Github link - https://github.com/rupeshknn/tac-hep-gpus/tree/main/Project


```
/*

compile: "c++ -o c_code c_code.cpp"
run check: "./c_code -check"
run code with size 512 (default): "./c_code"
run code with size 2048: "./c_code -size 2048"

*/

#include <stdio.h>
#include <stdlib.h>
#include <cstring>
#include <chrono>

int* stencil_matmul(bool isrnad, int radius, const int DSIZE)
{
    int *h_A, *h_B, *h_Ac, *h_Bc, *h_C;
    int print_num = 3;
    // Create and allocate memory for host and device pointers
    h_A = new int[DSIZE * DSIZE];
    h_B = new int[DSIZE * DSIZE];
    h_Ac = new int[DSIZE * DSIZE];
    h_Bc = new int[DSIZE * DSIZE];
    h_C = new int[DSIZE * DSIZE];

    // Fill in the matrices
    for (int i = 0; i < DSIZE; i++) {
        for (int j = 0; j < DSIZE; j++) {
            if (isrnad){
                h_A[i*DSIZE + j] = rand() % 10;
                h_B[i*DSIZE + j] = rand() % 10;
            } else{
                h_A[i*DSIZE + j] = 1;
                h_B[i*DSIZE + j] = 1;
            }
            h_Ac[i*DSIZE + j] = h_A[i*DSIZE + j];
            h_Bc[i*DSIZE + j] = h_B[i*DSIZE + j];
            h_C[i*DSIZE + j] = 0;
        }
    }
    int tempA = 0, tempB = 0;
    for (int idx = radius; idx < DSIZE-radius; idx++ ){
        for (int idy = radius; idy < DSIZE-radius; idy++ ){
            tempA = -h_A[idx*DSIZE + idy];
            tempB = -h_B[idx*DSIZE + idy];
            for (int idr = -radius; idr < radius+1; idr++ ){
                tempA += h_A[(idx+idr)*DSIZE + idy] + h_A[idx*DSIZE + idy+idr];
                tempB += h_B[(idx+idr)*DSIZE + idy] + h_B[idx*DSIZE + idy+idr];
            }
            h_Ac[idx*DSIZE + idy] = tempA;
            h_Bc[idx*DSIZE + idy] = tempB;
        }
    }

    for (int i=0; i<DSIZE; i++){
        for (int j=0; j<DSIZE; j++){
            h_C[i*DSIZE+j] = 0;
            for (int k=0; k<DSIZE; k++){
                h_C[i*DSIZE+j] += h_Ac[i*DSIZE+k]*h_Bc[k*DSIZE+j];
            }
        }
    }

    return h_C;
}

int main(int argc, char const *argv[]){
    bool check = false, dsize_set = false;
    uint DSIZE;

    if ( argc > 1){
        if (strcmp( argv[1], "-check") == 0){
            check = true;
        }
        if (strcmp( argv[1], "-size") == 0){
            DSIZE = std::atoi(argv[2]);
            dsize_set = true;
        }
    }
    int print_num = 10;
    int * C;
    if (check){
        DSIZE = 10;
```

```
        C = stencil_matmul(false, 1, DSIZE);
        if (C[0] != 10)
            printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,0, C[0], 10);
        else if (C[1] != 42)
            printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,1, C[1], 42);
        else if (C[11] != 202)
            printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 2,1, C[11], 202);
        else
            printf("Sucess!\n");

        printf("C = [\n");
        for (int i = 0; i < print_num; i++) {
            printf("     [");
            for (int j = 0; j < print_num; j++) {
                printf("%3d, ", C[DSIZE*j + i]);
            }
        printf("\b\b  ]\n");
        }
        printf("    ]\n");

    } else{
        DSIZE = dsize_set ? DSIZE: 512;
        printf("the dsize is %d\n", DSIZE);
        const int radius = 3;
        auto start = std::chrono::steady_clock::now();

        C = stencil_matmul(true, radius, DSIZE);

        auto finish = std::chrono::steady_clock::now();
        double elapsed_seconds = std::chrono::duration_cast<std::chrono::duration<double>>(finish - start).count();
        printf("time to run = %.2f S\n\n", elapsed_seconds);
    }

}
```

```
/*
# Basic cuda implementation

compile: "nvcc -o cuda_code cuda_code.cu"
run check: "./cuda_code -check"
run code with size 512 (default): "./cuda_code"
run code with size 4096: "./cuda_code -size 4096"

nsys comment: Most time speant on memory allocation and movement (about 90%)
*/

#include <cuda.h>
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <chrono>

#define CUDA_CHECK(call) \
    do { \
        cudaError_t err = call; \
        if (err != cudaSuccess) { \
            std::cerr << "CUDA error at " << __FILE__ << ":" << __LINE__ << ": " \
                      << cudaGetErrorString(err) << std::endl; \
            exit(EXIT_FAILURE); \
        } \
    } while (0)

// Stencil kernel
__global__ void stencilKernel(const int* d_A, int* d_Ac, int DSIZE, int radius) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;

    if (idx >= radius && idx < DSIZE - radius && idy >= radius && idy < DSIZE - radius) {
        int temp = -d_A[idx * DSIZE + idy];
        for (int r = -radius; r < radius+1; r++) {
            temp += d_A[(idx + r) * DSIZE + idy] + d_A[idx * DSIZE + idy + r];
        }
        d_Ac[idx * DSIZE + idy] = temp;
    }
}

// Matrix multiplication kernel
__global__ void matmulKernel(const int* d_Ac, const int* d_Bc, int* d_C, int DSIZE) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < DSIZE && col < DSIZE) {
        int sum = 0;
        for (int k = 0; k < DSIZE; ++k) {
            sum += d_Ac[row * DSIZE + k] * d_Bc[k * DSIZE + col];
        }
        d_C[row * DSIZE + col] = sum;
    }
}

// Host function
int* stencilMatmul(bool isRand, int radius, const int DSIZE) {
    // Allocate host memory
    int *h_A, *h_B, *h_Ac, *h_Bc, *h_C;
    h_A = new int[DSIZE * DSIZE];
    h_B = new int[DSIZE * DSIZE];
    h_Ac = new int[DSIZE * DSIZE];
    h_Bc = new int[DSIZE * DSIZE];
    h_C = new int[DSIZE * DSIZE];

    // Initialize matrices
    for (int i = 0; i < DSIZE; ++i) {
        for (int j = 0; j < DSIZE; ++j) {
            h_A[i * DSIZE + j] = isRand ? rand() % 10 : 1;
            h_B[i * DSIZE + j] = isRand ? rand() % 10 : 1;
            h_Ac[i * DSIZE + j] = h_A[i * DSIZE + j];
            h_Bc[i * DSIZE + j] = h_B[i * DSIZE + j];
            h_C[i * DSIZE + j] = 0;
        }
    }

    // Allocate device memory
    int *d_A, *d_B, *d_Ac, *d_Bc, *d_C;
    CUDA_CHECK(cudaMalloc((void**)&d_A, DSIZE * DSIZE * sizeof(int)));
    CUDA_CHECK(cudaMalloc((void**)&d_B, DSIZE * DSIZE * sizeof(int)));
    CUDA_CHECK(cudaMalloc((void**)&d_Ac, DSIZE * DSIZE * sizeof(int)));
    CUDA_CHECK(cudaMalloc((void**)&d_Bc, DSIZE * DSIZE * sizeof(int)));
    CUDA_CHECK(cudaMalloc((void**)&d_C, DSIZE * DSIZE * sizeof(int)));
```

```

    // Copy data to device
    CUDA_CHECK(cudaMemcpy(d_A, h_A, DSIZE * DSIZE * sizeof(int), cudaMemcpyHostToDevice));
    CUDA_CHECK(cudaMemcpy(d_B, h_B, DSIZE * DSIZE * sizeof(int), cudaMemcpyHostToDevice));
    CUDA_CHECK(cudaMemcpy(d_Ac, h_Ac, DSIZE * DSIZE * sizeof(int), cudaMemcpyHostToDevice));
    CUDA_CHECK(cudaMemcpy(d_Bc, h_Bc, DSIZE * DSIZE * sizeof(int), cudaMemcpyHostToDevice));
    CUDA_CHECK(cudaMemcpy(d_C, h_C, DSIZE * DSIZE * sizeof(int), cudaMemcpyHostToDevice));

    // Kernel configurations
    dim3 blockDim(16, 16);
    dim3 gridDim((DSIZE + blockDim.x - 1) / blockDim.x, (DSIZE + blockDim.y - 1) / blockDim.y);

    // Launch stencil kernels
    stencilKernel<<<gridDim, blockDim>>>(d_A, d_Ac, DSIZE, radius);
    stencilKernel<<<gridDim, blockDim>>>(d_B, d_Bc, DSIZE, radius);
    CUDA_CHECK(cudaDeviceSynchronize());

    // Launch matrix multiplication kernel
    matmulKernel<<<gridDim, blockDim>>>(d_Ac, d_Bc, d_C, DSIZE);
    CUDA_CHECK(cudaDeviceSynchronize());

    // Copy result back to host
    // CUDA_CHECK(cudaMemcpy(h_C, d_Ac, DSIZE * DSIZE * sizeof(int), cudaMemcpyDeviceToHost));
    CUDA_CHECK(cudaMemcpy(h_C, d_C, DSIZE * DSIZE * sizeof(int), cudaMemcpyDeviceToHost));

    // Free device memory
    CUDA_CHECK(cudaFree(d_A));
    CUDA_CHECK(cudaFree(d_B));
    CUDA_CHECK(cudaFree(d_Ac));
    CUDA_CHECK(cudaFree(d_Bc));
    CUDA_CHECK(cudaFree(d_C));

    // Free unused host memory
    delete[] h_A;
    delete[] h_B;
    delete[] h_Ac;
    delete[] h_Bc;

    return h_C;
}

int main(int argc, char const *argv[]) {
    bool check = false, dsize_set = false;
    uint DSIZE;

    if ( argc > 1){
        if (strcmp( argv[1], "-check") == 0){
            check = true;
        }
        if (strcmp( argv[1], "-size") == 0){
            DSIZE = std::atoi(argv[2]);
            dsize_set = true;
        }
    }
    int print_num = 10;
    int * C;
    if (check){
        DSIZE = 10;
        C = stencilMatmul(false, 1, DSIZE);
        if (C[0] != 10)
            printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,0, C[0], 10);
        else if (C[1] != 42)
            printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,1, C[1], 42);
        else if (C[11] != 202)
            printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 2,1, C[11], 202);
        else
            printf("Sucess!\n");

        printf("C = [\n");
        for (int i = 0; i < print_num; i++) {
            printf("     [");
            for (int j = 0; j < print_num; j++) {
                printf("%3d, ", C[DSIZE*j + i]);
            }
            printf("\b\b  ]\n");
        }
        printf("    ]\n");
    } else{
        DSIZE = dsize_set ? DSIZE: 512;
        printf("the dsize is %d\n", DSIZE);
        const int radius = 3;

        auto start = std::chrono::steady_clock::now();
```

```

        C = stencilMatmul(true, radius, DSIZE);

        auto finish = std::chrono::steady_clock::now();
        double elapsed_seconds = std::chrono::duration_cast<std::chrono::duration<double>>(finish - start).count();
        printf("time to run = %.2f\n\n", elapsed_seconds);
    }

    // Free unified memory for result
    delete[] C;

    return 0;
}
```

```
/*
# Managed memory
compile: "nvcc -o cuda_mm cuda_mm.cu"
run check: "./cuda_mm -check"
run code with size 512 (default): "./cuda_mm"
run code with size 4096: "./cuda_mm -size 4096"

nsys comment: Most time speant on memory allocation and movement (about 87%) slight improvement to cuda_code.
*/

#include <cuda.h>
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <chrono>

#define CUDA_CHECK(call) \
    do { \
        cudaError_t err = call; \
        if (err != cudaSuccess) { \
            std::cerr << "CUDA error at " << __FILE__ << ":" << __LINE__ << ": " \
                      << cudaGetErrorString(err) << std::endl; \
            exit(EXIT_FAILURE); \
        } \
    } while (0)

// Stencil kernel
__global__ void stencilKernel(const int* A, int* Ac, int DSIZE, int radius) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;

    if (idx >= radius && idx < DSIZE - radius && idy >= radius && idy < DSIZE - radius) {
        int temp = -A[idx * DSIZE + idy];
        for (int r = -radius; r < radius+1; r++) {
            temp += A[(idx + r) * DSIZE + idy] + A[idx * DSIZE + idy + r];
        }
        Ac[idx * DSIZE + idy] = temp;
    }
}

// Matrix multiplication kernel
__global__ void matmulKernel(const int* Ac, const int* Bc, int* C, int DSIZE) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < DSIZE && col < DSIZE) {
        int sum = 0;
        for (int k = 0; k < DSIZE; ++k) {
            sum += Ac[row * DSIZE + k] * Bc[k * DSIZE + col];
        }
        C[row * DSIZE + col] = sum;
    }
}

// Host function
int* stencilMatmul(bool isRand, int radius, const int DSIZE) {
    // Unified memory allocation
    int *A, *B, *Ac, *Bc, *C;
    CUDA_CHECK(cudaMallocManaged(&A, DSIZE * DSIZE * sizeof(int)));
    CUDA_CHECK(cudaMallocManaged(&B, DSIZE * DSIZE * sizeof(int)));
    CUDA_CHECK(cudaMallocManaged(&Ac, DSIZE * DSIZE * sizeof(int)));
    CUDA_CHECK(cudaMallocManaged(&Bc, DSIZE * DSIZE * sizeof(int)));
    CUDA_CHECK(cudaMallocManaged(&C, DSIZE * DSIZE * sizeof(int)));

    // Initialize matrices
    for (int i = 0; i < DSIZE; ++i) {
        for (int j = 0; j < DSIZE; ++j) {
            A[i * DSIZE + j] = isRand ? rand() % 10 : 1;
            B[i * DSIZE + j] = isRand ? rand() % 10 : 1;
            Ac[i * DSIZE + j] = A[i * DSIZE + j];
            Bc[i * DSIZE + j] = B[i * DSIZE + j];
            C[i * DSIZE + j] = 0;
        }
    }

    // Kernel configurations
    dim3 blockDim(16, 16);
    dim3 gridDim((DSIZE + blockDim.x - 1) / blockDim.x, (DSIZE + blockDim.y - 1) / blockDim.y);

    // Launch stencil kernels
    stencilKernel<<<gridDim, blockDim>>>(A, Ac, DSIZE, radius);
    stencilKernel<<<gridDim, blockDim>>>(B, Bc, DSIZE, radius);
```

```
    CUDA_CHECK(cudaDeviceSynchronize());

    // Launch matrix multiplication kernel
    matmulKernel<<<gridDim, blockDim>>>(Ac, Bc, C, DSIZE);
    CUDA_CHECK(cudaDeviceSynchronize());

    // Free unified memory
    CUDA_CHECK(cudaFree(A));
    CUDA_CHECK(cudaFree(B));
    CUDA_CHECK(cudaFree(Ac));
    CUDA_CHECK(cudaFree(Bc));

    return C; // Return result (managed memory pointer)
}

int main(int argc, char const *argv[]) {
    bool check = false, dsize_set = false;
    uint DSIZE;

    if ( argc > 1){
        if (strcmp( argv[1], "-check") == 0){
            check = true;
        }
        if (strcmp( argv[1], "-size") == 0){
            DSIZE = std::atoi(argv[2]);
            dsize_set = true;
        }
    }
    int print_num = 10;
    int * C;
    if (check){
        DSIZE = 10;
        C = stencilMatmul(false, 1, DSIZE);
        if (C[0] != 10)
            printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,0, C[0], 10);
        else if (C[1] != 42)
            printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,1, C[1], 42);
        else if (C[11] != 202)
            printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 2,1, C[11], 202);
        else
            printf("Sucess!\n");

        printf("C = [\n");
        for (int i = 0; i < print_num; i++) {
            printf("     [");
            for (int j = 0; j < print_num; j++) {
                printf("%3d, ", C[DSIZE*j + i]);
            }
            printf("\b\b  ]\n");
        }
        printf("    ]\n");
    } else{
        DSIZE = dsize_set ? DSIZE: 512;
        printf("the dsize is %d\n", DSIZE);
        const int radius = 3;

        auto start = std::chrono::steady_clock::now();

        C = stencilMatmul(true, radius, DSIZE);

        auto finish = std::chrono::steady_clock::now();
        double elapsed_seconds = std::chrono::duration_cast<std::chrono::duration<double>>(finish - start).count();
        printf("time to run = %.2f\n\n", elapsed_seconds);
    }

    // Free unified memory for result
    CUDA_CHECK(cudaFree(C));

    return 0;
}
```

```
/*
# Shared memory (both stencil and matrix-mul) and non-default cuda streams.

compile: "nvcc -o cuda_final cuda_final.cu"
run check: "./cuda_final -check"
run code with size 512 (default): "./cuda_final"
run code with size 4096: "./cuda_final -size 4096"

nsys comment: significantly faster and less time spent on mem alloc.
*/

#include <cuda.h>
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <chrono>

#define CUDA_CHECK(call) \
    do { \
        cudaError_t err = call; \
        if (err != cudaSuccess) { \
            std::cerr << "CUDA error at " << __FILE__ << ":" << __LINE__ << ": " \
                      << cudaGetErrorString(err) << std::endl; \
            exit(EXIT_FAILURE); \
        } \
    } while (0)

// Stencil kernel with shared memory
__global__ void stencilKernelShared(const int* A, int* Ac, int DSIZE, int radius) {
    extern __shared__ int shared[];
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int idx = blockIdx.x * blockDim.x + tx;
    int idy = blockIdx.y * blockDim.y + ty;

    int localIdx = tx + radius;
    int localIdy = ty + radius;

    // Copy to shared memory (with halo)
    if (idx < DSIZE && idy < DSIZE) {
        shared[localIdy * (blockDim.x + 2 * radius) + localIdx] = A[idy * DSIZE + idx];
    }

    // Load halo regions
    if (tx < radius) {
        if (idx >= radius) {
            shared[localIdy * (blockDim.x + 2 * radius) + tx] = A[idy * DSIZE + (idx - radius)];
        }
        if (idx + blockDim.x < DSIZE) {
            shared[localIdy * (blockDim.x + 2 * radius) + (localIdx + blockDim.x)] =
                A[idy * DSIZE + (idx + blockDim.x)];
        }
    }
    if (ty < radius) {
        if (idy >= radius) {
            shared[ty * (blockDim.x + 2 * radius) + localIdx] = A[(idy - radius) * DSIZE + idx];
        }
        if (idy + blockDim.y < DSIZE) {
            shared[(localIdy + blockDim.y) * (blockDim.x + 2 * radius) + localIdx] =
                A[(idy + blockDim.y) * DSIZE + idx];
        }
    }

    __syncthreads();

    // Apply stencil
    if (idx >= radius && idx < DSIZE - radius && idy >= radius && idy < DSIZE - radius) {
        int temp = -shared[localIdy * (blockDim.x + 2 * radius) + localIdx];
        for (int r = -radius; r <= radius; ++r) {
            temp += shared[(localIdy + r) * (blockDim.x + 2 * radius) + localIdx];
            temp += shared[localIdy * (blockDim.x + 2 * radius) + (localIdx + r)];
        }
        Ac[idy * DSIZE + idx] = temp;
    }
}

// Matrix multiplication kernel
__global__ void matmulSharedKernel(const int* A, const int* B, int* C, int DSIZE) {
    extern __shared__ int shared[];
    int* tileA = shared;
    int* tileB = shared + blockDim.x * blockDim.y;

```

```
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int row = blockIdx.y * blockDim.y + ty;
    int col = blockIdx.x * blockDim.x + tx;

    int temp = 0;

    // Loop over tiles
    for (int t = 0; t < (DSIZE + blockDim.x - 1) / blockDim.x; ++t) {
        // Load tiles into shared memory
        if (row < DSIZE && t * blockDim.x + tx < DSIZE) {
            tileA[ty * blockDim.x + tx] = A[row * DSIZE + t * blockDim.x + tx];
        } else {
            tileA[ty * blockDim.x + tx] = 0;
        }
        if (t * blockDim.y + ty < DSIZE && col < DSIZE) {
            tileB[ty * blockDim.x + tx] = B[(t * blockDim.y + ty) * DSIZE + col];
        } else {
            tileB[ty * blockDim.x + tx] = 0;
        }

        __syncthreads();

        // Perform partial computation for the tile
        for (int k = 0; k < blockDim.x; ++k) {
            temp += tileA[ty * blockDim.x + k] * tileB[k * blockDim.x + tx];
        }

        __syncthreads();
    }

    // Write result to global memory
    if (row < DSIZE && col < DSIZE) {
        C[row * DSIZE + col] = temp;
    }
}

// Host function
int* stencilMatmul(bool isRand, int radius, const int DSIZE) {
    // Unified memory allocation
    int *A, *B, *Ac, *Bc, *C;
    CUDA_CHECK(cudaMallocManaged(&A, DSIZE * DSIZE * sizeof(int)));
    CUDA_CHECK(cudaMallocManaged(&B, DSIZE * DSIZE * sizeof(int)));
    CUDA_CHECK(cudaMallocManaged(&Ac, DSIZE * DSIZE * sizeof(int)));
    CUDA_CHECK(cudaMallocManaged(&Bc, DSIZE * DSIZE * sizeof(int)));
    CUDA_CHECK(cudaMallocManaged(&C, DSIZE * DSIZE * sizeof(int)));

    // Initialize matrices
    for (int i = 0; i < DSIZE; ++i) {
        for (int j = 0; j < DSIZE; ++j) {
            A[i * DSIZE + j] = isRand ? rand() % 10 : 1;
            B[i * DSIZE + j] = isRand ? rand() % 10 : 1;
            Ac[i * DSIZE + j] = A[i * DSIZE + j];
            Bc[i * DSIZE + j] = B[i * DSIZE + j];
            C[i * DSIZE + j] = 0;
        }
    }

    // Kernel configurations
    dim3 blockDim(16, 16);
    // dim3 gridDim((DSIZE + blockDim.x - 1) / blockDim.x, (DSIZE + blockDim.y - 1) / blockDim.y);
    dim3 gridDim(32, 32);
    int sharedMemSize = (blockDim.x + 2 * radius) * (blockDim.y + 2 * radius) * sizeof(int);
    int sharedMemMatmul = 2 * blockDim.x * blockDim.y * sizeof(int);
    // Create CUDA streams
    cudaStream_t stream1, stream2;
    CUDA_CHECK(cudaStreamCreate(&stream1));
    CUDA_CHECK(cudaStreamCreate(&stream2));
    printf("Grid : {%d, %d} blocks. Blocks : {%d, %d} threads.\n", gridDim.x, gridDim.y, blockDim.x, blockDim.y);
    // Launch stencil kernels on different streams
    stencilKernelShared<<<gridDim, blockDim, sharedMemSize, stream1>>>(A, Ac, DSIZE, radius);
    stencilKernelShared<<<gridDim, blockDim, sharedMemSize, stream2>>>(B, Bc, DSIZE, radius);

    // Synchronize stencil streams
    CUDA_CHECK(cudaStreamSynchronize(stream1));
    CUDA_CHECK(cudaStreamSynchronize(stream2));

    // Launch matrix multiplication kernel
    matmulSharedKernel<<<gridDim, blockDim, sharedMemMatmul>>>(Ac, Bc, C, DSIZE);
    CUDA_CHECK(cudaDeviceSynchronize());

    // Free CUDA streams
```

```
    CUDA_CHECK(cudaStreamDestroy(stream1));
    CUDA_CHECK(cudaStreamDestroy(stream2));

    // Free unified memory
    CUDA_CHECK(cudaFree(A));
    CUDA_CHECK(cudaFree(B));
    CUDA_CHECK(cudaFree(Ac));
    CUDA_CHECK(cudaFree(Bc));

    return C; // Return result (managed memory pointer)
}

int main(int argc, char const *argv[]) {
    bool check = false, dsize_set = false;
    uint DSIZE;

    if ( argc > 1){
        if (strcmp( argv[1], "-check") == 0){
            check = true;
        }
        if (strcmp( argv[1], "-size") == 0){
            DSIZE = std::atoi(argv[2]);
            dsize_set = true;
        }
    }
    int print_num = 10;
    int * C;
    if (check){
        DSIZE = 10;
        C = stencilMatmul(false, 1, DSIZE);
        if (C[0] != 10)
            printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,0, C[0], 10);
        else if (C[1] != 42)
            printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,1, C[1], 42);
        else if (C[11] != 202)
            printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 2,1, C[11], 202);
        else
            printf("Sucess!\n");

        printf("C = [\n");
        for (int i = 0; i < print_num; i++) {
            printf("     [");
            for (int j = 0; j < print_num; j++) {
                printf("%3d, ", C[DSIZE*j + i]);
            }
            printf("\b\b  ]\n");
        }
        printf("    ]\n");
    } else{
        DSIZE = dsize_set ? DSIZE: 512;
        printf("the dsize is %d\n", DSIZE);
        const int radius = 3;

        auto start = std::chrono::steady_clock::now();

        C = stencilMatmul(true, radius, DSIZE);

        auto finish = std::chrono::steady_clock::now();
        double elapsed_seconds = std::chrono::duration_cast<std::chrono::duration<double>>(finish - start).count();
        printf("time to run = %.2f\n\n", elapsed_seconds);
    }

    // Free unified memory for result
    CUDA_CHECK(cudaFree(C));

    return 0;
}
```

```cpp
/*
# Alpaka implementation.

cuda compile: "nvcc -x cu -std=c++17 -O2 -g --expt-relaxed-constexpr -I$ALPAKA_BASE/include -DALPAKA_ACC_GPU_CUDA_ENABLED alpaka.cpp -o alpaka_cuda"
run check: "./alpaka_cuda -check"
run code with size 512 (default): "./alpaka_cuda"
run code with size 4096: "./alpaka_cuda -size 4096"

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"
run check: "./alpaka_cpu -check"
run code with size 512 (default): "./alpaka_cpu"
run code with size 2048: "./alpaka_cpu -size 2048"
 */

#include <cassert>
#include <cstdio>
#include <random>

#include <alpaka/alpaka.hpp>

#include "config.h"
#include "WorkDiv.hpp"
#include <chrono>


struct stencil2D {
    template <typename TAcc, typename T>
    ALPAKA_FN_ACC void operator()(TAcc const& acc,
                                  T const* __restrict__ d_A,
                                  T* __restrict__ d_Aout,
                                  int radius,
                                  Vec2D size) const {
        for (auto ndindex : alpaka::uniformElementsND(acc, size)) {
            auto idx = ndindex[0];
            auto idy = ndindex[1];
            auto DSIZE = size[1];
            // auto index = (ndindex[0] * size[1] + ndindex[1])

            if (idx >= radius && idx < DSIZE - radius && idy >= radius && idy < DSIZE - radius) {
                int temp = -d_A[idx * DSIZE + idy];
                for (int r = -radius; r < radius+1; r++) {
                    temp += d_A[(idx + r) * DSIZE + idy] + d_A[idx * DSIZE + idy + r];
                }
                d_Aout[idx * DSIZE + idy] = temp;
            }
        }
    }
};

struct matrixmul {
    template <typename TAcc, typename T>
    ALPAKA_FN_ACC void operator()(TAcc const& acc,
                                  T const* __restrict__ d_A,
                                  T const* __restrict__ d_B,
                                  T* __restrict__ d_ABout,
                                  Vec2D size) const {
        for (auto ndindex : alpaka::uniformElementsND(acc, size)) {
            auto idx = ndindex[0];
            auto idy = ndindex[1];
            auto DSIZE = size[1];
            // auto index = (ndindex[0] * size[1] + ndindex[1])
            if (idy < DSIZE && idx < DSIZE) {
                    int sum = 0;
                    for (int k = 0; k < DSIZE; ++k) {
                        sum += d_A[idy * DSIZE + k] * d_B[k * DSIZE + idx];
                    }
                    d_ABout[idy * DSIZE + idx] = sum;
            }

        }
    }
};

// Host function
void stencilMatmul(Host host, Platform platform, Device device, bool isRand, int radius, const int DSIZE, int* out) {

    // 3-dimensional and linearised buffer size
    Vec2D ndsize = {DSIZE, DSIZE};
    uint32_t size = ndsize.prod();
    auto h_A = alpaka::allocMappedBuf<int, uint32_t>(host, platform, size);
```

```cpp
    auto h_B = alpaka::allocMappedBuf<int, uint32_t>(host, platform, size);
    auto h_As = alpaka::allocMappedBuf<int, uint32_t>(host, platform, size);
    auto h_Bs = alpaka::allocMappedBuf<int, uint32_t>(host, platform, size);
    auto h_C = alpaka::allocMappedBuf<int, uint32_t>(host, platform, size);

    for (uint32_t i = 0; i < size; ++i) {
        h_A[i] = isRand ? rand() % 10 : 1;
        h_B[i] = isRand ? rand() % 10 : 1;
        h_As[i] = h_A[i];
        h_Bs[i] = h_B[i];
    }

    // run the test the given device
    auto queue = Queue{device};

    // allocate input and output buffers on the device
    auto d_A = alpaka::allocAsyncBuf<int, uint32_t>(queue, size);
    auto d_B = alpaka::allocAsyncBuf<int, uint32_t>(queue, size);
    auto d_As = alpaka::allocAsyncBuf<int, uint32_t>(queue, size);
    auto d_Bs = alpaka::allocAsyncBuf<int, uint32_t>(queue, size);
    auto d_C = alpaka::allocAsyncBuf<int, uint32_t>(queue, size);

    // copy the input data to the device; the size is known from the buffer objects
    alpaka::memcpy(queue, d_A, h_A);
    alpaka::memcpy(queue, d_B, h_B);
    alpaka::memcpy(queue, d_As, h_As);
    alpaka::memcpy(queue, d_Bs, h_Bs);

    alpaka::memset(queue, d_C, 0x00);

    // launch the 3-dimensional kernel
    auto div = makeWorkDiv<Acc2D>({32, 32}, {16, 16});
    std::cout << "Testing VectorAddKernel3D with vector indices with a grid of "
              << alpaka::getWorkDiv<alpaka::Grid, alpaka::Blocks>(div) << " blocks x "
              << alpaka::getWorkDiv<alpaka::Block, alpaka::Threads>(div) << " threads x "
              << alpaka::getWorkDiv<alpaka::Thread, alpaka::Elems>(div) << " elements...\n";
    alpaka::exec<Acc2D>(
        queue, div, stencil2D{}, d_A.data(), d_As.data(), radius, ndsize);
    alpaka::exec<Acc2D>(
        queue, div, stencil2D{}, d_B.data(), d_Bs.data(), radius, ndsize);
    alpaka::wait(queue);
    alpaka::exec<Acc2D>(
        queue, div, matrixmul{}, d_As.data(), d_Bs.data(), d_C.data(), ndsize);

    // copy the results from the device to the host
    alpaka::memcpy(queue, h_C, d_C);

    // wait for all the operations to complete
    alpaka::wait(queue);
    // alpaka::memcpy(queue, out, h_C);
    for (uint32_t i = 0; i < size; ++i) {
        out[i] = h_C[i];
    }
}

int main(int argc, char const *argv[]) {
    // initialise the accelerator platform
    Platform platform;
    // require at least one device
    std::uint32_t n = alpaka::getDevCount(platform);
    if (n == 0) {
        exit(EXIT_FAILURE);
    }

    // use the single host device
    HostPlatform host_platform;
    Host host = alpaka::getDevByIdx(host_platform, 0u);
    std::cout << "Host:    " << alpaka::getName(host) << '\n';

    // use the first device
    Device device = alpaka::getDevByIdx(platform, 0u);
    std::cout << "Device: " << alpaka::getName(device) << '\n';


    bool check = false, dsize_set = false;
    uint DSIZE;

    if ( argc > 1){
        if (strcmp( argv[1], "-check") == 0){
            check = true;
        }
        if (strcmp( argv[1], "-size") == 0){
            DSIZE = std::atoi(argv[2]);
```

```
            dsize_set = true;
        }
    }
    int print_num = 10;
    int * C;
    if (check){
        DSIZE = 10;
        printf("Matrix Size - %d\n", DSIZE);
        C = new int[DSIZE * DSIZE];
        stencilMatmul(host, platform, device, false, 1, DSIZE, C);
        if (C[0] != 10)
            printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,0, C[0], 10);
        else if (C[1] != 42)
            printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 0,1, C[1], 42);
        else if (C[11] != 202)
            printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", 2,1, C[11], 202);
        else
            printf("Sucess!\n");

        // print test result
        printf("C = [\n");
        for (int i = 0; i < print_num; i++) {
            printf("     [");
            for (int j = 0; j < print_num; j++) {
                printf("%3d, ", C[DSIZE*j + i]);
            }
        printf("\b\b  ]\n");
        }
        printf("    ]\n");
    } else{
        // set DSIZE from CLI arg.
        DSIZE = dsize_set ? DSIZE: 512;
        printf("Matrix Size - %d\n", DSIZE);

        // Start clock
        auto start = std::chrono::steady_clock::now();
        const int radius = 3;
        C = new int[DSIZE * DSIZE];
        stencilMatmul(host, platform, device, true, radius, DSIZE, C);

        // Stop clock
        auto finish = std::chrono::steady_clock::now();
        double elapsed_seconds = std::chrono::duration_cast<std::chrono::duration<double>>(finish - start).count();
        printf("time to run = %.2f S\n\n", elapsed_seconds);
    }
    // Free host memory
    delete[] C;

    return 0;
}
```