Uebung 2

Patrik Plewka (3761940) Matthias Riegler (3761312)

1a

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>

#define N (2048 * 2048)       // N
#define THREADS_PER_BLOCK 128 // B

// random_floats fills an array of floats; not that random :-)
void random_floats(float *a, int array_size)
{
    for (int i = 0; i < array_size; i++)
    {
        a[i] = (float)i;
    }
}

// add is the default vector add
__global__ void add(float *a, float *b, float *c)
{

    int idx = blockIdx.x + threadIdx.x * gridDim.x;

    // Not needed, just for safety!
    c[idx] = a[idx] + b[idx];
}

// main is the main entrypoint.
int main(void)
{
    float *a, *b, *c;       // host copies of a, b, c
    float *d_a, *d_b, *d_c; // device copies of a, b, c
    float size = N * sizeof(float);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for cpu
    a = (float *)malloc(size);
    random_floats(a, N);
    b = (float *)malloc(size);
```

```
    random_floats(b, N);
    c = (float *)malloc(size);

    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
    // Launch add() kernel on GPU
    add<<<N / THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);
    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
    printf("%f\n", c[N - 1]);

    // Cleanup
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    free(a);
    free(b);
    free(c);
    return 0;
}
```

1b

- Wenn x<M/2: p(y,x) = 2*(x*N+y)
- Wenn x>=M/2: p(y,x) = 2*((x-M/2)*N+y)+1

Einfach x und y (und M und N) vertauschen -> zeilenweise wird dann zu spaltenweise

-> geht nur bei gerader Anzahl Spalten

2

```
#include <iostream>

__device__ int proj(unsigned int block_Idx, unsigned int thread_Idx) {
    return (int)(thread_Idx + block_Idx * blockDim.x);
}

__global__ void vecads(float *a, float *b, float *c, int n)
{
    int idx;
    idx = proj(blockIdx.x,threadIdx.x);
    if(idx<n){
        c[idx] = a[idx] + (((idx & 1) * 2) - 1) * b[idx];
    }
}

// a[i] + b[i] - (a[i] - b[i]) = 2 * b[i]

// a[i] +/- b[i]
// +/- kann man über mehrere Arten erreichen:
// a[i] + -1^(not Bedingung) * b[i]
// a[i] + ((Bedingung * 2) - 1)) * b[i]
// a[i] + (Bedingung - not Bedingung) * b[i]
```

```
/*
 * Es führt zu keiner Einschränkgung des Wertebereichs, da im Vergleich zur alten Berechnung
 * folgendes gilt:
 * Der letzte Rechenschritt ist ähnlich wie vor der Änderung:
 *   statt a[i] + b[i] oder a[i] - b[i] wird a[i] + b[i] oder a[i] + (-b[i]) berechnet.
 * Diese letzte Rechenschritt schränkt den Wertebereich so ein, dass a[i] + b[i] (bzw. a[i] - b[i])
 * zu keinem over oder underflow führen dürfen.
 * Solange keiner der Schritte davor den Wertebereich mehr einschränken kann,
 * führt die Veränderung zu keiner weiteren Einscränkung des Wertebereichs.
 *
 * a[i] + -1^(not Bedingung) * b[i]:
 * Bedingung -> entweder 0 oder 1
 * not Bedingung -> entweder 0 oder 1
 * -1^(not Bedingung) -> entweder 1 oder -1
 * -1^(not Bedingung) * b[i] -> entweder b[i] oder -b[i]
 * -> keine Einschränkung des Wertebereichs
 *
 * a[i] + ((Bedingung * 2) - 1)) * b[i]:
 * Bedingung -> entweder 0 oder 1
 * (Bedingung * 2) -> entweder 0 oder 2
 * ((Bedingung * 2) - 1) -> entweder -1 oder 1
 * ((Bedingung * 2) - 1) * b[i] -> entweder -b[i] oder b[i]
 * -> keine Einschränkung des Wertebereichs
 *
 * a[i] + (Bedingung - not Bedingung) * b[i]:
 * Bedingung -> entweder 0 oder 1
 * not Bedingung -> entweder 0 oder 1
 * (Bedingung - not Bedingung) -> entweder -1 oder 1
 * (Bedingung - not Bedingung) * b[i] -> entweder -b[i] oder b[i]
 * -> keine Einschränkung des Wertebereichs
 */
#define N 128

int main(){
    float a[N];
    float b[N];
    float c[N];
    for(int i = 0; i < N; ++i){
        a[i] = i;
        b[i] = 2 * i;
        c[i] = -4793;
    }
    float *device_a, *device_b, *device_c;
    cudaMalloc((void**)&device_a, N * sizeof(float));
    cudaMalloc((void**)&device_b, N * sizeof(float));
    cudaMalloc((void**)&device_c, N * sizeof(float));

    cudaMemcpy(device_a, a, N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(device_b, b, N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(device_c, c, N * sizeof(float), cudaMemcpyHostToDevice);
    vecads<<<4, 32>>>(device_a, device_b, device_c, N);
    cudaMemcpy(c, device_c, N * sizeof(float), cudaMemcpyDeviceToHost);
    for(int i = 0; i < N; ++i){
        std::cout << c[i] << std::endl;
    }
}
```

3

```
#define ROWS 16 // N
#define COLS 10 // M

__global__ void compact(int *a, int *list_x, int *list_y)
{
    __shared__ int num[COLS]; // Shared array

    int idx = threadIdx.x; // Index

    // Count for every row how many entries with 16 exist
    // store in num array (shared state)
    if (idx < COLS)  // executed by COLS threads
    {
        int counter = 0;
        // Each thread iterates through all rows at given position
        for (int i = 0; i < ROWS; i++)
        {
            // Check if idx == 16 -> if so, increase counter
            counter += a[i] == 16;
        }
        // Store final count in shared array
        num[idx] = counter;
    }

    // one of the threads(!) executes after barrier prefix Sum
    __syncthreads();
    if (threadIdx.x == 0)
    {
        prefix_sum(num, COLS);
    }

    // another barrier and store in global list_x and list_y
    __syncthreads();
    if (idx < COLS)  // executed by COLS threads
    {
        int calcIdx = num[idx];
        for (int i = 0; i < ROWS; i++)
        {
            if (a[i * COLS + idx] == 16)
            {
                list_x[calcIdx] = i;
                list_y[calcIdx] = idx;
                calcIdx++;
            }
        }
    }
}
```