key points

help decrease computing $O(N^2)$

double-buffering: use two buffers of the shared memory, thus separate reading and writing section so that it can decrease one time of __syncthread().

1

1.1

for the array: [4 6 7 1 2 8 5 2] the step in progress is [10 13 8 3 10 13 7],[4 10 14 18 16 18 16 17]

so the final result should be:[4 10 14 18 20 28 29 30]

1.2

```
__global__ void Kogge_Stone_scan_kernel(float *X, float *Y, unsigned int N)
    {
    __shared__ float XY[SECTION_SIZE];
    __shared__ float XY_1[SECTION_SIZE];
    if(i < N) {
        XY[threadIdx.x] = X[i];
        XY_1[threadIdx.x] = X[i];
    } else {
        XY[threadIdx.x] = 0.0f;
        XY_1[threadIdx.x] = 0.0f;
    }
    bool choose = true;
    for(unsigned int stride = 1; stride < blockDim.x; stride *= 2){
        __syncthreads();
        if(choose){
            if(threadIdx.x >= stride){
                XY_1[threadIdx.x] = XY[threadIdx.x] + XY[threadIdx.x -
                    stride];
            }
            choose = false;
        } else {
            if(threadIdx.x >= stride){
                XY[threadIdx.x] = XY_1[threadIdx.x] + XY_1[threadIdx.x-
                    stride];
            }
            choose = true;
        }
        if(i < N){
            if(choose){
                Y[i] = XY[threadIdx.x];
            } else {
                Y[i] = XY_1[threadIdx.x];
            }
        }
    }
}
```

1.3

for different stride values n, it will calculate from (2k-1)n to (2k)n.

1.4

it is about $2048 * log_2 2048$

1.5

~~for each step, the array is: [10 8 10 7], [18 17],[25],and [20], [17 10 15], so the final result should be [4,10,18,18,20,28,33,35]~~ [4 10 7 8 2 10 5 7],[4 10 18 8 2 18 5 15],[4 10 18 8 20 18 28 15],[4 10 18 20 28 28 33],[4 10 18 18 20 28 33 35],[4 10 18 18 20 28 33 35]

for stride from 1,2,4(Upsweep phase) to 4,2,1(DownSweep phase).

1.6

$\sum_{i=1}^{log_2 n} \frac{n}{2^i}$

1.7

```
__global__ void Kogge_Stone_scan_kernel(float *X, float *Y, unsigned int N)
    {
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    __shared__ float XY[blockDim.x+1];
    if(i > 0 && i < N){
        XY[threadIdx.x] = X[i-1];// Shift by 1 for exclusive behavior
    } else {
        XY[threadIdx.x] = 0.0f;
    }
    for(unsigned int stride = 1; stride < blockDim.x; stride *= 2){
        __syncthreads();
        float temp;
        if(threadIdx.x >= stride){
            temp = XY[threadIdx.x] + XY[threadIdx.x - stride];
        }
        __syncthreads();
        if(threadIdx.x >= stride){
            XY[threadIdx.x] = temp;
        }
    }
    if(i < N){
        Y[i] = XY[threadIdx.x];
    }
}
```

1.8

```
#include <cuda_runtime.h>
#include <iostream>

__global__ void reduceKernel(float* input, float* output, int* flags, int n
    ) {
    extern __shared__ float sdata[];
    int tid = threadIdx.x;
    int i = blockIdx.x * blockDim.x + tid;

    if (i < n) {
        sdata[tid] = input[i];
    } else {
        sdata[tid] = 0;
    }
    __syncthreads();

    // Up-sweep phase
    for (int stride = 1; stride < blockDim.x; stride *= 2) {
        int index = (tid + 1) * stride * 2 - 1;
        if (index < blockDim.x && i < n && flags[index] == flags[index -
                stride]) {
```

```
            sdata[index] += sdata[index - stride];
        }
        __syncthreads();
    }

    // Write the block-level result to output
    if (tid == blockDim.x - 1 && i < n) {
        output[blockIdx.x] = sdata[tid];
    }
}

__global__ void propagateKernel(float* output, int* flags, int n) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if (tid >= n) return;

    // Segmented scan at block level
    for (int stride = 1; stride < n; stride *= 2) {
        __syncthreads();
        if (tid >= stride && flags[tid] == flags[tid - stride]) {
            output[tid] += output[tid - stride];
        }
    }
}

__global__ void downsweepKernel(float* input, float* output, int* flags,
    int n) {
   int tid = threadIdx.x + blockIdx.x * blockDim.x;

   if (tid >= n) return;

   float val = output[tid];
   for (int i = 0; i < n; i++) {
       if (flags[i] == flags[tid]) {
           output[tid] += input[i];
       }
   }
}

void segmentedParallelScan(float* h_input, float* h_output, int* h_flags,
      int n) {
    // Allocate device memory
    float *d_input, *d_output;
    int *d_flags;
    cudaMalloc(&d_input, n * sizeof(float));
    cudaMalloc(&d_output, n * sizeof(float));
    cudaMalloc(&d_flags, n * sizeof(int));

    // Copy input data to device
    cudaMemcpy(d_input, h_input, n * sizeof(float),
       cudaMemcpyHostToDevice);
    cudaMemcpy(d_flags, h_flags, n * sizeof(int), cudaMemcpyHostToDevice);

    // Launch reduce kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;
    reduceKernel<<<blocksPerGrid, threadsPerBlock>>>(d_input, d_output,
           d_flags, n);

    // Launch propagate kernel
    propagateKernel<<<blocksPerGrid, threadsPerBlock>>>(d_output,   d_flags
       , n);
    // Launch downsweep kernel
    downsweepKernel<<<blocksPerGrid, threadsPerBlock>>>(d_input,
        d_output, d_flags, n);

    // Copy output data back to host
    cudaMemcpy(h_output, d_output, n * sizeof(float),
        cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_input);
    cudaFree(d_output);
    cudaFree(d_flags);
}
```