3-parallel-reduction-cu

May 8, 2025

[21]:
```
!pip install git+https://github.com/afnan47/cuda.git
```

```
Collecting git+https://github.com/afnan47/cuda.git
  Cloning https://github.com/afnan47/cuda.git to /tmp/pip-req-build-j25w8mkj
  Running command git clone --filter=blob:none --quiet
https://github.com/afnan47/cuda.git /tmp/pip-req-build-j25w8mkj
  Resolved https://github.com/afnan47/cuda.git to commit
aac710a35f52bb78ab34d2e52517237941399eff
  Preparing metadata (setup.py) … done
Building wheels for collected packages: NVCCPlugin
  Building wheel for NVCCPlugin (setup.py) … done
  Created wheel for NVCCPlugin: filename=NVCCPlugin-0.0.2-py3-none-any.whl
size=4290
sha256=c8f6e034779b7f0ee8c1ae200e11bd714d79a77cfdafa9d2d8e593d1a96c8fd3
  Stored in directory: /tmp/pip-ephem-wheel-cache-
jia_5g_f/wheels/bc/4e/e0/2d86bd15f671dbeb32144013f1159dba09757fde36dc51a963
Successfully built NVCCPlugin
Installing collected packages: NVCCPlugin
Successfully installed NVCCPlugin-0.0.2
```

[22]:
```
%load_ext nvcc_plugin
```

```
created output directory at /content/src
Out bin /content/result.out
```

[26]:
```
%%cu
// WARNING: DO NOT COPY THIS CODE, INSTEAD DOWNLOAD IT TO AVOID ERRORS.
#include <stdio.h>

#define BLOCK_SIZE 256

// Kernel for parallel reduction using min operation
__global__ void reduceMin(int* input, int* output, int size) {
    __shared__ int sdata[BLOCK_SIZE];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    // Load data into shared memory
```

```
    if (i < size) {
        sdata[tid] = input[i];
    } else {
        sdata[tid] = INT_MAX;
    }

    __syncthreads();

    // Perform reduction within each block
    for (unsigned int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
        if (tid < stride) {
            sdata[tid] = min(sdata[tid], sdata[tid + stride]);
        }
        __syncthreads();
    }

    // Write the result for this block to global memory
    if (tid == 0) {
        output[blockIdx.x] = sdata[0];
    }
}

// Kernel for parallel reduction using max operation
__global__ void reduceMax(int* input, int* output, int size) {
    __shared__ int sdata[BLOCK_SIZE];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    // Load data into shared memory
    if (i < size) {
        sdata[tid] = input[i];
    } else {
        sdata[tid] = INT_MIN;
    }

    __syncthreads();

    // Perform reduction within each block
    for (unsigned int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
        if (tid < stride) {
            sdata[tid] = max(sdata[tid], sdata[tid + stride]);
        }
        __syncthreads();
    }

    // Write the result for this block to global memory
    if (tid == 0) {
```

```
        output[blockIdx.x] = sdata[0];
    }
}

// Kernel for parallel reduction using sum operation
__global__ void reduceSum(int* input, int* output, int size) {
    __shared__ int sdata[BLOCK_SIZE];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    // Load data into shared memory
    if (i < size) {
        sdata[tid] = input[i];
    } else {
        sdata[tid] = 0;
    }

    __syncthreads();

    // Perform reduction within each block
    for (unsigned int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
        if (tid < stride) {
            sdata[tid] += sdata[tid + stride];
        }
        __syncthreads();
    }

    // Write the result for this block to global memory
    if (tid == 0) {
        output[blockIdx.x] = sdata[0];
    }
}

// Kernel for parallel reduction using average operation
__global__ void reduceAverage(int* input, float* output, int size) {
    __shared__ float sdata[BLOCK_SIZE];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    // Load data into shared memory
    if (i < size) {
        sdata[tid] = static_cast<float>(input[i]);
    } else {
        sdata[tid] = 0.0f;
    }

    __syncthreads();
```

```
    // Perform reduction within each block
    for (unsigned int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
        if (tid < stride) {
            sdata[tid] += sdata[tid + stride];
        }
        __syncthreads();
    }

    // Write the result for this block to global memory
    if (tid == 0) {
        output[blockIdx.x] = sdata[0] / static_cast<float>(size);
    }
}

int main() {
    // Input array
    const int array_size = 256;
    int input[array_size];

    // Initialize input array
    for (int i = 0; i < array_size; ++i) {
        input[i] = i + 1;
    }

    // Allocate device memory
    int* d_input;
    int* d_output_min;
    int* d_output_max;
    int* d_output_sum;
    float* d_output_avg;
    cudaMalloc((void**)&d_input, sizeof(int) * array_size);
    cudaMalloc((void**)&d_output_min, sizeof(int) * array_size);
    cudaMalloc((void**)&d_output_max, sizeof(int) * array_size);
    cudaMalloc((void**)&d_output_sum, sizeof(int) * array_size);
    cudaMalloc((void**)&d_output_avg, sizeof(float) * array_size);

    // Copy input array to device memory
    cudaMemcpy(d_input, input, sizeof(int) * array_size, cudaMemcpyHostToDevice);

    // Determine the number of threads and blocks
    int threads_per_block = BLOCK_SIZE;
    int blocks_per_grid = (array_size + threads_per_block - 1) / threads_per_block;

    // Launch the kernels for parallel reduction
```

```
    reduceMin<<<blocks_per_grid, threads_per_block>>>(d_input, d_output_min,␣
↪array_size);
    reduceMax<<<blocks_per_grid, threads_per_block>>>(d_input, d_output_max,␣
↪array_size);
    reduceSum<<<blocks_per_grid, threads_per_block>>>(d_input, d_output_sum,␣
↪array_size);
    reduceAverage<<<blocks_per_grid, threads_per_block>>>(d_input,␣
↪d_output_avg, array_size);

    // Copy the results back to the host
    int min_result, max_result, sum_result;
    float avg_result;
    cudaMemcpy(&min_result, d_output_min, sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(&max_result, d_output_max, sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(&sum_result, d_output_sum, sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(&avg_result, d_output_avg, sizeof(float),␣
↪cudaMemcpyDeviceToHost);

    // Print the results
    printf("Minimum value: %d\n", min_result);
    printf("Maximum value: %d\n", max_result);
    printf("Sum: %d\n", sum_result);
    printf("Average: %.2f\n", avg_result);

    // Free device memory
    cudaFree(d_input);
    cudaFree(d_output_min);
    cudaFree(d_output_max);
    cudaFree(d_output_sum);
    cudaFree(d_output_avg);

    return 0;
}
```

```
Minimum value: 0
Maximum value: 0
Sum: 0
Average: 0.00
```