

```
1: /*
2:
3: compile: "c++ -o c_code c_code.cpp"
4: run check: "./c_code -check"
5: run code with size 512 (default): "./c_code"
6: run code with size 2048: "./c_code -size 2048"
7:
8: */
9:
10: #include <stdio.h>
11: #include <stdlib.h>
12: #include <cstring>
13: #include <chrono>
14:
15: int* stencil_matmul(bool isrnad, int radius, const int DSIZE)
16: {
17:     int *h_A, *h_B, *h_Ac, *h_Bc, *h_C;
18:     int print_num = 3;
19:     // Create and allocate memory for host and device pointers
20:     h_A = new int[DSIZE * DSIZE];
21:     h_B = new int[DSIZE * DSIZE];
22:     h_Ac = new int[DSIZE * DSIZE];
23:     h_Bc = new int[DSIZE * DSIZE];
24:     h_C = new int[DSIZE * DSIZE];
25:
26:     // Fill in the matrices
27:     for (int i = 0; i < DSIZE; i++) {
28:         for (int j = 0; j < DSIZE; j++) {
29:             if (isrnad){
30:                 h_A[i*DSIZE + j] = rand() % 10;
31:                 h_B[i*DSIZE + j] = rand() % 10;
32:             } else{
33:                 h_A[i*DSIZE + j] = 1;
34:                 h_B[i*DSIZE + j] = 1;
35:             }
36:             h_Ac[i*DSIZE + j] = h_A[i*DSIZE + j];
37:             h_Bc[i*DSIZE + j] = h_B[i*DSIZE + j];
38:             h_C[i*DSIZE + j] = 0;
39:         }
40:     }
41:     int tempA = 0, tempB = 0;
42:     for (int idx = radius; idx < DSIZE-radius; idx++) {
43:         for (int idy = radius; idy < DSIZE-radius; idy++) {
44:             tempA = -h_A[idx*DSIZE + idy];
45:             tempB = -h_B[idx*DSIZE + idy];
46:             for (int idr = -radius; idr < radius+1; idr++) {
47:                 tempA += h_A[(idx+idr)*DSIZE + idy] + h_A[idx*DSIZE + idy+idr];
48:                 tempB += h_B[(idx+idr)*DSIZE + idy] + h_B[idx*DSIZE + idy+idr];
49:             }
50:             h_Ac[idx*DSIZE + idy] = tempA;
51:             h_Bc[idx*DSIZE + idy] = tempB;
52:         }
53:     }
54:
55:     for (int i=0; i<DSIZE; i++){
56:         for (int j=0; j<DSIZE; j++){
57:             h_C[i*DSIZE+j] = 0;
58:             for (int k=0; k<DSIZE; k++){
59:                 h_C[i*DSIZE+j] += h_Ac[i*DSIZE+k]*h_Bc[k*DSIZE+j];
60:             }
61:         }
62:     }
63:
64:     return h_C;
65: }
66:
67: int main(int argc, char const *argv[]){
68:     bool check = false, dsize_set = false;
69:     uint DSIZE;
70:
71:     if (argc > 1){
72:         if (strcmp(argv[1], "-check") == 0){
73:             check = true;
74:         }
75:         if (strcmp(argv[1], "-size") == 0){
76:             DSIZE = std::atoi(argv[2]);
77:             dsize_set = true;
78:         }
79:     }
80:     int print_num = 10;
81:     int * C;
82:     if (check){
83:         DSIZE = 10;
```

```
84:     C = stencil_matmul(false, 1, DSIZE);
85:     if (C[0] != 10)
86:         printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,0, C[0], 10);
87:     else if (C[1] != 42)
88:         printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,1, C[1], 42);
89:     else if (C[11] != 202)
90:         printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 2,1, C[11], 202);
91:     else
92:         printf("Sucess!\n");
93:
94:     printf("C = [\n");
95:     for (int i = 0; i < print_num; i++) {
96:         printf("    ");
97:         for (int j = 0; j < print_num; j++) {
98:             printf("%3d, ", C[DSIZE*j + i]);
99:         }
100:         printf("\b\b ]\n");
101:     }
102:     printf("    ]\n");
103:
104: } else{
105:     DSIZE = dsize_set ? DSIZE: 512;
106:     printf("the dsize is %d\n", DSIZE);
107:     const int radius = 3;
108:     auto start = std::chrono::steady_clock::now();
109:
110:     C = stencil_matmul(true, radius, DSIZE);
111:
112:     auto finish = std::chrono::steady_clock::now();
113:     double elapsed_seconds = std::chrono::duration_cast<std::chrono::duration<double>>(finish - start).
count();
114:     printf("time to run = %.2f S\n\n", elapsed_seconds);
115: }
116:
117: }
```

```
1: #include <cuda.h>
2: #include <iostream>
3: #include <cstdlib>
4: #include <cmath>
5:
6: #define CUDA_CHECK(call) \
7:     do { \
8:         cudaError_t err = call; \
9:         if (err != cudaSuccess) { \
10:             std::cerr << "CUDA error at " << __FILE__ << ":" << __LINE__ << ": " \
11:                 << cudaGetErrorString(err) << std::endl; \
12:             exit(EXIT_FAILURE); \
13:         } \
14:     } while (0)
15:
16: // Stencil kernel
17: __global__ void stencilKernel(const int* d_A, int* d_Ac, int DSIZE, int radius) {
18:     int idx = blockIdx.x * blockDim.x + threadIdx.x;
19:     int idy = blockIdx.y * blockDim.y + threadIdx.y;
20:
21:     if (idx >= radius && idx < DSIZE - radius && idy >= radius && idy < DSIZE - radius) {
22:         int temp = -d_A[idx * DSIZE + idy];
23:         for (int r = -radius; r < radius+1; r++) {
24:             temp += d_A[(idx + r) * DSIZE + idy] + d_A[idx * DSIZE + idy + r];
25:         }
26:         d_Ac[idx * DSIZE + idy] = temp;
27:     }
28: }
29:
30: // Matrix multiplication kernel
31: __global__ void matmulKernel(const int* d_Ac, const int* d_Bc, int* d_C, int DSIZE) {
32:     int row = blockIdx.y * blockDim.y + threadIdx.y;
33:     int col = blockIdx.x * blockDim.x + threadIdx.x;
34:
35:     if (row < DSIZE && col < DSIZE) {
36:         int sum = 0;
37:         for (int k = 0; k < DSIZE; ++k) {
38:             sum += d_Ac[row * DSIZE + k] * d_Bc[k * DSIZE + col];
39:         }
40:         d_C[row * DSIZE + col] = sum;
41:     }
42: }
43:
44: // Host function
45: int* stencilMatmul(bool isRand, int radius, const int DSIZE) {
46:     // Allocate host memory
47:     int *h_A, *h_B, *h_Ac, *h_Bc, *h_C;
48:     h_A = new int[DSIZE * DSIZE];
49:     h_B = new int[DSIZE * DSIZE];
50:     h_Ac = new int[DSIZE * DSIZE];
51:     h_Bc = new int[DSIZE * DSIZE];
52:     h_C = new int[DSIZE * DSIZE];
53:
54:     // Initialize matrices
55:     for (int i = 0; i < DSIZE; ++i) {
56:         for (int j = 0; j < DSIZE; ++j) {
57:             h_A[i * DSIZE + j] = isRand ? rand() % 10 : 1;
58:             h_B[i * DSIZE + j] = isRand ? rand() % 10 : 1;
59:             h_Ac[i * DSIZE + j] = h_A[i * DSIZE + j];
60:             h_Bc[i * DSIZE + j] = h_B[i * DSIZE + j];
61:             h_C[i * DSIZE + j] = 0;
62:         }
63:     }
64:
65:     // Allocate device memory
66:     int *d_A, *d_B, *d_Ac, *d_Bc, *d_C;
67:     CUDA_CHECK(cudaMalloc((void**)&d_A, DSIZE * DSIZE * sizeof(int)));
68:     CUDA_CHECK(cudaMalloc((void**)&d_B, DSIZE * DSIZE * sizeof(int)));
69:     CUDA_CHECK(cudaMalloc((void**)&d_Ac, DSIZE * DSIZE * sizeof(int)));
70:     CUDA_CHECK(cudaMalloc((void**)&d_Bc, DSIZE * DSIZE * sizeof(int)));
71:     CUDA_CHECK(cudaMalloc((void**)&d_C, DSIZE * DSIZE * sizeof(int)));
72:
73:     // Copy data to device
74:     CUDA_CHECK(cudaMemcpy(d_A, h_A, DSIZE * DSIZE * sizeof(int), cudaMemcpyHostToDevice));
75:     CUDA_CHECK(cudaMemcpy(d_B, h_B, DSIZE * DSIZE * sizeof(int), cudaMemcpyHostToDevice));
76:     CUDA_CHECK(cudaMemcpy(d_Ac, h_Ac, DSIZE * DSIZE * sizeof(int), cudaMemcpyHostToDevice));
77:     CUDA_CHECK(cudaMemcpy(d_Bc, h_Bc, DSIZE * DSIZE * sizeof(int), cudaMemcpyHostToDevice));
78:     CUDA_CHECK(cudaMemcpy(d_C, h_C, DSIZE * DSIZE * sizeof(int), cudaMemcpyHostToDevice));
79:
80:     // Kernel configurations
81:     dim3 blockDim(16, 16);
82:     dim3 gridDim((DSIZE + blockDim.x - 1) / blockDim.x, (DSIZE + blockDim.y - 1) / blockDim.y);
83: }
```

```
84: // Launch stencil kernels
85: stencilKernel<<<gridDim, blockDim>>>(d_A, d_Ac, DSIZE, radius);
86: stencilKernel<<<gridDim, blockDim>>>(d_B, d_Bc, DSIZE, radius);
87: CUDA_CHECK(cudaDeviceSynchronize());
88:
89: // Launch matrix multiplication kernel
90: matmulKernel<<<gridDim, blockDim>>>(d_Ac, d_Bc, d_C, DSIZE);
91: CUDA_CHECK(cudaDeviceSynchronize());
92:
93: // Copy result back to host
94: // CUDA_CHECK(cudaMemcpy(h_C, d_Ac, DSIZE * DSIZE * sizeof(int), cudaMemcpyDeviceToHost));
95: CUDA_CHECK(cudaMemcpy(h_C, d_C, DSIZE * DSIZE * sizeof(int), cudaMemcpyDeviceToHost));
96:
97: // Free device memory
98: CUDA_CHECK(cudaFree(d_A));
99: CUDA_CHECK(cudaFree(d_B));
100: CUDA_CHECK(cudaFree(d_Ac));
101: CUDA_CHECK(cudaFree(d_Bc));
102: CUDA_CHECK(cudaFree(d_C));
103:
104: // Free unused host memory
105: delete[] h_A;
106: delete[] h_B;
107: delete[] h_Ac;
108: delete[] h_Bc;
109:
110: return h_C;
111: }
112:
113: int main(int argc, char const *argv[]) {
114:     bool check = false;
115:     if (argc > 1 && strcmp(argv[1], "-check") == 0) {
116:         check = true;
117:     }
118:     int DSIZE;
119:     int print_num = 10;
120:     int * C;
121:     if (check) {
122:         DSIZE = 10;
123:         C = stencilMatmul(false, 1, DSIZE);
124:         if (C[0] != 10)
125:             printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,0, C[0], 10);
126:         if (C[1] != 42)
127:             printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,1, C[1], 42);
128:         if (C[11] != 202)
129:             printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 2,1, C[11], 202);
130:     } else {
131:         DSIZE = 512;
132:         const int radius = 3;
133:         C = stencilMatmul(true, radius, DSIZE);
134:     }
135:
136:     printf("C = \n");
137:     for (int i = 0; i < print_num; i++) {
138:         printf("    ");
139:         for (int j = 0; j < print_num; j++) {
140:             printf("%3d, ", C[DSIZE*j + i]);
141:         }
142:         printf("\b\b ]\n");
143:     }
144:     printf("    ]\n");
145:     // Free host memory
146:     delete[] C;
147:
148:     return 0;
149: }
```

```
1: #include <cuda.h>
2: #include <iostream>
3: #include <cstdlib>
4: #include <cmath>
5:
6: #define CUDA_CHECK(call) \
7:     do { \
8:         cudaError_t err = call; \
9:         if (err != cudaSuccess) { \
10:             std::cerr << "CUDA error at " << __FILE__ << ":" << __LINE__ << ": " \
11:                 << cudaGetErrorString(err) << std::endl; \
12:             exit(EXIT_FAILURE); \
13:         } \
14:     } while (0)
15:
16: // Stencil kernel
17: __global__ void stencilKernel(const int* A, int* Ac, int DSIZE, int radius) {
18:     int idx = blockIdx.x * blockDim.x + threadIdx.x;
19:     int idy = blockIdx.y * blockDim.y + threadIdx.y;
20:
21:     if (idx >= radius && idx < DSIZE - radius && idy >= radius && idy < DSIZE - radius) {
22:         int temp = -A[idx * DSIZE + idy];
23:         for (int r = -radius; r < radius+1; r++) {
24:             temp += A[(idx + r) * DSIZE + idy] + A[idx * DSIZE + idy + r];
25:         }
26:         Ac[idx * DSIZE + idy] = temp;
27:     }
28: }
29:
30: // Matrix multiplication kernel
31: __global__ void matmulKernel(const int* Ac, const int* Bc, int* C, int DSIZE) {
32:     int row = blockIdx.y * blockDim.y + threadIdx.y;
33:     int col = blockIdx.x * blockDim.x + threadIdx.x;
34:
35:     if (row < DSIZE && col < DSIZE) {
36:         int sum = 0;
37:         for (int k = 0; k < DSIZE; ++k) {
38:             sum += Ac[row * DSIZE + k] * Bc[k * DSIZE + col];
39:         }
40:         C[row * DSIZE + col] = sum;
41:     }
42: }
43:
44: // Host function
45: int* stencilMatmul(bool isRand, int radius, const int DSIZE) {
46:     // Unified memory allocation
47:     int *A, *B, *Ac, *Bc, *C;
48:     CUDA_CHECK(cudaMallocManaged(&A, DSIZE * DSIZE * sizeof(int)));
49:     CUDA_CHECK(cudaMallocManaged(&B, DSIZE * DSIZE * sizeof(int)));
50:     CUDA_CHECK(cudaMallocManaged(&Ac, DSIZE * DSIZE * sizeof(int)));
51:     CUDA_CHECK(cudaMallocManaged(&Bc, DSIZE * DSIZE * sizeof(int)));
52:     CUDA_CHECK(cudaMallocManaged(&C, DSIZE * DSIZE * sizeof(int)));
53:
54:     // Initialize matrices
55:     for (int i = 0; i < DSIZE; ++i) {
56:         for (int j = 0; j < DSIZE; ++j) {
57:             A[i * DSIZE + j] = isRand ? rand() % 10 : 1;
58:             B[i * DSIZE + j] = isRand ? rand() % 10 : 1;
59:             Ac[i * DSIZE + j] = A[i * DSIZE + j];
60:             Bc[i * DSIZE + j] = B[i * DSIZE + j];
61:             C[i * DSIZE + j] = 0;
62:         }
63:     }
64:
65:     // Kernel configurations
66:     dim3 blockDim(16, 16);
67:     dim3 gridDim((DSIZE + blockDim.x - 1) / blockDim.x, (DSIZE + blockDim.y - 1) / blockDim.y);
68:
69:     // Launch stencil kernels
70:     stencilKernel<<<gridDim, blockDim>>>(A, Ac, DSIZE, radius);
71:     stencilKernel<<<gridDim, blockDim>>>(B, Bc, DSIZE, radius);
72:     CUDA_CHECK(cudaDeviceSynchronize());
73:
74:     // Launch matrix multiplication kernel
75:     matmulKernel<<<gridDim, blockDim>>>(Ac, Bc, C, DSIZE);
76:     CUDA_CHECK(cudaDeviceSynchronize());
77:
78:     // Free unified memory
79:     CUDA_CHECK(cudaFree(A));
80:     CUDA_CHECK(cudaFree(B));
81:     CUDA_CHECK(cudaFree(Ac));
82:     CUDA_CHECK(cudaFree(Bc));
83: }
```

```
84:     return C; // Return result (managed memory pointer)
85: }
86:
87: int main(int argc, char const *argv[]) {
88:     bool check = false;
89:     if (argc > 1 && strcmp(argv[1], "-check") == 0){
90:         check = true;
91:     }
92:     int DSIZE;
93:     int print_num = 10;
94:     int * C;
95:     if (check){
96:         DSIZE = 10;
97:         C = stencilMatmul(false, 1, DSIZE);
98:         if (C[0] != 10)
99:             printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,0, C[0], 10);
100:        if (C[1] != 42)
101:            printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,1, C[1], 42);
102:        if (C[11] != 202)
103:            printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 2,1, C[11], 202);
104:    } else{
105:        DSIZE = 512;
106:        const int radius = 3;
107:        C = stencilMatmul(true, radius, DSIZE);
108:    }
109:
110:    printf("C = [\n");
111:    for (int i = 0; i < print_num; i++) {
112:        printf("    ");
113:        for (int j = 0; j < print_num; j++) {
114:            printf("%3d, ", C[DSIZE*j + i]);
115:        }
116:        printf("\b\b ]\n");
117:    }
118:    printf("    ]\n");
119:
120:    // Free unified memory for result
121:    CUDA_CHECK(cudaFree(C));
122:
123:    return 0;
124: }
```

```
1: #include <cuda.h>
2: #include <iostream>
3: #include <cstdlib>
4: #include <cmath>
5:
6: #define CUDA_CHECK(call) \
7:     do { \
8:         cudaError_t err = call; \
9:         if (err != cudaSuccess) { \
10:             std::cerr << "CUDA error at " << __FILE__ << ":" << __LINE__ << ": " \
11:                 << cudaGetErrorString(err) << std::endl; \
12:             exit(EXIT_FAILURE); \
13:         } \
14:     } while (0)
15:
16: // Stencil kernel with shared memory
17: __global__ void stencilKernelShared(const int* A, int* Ac, int DSIZE, int radius) {
18:     extern __shared__ int shared[];
19:     int tx = threadIdx.x;
20:     int ty = threadIdx.y;
21:
22:     int idx = blockIdx.x * blockDim.x + tx;
23:     int idy = blockIdx.y * blockDim.y + ty;
24:
25:     int localIdx = tx + radius;
26:     int localIdy = ty + radius;
27:
28:     // Copy to shared memory (with halo)
29:     if (idx < DSIZE && idy < DSIZE) {
30:         shared[localIdy * (blockDim.x + 2 * radius) + localIdx] = A[idy * DSIZE + idx];
31:     }
32:
33:     // Load halo regions
34:     if (tx < radius) {
35:         if (idx >= radius) {
36:             shared[localIdy * (blockDim.x + 2 * radius) + tx] = A[idy * DSIZE + (idx - radius)];
37:         }
38:         if (idx + blockDim.x < DSIZE) {
39:             shared[localIdy * (blockDim.x + 2 * radius) + (localIdx + blockDim.x)] =
40:                 A[idy * DSIZE + (idx + blockDim.x)];
41:         }
42:     }
43:     if (ty < radius) {
44:         if (idy >= radius) {
45:             shared[ty * (blockDim.x + 2 * radius) + localIdx] = A[(idy - radius) * DSIZE + idx];
46:         }
47:         if (idy + blockDim.y < DSIZE) {
48:             shared[(localIdy + blockDim.y) * (blockDim.x + 2 * radius) + localIdx] =
49:                 A[(idy + blockDim.y) * DSIZE + idx];
50:         }
51:     }
52:
53:     __syncthreads();
54:
55:     // Apply stencil
56:     if (idx >= radius && idx < DSIZE - radius && idy >= radius && idy < DSIZE - radius) {
57:         int temp = -shared[localIdy * (blockDim.x + 2 * radius) + localIdx];
58:         for (int r = -radius; r <= radius; ++r) {
59:             temp += shared[(localIdy + r) * (blockDim.x + 2 * radius) + localIdx];
60:             temp += shared[localIdy * (blockDim.x + 2 * radius) + (localIdx + r)];
61:         }
62:         Ac[idy * DSIZE + idx] = temp;
63:     }
64: }
65:
66: // Matrix multiplication kernel
67: __global__ void matmulKernel(const int* Ac, const int* Bc, int* C, int DSIZE) {
68:     int row = blockIdx.y * blockDim.y + threadIdx.y;
69:     int col = blockIdx.x * blockDim.x + threadIdx.x;
70:
71:     if (row < DSIZE && col < DSIZE) {
72:         int sum = 0;
73:         for (int k = 0; k < DSIZE; ++k) {
74:             sum += Ac[row * DSIZE + k] * Bc[k * DSIZE + col];
75:         }
76:         C[row * DSIZE + col] = sum;
77:     }
78: }
79:
80: // Host function
81: int* stencilMatmul(bool isRand, int radius, const int DSIZE) {
82:     // Unified memory allocation
83:     int *A, *B, *Ac, *Bc, *C;
```

```
84:     CUDA_CHECK(cudaMallocManaged(&A, DSIZE * DSIZE * sizeof(int)));
85:     CUDA_CHECK(cudaMallocManaged(&B, DSIZE * DSIZE * sizeof(int)));
86:     CUDA_CHECK(cudaMallocManaged(&Ac, DSIZE * DSIZE * sizeof(int)));
87:     CUDA_CHECK(cudaMallocManaged(&Bc, DSIZE * DSIZE * sizeof(int)));
88:     CUDA_CHECK(cudaMallocManaged(&C, DSIZE * DSIZE * sizeof(int)));
89:
90:     // Initialize matrices
91:     for (int i = 0; i < DSIZE; ++i) {
92:         for (int j = 0; j < DSIZE; ++j) {
93:             A[i * DSIZE + j] = isRand ? rand() % 10 : 1;
94:             B[i * DSIZE + j] = isRand ? rand() % 10 : 1;
95:             Ac[i * DSIZE + j] = A[i * DSIZE + j];
96:             Bc[i * DSIZE + j] = B[i * DSIZE + j];
97:             C[i * DSIZE + j] = 0;
98:         }
99:     }
100:
101:     // Kernel configurations
102:     dim3 blockDim(16, 16);
103:     dim3 gridDim((DSIZE + blockDim.x - 1) / blockDim.x, (DSIZE + blockDim.y - 1) / blockDim.y);
104:     int sharedMemSize = (blockDim.x + 2 * radius) * (blockDim.y + 2 * radius) * sizeof(int);
105:
106:     // Create CUDA streams
107:     cudaStream_t stream1, stream2;
108:     CUDA_CHECK(cudaStreamCreate(&stream1));
109:     CUDA_CHECK(cudaStreamCreate(&stream2));
110:
111:     // Launch stencil kernels on different streams
112:     stencilKernelShared<<<gridDim, blockDim, sharedMemSize, stream1>>>(A, Ac, DSIZE, radius);
113:     stencilKernelShared<<<gridDim, blockDim, sharedMemSize, stream2>>>(B, Bc, DSIZE, radius);
114:
115:     // Synchronize stencil streams
116:     CUDA_CHECK(cudaStreamSynchronize(stream1));
117:     CUDA_CHECK(cudaStreamSynchronize(stream2));
118:
119:     // Launch matrix multiplication kernel
120:     matmulKernel<<<gridDim, blockDim>>>(Ac, Bc, C, DSIZE);
121:     CUDA_CHECK(cudaDeviceSynchronize());
122:
123:     // Free CUDA streams
124:     CUDA_CHECK(cudaStreamDestroy(stream1));
125:     CUDA_CHECK(cudaStreamDestroy(stream2));
126:
127:     // Free unified memory
128:     CUDA_CHECK(cudaFree(A));
129:     CUDA_CHECK(cudaFree(B));
130:     CUDA_CHECK(cudaFree(Ac));
131:     CUDA_CHECK(cudaFree(Bc));
132:
133:     return C; // Return result (managed memory pointer)
134: }
135:
136: int main(int argc, char const *argv[]) {
137:     bool check = false;
138:     if (argc > 1 && strcmp(argv[1], "-check") == 0) {
139:         check = true;
140:     }
141:     int DSIZE;
142:     int print_num = 10;
143:     int * C;
144:     if (check) {
145:         DSIZE = 10;
146:         C = stencilMatmul(false, 1, DSIZE);
147:         if (C[0] != 10)
148:             printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,0, C[0], 10);
149:         if (C[1] != 42)
150:             printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,1, C[1], 42);
151:         if (C[11] != 202)
152:             printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 2,1, C[11], 202);
153:     } else {
154:         DSIZE = 512;
155:         const int radius = 3;
156:         C = stencilMatmul(true, radius, DSIZE);
157:     }
158:
159:     printf("C = [\n");
160:     for (int i = 0; i < print_num; i++) {
161:         printf("    ");
162:         for (int j = 0; j < print_num; j++) {
163:             printf("%3d, ", C[DSIZE*j + i]);
164:         }
165:         printf("\b\b ]\n");
166:     }
```



```
167:     printf("    ]\n");
168:
169:     // Free unified memory for result
170:     CUDA_CHECK(cudaFree(C));
171:
172:     return 0;
173: }
```

```
1: /*
2: compile: "nvcc -o cuda_final cuda_final.cu"
3: run check: "./cuda_final -check"
4: run code with size 512 (default): "./cuda_final"
5: run code with size 4096: "./cuda_final -size 4096"
6: */
7:
8: #include <cuda.h>
9: #include <iostream>
10: #include <cstdlib>
11: #include <cmath>
12: #include <chrono>
13:
14: #define CUDA_CHECK(call) \
15:     do { \
16:         cudaError_t err = call; \
17:         if (err != cudaSuccess) { \
18:             std::cerr << "CUDA error at " << __FILE__ << " " << __LINE__ << " " \
19:                 << cudaGetErrorString(err) << std::endl; \
20:             exit(EXIT_FAILURE); \
21:         } \
22:     } while (0)
23:
24: // Stencil kernel with shared memory
25: __global__ void stencilKernelShared(const int* A, int* Ac, int DSIZE, int radius) {
26:     extern __shared__ int shared[];
27:     int tx = threadIdx.x;
28:     int ty = threadIdx.y;
29:
30:     int idx = blockIdx.x * blockDim.x + tx;
31:     int idy = blockIdx.y * blockDim.y + ty;
32:
33:     int localIdx = tx + radius;
34:     int localIdy = ty + radius;
35:
36:     // Copy to shared memory (with halo)
37:     if (idx < DSIZE && idy < DSIZE) {
38:         shared[localIdy * (blockDim.x + 2 * radius) + localIdx] = A[idy * DSIZE + idx];
39:     }
40:
41:     // Load halo regions
42:     if (tx < radius) {
43:         if (idx >= radius) {
44:             shared[localIdy * (blockDim.x + 2 * radius) + tx] = A[idy * DSIZE + (idx - radius)];
45:         }
46:         if (idx + blockDim.x < DSIZE) {
47:             shared[localIdy * (blockDim.x + 2 * radius) + (localIdx + blockDim.x)] =
48:                 A[idy * DSIZE + (idx + blockDim.x)];
49:         }
50:     }
51:     if (ty < radius) {
52:         if (idy >= radius) {
53:             shared[ty * (blockDim.x + 2 * radius) + localIdx] = A[(idy - radius) * DSIZE + idx];
54:         }
55:         if (idy + blockDim.y < DSIZE) {
56:             shared[(localIdy + blockDim.y) * (blockDim.x + 2 * radius) + localIdx] =
57:                 A[(idy + blockDim.y) * DSIZE + idx];
58:         }
59:     }
60:
61:     __syncthreads();
62:
63:     // Apply stencil
64:     if (idx >= radius && idx < DSIZE - radius && idy >= radius && idy < DSIZE - radius) {
65:         int temp = -shared[localIdy * (blockDim.x + 2 * radius) + localIdx];
66:         for (int r = -radius; r <= radius; ++r) {
67:             temp += shared[(localIdy + r) * (blockDim.x + 2 * radius) + localIdx];
68:             temp += shared[localIdy * (blockDim.x + 2 * radius) + (localIdx + r)];
69:         }
70:         Ac[idy * DSIZE + idx] = temp;
71:     }
72: }
73:
74: // Matrix multiplication kernel
75: __global__ void matmulSharedKernel(const int* A, const int* B, int* C, int DSIZE) {
76:     extern __shared__ int shared[];
77:     int* tileA = shared;
78:     int* tileB = shared + blockDim.x * blockDim.y;
79:
80:     int tx = threadIdx.x;
81:     int ty = threadIdx.y;
82:     int row = blockIdx.y * blockDim.y + ty;
83:     int col = blockIdx.x * blockDim.x + tx;
```

```

84:
85:     int temp = 0;
86:
87:     // Loop over tiles
88:     for (int t = 0; t < (DSIZE + blockDim.x - 1) / blockDim.x; ++t) {
89:         // Load tiles into shared memory
90:         if (row < DSIZE && t * blockDim.x + tx < DSIZE) {
91:             tileA[ty * blockDim.x + tx] = A[row * DSIZE + t * blockDim.x + tx];
92:         } else {
93:             tileA[ty * blockDim.x + tx] = 0;
94:         }
95:         if (t * blockDim.y + ty < DSIZE && col < DSIZE) {
96:             tileB[ty * blockDim.x + tx] = B[(t * blockDim.y + ty) * DSIZE + col];
97:         } else {
98:             tileB[ty * blockDim.x + tx] = 0;
99:         }
100:
101:         __syncthreads();
102:
103:         // Perform partial computation for the tile
104:         for (int k = 0; k < blockDim.x; ++k) {
105:             temp += tileA[ty * blockDim.x + k] * tileB[k * blockDim.x + tx];
106:         }
107:
108:         __syncthreads();
109:     }
110:
111:     // Write result to global memory
112:     if (row < DSIZE && col < DSIZE) {
113:         C[row * DSIZE + col] = temp;
114:     }
115: }
116:
117: // Host function
118: int* stencilMatmul(bool isRand, int radius, const int DSIZE) {
119:     // Unified memory allocation
120:     int *A, *B, *Ac, *Bc, *C;
121:     CUDA_CHECK(cudaMallocManaged(&A, DSIZE * DSIZE * sizeof(int)));
122:     CUDA_CHECK(cudaMallocManaged(&B, DSIZE * DSIZE * sizeof(int)));
123:     CUDA_CHECK(cudaMallocManaged(&Ac, DSIZE * DSIZE * sizeof(int)));
124:     CUDA_CHECK(cudaMallocManaged(&Bc, DSIZE * DSIZE * sizeof(int)));
125:     CUDA_CHECK(cudaMallocManaged(&C, DSIZE * DSIZE * sizeof(int)));
126:
127:     // Initialize matrices
128:     for (int i = 0; i < DSIZE; ++i) {
129:         for (int j = 0; j < DSIZE; ++j) {
130:             A[i * DSIZE + j] = isRand ? rand() % 10 : 1;
131:             B[i * DSIZE + j] = isRand ? rand() % 10 : 1;
132:             Ac[i * DSIZE + j] = A[i * DSIZE + j];
133:             Bc[i * DSIZE + j] = B[i * DSIZE + j];
134:             C[i * DSIZE + j] = 0;
135:         }
136:     }
137:
138:     // Kernel configurations
139:     dim3 blockDim(16, 16);
140:     // dim3 gridDim((DSIZE + blockDim.x - 1) / blockDim.x, (DSIZE + blockDim.y - 1) / blockDim.y);
141:     dim3 gridDim(32, 32);
142:     int sharedMemSize = (blockDim.x + 2 * radius) * (blockDim.y + 2 * radius) * sizeof(int);
143:     int sharedMemMatmul = 2 * blockDim.x * blockDim.y * sizeof(int);
144:     // Create CUDA streams
145:     cudaStream_t stream1, stream2;
146:     CUDA_CHECK(cudaStreamCreate(&stream1));
147:     CUDA_CHECK(cudaStreamCreate(&stream2));
148:     printf("Grid : {%d, %d} blocks. Blocks : {%d, %d} threads.\n", gridDim.x, gridDim.y, blockDim.x, blockDim.y);
149:     // Launch stencil kernels on different streams
150:     stencilKernelShared<<<gridDim, blockDim, sharedMemSize, stream1>>>(A, Ac, DSIZE, radius);
151:     stencilKernelShared<<<gridDim, blockDim, sharedMemSize, stream2>>>(B, Bc, DSIZE, radius);
152:
153:     // Synchronize stencil streams
154:     CUDA_CHECK(cudaStreamSynchronize(stream1));
155:     CUDA_CHECK(cudaStreamSynchronize(stream2));
156:
157:     // Launch matrix multiplication kernel
158:     matmulSharedKernel<<<gridDim, blockDim, sharedMemMatmul>>>(Ac, Bc, C, DSIZE);
159:     CUDA_CHECK(cudaDeviceSynchronize());
160:
161:     // Free CUDA streams
162:     CUDA_CHECK(cudaStreamDestroy(stream1));
163:     CUDA_CHECK(cudaStreamDestroy(stream2));
164:
165:     // Free unified memory

```

```
166:     CUDA_CHECK(cudaFree(A));
167:     CUDA_CHECK(cudaFree(B));
168:     CUDA_CHECK(cudaFree(Ac));
169:     CUDA_CHECK(cudaFree(Bc));
170:
171:     return C; // Return result (managed memory pointer)
172: }
173:
174: int main(int argc, char const *argv[]) {
175:     bool check = false, dsize_set = false;
176:     uint DSIZE;
177:
178:     if (argc > 1){
179:         if (strcmp(argv[1], "-check") == 0){
180:             check = true;
181:         }
182:         if (strcmp(argv[1], "-size") == 0){
183:             DSIZE = std::atoi(argv[2]);
184:             dsize_set = true;
185:         }
186:     }
187:     int print_num = 10;
188:     int * C;
189:     if (check){
190:         DSIZE = 10;
191:         C = stencilMatmul(false, 1, DSIZE);
192:         if (C[0] != 10)
193:             printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,0, C[0], 10);
194:         else if (C[1] != 42)
195:             printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,1, C[1], 42);
196:         else if (C[11] != 202)
197:             printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 2,1, C[11], 202);
198:         else
199:             printf("Sucess!\n");
200:
201:         printf("C = [\n");
202:         for (int i = 0; i < print_num; i++) {
203:             printf("    ");
204:             for (int j = 0; j < print_num; j++) {
205:                 printf("%3d, ", C[DSIZE*j + i]);
206:             }
207:             printf("\b\b ]\n");
208:         }
209:         printf("]\n");
210:     } else{
211:         DSIZE = dsize_set ? DSIZE: 512;
212:         printf("the dsize is %d\n", DSIZE);
213:         const int radius = 3;
214:
215:         auto start = std::chrono::steady_clock::now();
216:
217:         C = stencilMatmul(true, radius, DSIZE);
218:
219:         auto finish = std::chrono::steady_clock::now();
220:         double elapsed_seconds = std::chrono::duration_cast<std::chrono::duration<double>>(finish - start).
count();
221:         printf("time to run = %.2f\n\n", elapsed_seconds);
222:     }
223:
224:     // Free unified memory for result
225:     CUDA_CHECK(cudaFree(C));
226:
227:     return 0;
228: }
```

```

1: /*
2:  cuda compile: "nvcc -x cu -std=c++17 -O2 -g --expt-relaxed-constexpr -I$ALPAKA_BASE/include -DALPAKA_ACC_GP
U_CUDA_ENABLED alpaka.cpp -o alpaka_cuda"
3:  run check: "./alpaka_cuda -check"
4:  run code with size 512 (default): "./alpaka_cuda"
5:  run code with size 4096: "./alpaka_cuda -size 4096"
6:
7:  cpu compile: "g++ -std=c++17 -O2 -g -I$ALPAKA_BASE/include -DALPAKA_ACC_CPU_B_SEQ_T_SEQ_ENABLED alpaka.cpp
-o alpaka_cpu"
8:  run check: "./alpaka_cpu -check"
9:  run code with size 512 (default): "./alpaka_cpu"
10: run code with size 2048: "./alpaka_cpu -size 2048"
11: */
12:
13: #include <cassert>
14: #include <cstdio>
15: #include <random>
16:
17: #include <alpaka/alpaka.hpp>
18:
19: #include "config.h"
20: #include "WorkDiv.hpp"
21: #include <chrono>
22:
23:
24: struct stencil2D {
25:     template <typename TAcc, typename T>
26:     ALPAKA_FN_ACC void operator()(TAcc const& acc,
27:                                   T const* __restrict__ d_A,
28:                                   T* __restrict__ d_Aout,
29:                                   int radius,
30:                                   Vec2D size) const {
31:         for (auto ndindex : alpaka::uniformElementsND(acc, size)) {
32:             auto idx = ndindex[0];
33:             auto idy = ndindex[1];
34:             auto DSIZE = size[1];
35:             // auto index = (ndindex[0] * size[1] + ndindex[1])
36:
37:             if (idx >= radius && idx < DSIZE - radius && idy >= radius && idy < DSIZE - radius) {
38:                 int temp = -d_A[idx * DSIZE + idy];
39:                 for (int r = -radius; r < radius+1; r++) {
40:                     temp += d_A[(idx + r) * DSIZE + idy] + d_A[idx * DSIZE + idy + r];
41:                 }
42:                 d_Aout[idx * DSIZE + idy] = temp;
43:             }
44:         }
45:     }
46: };
47:
48: struct matrixmul {
49:     template <typename TAcc, typename T>
50:     ALPAKA_FN_ACC void operator()(TAcc const& acc,
51:                                   T const* __restrict__ d_A,
52:                                   T const* __restrict__ d_B,
53:                                   T* __restrict__ d_About,
54:                                   Vec2D size) const {
55:         for (auto ndindex : alpaka::uniformElementsND(acc, size)) {
56:             auto idx = ndindex[0];
57:             auto idy = ndindex[1];
58:             auto DSIZE = size[1];
59:             // auto index = (ndindex[0] * size[1] + ndindex[1])
60:             if (idy < DSIZE && idx < DSIZE) {
61:                 int sum = 0;
62:                 for (int k = 0; k < DSIZE; ++k) {
63:                     sum += d_A[idy * DSIZE + k] * d_B[k * DSIZE + idx];
64:                 }
65:                 d_About[idy * DSIZE + idx] = sum;
66:             }
67:         }
68:     }
69: };
70: };
71:
72: // Host function
73: void stencilMatmul(Host host, Platform platform, Device device, bool isRand, int radius, const int DSIZE, i
nt* out) {
74:
75:     // 3-dimensional and linearised buffer size
76:     Vec2D ndsize = {DSIZE, DSIZE};
77:     uint32_t size = ndsize.prod();
78:     auto h_A = alpaka::allocMappedBuf<int, uint32_t>(host, platform, size);
79:     auto h_B = alpaka::allocMappedBuf<int, uint32_t>(host, platform, size);
80:     auto h_As = alpaka::allocMappedBuf<int, uint32_t>(host, platform, size);

```

```

81:     auto h_Bs = alpaka::allocMappedBuf<int, uint32_t>(host, platform, size);
82:     auto h_C = alpaka::allocMappedBuf<int, uint32_t>(host, platform, size);
83:
84:     for (uint32_t i = 0; i < size; ++i) {
85:         h_A[i] = isRand ? rand() % 10 : 1;
86:         h_B[i] = isRand ? rand() % 10 : 1;
87:         h_As[i] = h_A[i];
88:         h_Bs[i] = h_B[i];
89:     }
90:
91:     // run the test the given device
92:     auto queue = Queue{device};
93:
94:     // allocate input and output buffers on the device
95:     auto d_A = alpaka::allocAsyncBuf<int, uint32_t>(queue, size);
96:     auto d_B = alpaka::allocAsyncBuf<int, uint32_t>(queue, size);
97:     auto d_As = alpaka::allocAsyncBuf<int, uint32_t>(queue, size);
98:     auto d_Bs = alpaka::allocAsyncBuf<int, uint32_t>(queue, size);
99:     auto d_C = alpaka::allocAsyncBuf<int, uint32_t>(queue, size);
100:
101:     // copy the input data to the device; the size is known from the buffer objects
102:     alpaka::memcpy(queue, d_A, h_A);
103:     alpaka::memcpy(queue, d_B, h_B);
104:     alpaka::memcpy(queue, d_As, h_As);
105:     alpaka::memcpy(queue, d_Bs, h_Bs);
106:
107:     alpaka::memset(queue, d_C, 0x00);
108:
109:     // launch the 3-dimensional kernel
110:     auto div = makeWorkDiv<Acc2D>({32, 32}, {16, 16});
111:     std::cout << "Testing VectorAddKernel3D with vector indices with a grid of "
112:         << alpaka::getWorkDiv<alpaka::Grid, alpaka::Blocks>(div) << " blocks x "
113:         << alpaka::getWorkDiv<alpaka::Block, alpaka::Threads>(div) << " threads x "
114:         << alpaka::getWorkDiv<alpaka::Thread, alpaka::Elements>(div) << " elements...\n";
115:     alpaka::exec<Acc2D>(
116:         queue, div, stencil2D{}, d_A.data(), d_As.data(), radius, ndsize);
117:     alpaka::exec<Acc2D>(
118:         queue, div, stencil2D{}, d_B.data(), d_Bs.data(), radius, ndsize);
119:     alpaka::wait(queue);
120:     alpaka::exec<Acc2D>(
121:         queue, div, matrixmul{}, d_As.data(), d_Bs.data(), d_C.data(), ndsize);
122:
123:     // copy the results from the device to the host
124:     alpaka::memcpy(queue, h_C, d_C);
125:
126:     // wait for all the operations to complete
127:     alpaka::wait(queue);
128:     // alpaka::memcpy(queue, out, h_C);
129:     for (uint32_t i = 0; i < size; ++i) {
130:         out[i] = h_C[i];
131:     }
132: }
133:
134: int main(int argc, char const *argv[]) {
135:     // initialise the accelerator platform
136:     Platform platform;
137:     // require at least one device
138:     std::uint32_t n = alpaka::getDevCount(platform);
139:     if (n == 0) {
140:         exit(EXIT_FAILURE);
141:     }
142:
143:     // use the single host device
144:     HostPlatform host_platform;
145:     Host host = alpaka::getDevByIdx(host_platform, 0u);
146:     std::cout << "Host: " << alpaka::getName(host) << '\n';
147:
148:     // use the first device
149:     Device device = alpaka::getDevByIdx(platform, 0u);
150:     std::cout << "Device: " << alpaka::getName(device) << '\n';
151:
152:
153:     bool check = false, dsize_set = false;
154:     uint DSIZE;
155:
156:     if (argc > 1) {
157:         if (strcmp(argv[1], "-check") == 0) {
158:             check = true;
159:         }
160:         if (strcmp(argv[1], "-size") == 0) {
161:             DSIZE = std::atoi(argv[2]);
162:             dsize_set = true;
163:         }

```

```
164:     }
165:     int print_num = 10;
166:     int * C;
167:     if (check){
168:         DSIZE = 10;
169:         printf("Matrix Size - %d\n", DSIZE);
170:         C = new int[DSIZE * DSIZE];
171:         stencilMatmul(host, platform, device, false, 1, DSIZE, C);
172:         if (C[0] != 10)
173:             printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,0, C[0], 10);
174:         else if (C[1] != 42)
175:             printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,1, C[1], 42);
176:         else if (C[11] != 202)
177:             printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 2,1, C[11], 202);
178:         else
179:             printf("Sucess!\n");
180:
181:         // print test result
182:         printf("C = [\n");
183:         for (int i = 0; i < print_num; i++) {
184:             printf("    ");
185:             for (int j = 0; j < print_num; j++) {
186:                 printf("%3d, ", C[DSIZE*j + i]);
187:             }
188:             printf("\b\b ]\n");
189:         }
190:         printf("    ]\n");
191:     } else{
192:         // set DSIZE from CLI arg.
193:         DSIZE = dsize_set ? DSIZE: 512;
194:         printf("Matrix Size - %d\n", DSIZE);
195:
196:         // Start clock
197:         auto start = std::chrono::steady_clock::now();
198:         const int radius = 3;
199:         C = new int[DSIZE * DSIZE];
200:         stencilMatmul(host, platform, device, true, radius, DSIZE, C);
201:
202:         // Stop clock
203:         auto finish = std::chrono::steady_clock::now();
204:         double elapsed_seconds = std::chrono::duration_cast<std::chrono::duration<double>>(finish - start).
count();
205:         printf("time to run = %.2f S\n\n", elapsed_seconds);
206:     }
207:     // Free host memory
208:     delete[] C;
209:
210:     return 0;
211: }
```