

# Parallelize Genetic Algorithm by Thrust Library

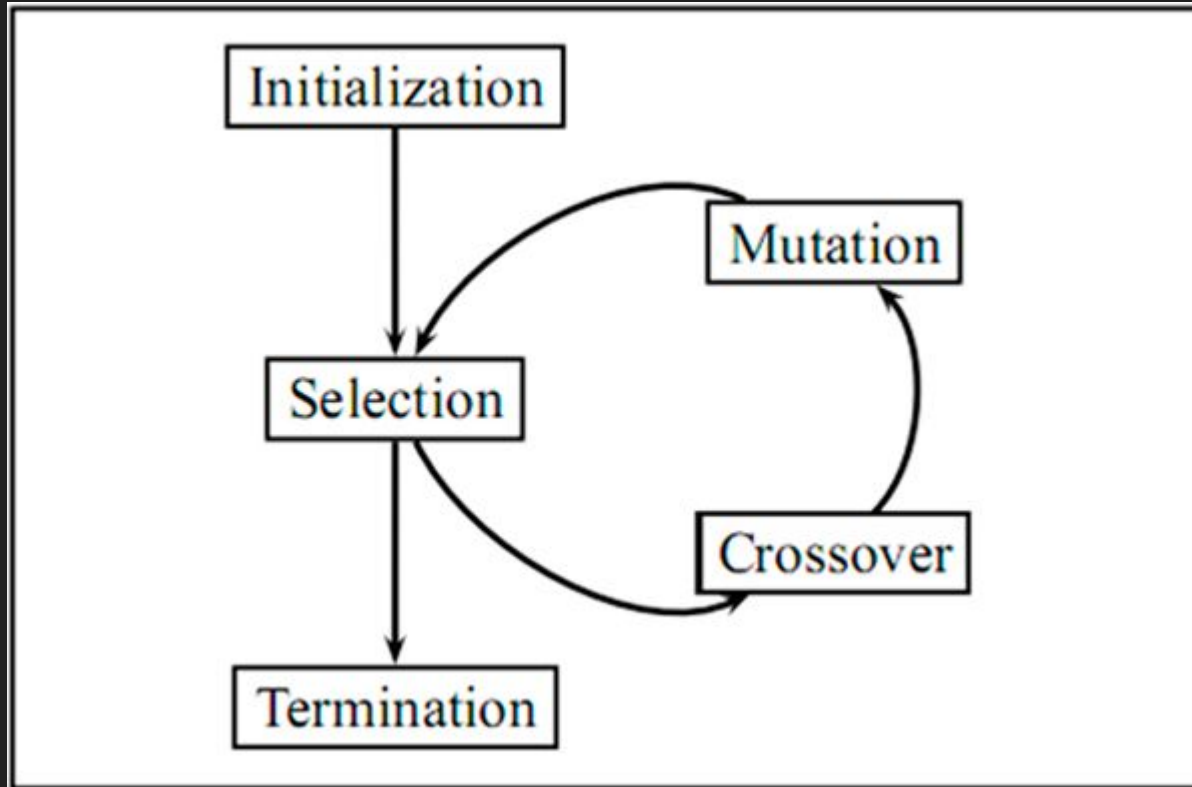
0556032 黃奕齊

0456024 蔡佩珊

# Introduction

- 基因演算法(Genetic Algorithm)是解最佳化問題的演算法，屬於演化計算(Evolutionary Computation)的一個分支。
- 此法藉由模擬生物演化的機制：將最佳化問題類比為自然環境，在此環境下存在一個群體(Population)，群體內有多個候選解，稱為個體(Individual)，每個候選解內容可表示成變量序列，稱為染色體(Chromosome)。
- 進化從隨機的初始群體開始，在每個世代評價個體的適應度(Fitness)，適應度高的個體通過天擇(Selection)可以進行交配(Crossover)及突變(Mutation)，遺傳生成下個世代的個體，周而復始，直到收斂至全域最佳解、局部最佳解或者滿足終止條件。

# Introduction



# Motivation

- 基因演算法的流程跟目標問題沒有太大的相關，因此很容易使用，常被用來解最佳化問題(Optimization Problem)以及搜索問題(Search Problem)。
- 但是當個體數量變多、染色體維度變大、適應度計算變複雜的時候，求解所需的收斂時間便會大幅增加。
- 因此，如何加速基因演算法是個值得重視的議題，而平行計算(Parallel computing)便是個最能保證提升效能的方法。

# PROPOSED APPROACHES

- 在基因演算法中，個體即為解，因此許多的運算操作都是以個體為單位進行。我們依照此特性所採取的平行化單位即是個體。以下為平行化細節，其中 $N_I$ 為個體數， $N_T$ 為執行緒(Thread)數。

# PROPOSED APPROACHES

- Initialization

- 個體的初始化是互相獨立且計算量相當，因此我們把個體均分至線程進行初始化。在本專的平行化架構下，每個執行緒：初始化  $N_I/N_T$  個個體。

- Evaluation

- 個體的適應度計算是互相獨立且計算量相當，因此我們把個體均分至線程進行適應度計算。在本次的平行化架構下，每個執行緒：計算  $N_I/N_T$  個個體的適應度。

# PROPOSED APPROACHES

- Selection, Crossover and Mutation

- 演化的過程有三個步驟，篩選、交配及突變。雖然篩選及突變是以單個個體為單位，但是考慮到：
  - 每個步驟的計算量不大
  - 記憶體配置(例如，在篩選的時候以個單個個體為單位進行，因為後面的交配步驟是以兩個個體為單位進行，為了能讓執行緒間能互相存取篩選出來的結果，勢必得將資料配置在執行緒的生命週期之外，但是這個資料除了這個操作外即無用途，造成記憶體資源浪費)
  - 執行緒啟動的代價(例如，在大部分的應用中，突變的機率趨近於零，為了那少量的突變操作，而要在每次演化都特地開執行緒會增加不必要的成本)
- 因此我們將三個步驟合併，以交配的兩個個體為單位進行。在本次的平行化架構下，每個執行緒：  
進行 $N_P/2N_T$ 次篩選出兩個父代、將兩個父代進行交配 產生兩個子代、將兩個子代進行突變。

# Implementation

- Thrust Library
  - 與STL相容的API
    - `for_each`
    - `transform`
    - `min_element`
    - ...
  - Container免去配置空間的繁雜程式碼
    - `device_vector`
    - `host_vector`
  - 自動根據輸入大小分配執行緒，不需要自行分配。





# Implementation

- 支援多種backend.
  - CPP (sequential)
  - OMP
  - TBB (multicore)
  - CUDA
- specify the flag in compile time:
  - `-DTHRUST_DEVICE_SYSTEM=THRUST_DEVICE_SYSTEM_{CPP/OMP/TBB/CUDA}`
- 不需要改動程式碼, 即可達到各種平行化。

# Implementation details

- 記憶體配置部分可直接利用`thrust::device_vector`, 省下`cudaMalloc`等繁雜的程式碼。但是因為`thrust::device_vector`不支援巢狀, 所以將所有空間配置好後, 還需要將每段空間的指標指定給每個個體。

# Implementation details

- 在基因演算法中，有這幾個主要的迴圈：
  - a. 初始化所有個體的迴圈
    - 使用 `thrust::for_each` 將 `for(int i=0; i<NI; i++)` 的迴圈平行化。
  - b. 最外層的 iteration 的迴圈
    - 此迴圈無法平行化，因為每個 iteration 都是根據前一個 iteration 的結果來進行。
  - c. 對所有個體進行 Selection, crossover, mutation 的迴圈
    - 使用 `thrust::for_each` 將 `for(int i=0; i<NI; i+=2)` 的迴圈平行化。
  - d. 計算所有個體的適應值的迴圈
    - 使用 `thrust::for_each` 將 `for(Individual &individual: individuals)` 的迴圈平行化。
  - e. 找尋最佳個體的迴圈
    - 使用 `thrust::min_element` 直接呼叫平行化的函式。

以上就是主要的實作細節。

# Experiment

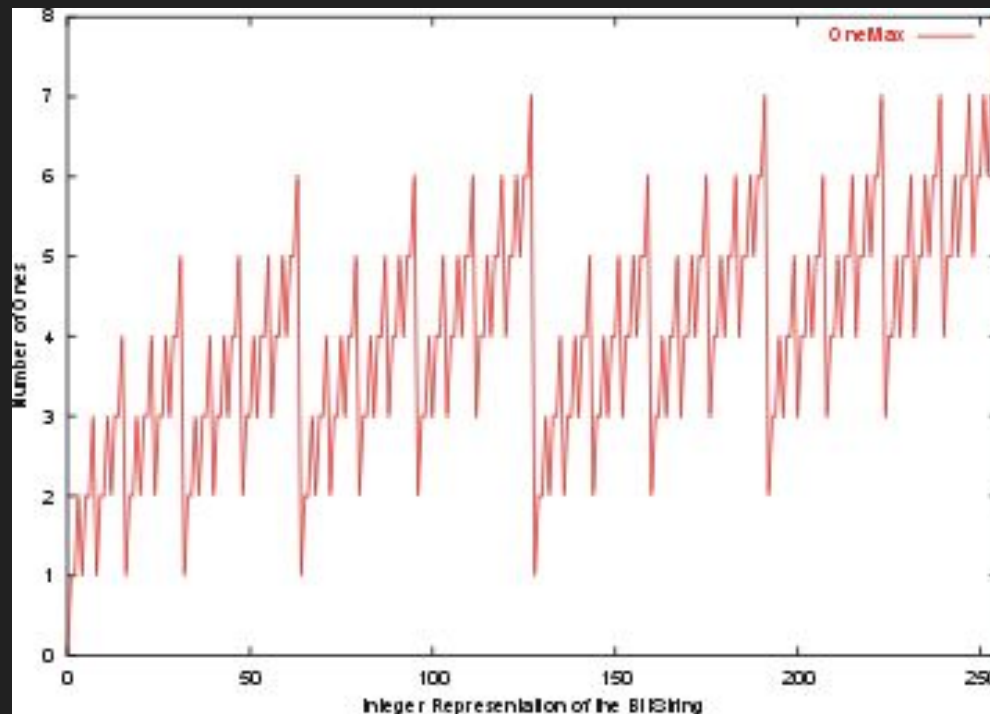
- CPU:
  - Intel(R) Xeon(R) CPU E3-1231 v3 @ 3.40GHz
- GPU:
  - NVIDIA GeForce GTX770

# Experiment

- P.S. 以下的population與gene size的單位均為千

# Experiment

- OneMax
  - minimum at  $\{0, \dots, 0\}$
  - maximum at  $\{1, \dots, 1\}$



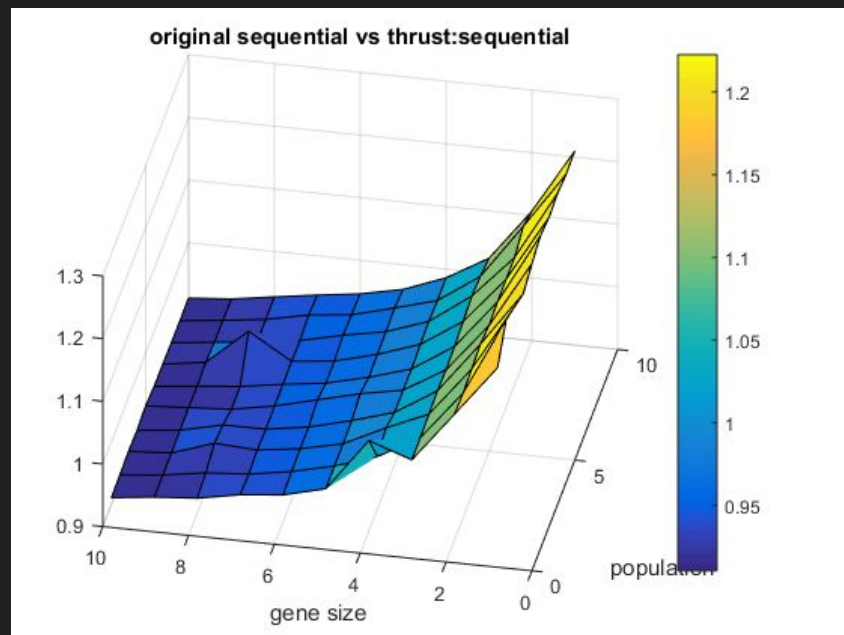
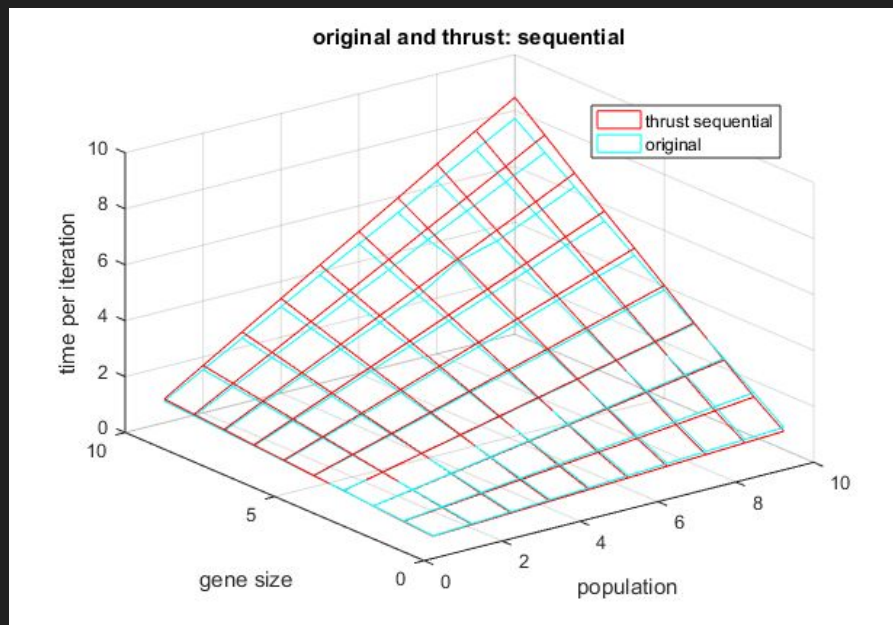
$$F(\vec{x}) = \sum_{i=1}^N x_i$$

# Onemax sequential

- thrust版在gene size小的時候花的時間最多約是原版的1.2倍，但在gene size大時約為原版的0.95倍。

左圖：原版與thrust sequential的time per iteration

右圖：時間比值。

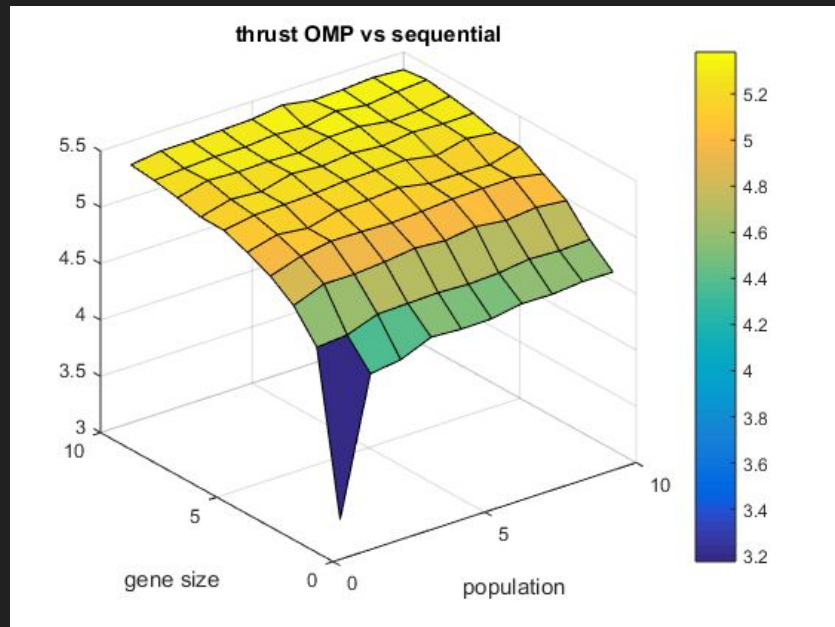
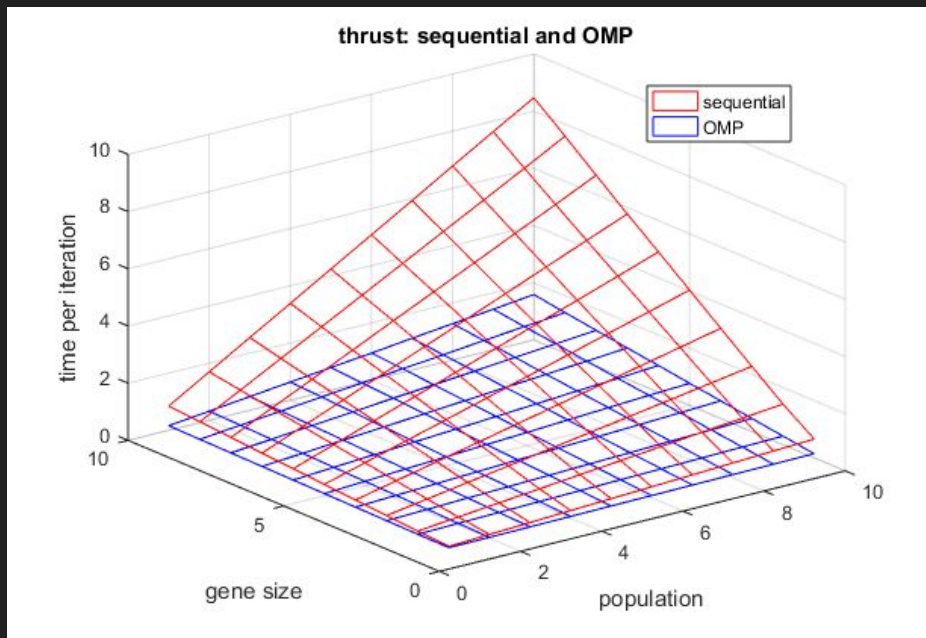


# Onemax OMP

- 大約達到3.2~5.3倍的speedup。除了population和gene size都很小的情況以外，幾乎都有4倍多的speedup。

左圖: time per iteration

右圖: OMP/sequential的時間比值



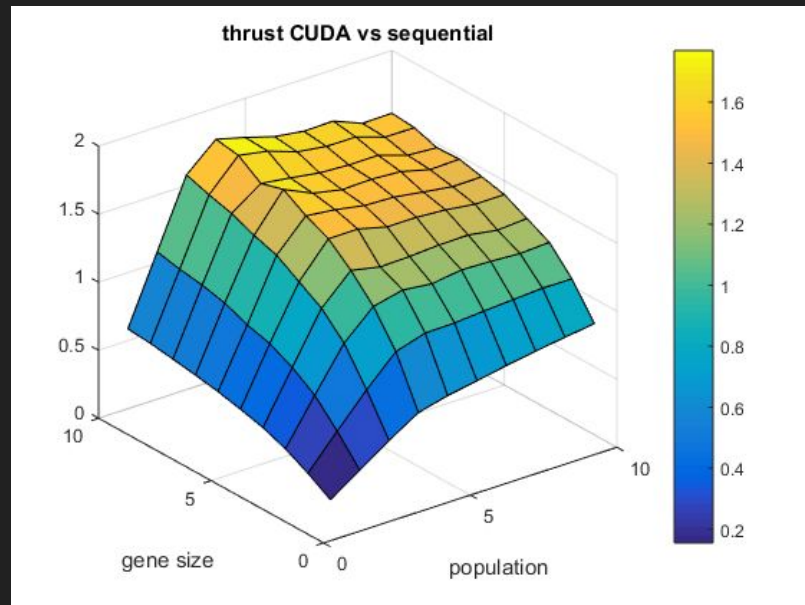
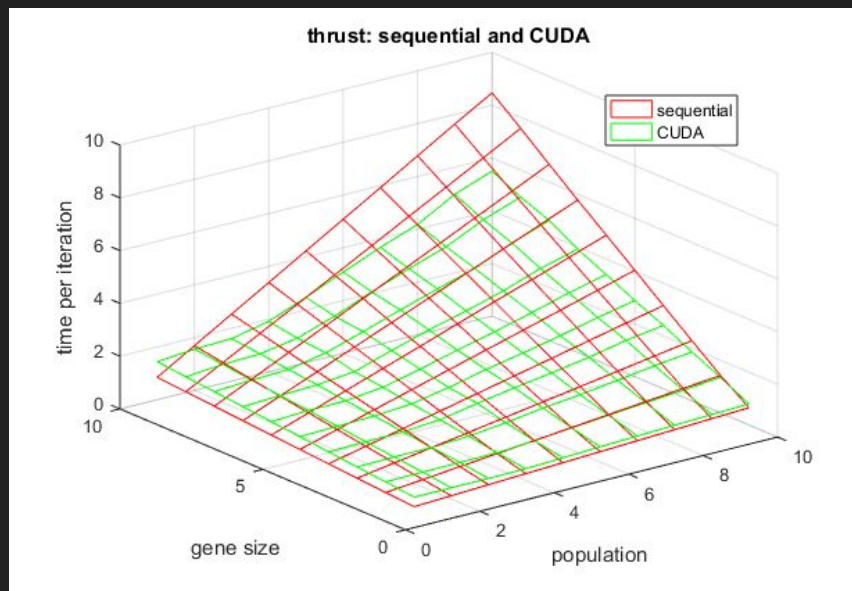


# Onemax CUDA

- 要在population和gene size都大於2000的情形才能有一點點的speedup(最多約1.7倍), 以下時反而比較慢(0.5倍左右)。

左圖: time per iteration

右圖: CUDA/sequential的時間比值

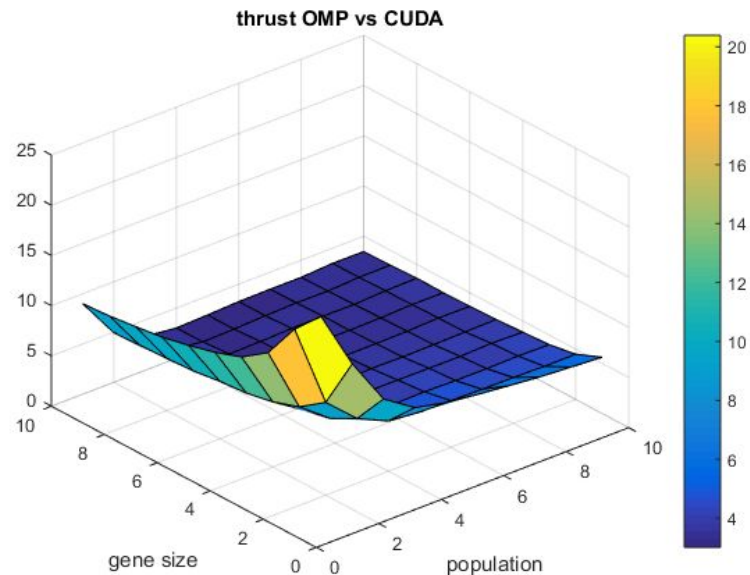
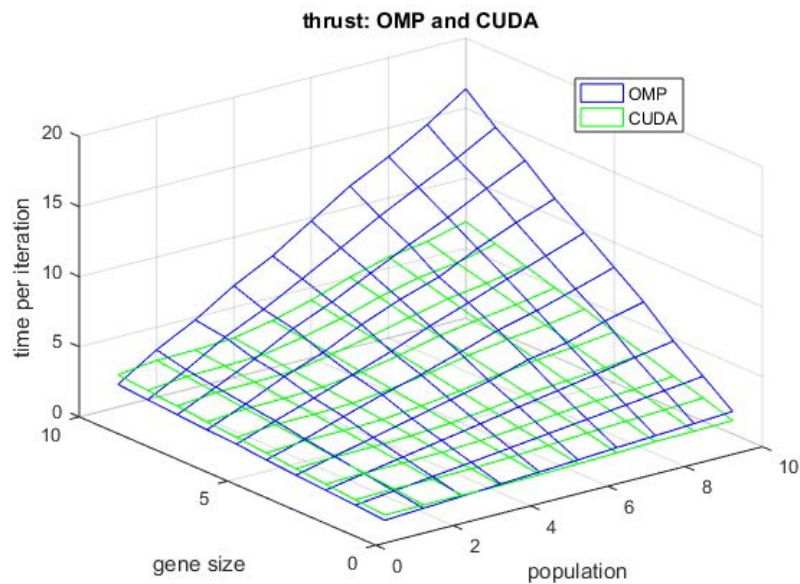


# Onemax OMP vs CUDA

- omp比cuda快了約3倍左右, 若population和gene size都夠小的時候甚至會差到20倍。

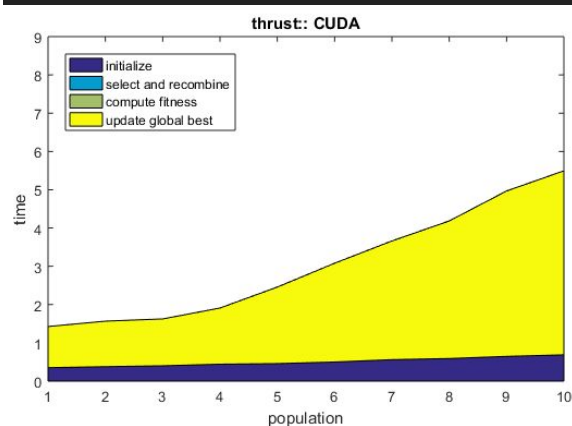
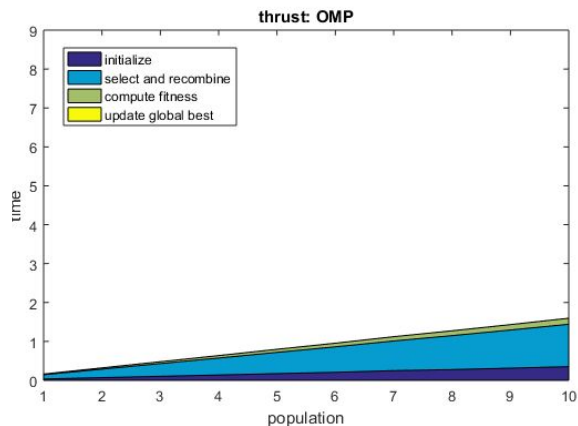
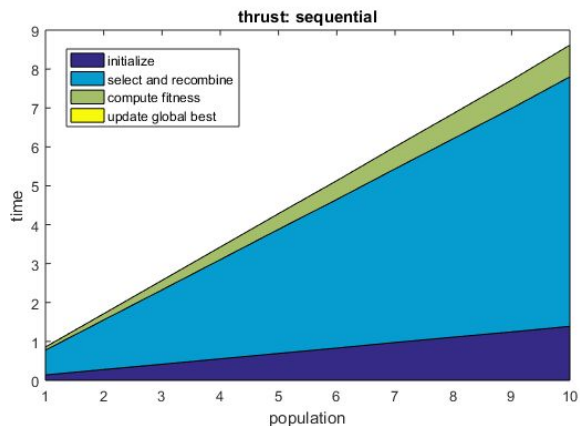
左圖: time per iteration

右圖: OMP/CUDA的時間比值



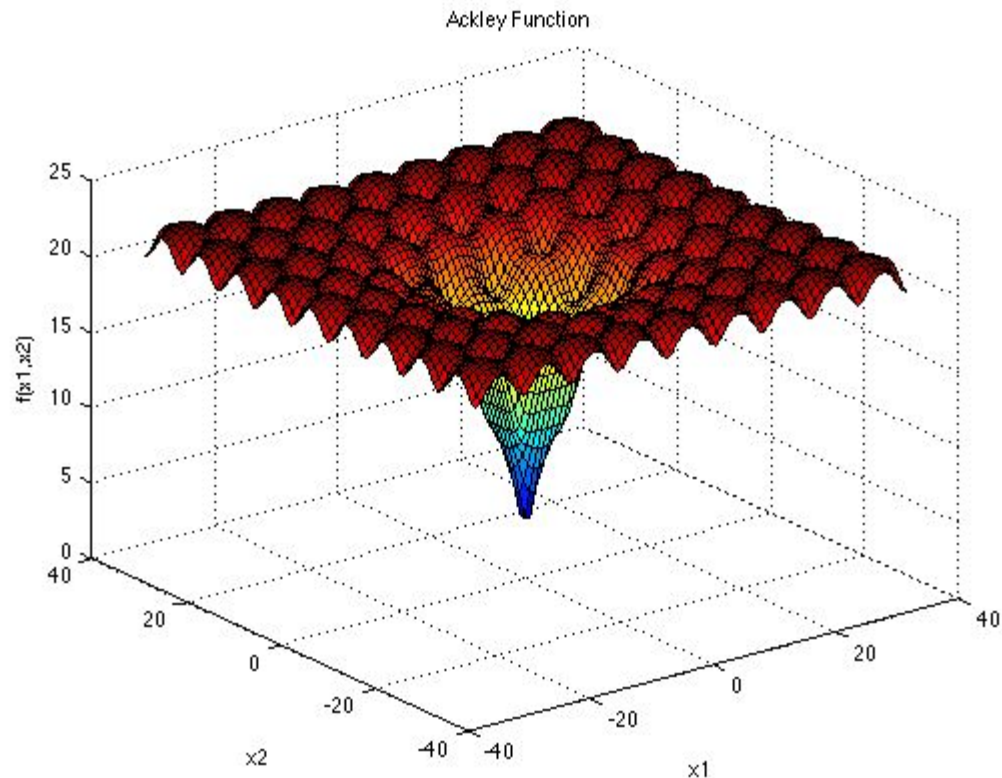
# Onemax 時間分割圖

- 在這個問題，sequential表現得很平穩，
- OMP也有顯著的效果。
- 但CUDA花費了非常大量的時間在update global best上。



# Experiment

- Ackley's Function
  - usually evaluated in  $[-32.768, 32.768]$
  - minimum at  $\{0, \dots, 0\}$



$$f(\mathbf{x}) = -a \exp \left( -b \sqrt{\frac{1}{d} \sum_{i=1}^d x_i^2} \right) - \exp \left( \frac{1}{d} \sum_{i=1}^d \cos(cx_i) \right) + a + \exp(1)$$

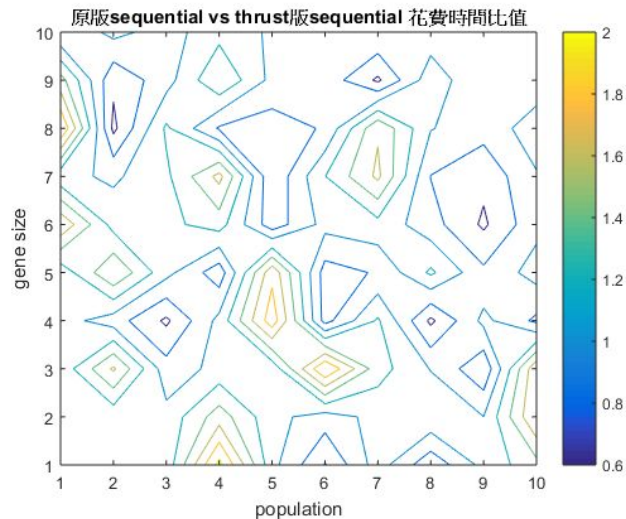
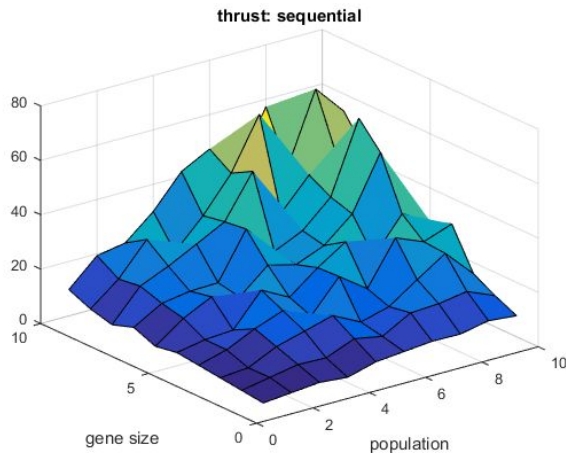
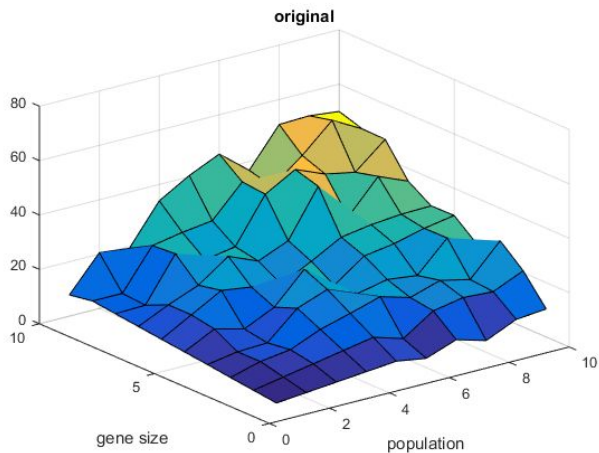
# Ackley sequential

- 因為原版和thrust版的sequential的時間非常不平滑，所以以下speedup將利用contour圖來表示。

左：原版的time per iteration

中：thrust: sequential 的 time per iteration

右(contour圖)：時間比值，thrust版花費時間大致相同，原版稍微快一點。



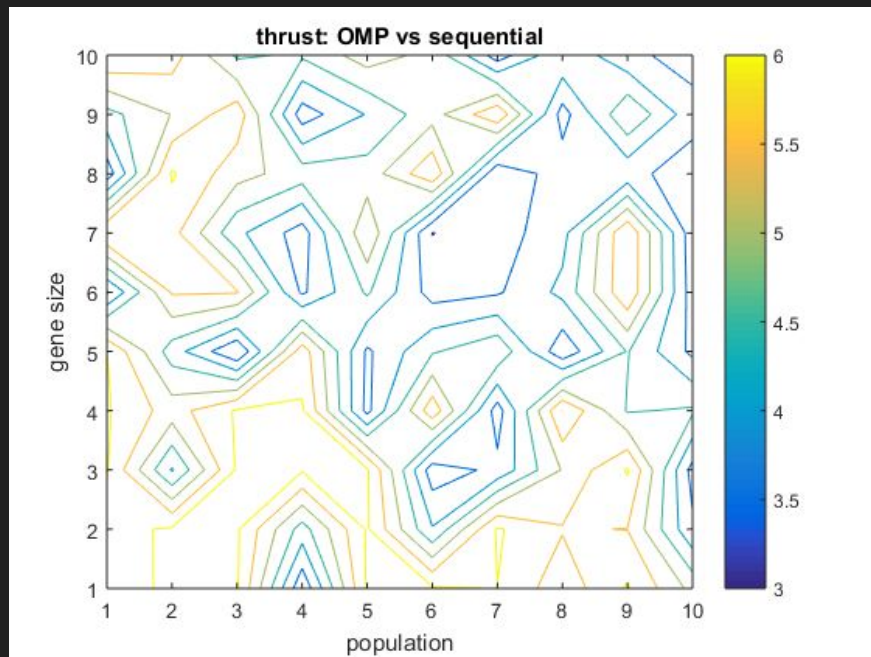
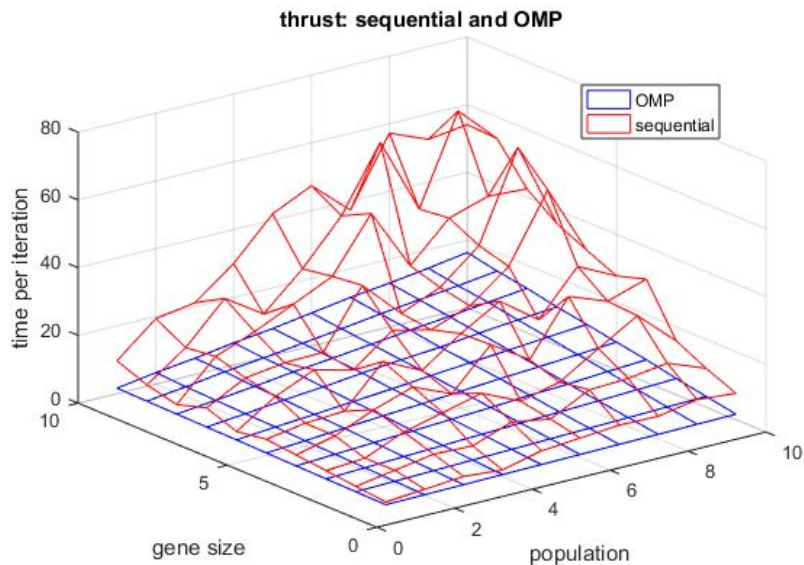


# Ackley OMP

- 大約達到3~6倍的speedup, 但較高的效果主要集中在gene size較小或population較小的情況, 在兩者都較大的情況speedup較不明顯。

左圖: time per iteration

右圖: speedup(OMP/sequential的時間比值)

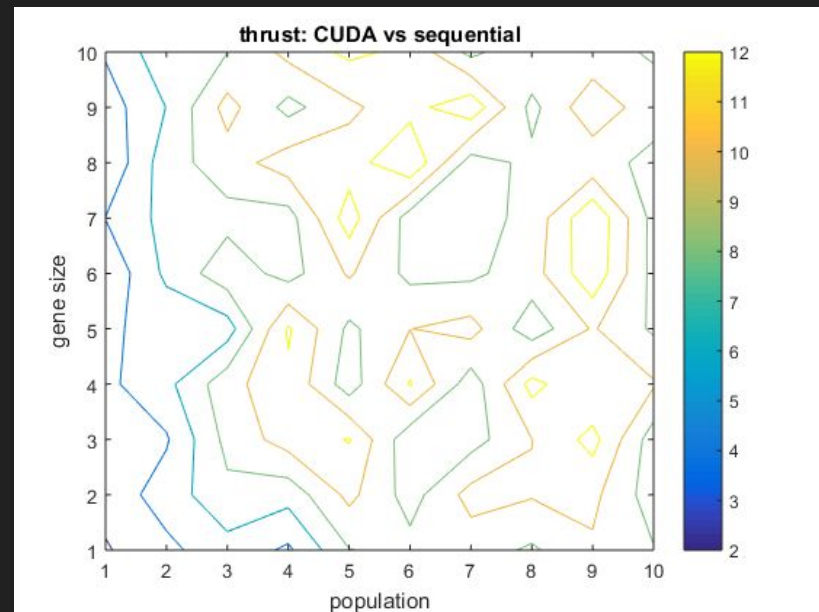
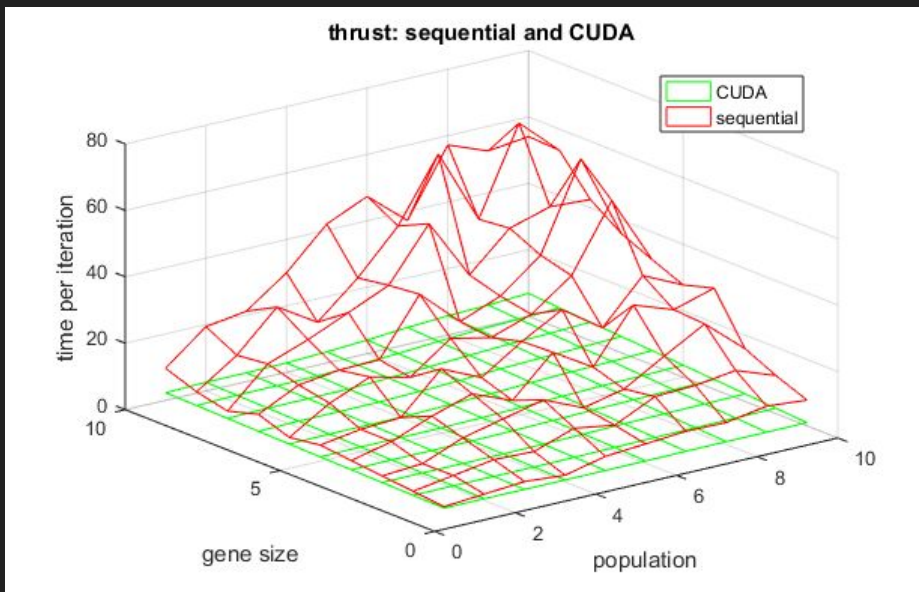


# Ackley CUDA

- 最多能達到12倍的speedup, 主要在population足夠大的時候。

左圖: time per iteration

右圖: speedup(CUDA/sequential 的時間比值)

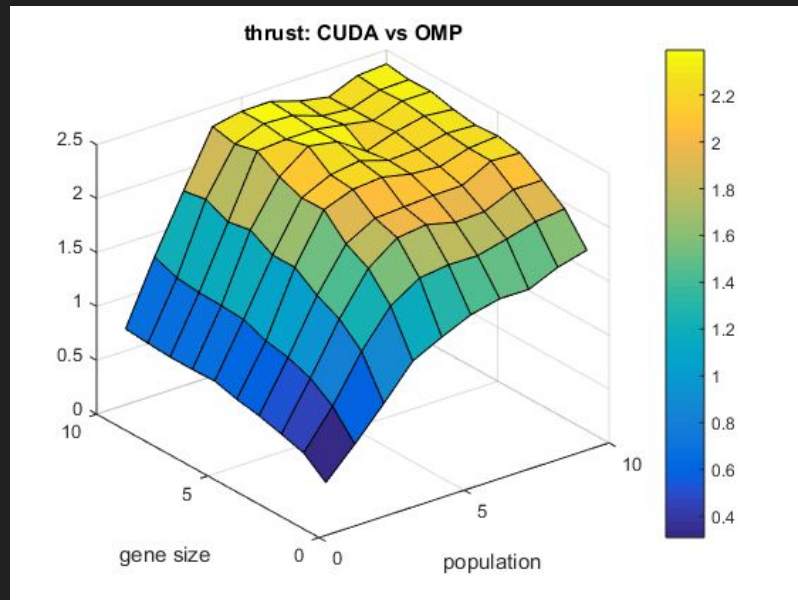
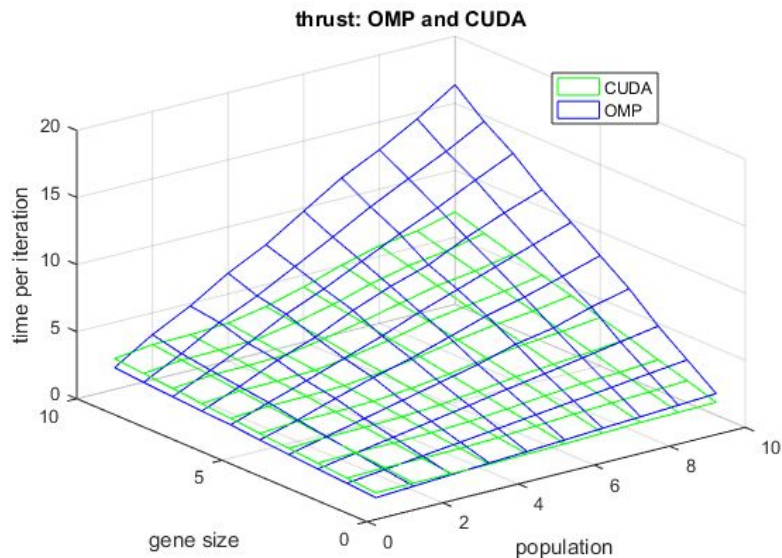


# Ackley OMP vs CUDA

- cuda在gene size和population都過了3000左右之後, 就都贏過omp, 差距大約為2倍。

左圖: time per iteration

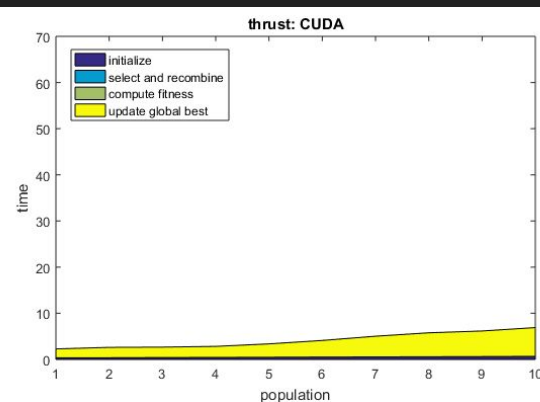
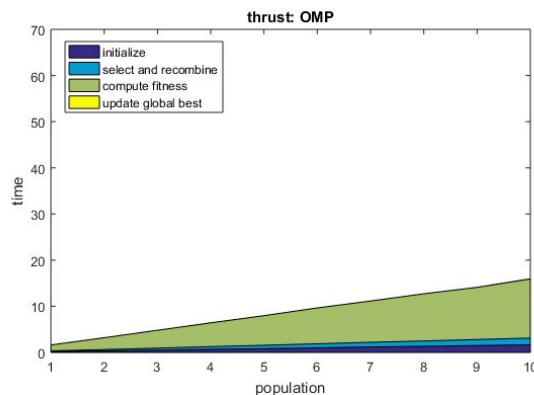
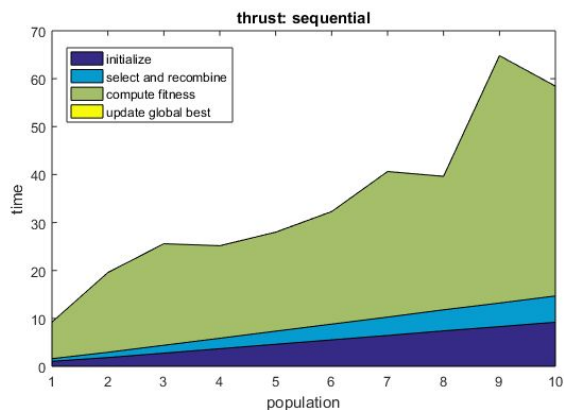
右圖: CUDA/OMP的時間比值





# Ackley 時間分割圖

- sequential的時間不平滑成長的原因在於compute fitness的部分。
- 很奇妙的OMP中卻沒有出現這個問題。
- 雖然CUDA在這個問題有最快的速度，但還是花費了大量的時間在update global best上。



# Conclusion

CUDA在update global best花費大量時間的可能原因：

1. device to host的copy, 因為要交由CPU檢查是否已達到目標的fitness
  - 拿掉copy後時間還是沒有改善
2. 每個iteration都呼叫的關係
  - 只在最後的iteration呼叫也沒有改善
3. min\_element本身不夠好?
  - 沒有確信, 網路上的資料也都顯示沒有這回事
4. array of structure的資料結構不好, 應該使用structure of array
  - 這是目前最可能的答案, 但是很可惜沒有時間可以做這麼重大的修改


# Conclusion

- Use thrust library!
  - no need to consider grid and block
  - no need to consider memory allocate
  - efficient algorithm
  - write once code, work everywhere.
- GPU need a large amount of data to get speed up
- Future work:
  - when size of gene is huge, may need more complicate parallelization.
  - for example: parallel compute the fitness in a grid

# Appendix

# Parallelization - part of implementation

```
void computePopulationFitness() {  
    for (int i = 0; i < population_size; ++i) {  
        problem.computeIndividualFitness(indivs[i]);  
    }  
}
```



```
class ComputeFitness{  
public:  
    P p; //need device pointer!  
    ComputeFitness(P p): p(p){}  
    __host__ __device__  
    void operator()(Individual &i){  
        p.computeIndividualFitness(i);  
    };  
};  
void computePopulationFitness() {  
    thrust::for_each(thrust::device_pointer_cast(indivs), thrust::device_pointer_cast(indivs+population_size), ComputeFitness(problem));  
}
```

# Parallelization - part of implementation

```
void updateBestIndividual() {  
    int best = 0;  
    double best_fitness = indivs[best].fitness;  
    for (int i = 1; i < population_size; ++i) {  
        if (indivs[i].fitness < best_fitness) {  
            best = i;  
            best_fitness = indivs[best].fitness;  
        }  
    }  
    best_indiv = indivs[best];  
}
```



```
void updateBestIndividual() {  
    best_indiv = *(thrust::min_element(thrust::device_pointer_cast(indivs), thrust::device_pointer_cast(indivs + population_size)));  
}
```

Para

```
void selectParent(int *p, int i) {
    int best = i;
    double best_fitness = indivs[best].fitness;
    for (int c = 1; c < selection_size; ++c) {
        int r=rand() % population_size;
        if (indivs[r].fitness<best_fitness) {
            best = r;
            best_fitness = indivs[best].fitness;
        }
    }
    *p = best;
}

void selectAndRecombine() {
    int p1, p2;
    for (int i = 0; i < population_size; i += 2) {
        selectParent(&p1, i);
        selectParent(&p2, i+1);

        if (drand() < crossover_probability) {
            Individual::crossover(indivs[p1], indivs[p2], new_indivs[i], new_indivs[i + 1]);

            problem.normalize(new_indivs[i]);
            problem.normalize(new_indivs[i + 1]);
        }
        if (drand() < mutation_probability) {
            Individual::mutation(new_indivs[i]);
            Individual::mutation(new_indivs[i + 1]);

            problem.normalize(new_indivs[i]);
            problem.normalize(new_indivs[i + 1]);
        }
    }
    swap(indivs, new_indivs);
}
```

```

class SelectAndRecombine{
private:
    P problem;
    Individual *indivs,*new_indivs;
    double pc, pm;
    int population_size;
    int selection_size;
    __host__ __device__
    void selectParent(int *p, int i) { //random!
        int best = i;
        double best_fitness = indivs[best].fitness;
        for (int c = 1; c < selection_size; ++c) {
            int r=indivs[i].rand() % population_size;
            if (indivs[r].fitness<best_fitness) {
                best = r;
                best_fitness = indivs[best].fitness;
            }
        }
        *p = best;
    }
public:
    SelectAndRecombine(P problem, Individual* indivs, Individual* new_indivs, double pc, double pm, int population_size, int selection_size)
        : problem(problem),indivs(indivs),new_indivs(new_indivs), pc(pc), pm(pm),population_size(population_size),selection_size(selection_size)
        ,__host__ __device__
    void operator()(int i){
        i*=2;
        int p1, p2;
        selectParent(&p1, i);
        selectParent(&p2, i+1);

        if (indivs[p1].drand() < pc) {
            Individual::crossover(indivs[p1], indivs[p2], new_indivs[i], new_indivs[i + 1]);
            problem.normalize(new_indivs[i]);
            problem.normalize(new_indivs[i + 1]);
        }
        if (indivs[p2].drand() < pm) {
            Individual::mutation(new_indivs[i]);
            Individual::mutation(new_indivs[i + 1]);

            problem.normalize(new_indivs[i]);
            problem.normalize(new_indivs[i + 1]);
        }
    }
};

void selectAndRecombine() {
    thrust::for_each(thrust::make_counting_iterator(0),
        thrust::make_counting_iterator(population_size/2),
        SelectAndRecombine(problem,indivs,new_indivs,
            crossover_probability, mutation_probability,
            population_size,selection_size));
    std::swap(indivs, new_indivs);
}

```