Parallel Programming Final Project

Group13 108032053 陳凱揚

Outline

- Problem description
- CPU - sequential
- GPU - optimize step by step
- Time distribution
- Profiling

Problem description

- Given **n** balloons. Each balloon is painted with a number on it represented by an array nums. You are asked to burst all the balloons.
- If you burst the **ith** balloon, you will get **nums[i-1]*nums[i]*nums[i+1]** coins. If **i-1** or **i+1** goes out of bounds of the array, then treat it as if there is a balloon with a **1** painted on it.
- Return the maximum coins you can collect by bursting the balloons wisely.
- Constraints: **1** <= **n** <= **10000**, **1** <= **nums[i]** <= **50**.

```
Input: nums = [3,1,5,8]
Output: 167
Explanation:
nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []
coins =  3*1*5    +   3*5*8    +  1*3*8   + 1*8*1 = 167
```

CPU - sequential

- Solved by dynamic programming.
- Define state
  - dp[i][j] is the maximum coins you can collect after bursting all balloons in [i + 1, j - 1] (not including i and j).
- State transition equation
  - dp[i][j] = 0, if j - i + 1 = 2.
  - dp[i][j] = max{ dp[i][k]+dp[k][j]+nums[i]*nums[k]*nums[j] } for all k in [i + 1, j - 1], if j - i + 1 > 2.
- Answer: dp[0][n - 1]
- Time complexity: $O(n^3)$

CPU - sequential

```
// solution
data[0] = data[n+1] = 1;
new_n = n+2;
for(int len = 3; len <= new_n; len++){
    for(int i = 0; i < new_n-len+1; i++){
        int j = i+len-1;
        for(int k = i+1; k < j; k++)
            dp[i][j] = std::max(dp[i][j], dp[i][k]+dp[k][j]+data[i]*data[k]*data[j]);
    }
}
```

GPU - optimize step by step

- Baseline (21.92s)
- DP reindexing (7.57s)
- Parallel maxreduce (6.82s)
- Two data per thread (5.99s)
- Multiple data per thread (4.07s)
- Unroll last warp (3.93s)
- Unroll all (3.90s)

Baseline (21.92s)

- Store input data to the constant memory in the GPU because it is read only.
- Parallel the second loop first because the first loop has dependency and the third loop should use atomicMax or perform max reduce operation, which will be optimized later.
- The number of threads per block is fixed to 1024 (#define NT 1024).

Baseline (21.92s)

```
__constant__ int data_GPU[N+2];
```

```
cudaMemcpyToSymbol(data_GPU, data, dp_n*sizeof(int));
cudaMalloc(&dp, dp_n*dp_n*sizeof(int));
cudaMemset(dp, 0, dp_n*dp_n*sizeof(int));
for(int len = 3; len <= dp_n; len++){
    int num = dp_n-len+1, block_num = (num+NT-1)/NT;
    max_reduce <<<block_num, NT>>> (len, dp_n, dp);
}
cudaMemcpy(&res, dp+(dp_n-1), sizeof(int), cudaMemcpyDeviceToHost);
```

```
__global__ void max_reduce(int len, int n, int *dp){
    int i = blockIdx.x*blockDim.x+threadIdx.x, j = i+len-1;
    if(i >= n-len+1) return;
    for(int k = i+1; k < j; k++)
        dp[i*n+j] = max(dp[i*n+j], dp[i*n+k]+dp[k*n+j]+data_GPU[i]*data_GPU[k]*data_GPU[j]);
}
```

DP reindexing (7.57s)

- Change the index of DP because the first loop is according to the length of subarray.
- Redefine state
  - dp[len][left_idx] is the maximum coins you can collect after bursting all balloons in [left_idx + 1, (left_idx+len-1) - 1].
- State transition equation stays the same.
- Answer: dp[n][0]

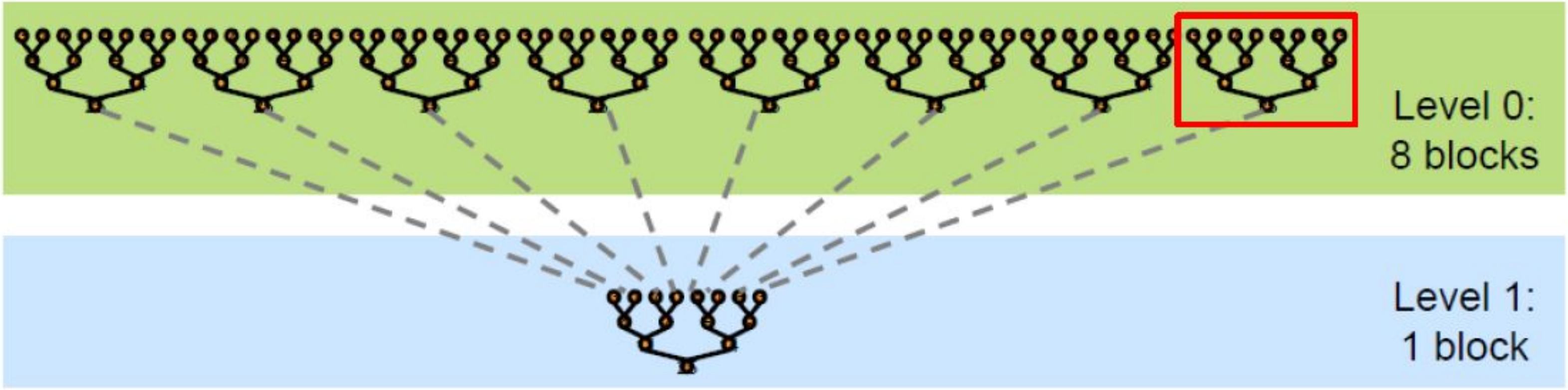
DP reindexing (7.57s)

```
cudaMemcpyToSymbol(data_GPU, data, dp_n*sizeof(int));
cudaMalloc(&dp, (dp_n+1)*dp_n*sizeof(int));
cudaMemset(dp, 0, (dp_n+1)*dp_n*sizeof(int));
for(int len = 3; len <= dp_n; len++){
    int num = dp_n-len+1, block_num = (num+NT-1)/NT;
    max_reduce <<<block_num, NT>>> (len, dp_n, dp);
}
cudaMemcpy(&res, dp+(dp_n*dp_n), sizeof(int), cudaMemcpyDeviceToHost);
```

```
__global__ void max_reduce(int len, int n, int *dp){
    int left_idx = blockIdx.x*blockDim.x+threadIdx.x, right_idx = left_idx+len-1;
    if(left_idx >= n-len+1) return;
    for(int j = 2; j < len; j++){
        int left_len = j, right_len = len-j+1, mid_idx = left_idx+left_len-1;
        dp[len*n+left_idx] = max(dp[len*n+left_idx], \
            dp[left_len*n+left_idx]+dp[right_len*n+mid_idx]+data_GPU[left_idx]*data_GPU[mid_idx]*data_GPU[right_idx]);
    }
}
```

Parallel maxreduce (6.82s)

- Use similar method from the NVIDIA slides which is mentioned in the class to parallel max reduce.
- Use two dimensions block, the second dimension is the block of level.
- Use an additional array to store reduced data.



Parallel maxreduce (6.82s)

```
for(int len = 3; len <= dp_n; len++){
    int blockX_num = dp_n-len+1, num_data = len-2;
    read_input <<<blockX_num, NT>>> (len, dp_n, dp, reduce_n, reduce_data);
    while(num_data > 1){
        int blockY_num = (num_data+NT-1)/NT;
        max_reduce <<<dim3(blockX_num, blockY_num), NT, NT*sizeof(int)>>> (reduce_n, reduce_data);
        num_data = blockY_num;
    }
    write_output <<<blockX_num, 1>>> (len, dp_n, dp, reduce_n, reduce_data);
}
```

```
__global__ void max_reduce(int n, int *reduce_data){
    extern __shared__ int sdata[];
    int left_idx = blockIdx.x, data_idx = blockIdx.y*blockDim.x+threadIdx.x, tid = threadIdx.x;
    sdata[tid] = reduce_data[left_idx*n+data_idx];
    __syncthreads();
    for(int s = blockDim.x/2; s > 0; s >>= 1){
        if(tid < s) sdata[tid] = max(sdata[tid], sdata[tid+s]);
        __syncthreads();
    }
    if(tid == 0) reduce_data[left_idx*n+blockIdx.y] = sdata[0];
}
```

Two data per thread (5.99s)

```
for(int len = 3; len <= dp_n; len++){
    int blockX_num = dp_n-len+1, num_data = len-2;
    read_input <<<blockX_num, NT>>> (len, dp_n, dp, reduce_n, reduce_data);
    while(num_data > 1){
        int blockY_num = (num_data+NT*2-1)/(NT*2);
        max_reduce <<<dim3(blockX_num, blockY_num), NT, NT*sizeof(int)>>> (reduce_n, reduce_data);
        num_data = blockY_num;
    }
    write_output <<<blockX_num, 1>>> (len, dp_n, dp, reduce_n, reduce_data);
}
```

```
__global__ void max_reduce(int n, int *reduce_data){
    extern __shared__ int sdata[];
    int left_idx = blockIdx.x, data_idx = blockIdx.y*(blockDim.x*2)+threadIdx.x, tid = threadIdx.x;
    sdata[tid] = max(reduce_data[left_idx*n+data_idx], reduce_data[left_idx*n+(data_idx+blockDim.x)]);
    __syncthreads();
    for(int s = blockDim.x/2; s > 0; s >>= 1){
        if(tid < s) sdata[tid] = max(sdata[tid], sdata[tid+s]);
        __syncthreads();
    }
    if(tid == 0) reduce_data[left_idx*n+blockIdx.y] = sdata[0];
}
```

Multiple data per thread (4.07s)

- Each thread loads **NUM*2** data, so there are only 1 block in each layer.
- The second dimension of block can be removed.
- Shared memory is large enough to store all reduced data.

```
int NUM = (dp_n+(NT*2)-1)/(NT*2);
for(int len = 3; len <= dp_n; len++){
    int block_num = dp_n-len+1, num_data = len-2;
    max_reduce <<<block_num, NT, NT*sizeof(int)>>> (NUM, num_data, len, dp_n, dp);
}
```

```
extern __shared__ int sdata[];
int left_idx = blockIdx.x, tid = threadIdx.x, val = 0;
for(int i = 0; i < NUM; i++){
    val = max(val, get_data(left_idx, len, i*(NT*2)+tid, num_data, n, dp));
    val = max(val, get_data(left_idx, len, i*(NT*2)+tid+NT, num_data, n, dp));
}
sdata[tid] = val;
__syncthreads();
```

Unroll last warp (3.93s)

- Use the same method from the NVIDIA slides to unroll last warp.
- There are 32 threads per warp.
- Instructions are SIMD synchronous within a warp

```
for(int s = blockDim.x/2; s > 32; s >>= 1){
    if(tid < s) sdata[tid] = max(sdata[tid], sdata[tid+s]);
    __syncthreads();
}
if(tid < 32) warp_reduce(sdata, tid);
```

```
__device__ void warp_reduce(volatile int *sdata, int tid){
    sdata[tid] = max(sdata[tid], sdata[tid+32]);
    sdata[tid] = max(sdata[tid], sdata[tid+16]);
    sdata[tid] = max(sdata[tid], sdata[tid+8]);
    sdata[tid] = max(sdata[tid], sdata[tid+4]);
    sdata[tid] = max(sdata[tid], sdata[tid+2]);
    sdata[tid] = max(sdata[tid], sdata[tid+1]);
}
```

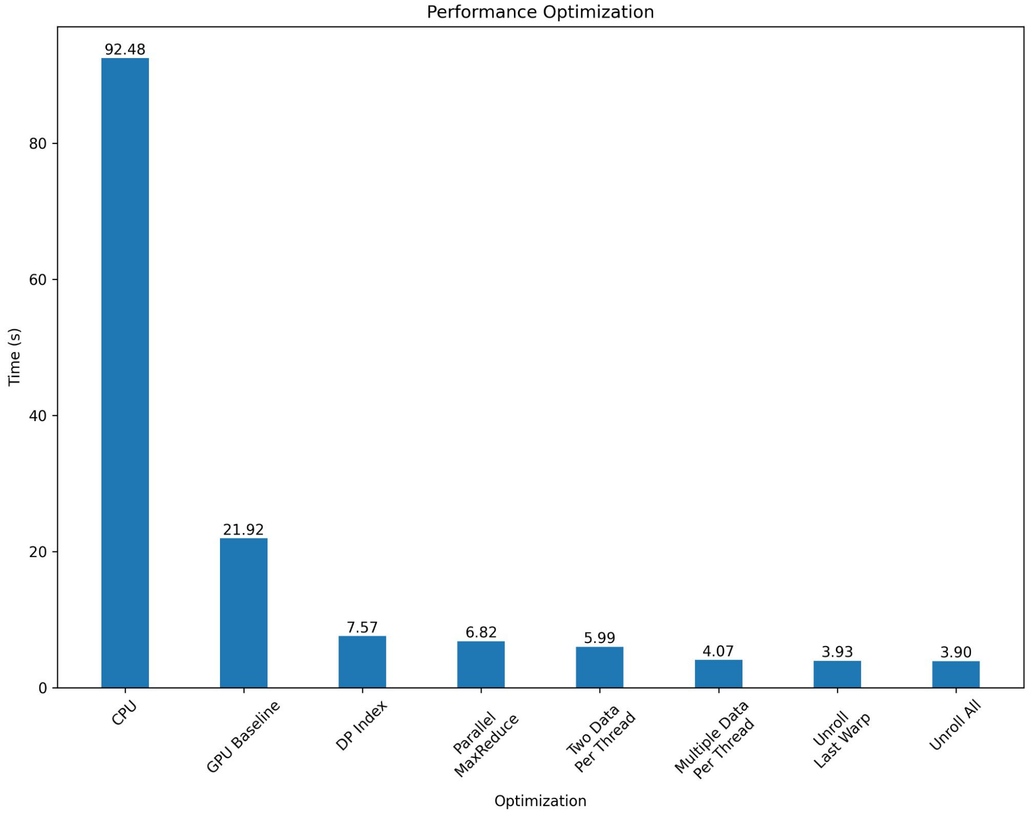
Unroll all (3.90s)

- Use the same method from the NVIDIA slides to unroll all loop.
- Use C++ template parameters

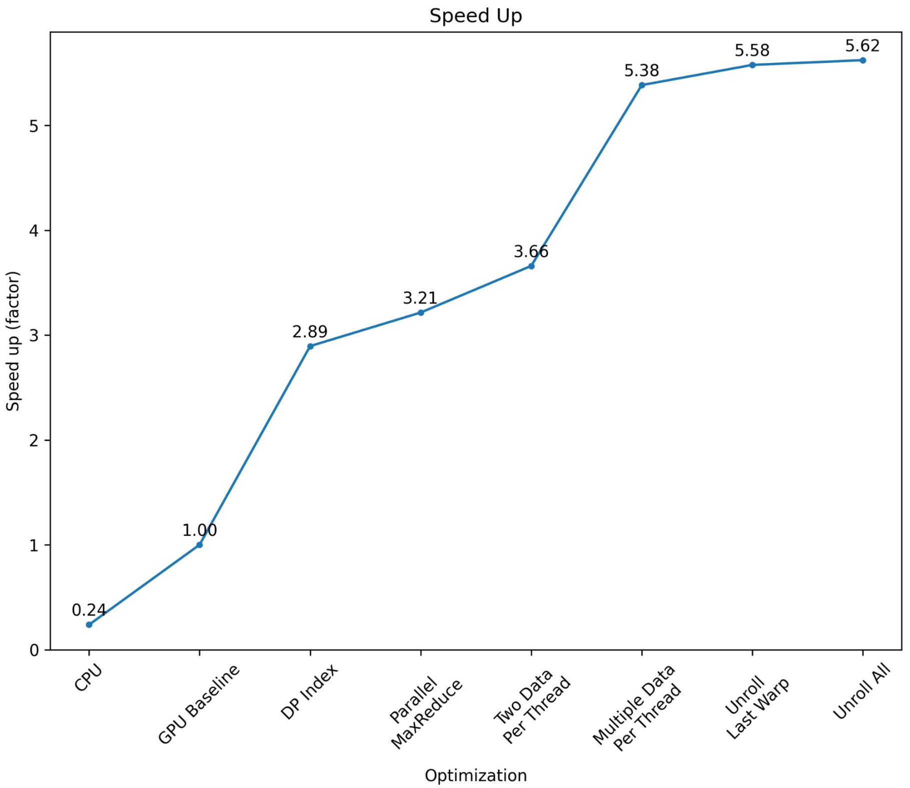
```
switch(NUM){
    case 1:
        max_reduce<1> <<<block_num, NT, NT*sizeof(int)>>> (
    case 2:
        max_reduce<2> <<<block_num, NT, NT*sizeof(int)>>> (
    case 3:
        max_reduce<3> <<<block_num, NT, NT*sizeof(int)>>> (
    case 4:
        max_reduce<4> <<<block_num, NT, NT*sizeof(int)>>> (
    case 5:
        max_reduce<5> <<<block_num, NT, NT*sizeof(int)>>> (
}
```

```
template <int NUM>
__global__ void max_reduce(int num_data, int len, int n, int *dp){
    extern __shared__ int sdata[];
    int left_idx = blockIdx.x, tid = threadIdx.x, val = 0;
    #pragma unroll
    for(int i = 0; i < NUM; i++){
        val = max(val, get_data(left_idx, len, i*(NT*2)+tid, num_data, n, dp));
        val = max(val, get_data(left_idx, len, i*(NT*2)+tid+NT, num_data, n, dp));
    }
    sdata[tid] = val;
    __syncthreads();
    if(tid < 512) sdata[tid] = max(sdata[tid], sdata[tid+512]);
    __syncthreads();
    if(tid < 256) sdata[tid] = max(sdata[tid], sdata[tid+256]);
    __syncthreads();
    if(tid < 128) sdata[tid] = max(sdata[tid], sdata[tid+128]);
    __syncthreads();
    if(tid < 64) sdata[tid] = max(sdata[tid], sdata[tid+64]);
    __syncthreads();
    if(tid < 32) warp_reduce(sdata, tid);
    if(tid == 0) dp[len*n+left_idx] = sdata[0];
}
```

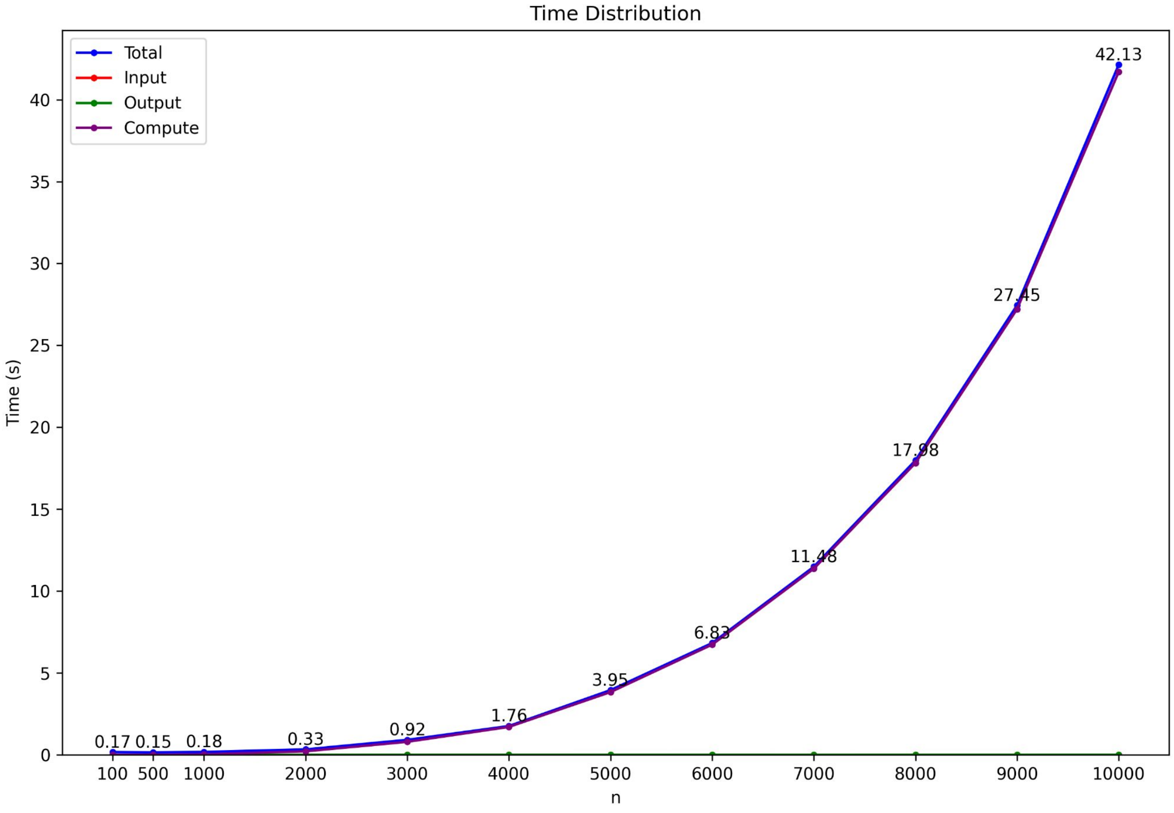
GPU - optimize step by step



GPU - optimize step by step



Time distribution



Profiling

```
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 (0)"
    Kernel: void max_reduce<int=3>(int, int, int, int*)
       5000                              sm_efficiency                  Multiprocessor Activity       4.35%      99.69%      98.63%
       5000                         achieved_occupancy                       Achieved Occupancy    0.493134    0.993579    0.900547
       5000                      shared_load_throughput            Shared Memory Load Throughput  546.76MB/s  192.66GB/s  28.226GB/s
       5000                     shared_store_throughput           Shared Memory Store Throughput  509.31MB/s  179.46GB/s  26.293GB/s
       5000                             gld_throughput                   Global Load Throughput  1.3196GB/s  464.74GB/s  322.34GB/s
       5000                             gst_throughput                  Global Store Throughput  1.8725MB/s  675.63MB/s  98.984MB/s
```

Reference

- https://leetcode.com/problems/burst-balloons/
- https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf