PP24 Final

Team36

組員: 程詩柔、謝之豫、熊恩伶

Firefly algorithm (visualization)

1. 設定螢火蟲總數(population),空間維度(dimen),訓練次數(max_iter)
2. 將所有螢火蟲隨機放在空間中並隨機給定亮度(fitness)
3. 每個iteration，每隻螢火蟲都要朝附近最亮的螢火蟲前進(位置更新)
4. 更新所有螢火蟲的亮度(fitness更新)

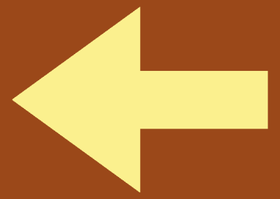
迴圈時間複雜度　max_iter * dimen * population * dimen * population

重要性&潛力:

應用層面非常多，firefly algorithm 不僅可以應用於路徑預測(導航)，還可以優化可再生能源系統(太陽能電池效率最大化)，基因調控網絡建模，藥物設計，影像處理等，希望本次成功優化的版本能夠幫助各種領域中需要使用firefly algorithm的研究。

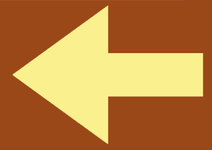
所有嘗試的優化方法

- SIMD
- OpenMP
- Pthread
- MPI
- CUDA



c++

- JAX



python

profiling後發現fun耗時最久，因此會focus在加速這個function

Stats System View

- MPI Event Trace
- NVTX GPU Projection Summary
- NVTX GPU Projection Trace
- NVTX Push/Pop Range Summary
- NVTX Push/Pop Range Trace
- NVTX Range Kernel Summary
- NVTX Range Summary
- NVTX Start/End Range Summary
- Network Devices Congestion
- NvVideo API Summary
- OS Runtime Summary

Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Range
98.9%	329.187 s	50	6.584 s	6.678 s	3.189 s	9.520 s	1.901 s	:fun() calculate fitness
1.1%	3.807 s	6125000	621 ns	320 ns	79 ns	28.061 ms	20.046 µs	:update firefly position
0.0%	403.836 µs	1	403.836 µs	403.836 µs	403.836 µs	403.836 µs	0 ns	:write result file
0.0%	231.226 µs	49	4.718 µs	4.020 µs	1.490 µs	21.599 µs	3.719 µs	:update best fitness
0.0%	134.489 µs	1	134.489 µs	134.489 µs	134.489 µs	134.489 µs	0 ns	:pop initialize
0.0%	4.390 µs	1	4.390 µs	4.390 µs	4.390 µs	4.390 µs	0 ns	:FA() initialize parameter

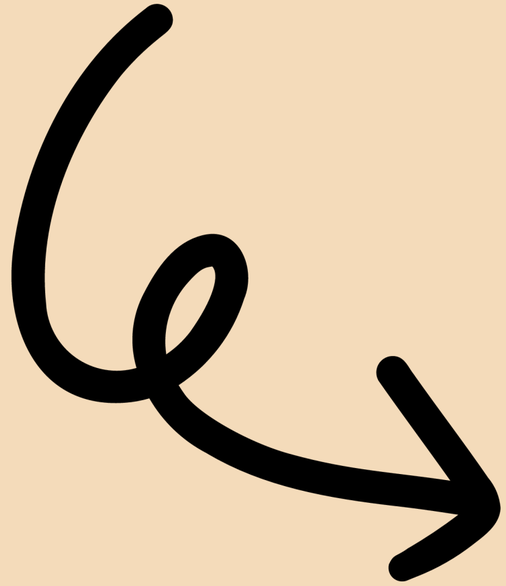
```cpp
vector<double> fun(const vector<vector<double>>& pop) {
    //nvtxRangePushA("fun() calculate fitness");
    vector<double> result;
    for (int i = 0; i < pop.size(); i++) {
        double funsum = 0;
        for (int j = 0; j < D; j++) {
            double x = pop[i][j];
            funsum += x * x - 10 * cos(2 * M_PI * x);
        }
        funsum += 10 * D;
        result.push_back(funsum);
    }
    return result;
    //nvtxRangePop();
}
```

fun()計算時間占99%

SIMD

Vectorization

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < D; j++) {
        double x = pop[i * D + j];
        fitness[i] += x * x - 10 * cos(2 * M_PI * x);
    }
}
```



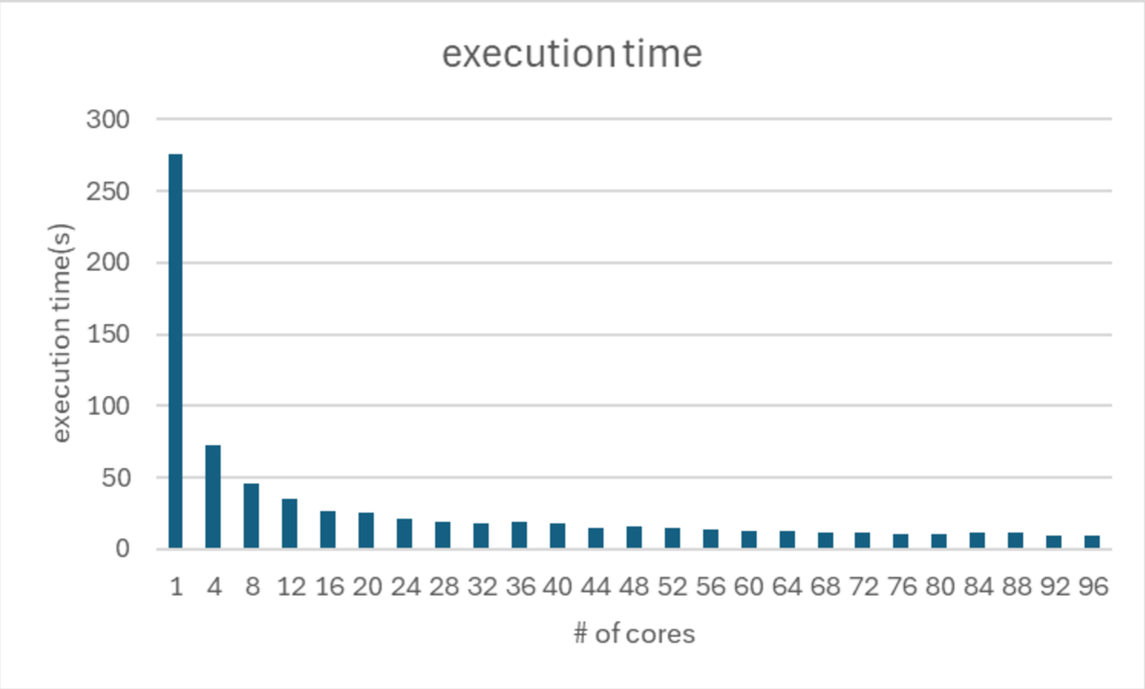
```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < D; j += 8) {
        int remaining = D - j;
        __mmask8 mask = (remaining >= 8) ? 0xFF : (1 << remaining) - 1;
        __m512d x = _mm512_maskz_loadu_pd(mask, &pop[i * D + j]);
        __m512d x_squared = _mm512_mul_pd(x, x);
        __m512d cos_term = _mm512_cos_pd(_mm512_mul_pd(vec_two_pi, x));
        __m512d result = _mm512_sub_pd(x_squared, _mm512_mul_pd(vec_ten, cos_term));
        fitness[i] += _mm512_mask_reduce_add_pd(mask, result);
    }
}
```

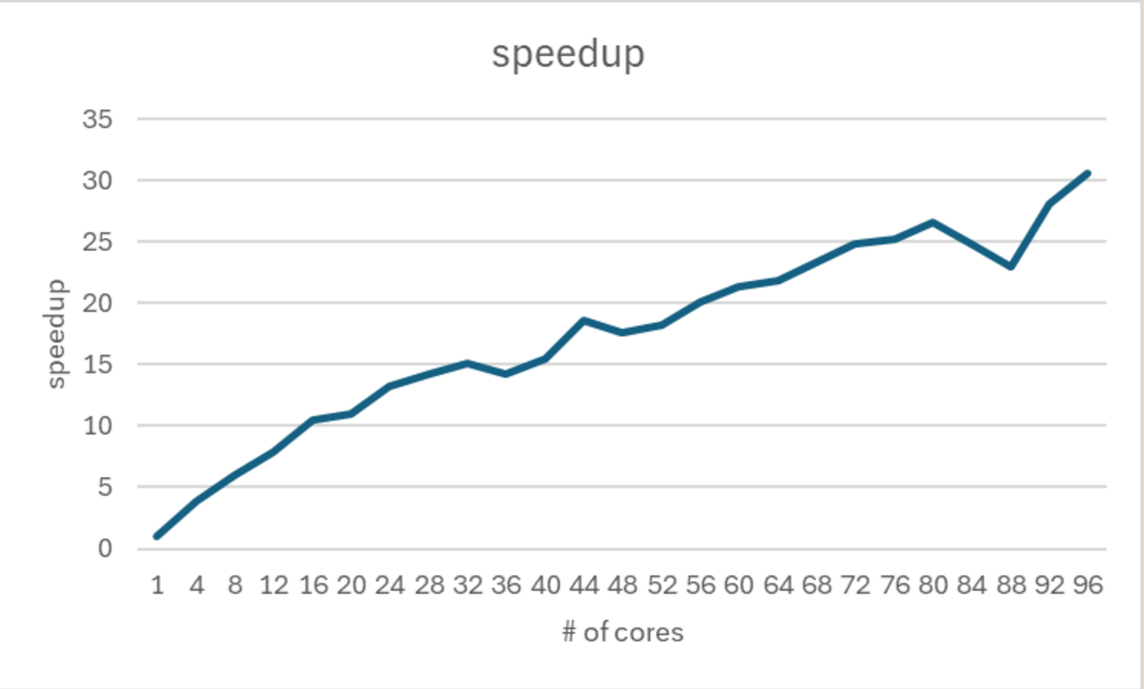
OpenMP

OpenMP

```
void fun2() {
    fitness.assign(N, 10 * D);
    #pragma omp parallel for collapse(2) num_threads(num_threads)
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < D; j += 8) {
            int remaining = D - j;
            __mmask8 mask = (remaining >= 8) ? 0xFF : (1 << remaining) - 1;
            __m512d x = _mm512_maskz_loadu_pd(mask, &pop[i * D + j]);
            __m512d x_squared = _mm512_mul_pd(x, x);
            __m512d cos_term = _mm512_cos_pd(_mm512_mul_pd(vec_two_pi, x));
            __m512d result = _mm512_sub_pd(x_squared, _mm512_mul_pd(vec_ten, cos_term));
            #pragma omp atomic
            fitness[i] += _mm512_mask_reduce_add_pd(mask, result);
        }
    }
}
```

Scalability





Pthread

Pthread

直接在每次呼叫fun時create threads並計算會浪費很多時間
在重複create以及join threads，因此我在一開始就先將
threads create好，等全部計算結束後再一起join

slave threads:

當cur_i小於N時表示還有工作等待執行，便透過fun3計算，算完後將finishes的數量加上剛剛算完的數量，當done是true時表示所有計算工作都結束，可以return了

```c
void *find_job(void *args) {
    int t = *(int *)args;
    int id;
    while (1) {
        if (done)
            break;
        pthread_mutex_lock(&mutex1);
        if (cur_i < N) {
            id = cur_i;
            cur_i += per_job;
        } else
            id = -1;
        pthread_mutex_unlock(&mutex1);
        if (id != -1) {
            int endd = id + per_job;
            if (endd > N)
                endd = N;
            fun3(id, endd);
            pthread_mutex_lock(&mutex2);
            finishes += endd - id;
            pthread_mutex_unlock(&mutex2);
        }
    }
    return NULL;
}
```

main thread:

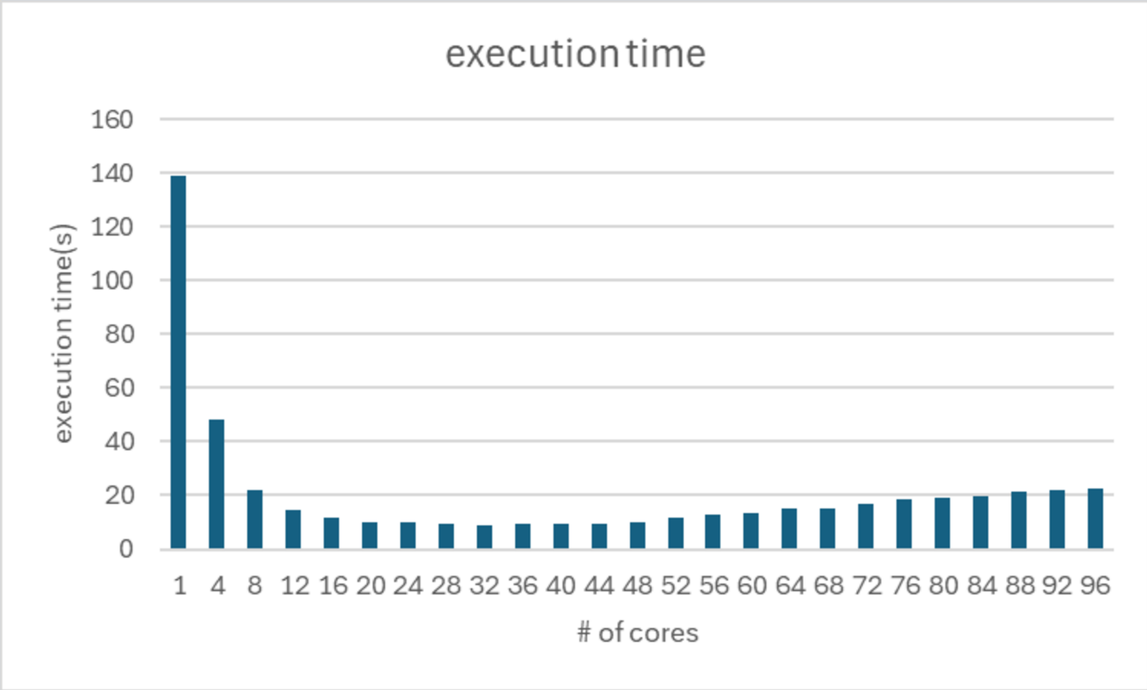
呼叫fun時將finishes及cur_i設成0讓slave threads開始計算，全部計算完後將done設為1，讓slave threads知道可以break了

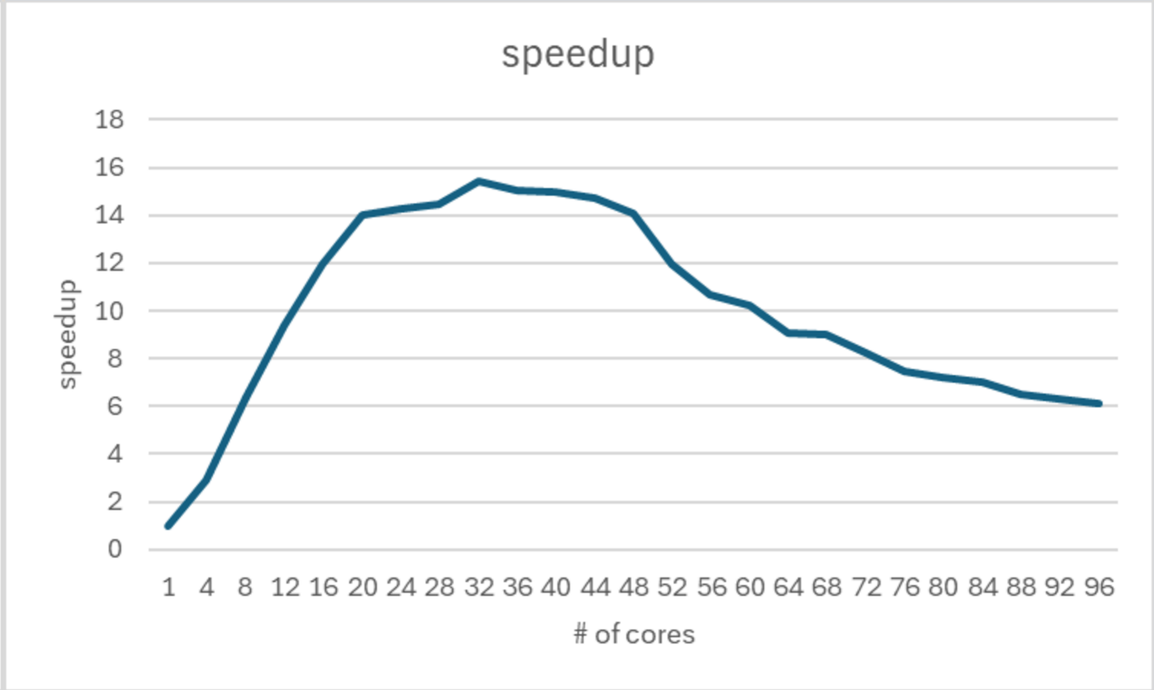
```
void fun() {
    // cout << "start fun\n";
    if (num_threads == 0) {
        fun3(0, N);
        return;
    }

    finishes = cur_i = 0;
    volatile bool ok = 0;
    while (!ok) {
        if (finishes >= N)
            ok = 1;
    }
    // cout << "end fun \n";
}
```

Scalability

當用更多cpu時，可能因為卡在find_job中的lock導致速度反而變慢





MPI

MPI

與pthread做法類似，rank 0當main，其餘rank當slave負責計算

slave ranks:

slave ranks會等在while loop中，當收到main rank發的flag時，若flag==-1表示計算全部結束，可以離開迴圈了，反之則進到fa.fun中進行計算。

在fa.fun中，會先等main rank將它需要的資料傳來，接著開始計算結果，最後再將結果傳回給main rank

```
int flag = 0;
while (1) {
    // MPI_Bcast(&flag, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Request request;
    MPI_Ibcast(&flag, 1, MPI_INT, 0, MPI_COMM_WORLD, &request);
    MPI_Wait(&request, MPI_STATUS_IGNORE);
    if (flag == -1)
        break;
    nvtxRangePush("calculating fun");
    fa.fun(pop, fitness, size, rank);
    nvtxRangePop();
}
```

```
int start = st[rank];
int end = en[rank];
MPI_Irecv(pop + start * D, (end - start) * D, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &req_recv[0]);
MPI_Wait(&req_recv[0], MPI_STATUS_IGNORE);
for (int i = start; i < end; ++i) {
    result[i] = 10 * D;
    for (int j = 0; j < D; j += 8) {
        int remaining = D - j;
        __mmask8 mask = (remaining >= 8) ? 0xFF : (1 << remaining) - 1;
        __m512d x = _mm512_maskz_loadu_pd(mask, &pop[i * D + j]);
        __m512d x_squared = _mm512_mul_pd(x, x);
        __m512d cos_term = _mm512_cos_pd(_mm512_mul_pd(vec_two_pi, x));
        __m512d res = _mm512_sub_pd(x_squared, _mm512_mul_pd(vec_ten, cos_term));
        result[i] += _mm512_mask_reduce_add_pd(mask, res);
    }
}
MPI_Send(result + start, end - start, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
```

main rank:

當呼叫fa.fun時，負責將pop中的資料送給相對應的ranks，再等待他們將結果送回

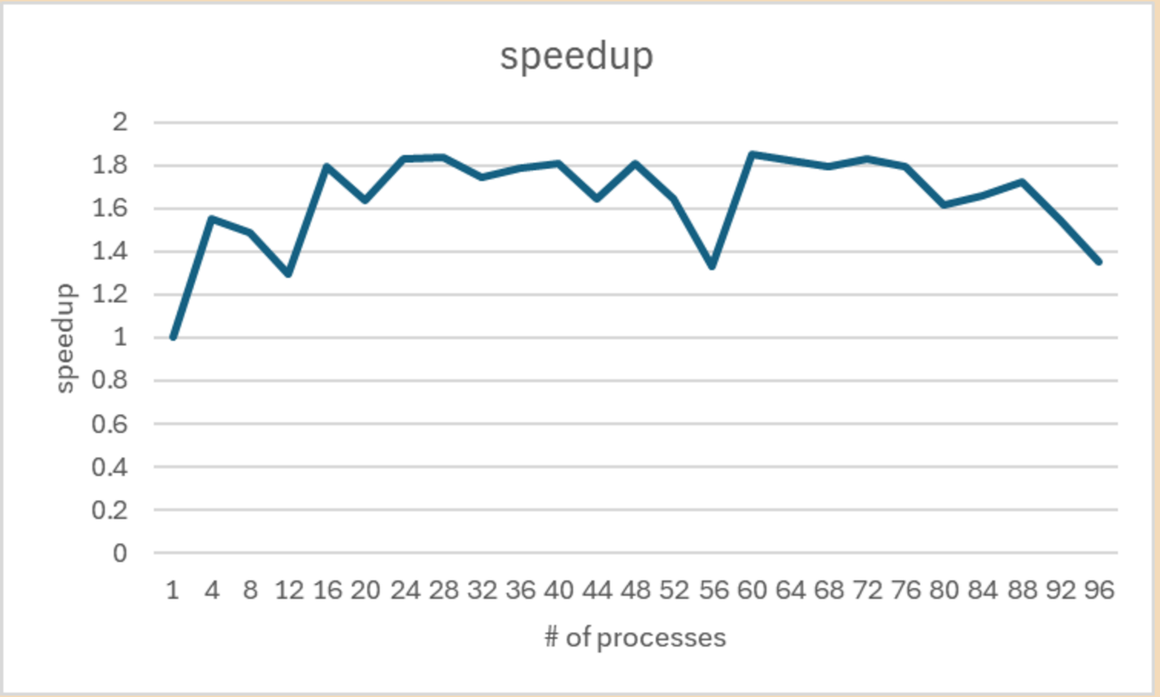
```
MPI_Request request;
MPI_Ibcast(&rank, 1, MPI_INT, 0, MPI_COMM_WORLD, &request);
for (int i = 1; i < size; ++i) {
    int start = st[i];
    int end = en[i];
    MPI_Isend(pop + start * D, (end - start) * D, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &req_send[i - 1]);
}
MPI_Wait(&request, MPI_STATUS_IGNORE);
MPI_Waitall(size - 1, req_send.data(), MPI_STATUSES_IGNORE);
for (int i = 1; i < size; ++i) {
    int start = st[i];
    int end = en[i];
    MPI_Irecv(result + start, end - start, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &req_recv[i - 1]);
}
MPI_Waitall(size - 1, req_recv.data(), MPI_STATUSES_IGNORE);
```

Profiling

problem size dimen:256 population:256 max_iter 5 (with 16 ranks)
以其中一個slave thread的結果可以看出，實際用來計算的時間只有所有fun執行時間的28.9%，其餘的部分都是資料的傳輸，這也導致他的scalability不是太好

Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Range
62.3%	378.019 ms	36533	10.347 µs	6.239 µs	3.480 µs	729.159 µs	14.519 µs	:wait for data
28.9%	175.579 ms	36533	4.806 µs	4.837 µs	4.433 µs	346.105 µs	2.433 µs	:computing
8.7%	52.994 ms	36533	1.450 µs	1.248 µs	651 ns	373.311 µs	3.581 µs	:send result back

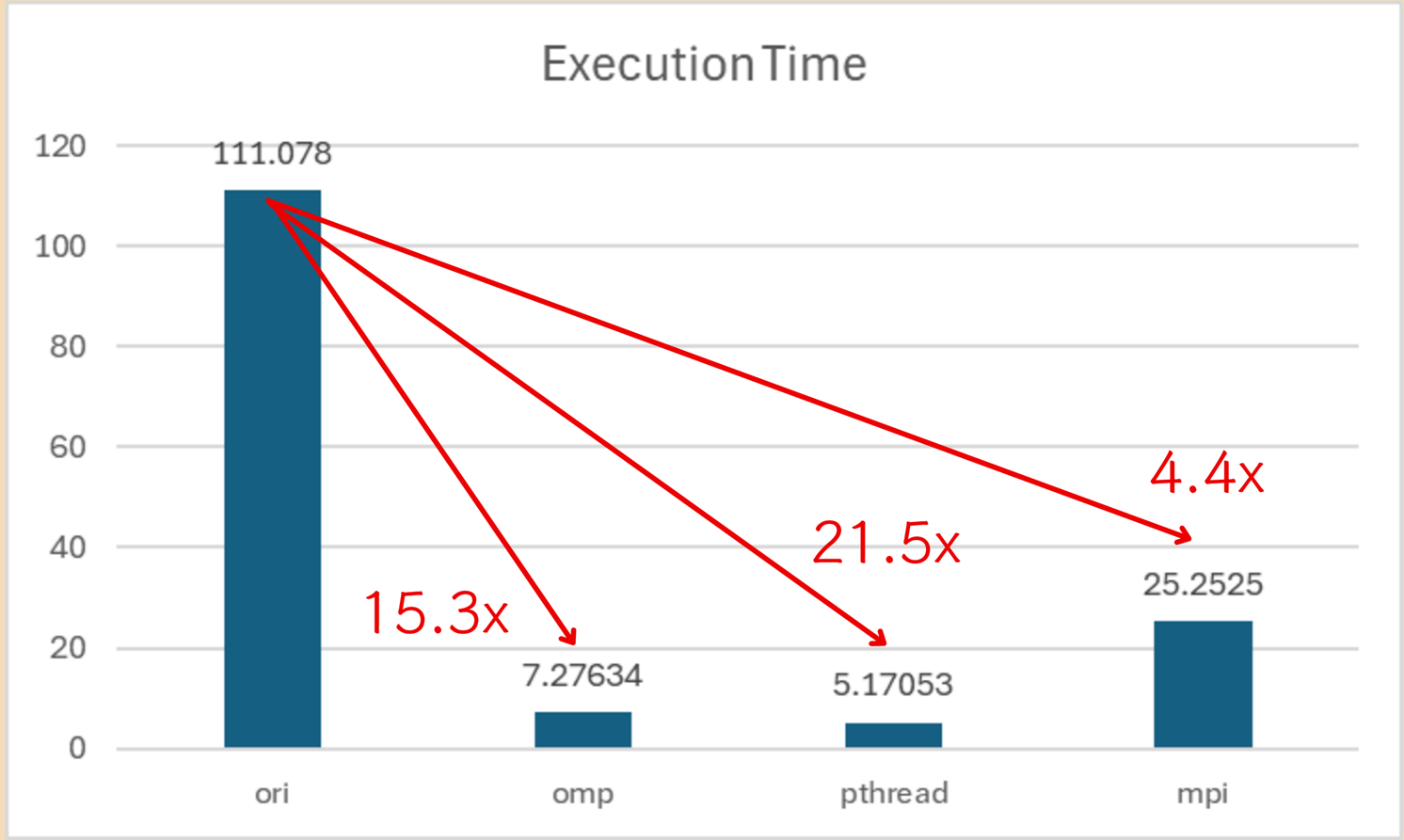
Scalability



Compares

problem size dimen:1024 population:512 max_iter 8

on QCT INTEL(R) XEON(R) PLATINUM 8568Y+



CUDA

CUDA-version1

bottleneck: fun()fitness亮度更新計算更新耗時最久

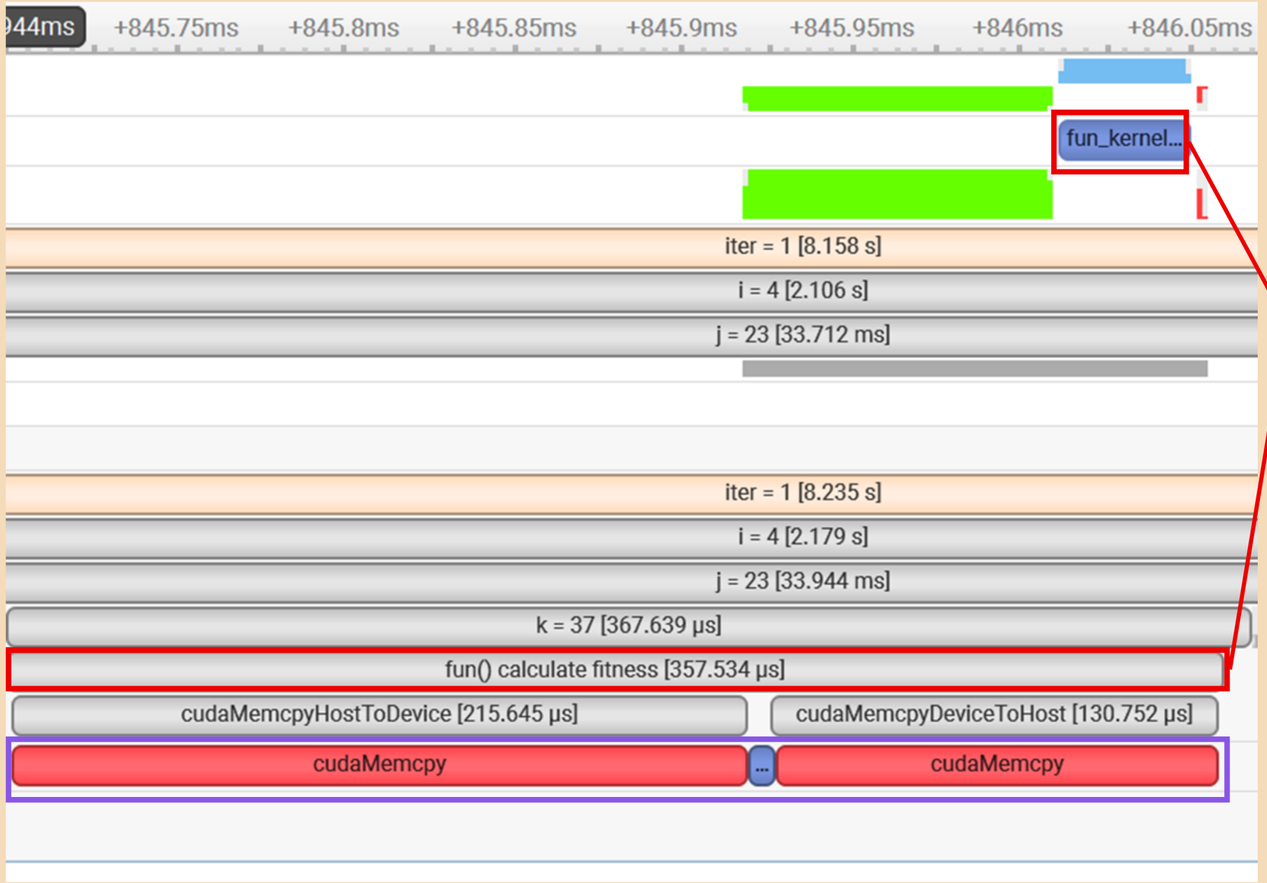
strategy:

1. 將fun()寫成kernel放上gpu做運算
2. 使用1024 threads population個數的blocks做平行運算
3. 使用warp-reduce做fitness summation

但是反而變慢了！

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < D; j++) {
        double x = pop[i * D + j];
        fitness[i] += x * x - 10 * cos(2 * M_PI * x);
    }
}
```

CUDA-version1



bottleneck:
kernel計算時間佔10%
剩餘90%均為Memcpy的時間

strategy:
透過將fun_kernel計算前後也放上gpu做運算節省Memcpy時間

CUDA-version2

```
if (fitness[i] > fitness[k]) {
    r_distance += pow(pop[i * fa.D + j] - pop[k * fa.D + j], 2);
    double Beta = fa.B * exp(-fa.G * r_distance);
    double xnew = pop[i * fa.D + j] + Beta * (pop[k * fa.D + j] - pop[i * fa.D + j]) + steps;

    xnew = min(max(xnew, fa.Lb[0]), fa.Ub[0]);
    pop[i * fa.D + j] = xnew;
```

update_pop_kernel

```
    // Update fitness after position update
    fitness = fa.fun(pop);
```

fun_kernel

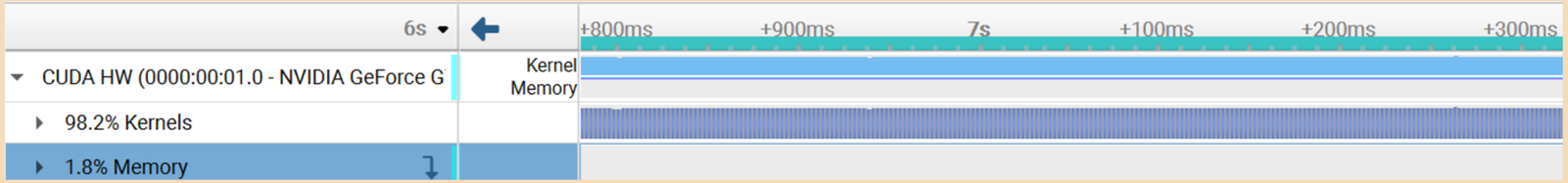
```
    auto best_iter = min_element(fitness.begin(), fitness.end()); //取得min fitness
    best_ = *best_iter; //取得min fitness
    int arr_ = distance(fitness.begin(), best_iter); //取得min fitness index位置

    for (int j = 0; j < fa.D; j++) {
        best_para_[j] = pop[arr_ * fa.D + j]; //將min_fitness整個dimention位置存入best_para_
    }
}
```

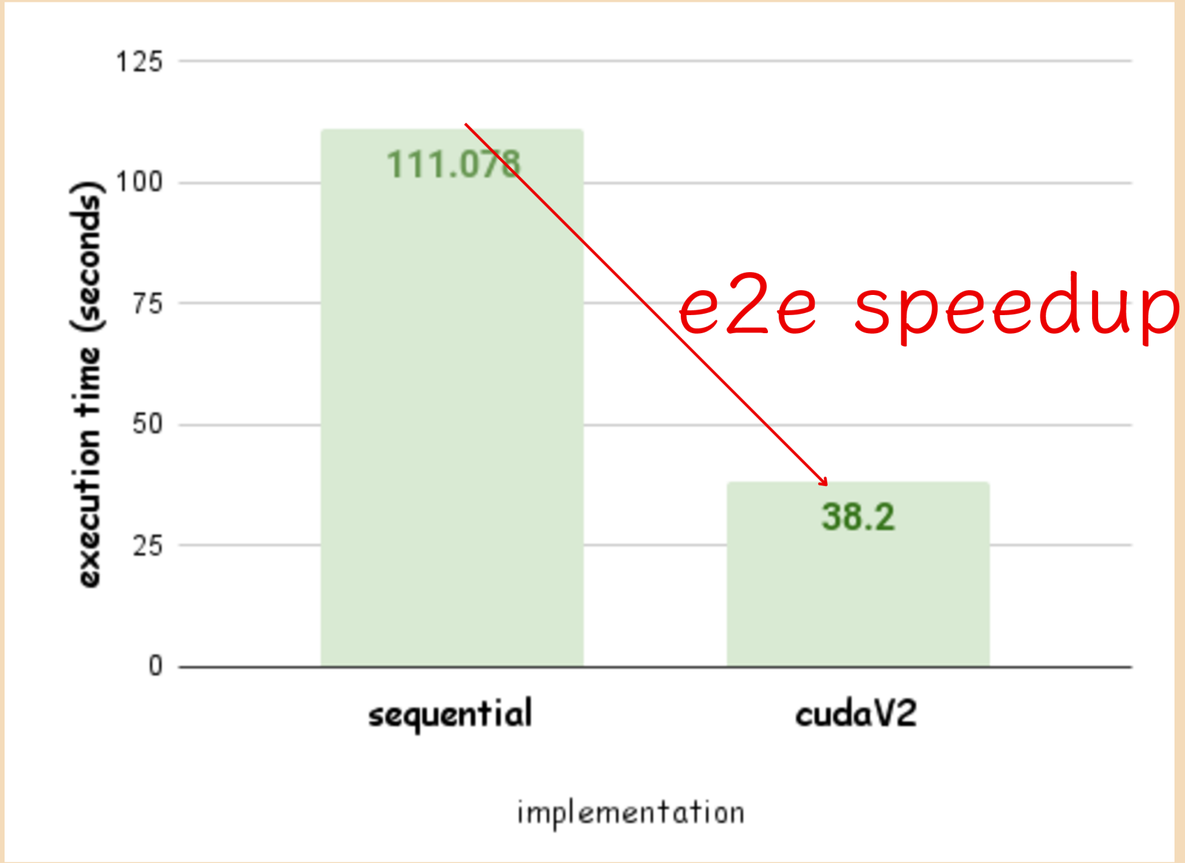
update_best_kernel

CUDA-version2

- 解決頻繁memcpy
- gpu utilization > 90%



problem size:
dimen:1024
population:512
max_iter 8

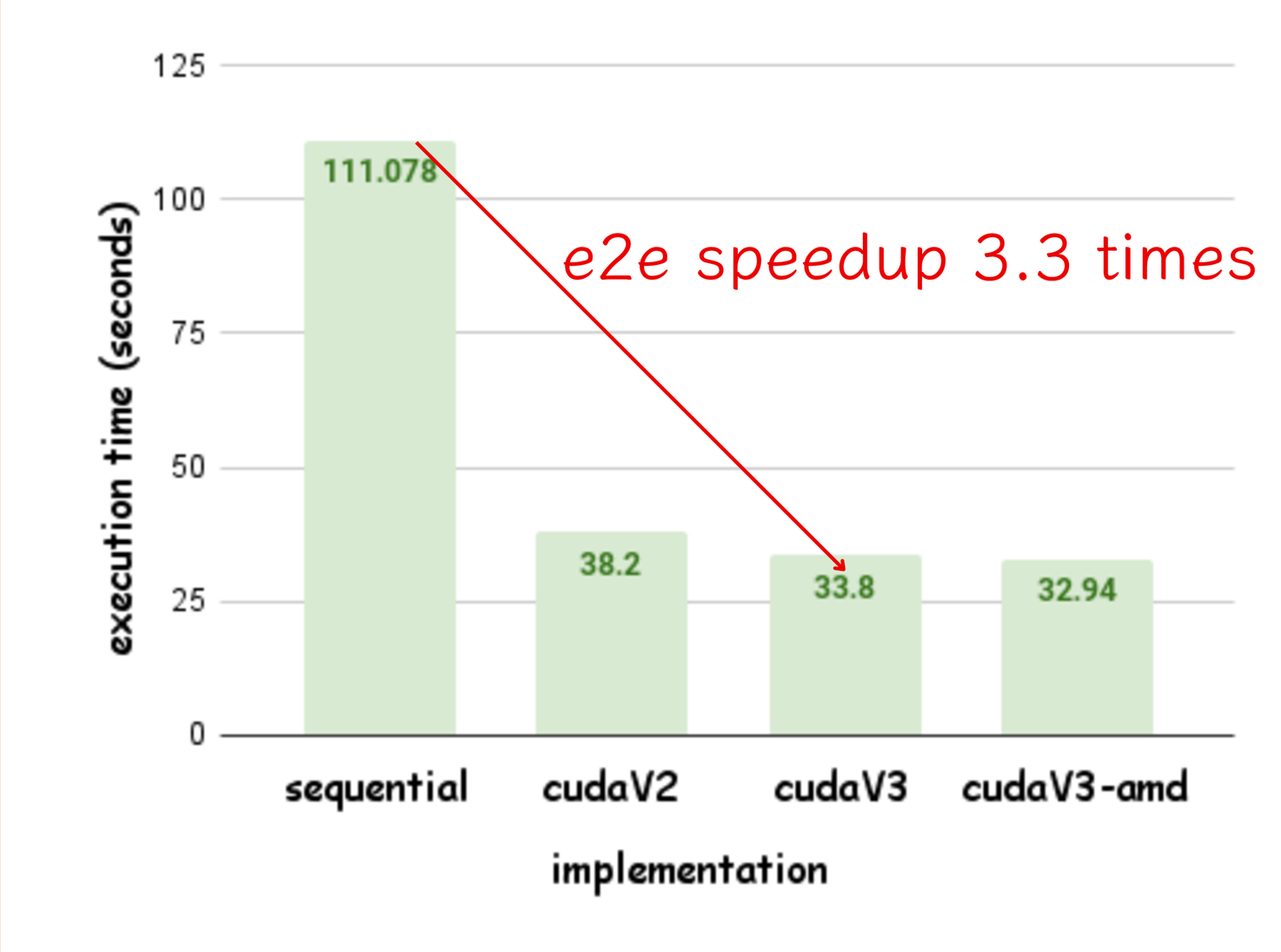


e2e speedup 2.9 times

CUDA-version3

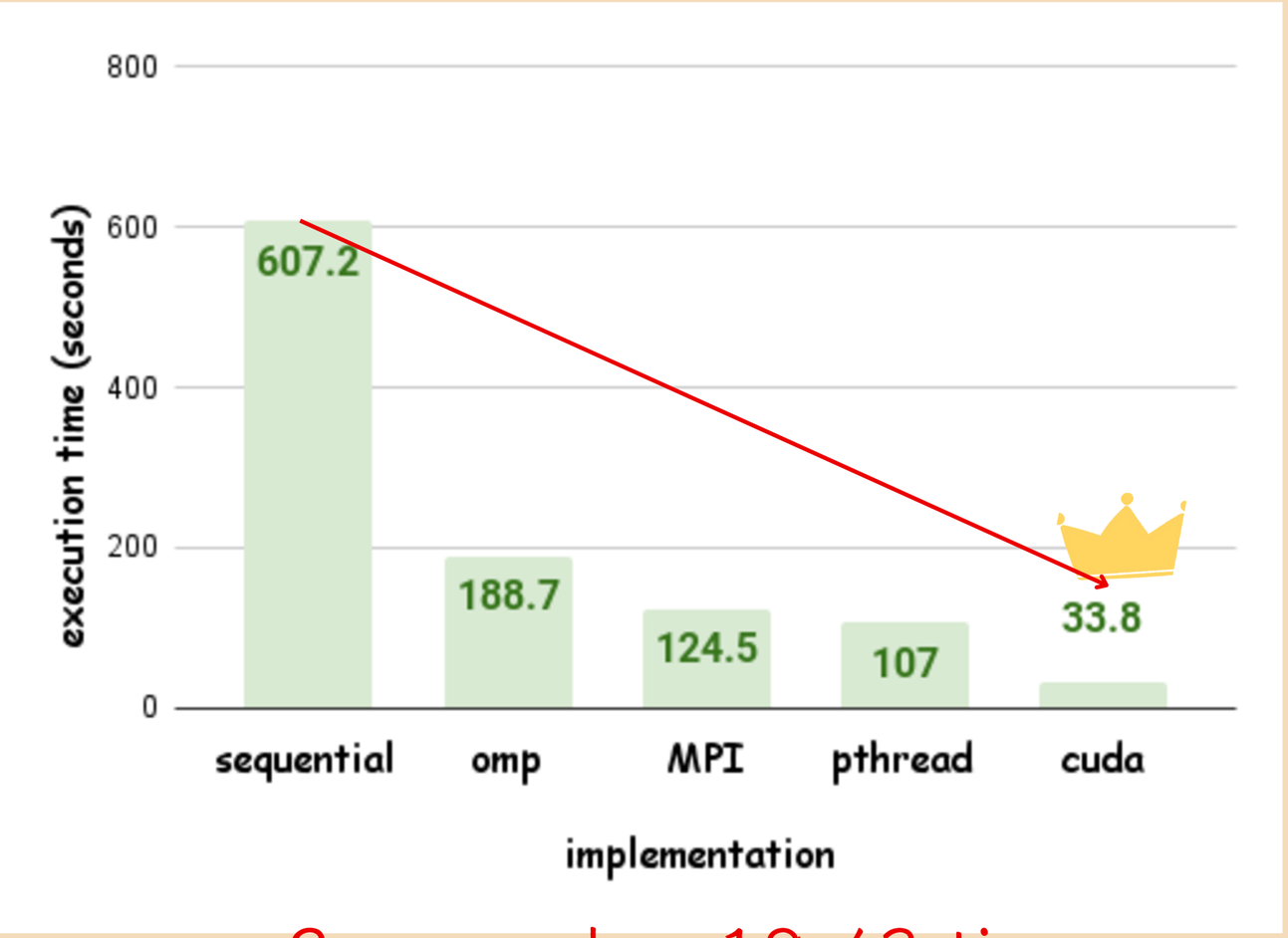
1. 使用((N + 256 - 1) / 256) blocks 256 threads平行計算 update_best_kernel
2. 在fun()實作coalesced memory access

problem size:
dimen:1024
population:512
max_iter 8

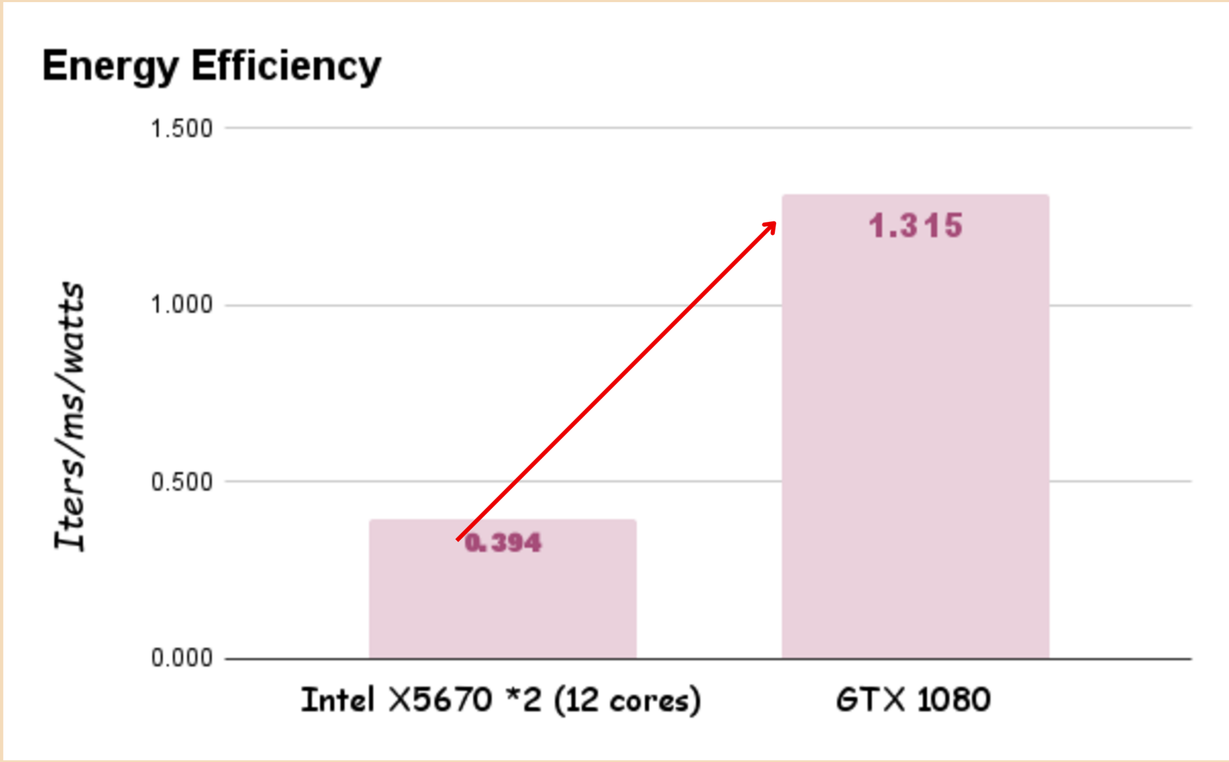


GTX1080*1 vs Intel X5670 *2 (12 cores)

problem size: dimen:1024 population:512 max_iter 8



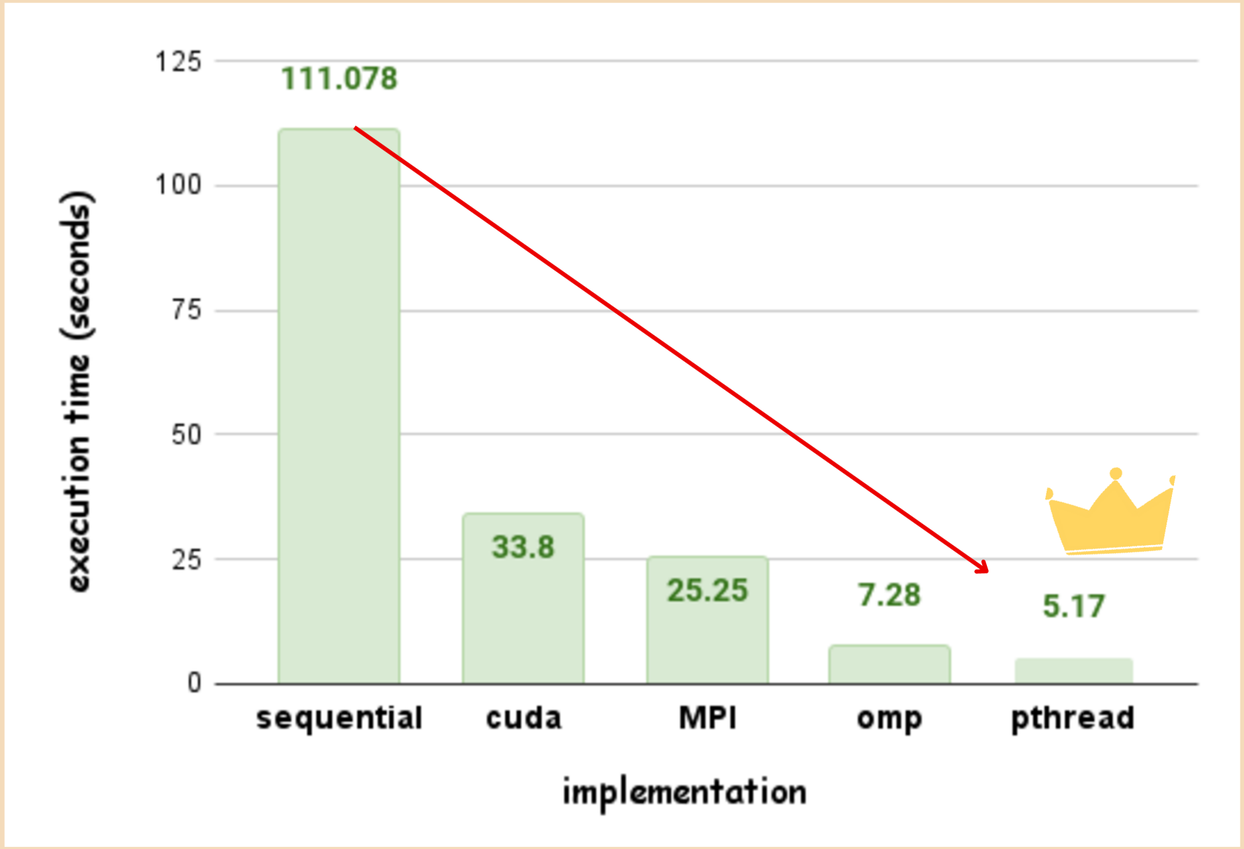
e2e speedup 18.43 times

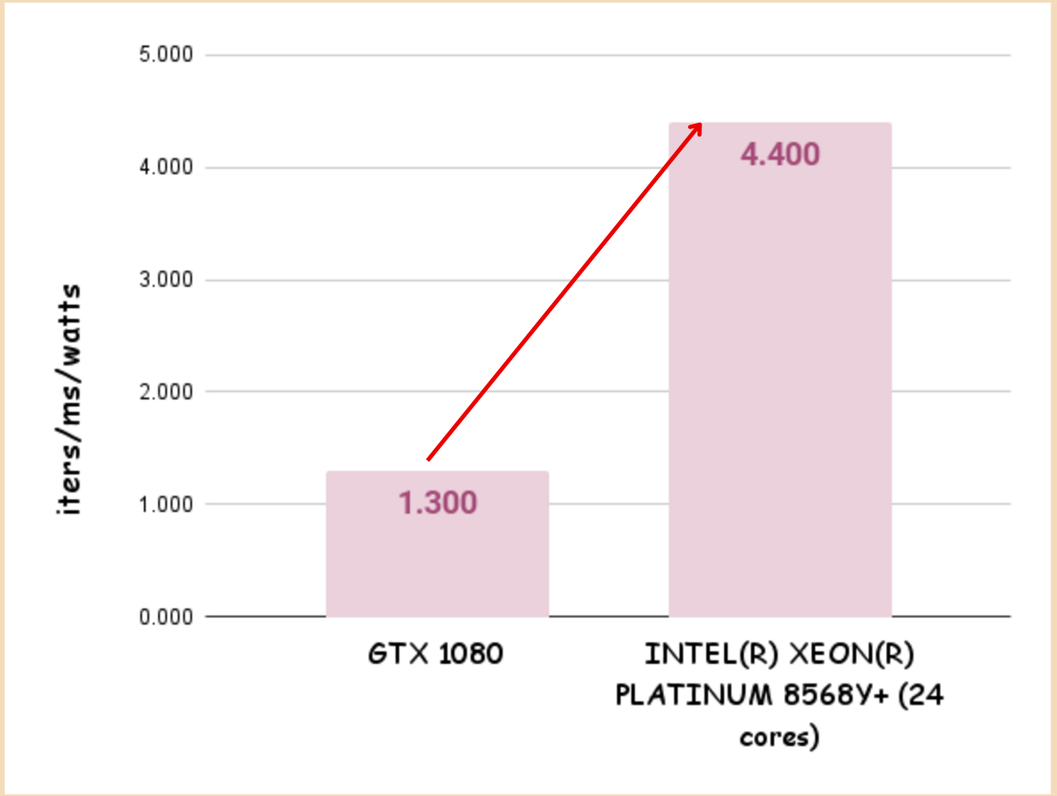


Energy efficiency improve 3.3 times

GTX1080*1 vs INTEL(R) XEON(R) PLATINUM 8568Y+ (24 cores)

problem size: dimen:1024 population:512 max_iter 8





e2e speedup 21.65 times

Energy efficiency improve 3.4 times

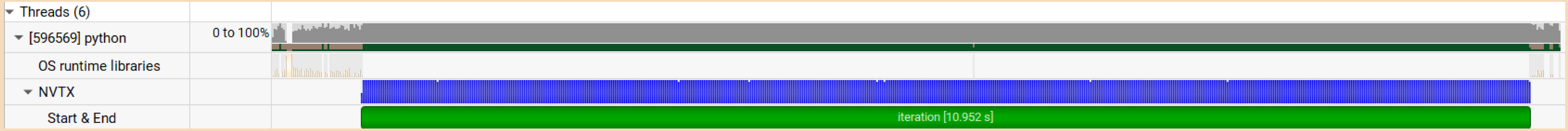
JAX

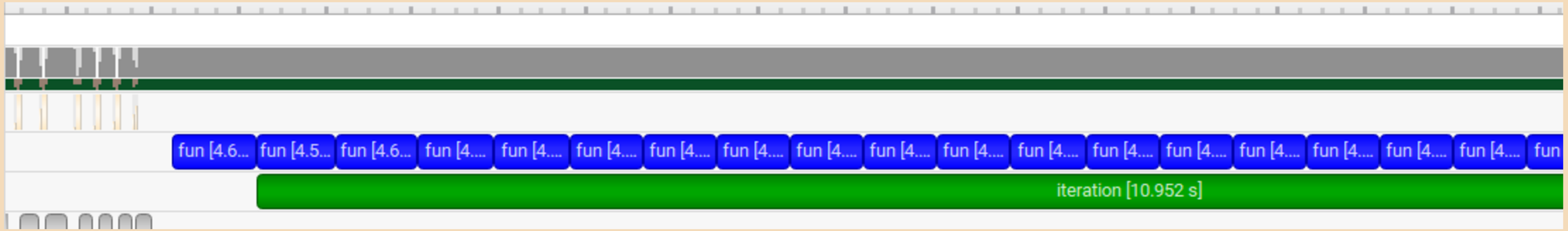
About JAX

- A library for array-oriented numerical computation
- Provides a unified NumPy-like interface
- Features built-in Just-In-Time (JIT) compilation
    - enables operations to execute on CPU/GPU/TPU

Since this algorithm deals with position information (shape=dimension) of each firefly, we think JAX is suitable to optimize our python code!

Python Code Observation





During the whole iteration, most of the time are spent at frequently called ‘fun’ function, which calculates the fitness of each firefly

Python Code Observation

With the observation of the original code, we found some places to optimize:

```python
def fun(self, pop):
    X = np.array(pop)
    funsum = 0
    for i in range(self.D):
        x = X[:,i]
        funsum += x**2 - 10*np.cos(2*np.pi*x)
    funsum += 10*self.D
    return list(funsum)
```

May take advantage of JAX

Python Code Observation

```
r_distance = 0
it = 0
while it < fa.it:
    random_steps = np.random.uniform(-0.5, 0.5, size=(fa.N, fa.D))
    rng = nvtx.start_range(message="iteration", color="green")
    for i in range(fa.N):
        for j in range(fa.D):
            steps = fa.A * random_steps[i][j] * abs(fa.Ub[0]-fa.Lb[0])
            for k in range(fa.N):
                if fitness[i] > fitness[k]:
                    # dist btwn firefly[i] and all other fireflies
                    r_distance += (pop[i][j] - pop[k][j])**2

                    Beta = fa.B*math.e**(-(fa.G*r_distance))
                    xnew = pop[i][j] + Beta*(pop[k][j] - pop[i][j]) + steps
                    if xnew > fa.Ub[0]:
                        xnew = fa.Ub[0]
                    elif xnew < fa.Lb[0]:
                        xnew = fa.Lb[0]
                    pop[i][j] = xnew
                    fitness = fa.fun(pop)
                    best_ = min(fitness)
                    arr_ = fitness.index(best_)
                    best_para_ = pop[arr_]
    nvtx.end_range(rng)

    best_list.append(best_)
    best_para_list.append(best_para_)
    it+=1
```

‘fun’ is called whenever a dimension of a firefly is updated

=> Inefficient

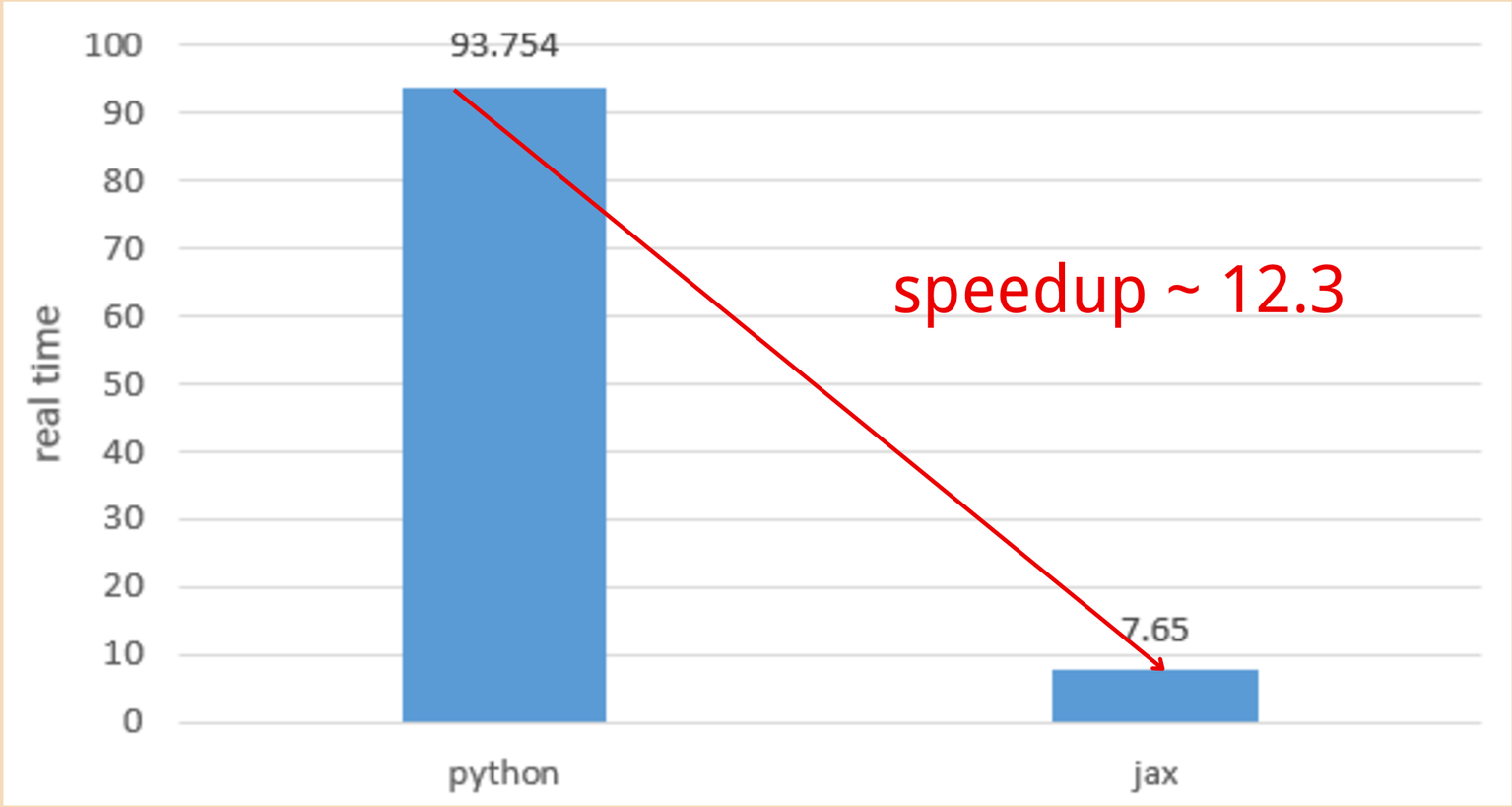
Experiment Design

- Convert the code with JAX
  - Use array to replace iterations
  - JIT + Execute ‘fun’ on GPU
- Reduce the times of calling ‘fun’
  - Precision may be affected

Testcase

- (Dimension, Population, Iterations) = (5, 5, 5)
  - For verifying correctness
- (Dimension, Population, Iterations) = (1024, 8, 5)
  - For testing optimization results

Results



(Dimension, Population, Iterations) = (1024, 8, 5)

Discussion

- 單純改寫使用jax.numpy效果不大 (DeviceArray on CPU)
  - 推測可能是ndarray轉DeviceArray的overhead
- 單純將‘fun’放至GPU上執行，結果反而變差非常多 (DeviceArray on GPU)
  - 推測是因為頻繁的kernel launch所造成的overhead
- 減少呼叫fun次數才是主要優化
  - 在小測資情況下，JAX與原版可以收斂到差不多的值，結果雖略有差異，但我們認為優化的效果大於一點準確度的犧牲；此外，JAX收斂得也較原版快

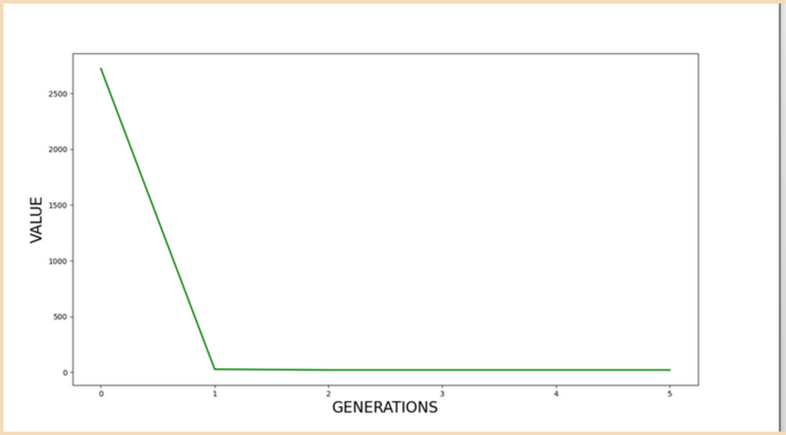
```
for i in range(fa.N):
    for j in range(fa.D):
        for k in range(fa.N):
                xnew = pop[i][j] + Beta*(pop[k][j] - pop[i][
                if xnew > fa.Ub[0]:
                    xnew = fa.Ub[0]
                elif xnew < fa.Lb[0]:
                    xnew = fa.Lb[0]
                pop[i][j] = xnew
                fitness = fa.fun(pop)
                best_ = min(fitness)
                arr_ = fitness.index(best_)
                best_para_ = pop[arr_]
```

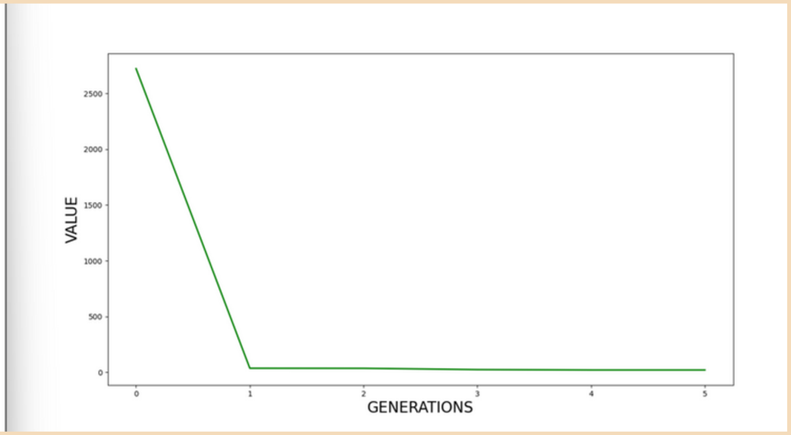
```
for i in range(fa.N):
    for k in range(fa.N):
        if fitness[i] > fitness[k]:
            pop, fitness = step_update(fa, pop, fitness, r_distance, steps_all, i, k)
            best_fitness = min(fitness)
            best_para = pop[jnp.argmin(fitness)]
```

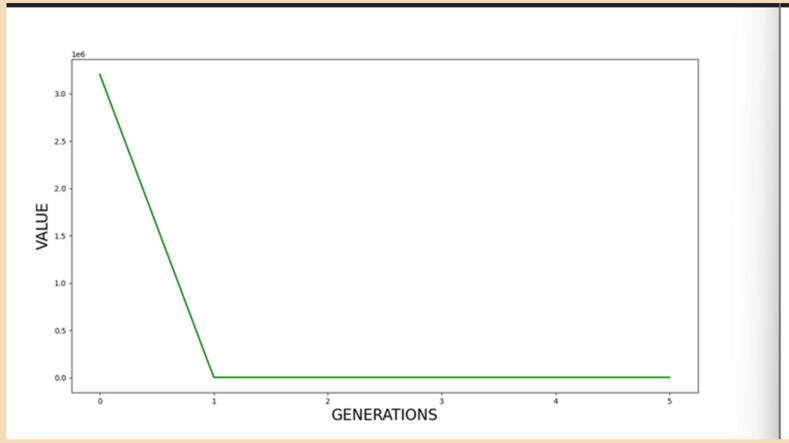
從每更一個dimension就叫一次fun
到每更一隻螢火蟲才叫一次

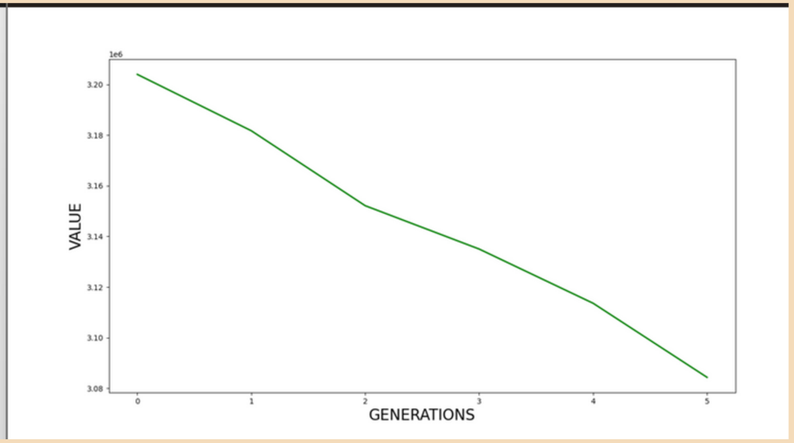
Discussion

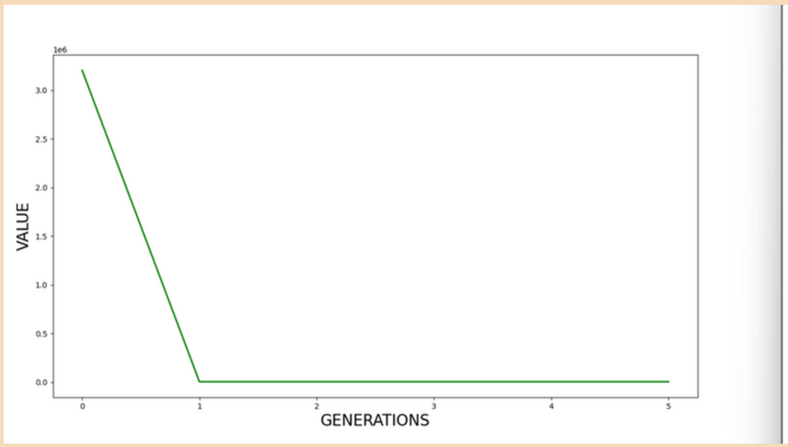
- 収斂結果 (JAX, python)
  - (5, 5, 5)
  - (1024, 8, 5)
  - (1024, 16, 5)

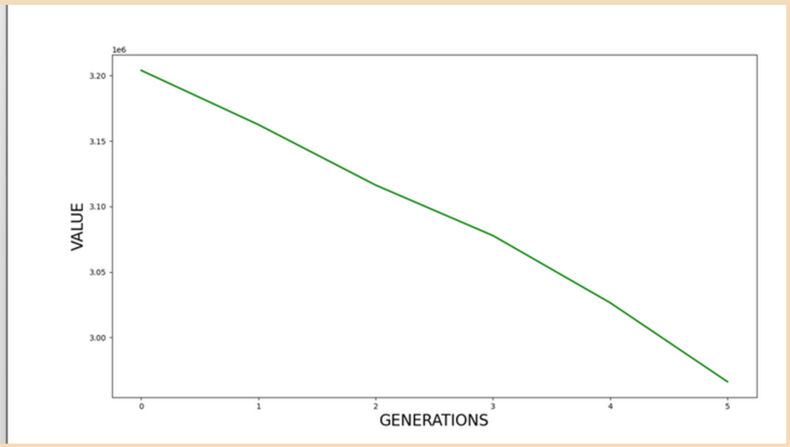












Discussion

- 不過就‘fun’來說，平均的執行時間卻變長了
  - 使用JAX需額外費時將ndarray轉為DeviceArray，以放到GPU上執行
  - 使用JIT，第一次的fun比原本所需的時間長(due to compilation)
  - 推測由於fun原本執行時間就不長，再加上上述的overhead，而導致這樣的結果

python

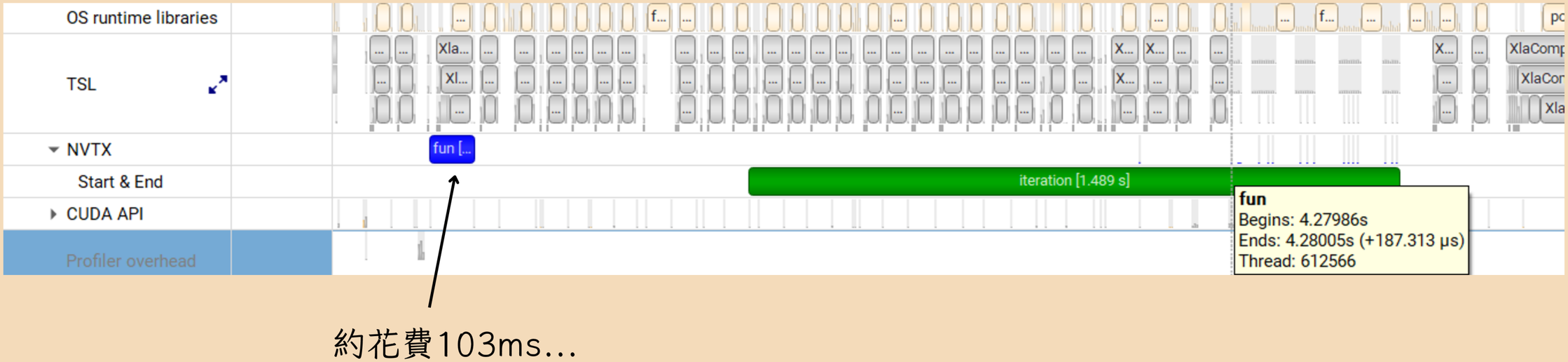
Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Range
100.0%	10.931 s	2524	4.331 ms	4.253 ms	4.163 ms	9.240 ms	406.081 µs	:fun

JAX

Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Range
1.9%	106.651 ms	17	6.274 ms	158.359 µs	138.005 µs	103.970 ms	25.176 ms	:fun

Discussion

- JIT的效果
  - 除了第一次對fun的呼叫花了較長時間外(由於做compilation關係)，後面呼叫的fun都能以很快的速度完成



約花費103ms...

- 其他優化：(沒有顯著較果)
  - 將整個update的步驟做JIT
  - 使用device_put將更新所需的random資料先搬到GPU上

Conclusion

c++

better!

GTX1080 vs INTEL(R) XEON(R) PLATINUM 8568Y+ (24 cores)

better!

GTX1080 vs Intel X5670 *2 (12 cores)

python

- 透過jax.numpy優化陣列運算，並對step_update(更新pop、fun)做JIT，並放上GPU執行
  - overhead考量: ndarray to DeviceArray, compilation time, frequent kernel launch
- 透過陣列運算取代迴圈，減少fun的呼叫次數，雖結果與原版略有不同，但收斂速度提升
- 此次的優化可能對dimension大、iteration多的測資有較顯著的效果