

./c_code.cpp

Fri Dec 06 03:57:35 2024

1

```

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: /*
2:  # Basic cuda implementation
3:
4:  compile: "nvcc -o cuda_code cuda_code.cu"
5:  run check: "./cuda_code -check"
6:  run code with size 512 (default): "./cuda_code"
7:  run code with size 4096: "./cuda_code -size 4096"
8:
9:  nsys comment: Most time speant on memory allocation and movement (about 90%)
10: */
11:
12: #include <cuda.h>
13: #include <iostream>
14: #include <cstdlib>
15: #include <cmath>
16: #include <chrono>
17:
18: #define CUDA_CHECK(call) \
19:     do { \
20:         cudaError_t err = call; \
21:         if (err != cudaSuccess) { \
22:             std::cerr << "CUDA error at " << __FILE__ << ":" << __LINE__ << ": " \
23:                 << cudaGetErrorString(err) << std::endl; \
24:             exit(EXIT_FAILURE); \
25:         } \
26:     } while (0)
27:
28: // Stencil kernel
29: __global__ void stencilKernel(const int* d_A, int* d_Ac, int DSIZE, int radius) {
30:     int idx = blockIdx.x * blockDim.x + threadIdx.x;
31:     int idy = blockIdx.y * blockDim.y + threadIdx.y;
32:
33:     if (idx >= radius && idx < DSIZE - radius && idy >= radius && idy < DSIZE - radius) {
34:         int temp = -d_A[idx * DSIZE + idy];
35:         for (int r = -radius; r < radius+1; r++) {
36:             temp += d_A[(idx + r) * DSIZE + idy] + d_A[idx * DSIZE + idy + r];
37:         }
38:         d_Ac[idx * DSIZE + idy] = temp;
39:     }
40: }
41:
42: // Matrix multiplication kernel
43: __global__ void matmulKernel(const int* d_Ac, const int* d_Bc, int* d_C, int DSIZE) {
44:     int row = blockIdx.y * blockDim.y + threadIdx.y;
45:     int col = blockIdx.x * blockDim.x + threadIdx.x;
46:
47:     if (row < DSIZE && col < DSIZE) {
48:         int sum = 0;
49:         for (int k = 0; k < DSIZE; ++k) {
50:             sum += d_Ac[row * DSIZE + k] * d_Bc[k * DSIZE + col];
51:         }
52:         d_C[row * DSIZE + col] = sum;
53:     }
54: }
55:
56: // Host function
57: int* stencilMatmul(bool isRand, int radius, const int DSIZE) {
58:     // Allocate host memory
59:     int *h_A, *h_B, *h_Ac, *h_Bc, *h_C;
60:     h_A = new int[DSIZE * DSIZE];
61:     h_B = new int[DSIZE * DSIZE];
62:     h_Ac = new int[DSIZE * DSIZE];
63:     h_Bc = new int[DSIZE * DSIZE];
64:     h_C = new int[DSIZE * DSIZE];
65:
66:     // Initialize matrices
67:     for (int i = 0; i < DSIZE; ++i) {
68:         for (int j = 0; j < DSIZE; ++j) {
69:             h_A[i * DSIZE + j] = isRand ? rand() % 10 : 1;
70:             h_B[i * DSIZE + j] = isRand ? rand() % 10 : 1;
71:             h_Ac[i * DSIZE + j] = h_A[i * DSIZE + j];
72:             h_Bc[i * DSIZE + j] = h_B[i * DSIZE + j];
73:             h_C[i * DSIZE + j] = 0;
74:         }
75:     }
76:
77:     // Allocate device memory
78:     int *d_A, *d_B, *d_Ac, *d_Bc, *d_C;
79:     CUDA_CHECK(cudaMalloc((void**)&d_A, DSIZE * DSIZE * sizeof(int)));
80:     CUDA_CHECK(cudaMalloc((void**)&d_B, DSIZE * DSIZE * sizeof(int)));
81:     CUDA_CHECK(cudaMalloc((void**)&d_Ac, DSIZE * DSIZE * sizeof(int)));
82:     CUDA_CHECK(cudaMalloc((void**)&d_Bc, DSIZE * DSIZE * sizeof(int)));
83:     CUDA_CHECK(cudaMalloc((void**)&d_C, DSIZE * DSIZE * sizeof(int)));

```

```
84:
85:     // Copy data to device
86:     CUDA_CHECK(cudaMemcpy(d_A, h_A, DSIZE * DSIZE * sizeof(int), cudaMemcpyHostToDevice));
87:     CUDA_CHECK(cudaMemcpy(d_B, h_B, DSIZE * DSIZE * sizeof(int), cudaMemcpyHostToDevice));
88:     CUDA_CHECK(cudaMemcpy(d_Ac, h_Ac, DSIZE * DSIZE * sizeof(int), cudaMemcpyHostToDevice));
89:     CUDA_CHECK(cudaMemcpy(d_Bc, h_Bc, DSIZE * DSIZE * sizeof(int), cudaMemcpyHostToDevice));
90:     CUDA_CHECK(cudaMemcpy(d_C, h_C, DSIZE * DSIZE * sizeof(int), cudaMemcpyHostToDevice));
91:
92:     // Kernel configurations
93:     dim3 blockDim(16, 16);
94:     dim3 gridDim((DSIZE + blockDim.x - 1) / blockDim.x, (DSIZE + blockDim.y - 1) / blockDim.y);
95:
96:     // Launch stencil kernels
97:     stencilKernel<<<gridDim, blockDim>>>(d_A, d_Ac, DSIZE, radius);
98:     stencilKernel<<<gridDim, blockDim>>>(d_B, d_Bc, DSIZE, radius);
99:     CUDA_CHECK(cudaDeviceSynchronize());
100:
101:     // Launch matrix multiplication kernel
102:     matmulKernel<<<gridDim, blockDim>>>(d_Ac, d_Bc, d_C, DSIZE);
103:     CUDA_CHECK(cudaDeviceSynchronize());
104:
105:     // Copy result back to host
106:     // CUDA_CHECK(cudaMemcpy(h_C, d_Ac, DSIZE * DSIZE * sizeof(int), cudaMemcpyDeviceToHost));
107:     CUDA_CHECK(cudaMemcpy(h_C, d_C, DSIZE * DSIZE * sizeof(int), cudaMemcpyDeviceToHost));
108:
109:     // Free device memory
110:     CUDA_CHECK(cudaFree(d_A));
111:     CUDA_CHECK(cudaFree(d_B));
112:     CUDA_CHECK(cudaFree(d_Ac));
113:     CUDA_CHECK(cudaFree(d_Bc));
114:     CUDA_CHECK(cudaFree(d_C));
115:
116:     // Free unused host memory
117:     delete[] h_A;
118:     delete[] h_B;
119:     delete[] h_Ac;
120:     delete[] h_Bc;
121:
122:     return h_C;
123: }
124:
125: int main(int argc, char const *argv[]) {
126:     bool check = false, dsize_set = false;
127:     uint DSIZE;
128:
129:     if (argc > 1){
130:         if (strcmp(argv[1], "--check") == 0){
131:             check = true;
132:         }
133:         if (strcmp(argv[1], "--size") == 0){
134:             DSIZE = std::atoi(argv[2]);
135:             dsize_set = true;
136:         }
137:     }
138:     int print_num = 10;
139:     int * C;
140:     if (check){
141:         DSIZE = 10;
142:         C = stencilMatmul(false, 1, DSIZE);
143:         if (C[0] != 10)
144:             printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,0, C[0], 10);
145:         else if (C[1] != 42)
146:             printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,1, C[1], 42);
147:         else if (C[11] != 202)
148:             printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 2,1, C[11], 202);
149:         else
150:             printf("Success!\n");
151:
152:         printf("C = [\n");
153:         for (int i = 0; i < print_num; i++) {
154:             printf("    [" );
155:             for (int j = 0; j < print_num; j++) {
156:                 printf("%3d, ", C[DSIZE*j + i]);
157:             }
158:             printf("\b\b ]\n");
159:         }
160:         printf("    ]\n");
161:     } else{
162:         DSIZE = dsize_set ? DSIZE : 512;
163:         printf("the dsize is %d\n", DSIZE);
164:         const int radius = 3;
165:
166:         auto start = std::chrono::steady_clock::now();
```

```
167:
168:         C = stencilMatmul(true, radius, DSIZE);
169:
170:         auto finish = std::chrono::steady_clock::now();
171:         double elapsed_seconds = std::chrono::duration_cast<std::chrono::duration<double>>(finish - start).
count();
172:         printf("time to run = %.2f\n\n", elapsed_seconds);
173:     }
174:
175:     // Free unified memory for result
176:     delete[] C;
177:
178:     return 0;
179: }
```

```

1:  /*
2:  # Managed memory
3:  compile: "nvcc -o cuda_mm cuda_mm.cu"
4:  run check: "./cuda_mm -check"
5:  run code with size 512 (default): "./cuda_mm"
6:  run code with size 4096: "./cuda_mm -size 4096"
7:
8:  nsys comment: Most time speant on memory allocation and movement (about 87%) slight improvement to cuda_cod
e.
9:  */
10:
11: #include <cuda.h>
12: #include <iostream>
13: #include <cstdlib>
14: #include <cmath>
15: #include <chrono>
16:
17: #define CUDA_CHECK(call) \
18:     do { \
19:         cudaError_t err = call; \
20:         if (err != cudaSuccess) { \
21:             std::cerr << "CUDA error at " << __FILE__ << ":" << __LINE__ << ": " \
22:                 << cudaGetErrorString(err) << std::endl; \
23:             exit(EXIT_FAILURE); \
24:         } \
25:     } while (0)
26:
27: // Stencil kernel
28: __global__ void stencilKernel(const int* A, int* Ac, int DSIZE, int radius) {
29:     int idx = blockIdx.x * blockDim.x + threadIdx.x;
30:     int idy = blockIdx.y * blockDim.y + threadIdx.y;
31:
32:     if (idx >= radius && idx < DSIZE - radius && idy >= radius && idy < DSIZE - radius) {
33:         int temp = -A[idx * DSIZE + idy];
34:         for (int r = -radius; r < radius+1; r++) {
35:             temp += A[(idx + r) * DSIZE + idy] + A[idx * DSIZE + idy + r];
36:         }
37:         Ac[idx * DSIZE + idy] = temp;
38:     }
39: }
40:
41: // Matrix multiplication kernel
42: __global__ void matmulKernel(const int* Ac, const int* Bc, int* C, int DSIZE) {
43:     int row = blockIdx.y * blockDim.y + threadIdx.y;
44:     int col = blockIdx.x * blockDim.x + threadIdx.x;
45:
46:     if (row < DSIZE && col < DSIZE) {
47:         int sum = 0;
48:         for (int k = 0; k < DSIZE; ++k) {
49:             sum += Ac[row * DSIZE + k] * Bc[k * DSIZE + col];
50:         }
51:         C[row * DSIZE + col] = sum;
52:     }
53: }
54:
55: // Host function
56: int* stencilMatmul(bool isRand, int radius, const int DSIZE) {
57:     // Unified memory allocation
58:     int *A, *B, *Ac, *Bc, *C;
59:     CUDA_CHECK(cudaMallocManaged(&A, DSIZE * DSIZE * sizeof(int)));
60:     CUDA_CHECK(cudaMallocManaged(&B, DSIZE * DSIZE * sizeof(int)));
61:     CUDA_CHECK(cudaMallocManaged(&Ac, DSIZE * DSIZE * sizeof(int)));
62:     CUDA_CHECK(cudaMallocManaged(&Bc, DSIZE * DSIZE * sizeof(int)));
63:     CUDA_CHECK(cudaMallocManaged(&C, DSIZE * DSIZE * sizeof(int)));
64:
65:     // Initialize matrices
66:     for (int i = 0; i < DSIZE; ++i) {
67:         for (int j = 0; j < DSIZE; ++j) {
68:             A[i * DSIZE + j] = isRand ? rand() % 10 : 1;
69:             B[i * DSIZE + j] = isRand ? rand() % 10 : 1;
70:             Ac[i * DSIZE + j] = A[i * DSIZE + j];
71:             Bc[i * DSIZE + j] = B[i * DSIZE + j];
72:             C[i * DSIZE + j] = 0;
73:         }
74:     }
75:
76:     // Kernel configurations
77:     dim3 blockDim(16, 16);
78:     dim3 gridDim((DSIZE + blockDim.x - 1) / blockDim.x, (DSIZE + blockDim.y - 1) / blockDim.y);
79:
80:     // Launch stencil kernels
81:     stencilKernel<<<gridDim, blockDim>>>(A, Ac, DSIZE, radius);
82:     stencilKernel<<<gridDim, blockDim>>>(B, Bc, DSIZE, radius);

```

```
83:     CUDA_CHECK(cudaDeviceSynchronize());
84:
85:     // Launch matrix multiplication kernel
86:     matmulKernel<<<gridDim, blockDim>>>(Ac, Bc, C, DSIZE);
87:     CUDA_CHECK(cudaDeviceSynchronize());
88:
89:     // Free unified memory
90:     CUDA_CHECK(cudaFree(A));
91:     CUDA_CHECK(cudaFree(B));
92:     CUDA_CHECK(cudaFree(Ac));
93:     CUDA_CHECK(cudaFree(Bc));
94:
95:     return C; // Return result (managed memory pointer)
96: }
97:
98: int main(int argc, char const *argv[]) {
99:     bool check = false, dsize_set = false;
100:     uint DSIZE;
101:
102:     if (argc > 1) {
103:         if (strcmp(argv[1], "-check") == 0) {
104:             check = true;
105:         }
106:         if (strcmp(argv[1], "-size") == 0) {
107:             DSIZE = std::atoi(argv[2]);
108:             dsize_set = true;
109:         }
110:     }
111:     int print_num = 10;
112:     int * C;
113:     if (check) {
114:         DSIZE = 10;
115:         C = stencilMatmul(false, 1, DSIZE);
116:         if (C[0] != 10)
117:             printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,0, C[0], 10);
118:         else if (C[1] != 42)
119:             printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,1, C[1], 42);
120:         else if (C[11] != 202)
121:             printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 2,1, C[11], 202);
122:         else
123:             printf("Sucess!\n");
124:
125:         printf("C = [\n");
126:         for (int i = 0; i < print_num; i++) {
127:             printf("    ");
128:             for (int j = 0; j < print_num; j++) {
129:                 printf("%3d, ", C[DSIZE*j + i]);
130:             }
131:             printf("\b\b ]\n");
132:         }
133:         printf("    ]\n");
134:     } else {
135:         DSIZE = dsize_set ? DSIZE: 512;
136:         printf("the dsize is %d\n", DSIZE);
137:         const int radius = 3;
138:
139:         auto start = std::chrono::steady_clock::now();
140:
141:         C = stencilMatmul(true, radius, DSIZE);
142:
143:         auto finish = std::chrono::steady_clock::now();
144:         double elapsed_seconds = std::chrono::duration_cast<std::chrono::duration<double>>(finish - start).
count();
145:         printf("time to run = %.2f\n\n", elapsed_seconds);
146:     }
147:
148:     // Free unified memory for result
149:     CUDA_CHECK(cudaFree(C));
150:
151:     return 0;
152: }
```

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



```

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

```

```
166:     CUDA_CHECK(cudaStreamDestroy(stream1));
167:     CUDA_CHECK(cudaStreamDestroy(stream2));
168:
169:     // Free unified memory
170:     CUDA_CHECK(cudaFree(A));
171:     CUDA_CHECK(cudaFree(B));
172:     CUDA_CHECK(cudaFree(Ac));
173:     CUDA_CHECK(cudaFree(Bc));
174:
175:     return C; // Return result (managed memory pointer)
176: }
177:
178: int main(int argc, char const *argv[]) {
179:     bool check = false, dsize_set = false;
180:     uint DSIZE;
181:
182:     if (argc > 1) {
183:         if (strcmp(argv[1], "-check") == 0) {
184:             check = true;
185:         }
186:         if (strcmp(argv[1], "-size") == 0) {
187:             DSIZE = std::atoi(argv[2]);
188:             dsize_set = true;
189:         }
190:     }
191:     int print_num = 10;
192:     int * C;
193:     if (check) {
194:         DSIZE = 10;
195:         C = stencilMatmul(false, 1, DSIZE);
196:         if (C[0] != 10)
197:             printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,0, C[0], 10);
198:         else if (C[1] != 42)
199:             printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,1, C[1], 42);
200:         else if (C[11] != 202)
201:             printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 2,1, C[11], 202);
202:         else
203:             printf("Success!\n");
204:
205:         printf("C = \n");
206:         for (int i = 0; i < print_num; i++) {
207:             printf("    ");
208:             for (int j = 0; j < print_num; j++) {
209:                 printf("%3d, ", C[DSIZE*j + i]);
210:             }
211:             printf("\b\b ]\n");
212:         }
213:         printf("    ]\n");
214:     } else {
215:         DSIZE = dsize_set ? DSIZE : 512;
216:         printf("the dsize is %d\n", DSIZE);
217:         const int radius = 3;
218:
219:         auto start = std::chrono::steady_clock::now();
220:
221:         C = stencilMatmul(true, radius, DSIZE);
222:
223:         auto finish = std::chrono::steady_clock::now();
224:         double elapsed_seconds = std::chrono::duration_cast<std::chrono::duration<double>>(finish - start).
count();
225:         printf("time to run = %.2f\n\n", elapsed_seconds);
226:     }
227:
228:     // Free unified memory for result
229:     CUDA_CHECK(cudaFree(C));
230:
231:     return 0;
232: }
```

```

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

```

```

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

```

```
164:         dsize_set = true;
165:     }
166: }
167: int print_num = 10;
168: int * C;
169: if (check){
170:     DSIZE = 10;
171:     printf("Matrix Size - %d\n", DSIZE);
172:     C = new int[DSIZE * DSIZE];
173:     stencilMatmul(host, platform, device, false, 1, DSIZE, C);
174:     if (C[0] != 10)
175:         printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,0, C[0], 10);
176:     else if (C[1] != 42)
177:         printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,1, C[1], 42);
178:     else if (C[11] != 202)
179:         printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 2,1, C[11], 202);
180:     else
181:         printf("Sucess!\n");
182:
183:     // print test result
184:     printf("C = [\n");
185:     for (int i = 0; i < print_num; i++) {
186:         printf("    ");
187:         for (int j = 0; j < print_num; j++) {
188:             printf("%3d, ", C[DSIZE*j + i]);
189:         }
190:         printf("\b\b ]\n");
191:     }
192:     printf("    ]\n");
193: } else{
194:     // set DSIZE from CLI arg.
195:     DSIZE = dsize_set ? DSIZE: 512;
196:     printf("Matrix Size - %d\n", DSIZE);
197:
198:     // Start clock
199:     auto start = std::chrono::steady_clock::now();
200:     const int radius = 3;
201:     C = new int[DSIZE * DSIZE];
202:     stencilMatmul(host, platform, device, true, radius, DSIZE, C);
203:
204:     // Stop clock
205:     auto finish = std::chrono::steady_clock::now();
206:     double elapsed_seconds = std::chrono::duration_cast<std::chrono::duration<double>>(finish - start).
count();
207:     printf("time to run = %.2f S\n\n", elapsed_seconds);
208: }
209: // Free host memory
210: delete[] C;
211:
212: return 0;
213: }
```