZE_X * DSIZE_Y*sizeof(float));
30:     cudaMalloc(&d_C, DSIZE_X * DSIZE_Y*sizeof(float));
31:
32:     // Fill in the matrices
33:     // FIXME
34:     for (int i = 0; i < DSIZE_X; i++) {
35:         for (int j = 0; j < DSIZE_Y; j++) {
36:             h_A[i*DSIZE_X + j] = rand()/(float)RAND_MAX;
37:             h_B[i*DSIZE_X + j] = rand()/(float)RAND_MAX;
38:             h_C[i*DSIZE_X + j] = 0;
39:         }
40:     }
41:
42:     // Copy from host to device
43:     // Copy from host to device
44:     cudaMemcpy(d_A, h_A, DSIZE_X * DSIZE_Y*sizeof(float), cudaMemcpyHostToDevice);
45:     cudaMemcpy(d_B, h_B, DSIZE_X * DSIZE_Y*sizeof(float), cudaMemcpyHostToDevice);
46:     cudaMemcpy(d_C, h_C, DSIZE_X * DSIZE_Y*sizeof(float), cudaMemcpyHostToDevice);
47:
48:     // Launch the kernel
49:     // dim3 is a built in CUDA type that allows you to define the block
50:     // size and grid size in more than 1 dimentions
51:     // Syntax : dim3(Nx,Ny,Nz)
52:     dim3 blockSize(block_size, block_size);
53:     dim3 gridSize(DSIZE_X/block_size, DSIZE_Y/block_size);
54:
55:     add_matrix<<<gridSize, blockSize>>>(d_A, d_B, d_C, DSIZE_X, DSIZE_Y);
56:
57:     // Copy back to host
58:     cudaMemcpy(h_C, d_C, DSIZE_X * DSIZE_Y*sizeof(float), cudaMemcpyDeviceToHost);
59:     // Print and check some elements to make the addition was succesfull
60:     printf("A = [");
61:     for (int i = 0; i < print_num; i++) {
62:         printf("[ ");
63:         for (int j = 0; j < print_num; j++) {
64:             printf("%f, ", h_A[DSIZE_Y*j + i]);
65:         }
66:         printf("]\n");
67:     }
68:     printf("]\n");
69:
70:     printf("B = [");
71:     for (int i = 0; i < print_num; i++) {
72:         printf("[ ");
73:         for (int j = 0; j < print_num; j++) {
74:             printf("%f, ", h_B[DSIZE_Y*j + i]);
75:         }
76:         printf("]\n");
77:     }
78:     printf("]\n");
79:
80:     printf("A+B = [");
81:     for (int i = 0; i < print_num; i++) {
82:         printf("[ ");
83:         for (int j = 0; j < print_num; j++) {
```

```
84:         printf("%f, ", h_C[DSIZE_Y*j + i]);
85:     }
86:     printf("]\n");
87: }
88: printf("]\n");
89: // Free the memory
90: cudaFree(d_A);
91: cudaFree(d_B);
92: cudaFree(d_C);
93:
94: free(h_A);
95: free(h_B);
96: free(h_C);
97: return 0;
98: }
```

```

1: #include <stdio.h>
2: #include <time.h>
3:
4: const int DSIZE = 256;
5: const float A_val = 3.0f;
6: const float B_val = 2.0f;
7:
8: // error checking macro
9: #define cudaCheckErrors(msg) \
10: do { \
11:     cudaError_t __err = cudaGetLastError(); \
12:     if (__err != cudaSuccess) { \
13:         fprintf(stderr, "Fatal error: %s (%s at %s:%d)\n", \
14:             msg, cudaGetErrorString(__err), \
15:             __FILE__, __LINE__); \
16:         fprintf(stderr, "*** FAILED - ABORTING\n"); \
17:         exit(1); \
18:     } \
19: } while (0)
20:
21: // Square matrix multiplication on CPU : C = A * B
22: void matrix_mul_cpu(const float *A, const float *B, float *C, int size) {
23:     //FIXME:
24:     for (int i=0; i<size; i++){
25:         for (int j=0; j<size; j++){
26:             C[i*size+j] = 0.0;
27:             for (int k=0; k<size; k++){
28:                 C[i*size+j] += A[i*size+k]*B[k*size+j];
29:             }
30:         }
31:     }
32: }
33:
34: // Square matrix multiplication on GPU : C = A * B
35: __global__ void matrix_mul_gpu(const float *A, const float *B, float *C, int size) {
36:
37:     //FIXME:
38:     // create thread x index
39:     // create thread y index
40:     int idx = threadIdx.x + blockDim.x * blockIdx.x;
41:     int idy = threadIdx.y + blockDim.y * blockIdx.y;
42:     // Make sure we are not out of range
43:     if ((idx < size) && (idy < size)) {
44:         float temp = 0;
45:         for (int i = 0; i < size; i++){
46:             //FIXME : Add dot product of row and column
47:             temp += A[idx*size+i]*B[i*size+idy];
48:         }
49:         C[idx*size+idy] = temp;
50:     }
51: }
52:
53: int main() {
54:
55:     float *h_A, *h_B, *h_C, *h_Ccpu, *d_A, *d_B, *d_C;
56:     int print_num = 3;
57:     // These are used for timing
58:     clock_t t0, t1, t2, t3;
59:     double t1sum=0.0;
60:     double t2sum=0.0;
61:     double t3sum=0.0;
62:
63:     // start timing
64:     t0 = clock();
65:
66:     // N*N matrices defined in 1 dimention
67:     // If you prefer to do this in 2-dimention cupdate accordingly
68:     h_A = new float[DSIZE*DSIZE];
69:     h_B = new float[DSIZE*DSIZE];
70:     h_C = new float[DSIZE*DSIZE];
71:     h_Ccpu = new float[DSIZE*DSIZE];
72:     for (int i = 0; i < DSIZE*DSIZE; i++){
73:         h_A[i] = A_val;
74:         h_B[i] = B_val;
75:         h_C[i] = 0;
76:         h_Ccpu[i] = 0;
77:     }
78:
79:     // Initialization timing
80:     t1 = clock();
81:     t1sum = ((double) (t1-t0))/CLOCKS_PER_SEC;
82:     printf("Init took %f seconds. Begin compute\n", t1sum);
83:

```

```
84: // Allocate device memory and copy input data from host to device
85: cudaMalloc(&d_A, DSIZE*DSIZE*sizeof(float));
86: cudaMalloc(&d_B, DSIZE*DSIZE*sizeof(float));
87: cudaMalloc(&d_C, DSIZE*DSIZE*sizeof(float));
88: cudaCheckErrors("Allocaiton");
89: //FIXME:Add all other allocations and copies from host to device
90: cudaMemcpy(d_A, h_A, DSIZE*DSIZE*sizeof(float), cudaMemcpyHostToDevice);
91: cudaMemcpy(d_B, h_B, DSIZE*DSIZE*sizeof(float), cudaMemcpyHostToDevice);
92: cudaMemcpy(d_C, h_C, DSIZE*DSIZE*sizeof(float), cudaMemcpyHostToDevice);
93: cudaCheckErrors("Memory copy Host->Device");
94: // Launch kernel
95: // Specify the block and grid dimentions
96: dim3 block(32,32); //FIXME
97: dim3 grid(DSIZE/32,DSIZE/32); //FIXME
98: matrix_mul_gpu<<grid, block>>>(d_A, d_B, d_C, DSIZE);
99: cudaCheckErrors("Kernel Launch");
100: // Copy results back to host
101: cudaMemcpy(h_C, d_C, DSIZE*DSIZE*sizeof(float), cudaMemcpyDeviceToHost);
102: cudaCheckErrors("Memory copy Device->Host");
103: // GPU timing
104: t2 = clock();
105: t2sum = ((double)(t2-t1))/CLOCKS_PER_SEC;
106: printf("Done. GPU Compute took %f seconds\n", t2sum);
107:
108: // FIXME
109: // Excecute and time the cpu matrix multiplication function
110: matrix_mul_cpu(h_A, h_B, h_Ccpu, DSIZE);
111:
112: // CPU timing
113: t3 = clock();
114: t3sum = ((double)(t3-t2))/CLOCKS_PER_SEC;
115: printf("Done. CPU Compute took %f seconds\n", t3sum);
116:
117:
118: printf("C_GPU = [");
119: for (int i = 0; i < print_num; i++) {
120:     printf("[ ");
121:     for (int j = 0; j < print_num; j++) {
122:         printf("%f, ", h_C[DSIZE*j + i]);
123:     }
124:     printf("]\n");
125: }
126: printf("]\n");
127:
128: printf("C_CPU = [");
129: for (int i = 0; i < print_num; i++) {
130:     printf("[ ");
131:     for (int j = 0; j < print_num; j++) {
132:         printf("%f, ", h_Ccpu[DSIZE*j + i]);
133:     }
134:     printf("]\n");
135: }
136: printf("]\n");
137:
138: // FIXME
139: // Free memory
140: cudaFree(d_A);
141: cudaFree(d_B);
142: cudaFree(d_C);
143:
144: free(h_A);
145: free(h_B);
146: free(h_C);
147: free(h_Ccpu);
148:
149: return 0;
150:
151: }
152:
153: // for DSIZE = 256
154: // Done. GPU Compute took 0.326956 seconds
155: // Done. CPU Compute took 0.117284 seconds
156:
157: // for DSIZE = 512
158: // Done. GPU Compute took 0.348977 seconds
159: // Done. CPU Compute took 1.072144 seconds
```