Minimum using parallel reduction

20 May 2025 10:02

Dry Run

Step-1 load global memory into shared memory
→ each thread copies 2 values

We have an array of size 512
filled with floats e.g {0.0, 1.0, 2.0 ---- 511.0}

threads per block = 256
so each thread handles 2 elements
& also we have 64 blocks

Element = 512 = 2* Num_threads
threads_per_block = 256
Blocks = 64

shared[tid] = input[tid]
shared[tid + blockDim.x] = input[tid + blockDim.x]

if tid == 0
shared[0] = input[0] = 0
shared[0+256] = input[256] = 256

So after all threads finish
shared[] = [0, 1, 2 ---- 512]

Step-2 Record Start Time

```
if (tid == 0)
    timer[bid] = clock();
```

Step-3 Parallel Reduction → d = 256 → 128 → 64 → 32 → 16 → 8 → 4 → 2 → 1
9 steps or $(\log_2(512)) = 9$

iteration 1, d = blockDim.x = 256

so we are comparing shared[tid] & shared[tid + d] (d = 256)

so for tid < 256
shared[0] = min(shared[0], shared[256]) = min(0, 256) = 0
shared[1] = min(shared[1], shared[257]) = min(1, 257) = 1
⋮
shared[255] = min(shared[255], shared[511]) = min(255, 511) = 255

for iteration, $d = \frac{d}{2} = \frac{256}{2} = 128$

& for tid < 128
shared[0] = min(shared[0], shared[128]) = min(0, 128) = 0
shared[1] = min(shared[1], shared[129]) = min(1, 129) = 1
⋮
shared[127] = min(shared[127], shared[255]) = min(127, 255) = 127

as we continue
we will be left with, d = 1

shared[0] = min(shared[0], shared[1]) = min(0, 1) = 0
↳ this stores the min value from input[0 ... 511]

Step-4 Write Output

```
if (tid == 0)
    output[bid] = shared[0]
```

Step-5 Record End Time

```
if Tid == 0
    timer[bid + gridDim.x] = clock()
```

Output ⇒ [_ , _ , ---- _]
output of block 0, output of block 1, output of block 63

timer[bid] = clock() ← start
timer[bid + gridDim.x] = clock() ← end

timer ⇒ [start 0, start 1 ---- start 63, end 0, end 1 ---- end 63]
start time of block 0, start time of block 1, end time of block 0, end time of block 1

$$\text{Elapsed Clocks} = \sum_{i=0}^{\text{Num\_blocks}} \text{timer}[i + \text{Num\_blocks}] - \text{timer}[i]$$

$$\text{Avg Elapsed Clocks} = \frac{\text{Elapsed Clocks}}{\text{Num\_blocks}}$$

1) Clock cycle = A single tick of the GPU's internal timer, used to measure time taken by operations

2) Minimum Reduction ⇒ A way to find the smallest number in a large dataset by using parallel comparisons.
eg. if we have [10, 5, 20, 8, 3]
then a minimum reduction would give us 3

3) extern __shared__ float shared[];
→ This is how you declare shared memory in CUDA dynamically.
→ threads in the same block can access this shared memory very quickly.
→ we use extern __shared__ when we don't know the size at compile time
→ We can pass the size when we launch the kernel, in the execution configuration.
Kernel <<< blocks, threads, shared-mem-size >>> (...)

input → 512 floats → [Num_threads * 2]
dinput → " " "
doutput → 1 float per block → [Num_blocks]
dtimer → 2 clock entries per block ⇒ so [2 × Num_blocks]
↳ start
↳ stop

tid = threadIdx.x
bid = blockIdx.x

4) clock_t ⇒ It is used to measure how many GPU clock cycles a certain operation takes
eg. if an operation took 1500000 cycles
and your gpu's clock rate is 1.5GHz (1.5×10^9 cycles/sec)
then time = $\frac{1500000}{1.5 \times 10^9}$ sec ⇒ 1 millisecond

```
#include<assert.h>
#include<stdint.h>
#include<stdio.h>
#include<cuda_runtime.h>


__global__ void Myfunc(float *input, float *output, clock_t *timer){
    extern __shared__ float shared[];

    const int tid = threadIdx.x;
    const int bid = blockIdx.x;

    // start clock
    if(tid==0){
        timer[bid] = clock();
    }

    // copying data from global memory to shared memory
    shared[tid] = input[tid];
    shared[tid + blockDim.x] = input[tid + blockDim.x];

    // parallel reduction
    for(int i=blockDim.x;i>0;i /= 2){
        __syncthreads();
        if(tid<i){
            float f0 = shared[tid];
            float f1 = shared[tid+i];

            shared[tid] = f1>f0? f0:f1;
        }
    }
```

```
Running NVIDIA GV100 in FUNCTIONAL mode...
Compiling...
Executing...
Average clocks/block = 9935.703125
Exit status: 0
```

```
    // storing the ouput
    if(tid==0){
        output[bid] = shared[0];
    }

    __syncthreads();

    // stop clock
    if(tid==0){
        timer[bid+gridDim.x] = clock();
    }



}

#define NUM_BLOCKS 64
#define NUM_THREADS 256
```

```
int main(){
    float *dinput = NULL;
    float *doutput = NULL;
    clock_t *dtimer = NULL;

    clock_t timer[NUM_BLOCKS * 2];
    float input[NUM_THREADS * 2];

    for(int i=0;i<NUM_THREADS*2;i++){
        input[i] = (float)i;
    }

    cudaMalloc(&dinput, sizeof(float) * NUM_THREADS*2);
    cudaMalloc(&doutput, sizeof(float)*NUM_BLOCKS);
    cudaMalloc(&dtimer, sizeof(clock_t)*NUM_BLOCKS*2);


    cudaMemcpy(dinput ,input ,sizeof(float)*NUM_THREADS * 2, cudaMemcpyHostToDevice);

    Myfunc<<<NUM_BLOCKS,NUM_THREADS,sizeof(float)*NUM_THREADS*2>>>(dinput, doutput, dtimer);

    cudaMemcpy(timer, dtimer, sizeof(clock_t)*NUM_BLOCKS*2, cudaMemcpyDeviceToHost);

    cudaFree(dinput);
    cudaFree(doutput);
    cudaFree(dtimer);
```

```
    long double Elapsedclocks = 0;
    for(int i=0;i<NUM_BLOCKS;i++){
        Elapsedclocks += (long double)(timer[i+NUM_BLOCKS] - timer[i]);
    }

    long double avgElapsedclocks = Elapsedclocks/NUM_BLOCKS;
    printf("Average clocks/block = %Lf\n", avgElapsedclocks);

    return 0;


}
```