**Iteration # 2 Code 2**

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#define N 512

\_\_global\_\_ void Reduction(float\* A, float\* B)

{

extern \_\_shared\_\_ float sharedMem[];

unsigned int t = threadIdx.x;

unsigned int start = blockIdx.x \* blockDim.x \* 2;

unsigned int i = start + t;

sharedMem[t] = A[i];

if (i + blockDim.x < N)

sharedMem[t] += A[i + blockDim.x];

for (unsigned int stride = blockDim.x >> 1; stride > 0; stride >>= 1) {

\_\_syncthreads();

if (t < stride) {

sharedMem[t] += sharedMem[t + stride];

}

}

if (t == 0) {

B[blockIdx.x] = sharedMem[0];

}

}

int main(int argc, char\*\* argv)

{

float\* hostA;

float\* hostB;

float\* deviceA;

float\* deviceB;

int Asize, Bsize;

Asize = N \* sizeof(float);

hostA = (float\*)malloc(Asize);

Bsize = sizeof(float); // Size for one result

hostB = (float\*)malloc(Bsize);

for (int i = 0; i < N; i++) {

hostA[i] = i + 1;

}

cudaMalloc((void\*\*)&deviceA, Asize);

cudaMalloc((void\*\*)&deviceB, Bsize);

cudaMemcpy(deviceA, hostA, Asize, cudaMemcpyHostToDevice);

// Adjusted block size and grid dimension calculation

dim3 blockDim(256);

dim3 gridDim((N + (2 \* blockDim.x) - 1) / (2 \* blockDim.x));

Reduction << <gridDim, blockDim, blockDim.x \* sizeof(float) >> > (deviceA, deviceB);

cudaDeviceSynchronize();

cudaMemcpy(hostB, deviceB, Bsize, cudaMemcpyDeviceToHost);

printf("\n\nVector A:\n");

for (int i = 0; i < N; i++) {

printf("%.0f ", hostA[i]);

}

printf("\n\nVector Sum:\n");

printf("%.0f\n", hostB[0]);

cudaFree(deviceA);

cudaFree(deviceB);

free(hostA);

free(hostB);

return 0;

}

**Analysis**

**A screenshot of a computer

Description automatically generated**

**Figure 1:** Occupancy Table

A graph on a black background

Description automatically generated

**Figure 2:** Threads per block

A graph with lines and numbers

Description automatically generated with medium confidence

**Figure 3:** Registers per thread

A graph with lines and dots

Description automatically generated

**Figure 4:** Shared memory per block

A screenshot of a computer

Description automatically generated

**Figure 5:** Summary

A screenshot of a computer

Description automatically generated

**Figure 6:** Results of Iteration 1

**Iteration # 3**

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#define N 512

// Kernel function for reduction

\_\_global\_\_ void Reduction(float\* A, float\* B)

{

extern \_\_shared\_\_ float sharedMem[];

unsigned int t = threadIdx.x;

unsigned int start = blockIdx.x \* blockDim.x \* 2;

unsigned int i = start + t;

// Load data into shared memory

sharedMem[t] = A[i];

if (i + blockDim.x < N)

sharedMem[t] += A[i + blockDim.x];

\_\_syncthreads();

// Use warp-level primitives for faster reduction within a warp

for (unsigned int stride = blockDim.x / 2; stride > 0; stride >>= 1) {

if (t < stride) {

sharedMem[t] += sharedMem[t + stride];

}

\_\_syncthreads();

}

// Store the result back to global memory

if (t == 0) {

B[blockIdx.x] = sharedMem[0];

}

}

int main(int argc, char\*\* argv)

{

float\* hostA;

float\* hostB;

float\* deviceA;

float\* deviceB;

int Asize, Bsize;

Asize = N \* sizeof(float);

hostA = (float\*)malloc(Asize);

Bsize = sizeof(float); // Size for one result

hostB = (float\*)malloc(Bsize);

for (int i = 0; i < N; i++) {

hostA[i] = i + 1;

}

cudaMalloc((void\*\*)&deviceA, Asize);

cudaMalloc((void\*\*)&deviceB, Bsize);

cudaMemcpy(deviceA, hostA, Asize, cudaMemcpyHostToDevice);

// Adjusted block size and grid dimension calculation

dim3 blockDim(256);

dim3 gridDim((N + (2 \* blockDim.x) - 1) / (2 \* blockDim.x));

Reduction << <gridDim, blockDim, blockDim.x \* sizeof(float) >> > (deviceA, deviceB);

cudaDeviceSynchronize();

cudaMemcpy(hostB, deviceB, Bsize, cudaMemcpyDeviceToHost);

printf("\n\nVector A:\n");

for (int i = 0; i < N; i++) {

printf("%.0f ", hostA[i]);

}

printf("\n\nVector Sum:\n");

printf("%.0f\n", hostB[0]);

cudaFree(deviceA);

cudaFree(deviceB);

free(hostA);

free(hostB);

return 0;

}

**Analysis**

A screenshot of a computer

Description automatically generated

**Figure 7:** Summary of Iteration 2

**Improvement Definition**

**Warp-level Primitives**: Instead of synchronizing all threads in a block using \_\_syncthreads() at each step of the reduction loop, we use warp-level primitives to reduce within a warp. This reduces the synchronization overhead, leading to faster execution.

**Memory Access Pattern**: We load data into shared memory in a coalesced manner to ensure efficient memory access. By accessing contiguous memory locations within each warp, we leverage memory coalescing for improved memory throughput.

**Iteration 3 Code 3**

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include "stdio.h"

#include "stdlib.h"

#include "math.h"

template <unsigned int blockSize>

\_\_device\_\_ void warpReduce(volatile float\* sdata, unsigned int tid) {

if (blockSize >= 64) {

sdata[tid] += sdata[tid + 32];

}

if (blockSize >= 32) {

sdata[tid] += sdata[tid + 16];

}

if (blockSize >= 16) {

sdata[tid] += sdata[tid + 8];

}

if (blockSize >= 8) {

sdata[tid] += sdata[tid + 4];

}

if (blockSize >= 4) {

sdata[tid] += sdata[tid + 2];

}

if (blockSize >= 2) {

sdata[tid] += sdata[tid + 1];

}

}

template <unsigned int blockSize>

\_\_global\_\_ void Reduction(float\* A, float\* B, int N) {

extern \_\_shared\_\_ float sdata[];

unsigned int tid = threadIdx.x;

unsigned int i = blockIdx.x \* (blockSize \* 2) + tid;

unsigned int gridSize = blockSize \* 2 \* gridDim.x;

unsigned int block = blockSize;

sdata[tid] = 0;

while (i < N) {

sdata[tid] += A[i] + A[i + blockSize];

i += gridSize;

}

\_\_syncthreads();

while (block >= 128) {

if (tid < block / 2) {

sdata[tid] += sdata[tid + block / 2];

}

block = block / 2;

\_\_syncthreads();

}

if (tid < 32) {

warpReduce<blockSize>(sdata, tid);

}

if (tid == 0) {

B[blockIdx.x] = sdata[0];

}

}

int main(int argc, char\*\* argv)

{

float\* hostA;

float\* hostB;

float\* deviceA;

float\* deviceB;

int Asize, Bsize;

int N = 2048;

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

Asize = N \* sizeof(float);

hostA = (float\*)malloc(Asize);

Bsize = sizeof(float); // Size for one result

hostB = (float\*)malloc(Bsize);

for (int i = 0; i < N; i++) {

hostA[i] = 1;

}

cudaMalloc((void\*\*)&deviceA, Asize);

cudaMalloc((void\*\*)&deviceB, Bsize);

cudaMemcpy(deviceA, hostA, Asize, cudaMemcpyHostToDevice);

dim3 blockDim(N / 2); //i had issues with the shared memory so i divided it into 2

dim3 gridDim(ceil((N + blockDim.x - 1) / blockDim.x)); // Adjusted grid dimension calculation

cudaEventRecord(start);

Reduction <1024> << <gridDim, blockDim, blockDim.x \* sizeof(float) >> > (deviceA, deviceB, N); // Shared memory size adjusted

cudaEventRecord(stop);

cudaDeviceSynchronize();

cudaMemcpy(hostB, deviceB, Bsize, cudaMemcpyDeviceToHost);

cudaEventSynchronize(stop);

float ms = 0;

cudaEventElapsedTime(&ms, start, stop);

printf("\n\nVector A:\n");

for (int i = 0; i < N; i++) {

printf("%.0f ", hostA[i]);

}

printf("\n\nVector Sum:\n");

printf("%.0f\n", hostB[0]);

printf("\nTime to run: %.4f ms", ms);

float bandwidth = N / (ms / 1000.0f);

printf("\nBandwidth: %.4f bytes/s", bandwidth);

cudaFree(deviceA);

cudaFree(deviceB);

free(hostA);

free(hostB);

return 0;

}

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A graph on a black background

Description automatically generated**

**A graph with a blue line

Description automatically generated**

**A graph with white lines

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**Iteration 4 Code 4**

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include "stdio.h"

#include "stdlib.h"

#include "math.h"

template <unsigned int blockSize>

\_\_device\_\_ void warpReduce(volatile float\* sdata, unsigned int tid) {

if (blockSize >= 64) sdata[tid] += sdata[tid + 32];

if (blockSize >= 32) sdata[tid] += sdata[tid + 16];

if (blockSize >= 16) sdata[tid] += sdata[tid + 8];

if (blockSize >= 8) sdata[tid] += sdata[tid + 4];

if (blockSize >= 4) sdata[tid] += sdata[tid + 2];

if (blockSize >= 2) sdata[tid] += sdata[tid + 1];

}

template <unsigned int blockSize>

\_\_global\_\_ void Reduction(float\* A, float\* B, int N) {

extern \_\_shared\_\_ float sdata[];

unsigned int tid = threadIdx.x;

unsigned int i = blockIdx.x \* blockDim.x \* 2 + tid;

unsigned int gridSize = blockDim.x \* 2 \* gridDim.x;

sdata[tid] = 0;

while (i < N) {

sdata[tid] += A[i] + A[i + blockDim.x];

i += gridSize;

}

\_\_syncthreads();

for (unsigned int s = blockDim.x / 2; s > 32; s >>= 1) {

if (tid < s) {

sdata[tid] += sdata[tid + s];

}

\_\_syncthreads();

}

if (tid < 32) {

warpReduce<blockSize>(sdata, tid);

}

if (tid == 0) {

B[blockIdx.x] = sdata[0];

}

}

int main(int argc, char\*\* argv) {

float\* hostA;

float\* hostB;

float\* deviceA;

float\* deviceB;

int Asize, Bsize;

int N = 2048;

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

Asize = N \* sizeof(float);

hostA = (float\*)malloc(Asize);

Bsize = sizeof(float); // Size for one result

hostB = (float\*)malloc(Bsize);

for (int i = 0; i < N; i++) {

hostA[i] = 1;

}

cudaMalloc((void\*\*)&deviceA, Asize);

cudaMalloc((void\*\*)&deviceB, Bsize);

cudaMemcpy(deviceA, hostA, Asize, cudaMemcpyHostToDevice);

dim3 blockDim(256); // Adjust block size for optimal performance

dim3 gridDim((N + blockDim.x \* 2 - 1) / (blockDim.x \* 2)); // Adjusted grid dimension calculation

cudaEventRecord(start);

Reduction<256> << <gridDim, blockDim, blockDim.x \* sizeof(float) >> > (deviceA, deviceB, N); // Shared memory size adjusted

cudaEventRecord(stop);

cudaDeviceSynchronize();

cudaMemcpy(hostB, deviceB, Bsize, cudaMemcpyDeviceToHost);

cudaEventSynchronize(stop);

float ms = 0;

cudaEventElapsedTime(&ms, start, stop);

printf("\n\nVector A:\n");

for (int i = 0; i < N; i++) {

printf("%.0f ", hostA[i]);

}

printf("\n\nVector Sum:\n");

printf("%.0f\n", hostB[0]);

printf("\nTime to run: %.4f ms", ms);

float bandwidth = N / (ms / 1000.0f);

printf("\nBandwidth: %.4f bytes/s", bandwidth);

cudaFree(deviceA);

cudaFree(deviceB);

free(hostA);

free(hostB);

return 0;

}

1. Reduce global memory accesses: Instead of reading A[i] and A[i + blockSize] separately, we can load them into shared memory first and then reduce them within the shared memory.
2. Simplify warpReduce function: Since we're only dealing with floats and the reductions are done within a warp, we can simplify the warpReduce function to reduce the number of if statements.
3. Optimize memory usage: Adjust the grid and block dimensions to maximize memory bandwidth utilization.

A screenshot of a computer

Description automatically generated

A graph on a computer screen

Description automatically generated

A graph on a black background

Description automatically generated

A graph with lines and text

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated