به نام خدا

دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

آزمایش شماره 6

زهرا لطیفی 9923069

آیدا احمدی پارسا 9923003

ابتدا الگوریتم اول را روی GPU پیاده‌سازی می‌کنیم.

الگوریتم پیاده شده دقیقا همانند الگوریتم موازی شده روی cpu است با این تفاوت که این بار متغیرها را روی GPU محاسبه می‌کنیم. اکنون به توضیح برخی از بخش‌های کلیدی کد می‌پردازیم.

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cuda.h>
#include <device_functions.h>
#include <cuda_device_runtime_api.h>
#include <device_launch_parameters.h>
#include <time.h>
```

```
#define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, const char *file, int line, bool abort =
true)
{
if (code != cudaSuccess)
{
fprintf(stderr, "GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
if (abort) exit(code);
}
}
```

1

```
void fill_array(int *a, size_t n);
void print_array(int *a, size_t n);
void parallel_prefix_sum(int *a, size_t n);

#define EXP_NUM 1
__global__ void prefix_sum_kernel(int *d_a, int *d_last_sums, int n, int
num_threads) {
int tid = threadIdx.x + blockIdx.x * blockDim.x;
if (tid >= num_threads) return;

int chunk_size = n / num_threads;
int start = tid * chunk_size;
int end = (tid == num_threads - 1) ? n : start + chunk_size;

// Local prefix sum over subarray
for (int i = start + 1; i < end; i++) {
d_a[i] += d_a[i - 1];
}
```

2

```
__syncthreads();

if (tid == 0) {
d_last_sums[0] = 0;
for (int i = 1; i < num_threads; i++) {
d_last_sums[i] = d_a[(i * chunk_size) - 1] + d_last_sums[i - 1];
}
}

__syncthreads();

if (tid != 0) {
for (int i = start; i < end; i++) {
d_a[i] += d_last_sums[tid];
}
}
}

int main(int argc, char *argv[]) {
double elapsed_time_sum = 0.0;

size_t n = 0;
printf("[-] Please enter N: ");
scanf("%lu", &n);

int *a = (int *)malloc(n * sizeof(int));

int *d_a, *d_last_sums;
gpuErrchk(cudaMalloc(&d_a, n * sizeof(int)));
gpuErrchk(cudaMemcpy(d_a, a, n * sizeof(int), cudaMemcpyHostToDevice));

int num_threads = 8;
gpuErrchk(cudaMalloc(&d_last_sums, num_threads * sizeof(int)));

for (int i = 0; i < EXP_NUM; i++) {
fill_array(a, n);
double starttime = clock();

parallel_prefix_sum(a, n);

double elapsedtime = (clock() - starttime) / (double)CLOCKS_PER_SEC;
elapsed_time_sum += elapsedtime;
}
```

3

```
printf("average running time : %f\n", elapsed_time_sum / EXP_NUM);

//gpuErrchk(cudaFree(d_a));
//gpuErrchk(cudaFree(d_last_sums));
free(a);

return EXIT_SUCCESS;
}

void parallel_prefix_sum(int *a, size_t n) {
int *d_a, *d_last_sums;
gpuErrchk(cudaMalloc(&d_a, n * sizeof(int)));
gpuErrchk(cudaMemcpy(d_a, a, n * sizeof(int), cudaMemcpyHostToDevice));


int num_threads = 8;
gpuErrchk(cudaMalloc(&d_last_sums, num_threads * sizeof(int)));

int blockSize = num_threads;
int numBlocks = 1;

prefix_sum_kernel << <numBlocks, blockSize,blockSize*sizeof(int) >> > (d_a,
d_last_sums, n, num_threads);

gpuErrchk(cudaGetLastError());

gpuErrchk(cudaDeviceSynchronize());


gpuErrchk(cudaMemcpy(a, d_a, n * sizeof(int), cudaMemcpyDeviceToHost));
print_array(a, n);
gpuErrchk(cudaFree(d_a));
gpuErrchk(cudaFree(d_last_sums));
}

void fill_array(int *a, size_t n) {
for (size_t i = 0; i < n; ++i) {
a[i] = i + 1;
}
}

void print_array(int *a, size_t n) {
printf("[-] array: ");
for (size_t i = 0; i < n; ++i) {
printf("%d, ", a[i]);
```

4

```
}
printf("\b\b  \n");
}
```

1: تعریف تابع تشخیص خطا برای GPU

2 : ابتدا ایندکس thread گلوبال را مقداردهی می‌کنیم. سپس مقدار هر چانک داده که قرار است توسط یک نخ پردازش شود را مشخص می‌کنیم. طی یک حلقه for مقدار prefix-sum محلی هر چانک را محاسبه می‌کنیم.

__syncthreads() اطمینان حاصل می‌کند که تمام نخ‌ها prefix-sum محلی خود را محاسبه کرده‌اند و برای پردازش مرحله بعدی و مرج کردن نتایج با یکدیگر آماده هستند.

سپس برای بدست آوردن نتایج نهایی باید تعیین کنیم که در thread اول قرار داریم یا خیر. اگر در thread اول باشیم نتایج بدست آمده در آن چانک صحیح است اما برای threadهای بعدی باید آخرین عنصر چانک قبلی با تمام درایه‌های چانک فعلی جمع شوند.

3 : یک آرایه با نام d_a در GPU تخصیص می‌دهیم و داده‌های CPU را بر روی GPU کپی می‌کنیم.

4 : در این بخش کرنل را صدا می‌زنیم و تابع prefix_sum را روی آن پیاده‌سازی می‌کنیم.

نتایج به شرح زیر است:

Thread 2			
Size	1MB÷4B	10MB÷4B	Speed up
Serial	0.000332	0.003162	
CPU	0.000134	0.001109	2.66
GPU	0.132	1.226	0.002547

Thread 4			
Size	1MB÷4B	10MB÷4B	Speed up
Serial	0.000332	0.003162	
CPU	0.000134	0.001109	2.66
GPU	0.072	0.668	0.004672

Thread 8			
Size	1MB÷4B	10MB÷4B	Speed up
Serial	0.000332	0.003162	
CPU	0.000134	0.001109	2.66
GPU	0.043	0.395	0.007863

سوال 1- برای آرایه‌های کوچک، هزینه سربار انتقال داده به و از GPU ممکن است بیشتر از مزایای محاسبات موازی باشد. برای آرایه‌های بزرگ، GPU می‌تواند سرعت بهتری را ارائه دهد. اما به طور کلی این الگوریتم برای پیاده‌سازی روی CPU مناسب‌تر است زیرا ساختار SIMD ندارد.

سوال 2- افزایش اندازه کار (تعداد عناصری که هر رشته پردازش می‌کند) می‌تواند کارایی اجرای GPU را با موارد زیر بهبود بخشد:

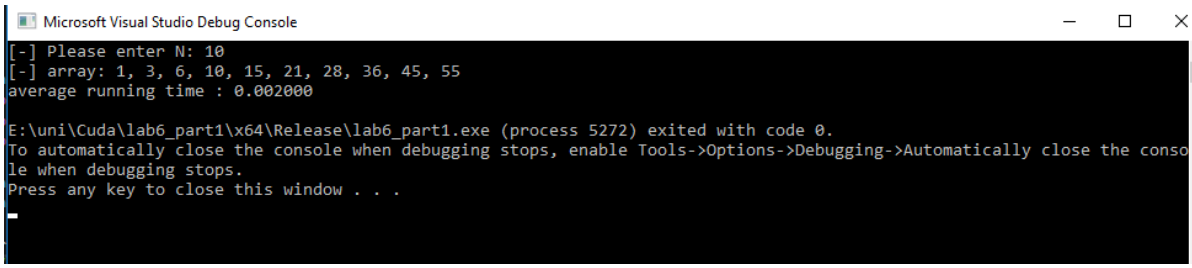
کاهش سربار: تکه‌های بزرگتر به معنای نیاز به نخ‌های کمتری است که باعث کاهش سربار مدیریت نخ می‌شود.

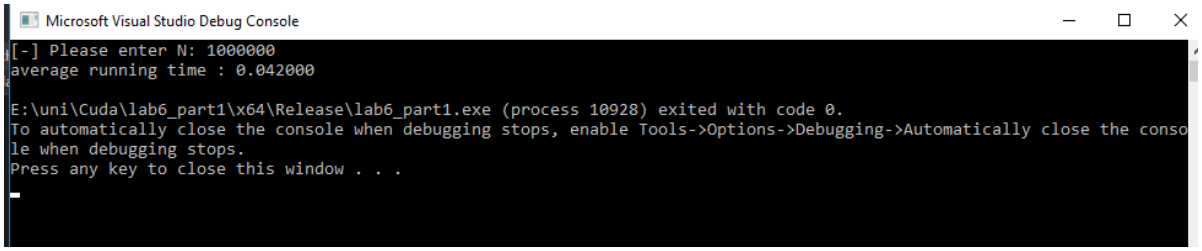
استفاده بهتر: با دادن کار بیشتر به هر رشته، منابع محاسباتی GPU به طور موثرتری استفاده می‌شود.

ادغام حافظه: تکه‌های بزرگتر می‌توانند به الگوهای دسترسی به حافظه کمک کنند و از پهنای باند حافظه GPU بهتر استفاده کنند.

با این حال، trade-off ای وجود دارد که باید مورد بررسی قرار گیرد:

اگر اندازه کار در هر رشته بیش از حد بزرگ شود، ممکن است منجر به استفاده ناکارآمد از قابلیت‌های پردازش موازی GPU شود.





بخش دوم: پیاده‌سازی با استفاده از الگوریتم Hillis and Steele

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cuda.h>
#define RUN_COUNT 5
double start_time, elapsed_time;

__global__ void prefixSumShared(int *data, int *prefixSum, int n) {
    const int index = blockIdx.x * blockDim.x + threadIdx.x;
    extern __shared__ int sharedMem[];

    // Load data into shared memory
    if (index < n) {
        sharedMem[threadIdx.x] = data[index];
    } else {
        sharedMem[threadIdx.x] = 0;
    }
    __syncthreads();

    // Perform parallel reduction in shared memory
    for (int d = 1; d < blockDim.x; d *= 2) {
        if (threadIdx.x >= d) {
            sharedMem[threadIdx.x] += sharedMem[threadIdx.x - d];
        }
        __syncthreads();
    }

    // Write the computed prefix sum to the output array
    if (index < n) {
        prefixSum[index] = sharedMem[threadIdx.x];
```

```
    }
}

__global__ void prefixSumNoShared(int* data, int* prefixSum, int n) {
    const int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index >= n) return;

    // Initialize the prefix sum with the first element
    prefixSum[index] = data[index];
    __syncthreads();

    for (int d = 1; d < n; d <<= 1) {
        int temp = 0;
        if (index >= d) {
            temp = prefixSum[index - d];
        }
        __syncthreads();

        if (index >= d) {
            prefixSum[index] += temp;
        }
        __syncthreads();
    }
}

int main() {
    int n;
    printf("Enter the length of the input array: ");
    scanf("%d", &n);

    // Allocate memory on host for data and prefix sum
    int *data = (int*)malloc(n * sizeof(int));
    int *prefixSum = (int*)malloc(n * sizeof(int));

    // Get input data from user
    for (int i = 0; i < n; i++) {
        data[i] = i;
    }

    // Choose between using shared memory or not
    int useSharedMemory;
    printf("Use shared memory (1) or not (0)? ");
    scanf("%d", &useSharedMemory);

    int blockSize = 32; // Adjust block size as needed
```

1

2

3

```
    int threadsPerBlock = (n + blockSize - 1) / blockSize;

    // Allocate memory on device for data and prefix sum
    int *d_data, *d_prefixSum;
    cudaMalloc(&d_data, n * sizeof(int));
    cudaMalloc(&d_prefixSum, n * sizeof(int));

    // Copy data to device
    cudaMemcpy(d_data, data, n * sizeof(int), cudaMemcpyHostToDevice);

    double total_time = 0;

    // Launch the appropriate kernel
    if (useSharedMemory) {
        for(int i = 0; i < RUN_COUNT; i++){
            start_time = clock();
            prefixSumShared<<<threadsPerBlock, blockSize,
blockSize*sizeof(int)>>>(d_data, d_prefixSum, n);
             // Synchronize the device to ensure completion
            cudaDeviceSynchronize();
            elapsed_time = clock() - start_time;
            total_time += elapsed_time;
        }
    }
    else {
        for(int i = 0; i < RUN_COUNT; i++){
            start_time = clock();
            prefixSumNoShared<<<threadsPerBlock, blockSize>>>(d_data,
d_prefixSum, n);
            // Synchronize the device to ensure completion
            cudaDeviceSynchronize();
            elapsed_time = clock() - start_time;
            total_time += elapsed_time;
        }
    }

    // Check for errors again
    cudaError_t err = cudaGetLastError();
    if (err != cudaSuccess) {
        printf("CUDA Error: %s\n", cudaGetErrorString(err));
        // Handle error...
    }

    // Copy prefix sum results back from device
    cudaMemcpy(prefixSum, d_prefixSum, n * sizeof(int), cudaMemcpyDeviceToHost);
```

```
    // Check for errors again
    err = cudaGetLastError();
    if (err != cudaSuccess) {
        printf("CUDA Error: %s\n", cudaGetErrorString(err));
        // Handle error...
    }

    // Print the prefix sum
    printf("%f\n", total_time/(CLOCKS_PER_SEC*RUN_COUNT));
    printf("Prefix sum: ");
    for (int i = 0; i < n; i++) {
        printf("%d\t", prefixSum[i]);
    }
}
```

1 : هر رشته عنصر مربوطه خود را در آرایه prefixSum با مقدار آرایه داده مقداردهی می‌کند.

__syncthreads() اطمینان می‌دهد که همه رشته‌ها این مقداردهی اولیه را قبل از حرکت به پایان رسانده‌اند.

2 : حلقه با دو برابر شدن d در هر بار تکرار می‌شود (d <<= 1) که نشان دهنده فاصله (گام) عناصری است که باید جمع شوند.

مرحله اول همگام سازی و خواندن:

هر رشته مقدار را از prefixSum[index - d] به یک temp متغیر موقت می‌خواند، اما فقط در صورتی که index >= d باشد. این کار از دسترسی خارج از محدوده جلوگیری می‌کند.

__syncthreads() اطمینان می‌دهد که تمام رشته‌ها خواندن مقادیر temp خود را قبل از هر به‌روزرسانی کامل کرده‌اند. این عمل از بروز شرایط مسابقه میان داده‌ها جلوگیری می‌کند که در آن برخی از رشته‌ها ممکن است مقادیر به‌روز شده را بخوانند در حالی که دیگران هنوز در حال محاسبه هستند.

مرحله دوم همگام سازی و نوشتن:

پس از همگام سازی، هر رشته به پیشوند Sum[index] خود temp اضافه می‌کند، اما فقط در صورتی که index >= d باشد.

3 : اندازه ورودی و تخصیص حافظه:

اندازه ورودی n را می‌خواند و حافظه روی host را برای آرایه‌های prefix sum اختصاص می‌دهد.

مقداردهی اولیه داده‌ها:

داده‌های ورودی را مقدار دهی اولیه می‌کند. برای سادگی، به هر عنصر مقدار شاخص خود را اختصاص می‌دهد.

حالت اجرای kernel را انتخاب می‌کند:

از کاربر می خواهد انتخاب کند که آیا از حافظه مشترک استفاده کند یا خیر.

محاسبه بلوک و موضوع:

تعداد بلوک ها و رشته های مورد نیاز را بر اساس اندازه ورودی n و اندازه بلوک 32 محاسبه می‌کند.

تخصیص حافظه در دستگاه:

حافظه روی GPU را برای داده‌های ورودی و آرایه‌های prefix sum اختصاص می‌دهد.

کپی داده‌ها در gpu:

داده های ورودی را از host به GPU کپی می کند.

اجرای کرنل و زمان بندی:

کرنل انتخاب شده (prefixSumShared یا prefixSumNoShared) را چندین بار اجرا می‌کند (RUN_COUNT) و زمان اجرا را با استفاده از clock() اندازه گیری می‌کند.

cudaDeviceSynchronize() تضمین می‌کند که اجرای هسته قبل از توقف clock کامل شده است.

میانگین زمان اجرا محاسبه می‌شود.

بررسی خطا:

خطاهای CUDA را پس از اجرای هسته و پس از کپی کردن نتایج به host بررسی می‌کند.

کپی کردن نتایج به host:

مجموع پیشوند محاسبه شده را از GPU به میزبان کپی می‌کند.

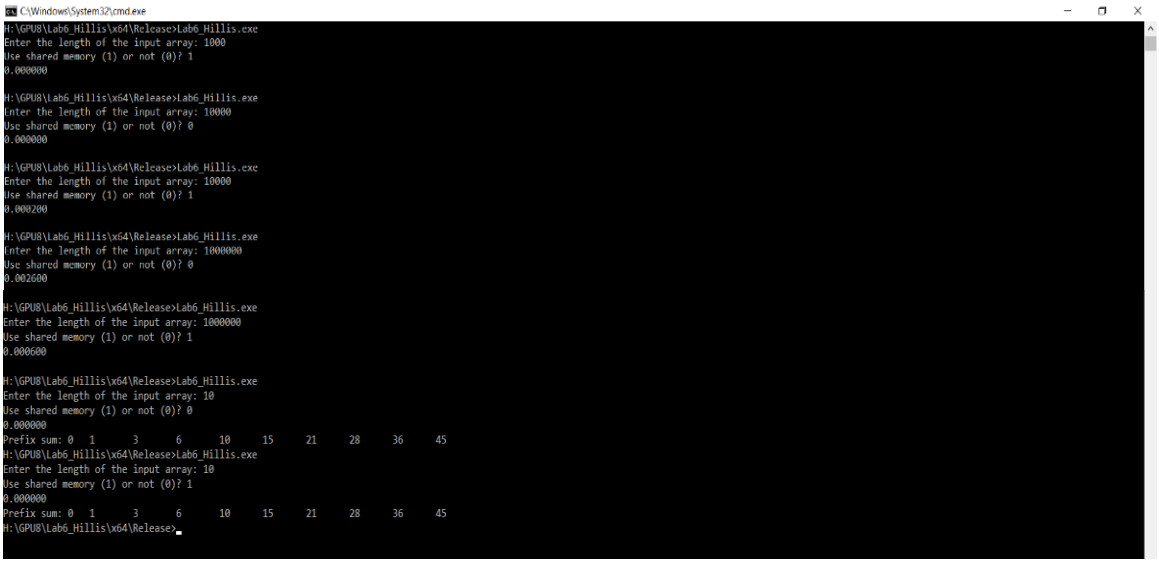
نتایج چاپ:

میانگین زمان اجرا و مجموع پیشوند محاسبه شده را چاپ می‌کند.

پاک کردن:

حافظه اختصاص داده شده در host و دیوایس را آزاد می‌کند.

نتایج به شرح زیر است:



سوال 3 :

اندازه warp:

واحد اصلی اجرای CUDA یک Warp است که از 32 رشته تشکیل شده است. انتخاب یک اندازه بلوک که مضربی از اندازه warp (32) باشد، تضمین می‌کند که warpها به طور کامل مورد استفاده قرار می‌گیرند، و نخ‌های بیکار را در یک warp به حداقل می‌رسانند.

Occupancy:

اشغال یک GPU به نسبت warpهای فعال در هر چند پردازنده به حداکثر تعداد warp هایی که می‌تواند فعال باشد اشاره دارد. اندازه بلوک 32 می‌تواند به دستیابی به بهره‌وری خوب کمک کند، مشروط بر اینکه منابع سخت افزاری (رجیسترها و حافظه مشترک) یک عامل محدود کننده نباشند.

انعطاف پذیری:

اندازه بلوک 32 انعطاف پذیری را فراهم می‌کند و تعداد نخ‌ها را در مقابل پیچیدگی اجرای هسته متعادل می‌کند. این نقطه شروع خوبی برای آزمایش است و می‌توان آن را بر اساس تنظیم عملکرد تنظیم کرد.

بخش a :

ارزیابی مقادیر انتخاب شده

آ. آیا مقادیر انتخاب شده بهتر از سایر مقادیر ممکن برای هر اندازه ورودی هستند؟

برای تعیین اینکه آیا مقادیر انتخاب شده بهینه هستند، باید در نظر بگیریم:

آزمایش و ارزیابی:

آزمایش اندازه های مختلف بلوک (به عنوان مثال، 64، 128، 256) و اندازه گیری عملکرد برای اندازه‌های ورودی مختلف برای شناسایی پیکربندی بهینه.

بهره برداری از منابع:

ارزیابی استفاده از منابع GPU (ثبت‌کننده‌ها، حافظه مشترک) برای اندازه‌های مختلف بلوک برای اطمینان از اینکه GPU به طور مؤثر و بدون اختلاف منابع مورد استفاده قرار می‌گیرد.

بخش b :

حافظه پین شده یا zero copy

حافظه پین شده (همچنین به عنوان حافظه قفل صفحه شناخته می‌شود) و حافظه zero copy می‌تواند عملکرد انتقال داده بین میزبان و دستگاه را افزایش دهد.

حافظه پین شده

تعریف:

حافظه پین شده حافظه میزبانی است که در حافظه فیزیکی قفل شده است و توسط سیستم عامل قابل صفحه بندی نیست. این امکان انتقال سریعتر داده‌ها بین host و دیوایس را فراهم می‌کند زیرا حافظه همیشه در RAM است.

مزایا:

سرعت انتقال سریعتر در مقایسه با حافظه قابل صفحه.

استفاده از عملیات کپی حافظه ناهمزمان را فعال می‌کند، که می‌تواند انتقال داده‌ها را با اجرای کرنل همپوشانی کند.

حافظه zero copy

تعریف:

حافظه zero copy به GPU اجازه می‌دهد تا مستقیماً به حافظه میزبان دسترسی داشته باشد بدون اینکه صریحاً آن را در حافظه دستگاه کپی کند. این می‌تواند برای کاهش ردپای حافظه و دستیابی به راندمان انتقال بالاتر برای موارد استفاده خاص مفید باشد.

مزایا:

نیاز به کپی های حافظه صریح را کاهش می‌دهد.

مناسب برای سناریوهایی که داده ها فقط یک بار خوانده می‌شوند یا یک بار نوشته می‌شوند.

سوال 4 :

دقت: double به دو برابر دقت بالاتر و بیت‌های بیشتری (64 بیتی) نسبت به float (32 بیتی) و int (32 بیتی) نیاز دارد.

Memory Bandwidth: مقدار داده‌ای که به و از حافظه GPU منتقل می‌شود. انواع داده‌های بزرگتر پهنای باند حافظه بیشتری می‌گیرند.

شدت حسابی: تعداد عملیات انجام شده در هر بایت داده منتقل شده. عملیات ممیز شناور می‌تواند از نظر محاسباتی فشرده‌تر از عملیات اعداد صحیح باشد.

استفاده از CUDA Core: برخی از معماری‌های GPU دارای هسته‌های جداگانه برای عملیات اعداد صحیح و ممیز شناور هستند که می‌تواند بر عملکرد تأثیر بگذارد.

دقت double :

پهنای باند حافظه: از آنجایی که دقت مضاعف به 64 بیت در هر عنصر نیاز دارد، پهنای باند حافظه در مقایسه با float و int بیشتر مورد استفاده قرار می گیرد.

بار محاسباتی: عملیات با دقت مضاعف از نظر محاسباتی فشرده‌تر است، که منجر به زمان اجرای بالقوه طولانی‌تر می‌شود.

تک دقیق (شناور):

پهنای باند حافظه: دقت شناور از 32 بیت در هر عنصر استفاده می‌کند، بنابراین از پهنای باند حافظه کمتری نسبت به دو برابر استفاده می‌کند.

بار محاسباتی: عملیات float از لحاظ محاسباتی کمتر از دو برابر است، اما بیشتر از int.

عدد صحیح (int):

پهنای باند حافظه: دقت int نیز از 32 بیت برای هر عنصر استفاده می کند، شبیه به شناور.

بار محاسباتی: عملیات اعداد صحیح معمولاً از نظر محاسباتی فشرده‌تر از عملیات ممیز شناور هستند که منجر به اجرای سریع‌تر می‌شود.

توضیح کرنل shared memory

```
if (index < n) {
     sharedMem[threadIdx.x] = data[index];
 } else {
     sharedMem[threadIdx.x] = 0;
 }
 __syncthreads();
```

شرط: بررسی می‌کند که آیا global thread index در محدوده داده‌های آرایه ورودی قرار دارد یا خیر.

اگر ایندکس < n باشد، رشته عنصر مربوطه را از داده در sharedMem کپی می‌کند.

اگر شاخص >= n باشد، رشته محل حافظه مشترک خود را به 0 مقداردهی می‌کند تا از خطاهای خارج از محدوده جلوگیری کند.

```
for (int d = 1; d < blockDim.x; d *= 2) {
     if (threadIdx.x >= d) {
         sharedMem[threadIdx.x] += sharedMem[threadIdx.x - d];
     }
     __syncthreads();
 }
```

حلقه بیرونی: در فواصل d که توان های 2 هستند (1، 2، 4، 8، ...) تکرار می‌شود.

شرط: بررسی می‌کند که آیا شاخص نخ در بلوک threadIdx.x بزرگتر یا مساوی d است یا خیر.

اگر درست باشد، رشته مقدار حافظه مشترک خود را با اضافه کردن مقدار در موقعیت‌های شاخص d قبل از آن به‌روزرسانی می‌کند: sharedMem[threadIdx.x] += sharedMem[threadIdx.x - d];

پس از بررسی و مقایسه نتایج بدست آمده، مشاهده شد که در روش دوم در حالت shared memory میزان تسریع با آرایه‌ای به اندازه 1000000 نسبت به حالت Distributed Memory 4.3 بوده اما برای ورودی با اندازه 10000 حالت Distributed memory سریع‌تر است.