大規模計算特論Ｂ　レポート課題

春山　椋

名古屋大学　情報学研究科
学生番号　251901088
2021 年 8 月 4 日

1　概要

行列サイズ 500, 800, 1000 の行列積に対して，Thread と ThreadBlock の形状を変化させて，それぞれ CUDA による並列化を行った．行列データは一次元配列によって表現し，列を変数 `i`，行を変数 `j` で表現した．行列積は三つのループを要する．外側から，それぞれ列に関するループ (`i`)，行に関するループ (`j`)，総和を取る際のループ (`k`) である．本稿では，次の二通りの方法で並列化を行い，両者の実行時間を比較した：

1. **列に関するループを ThreadBlock ごとに，行に関するループを Thread ごとに並列化する**
2. **列に関するループを Thread ごとに，行に関するループを ThreadBlock ごとに並列化する**

その結果，1. の方が 2. よりも短い実行時間で計算を行うことができることがわかった．これは，一次元配列のインデックスにおいては行列の行ごとに SharedMemory にキャッシュされるはずなので，メモリの局所性が有効活用され，キャッシュミスをおこしやすい 1. よりパフォーマンスが向上したためであると考えられる．以下では，この考察を導く詳細なデータを示す．

2　実行時間データの作成方法

この章では，実行時間データの作成方法を説明する．サンプルコード `mm5.cu` と `mm6.cu` をもとに，

1. 列に関するループを ThreadBlock ごとに，行に関するループを Thread ごとに設定し，**行列サイズ 500，800，1000 の行列積を計算するプログラム `mm5_500.cu`，`mm5_800.cu`，`mm5_1000.cu` をそれぞれ作成した．**

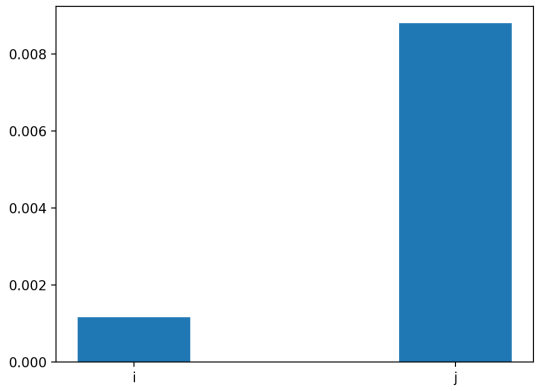


図 1: chart500.png

2. 列に関するループを Thread ごとに，行に関するループを ThreadBlock ごとに設定し，**行列サイズ 500，800，1000 の行列積を計算するプログラム** mm6_500.cu，mm6_800.cu，mm6_1000.cu **をそれぞれ作成した．**

上記のすべてのソースコードにおいて，行列サイズは変数 N で表現されている．以降では基本的に N = 500 と固定する．N = 800，1000 の場合も同様の操作を行った．

mm5_500.cu と mm6_500.cu をビルドしてバイナリファイル mm5_500.bin，mm6_500.bin をそれぞれ作成し，これらをそれぞれ 10 回ずつ実行し，実行時間データ data500 を作成した．このデータでは，一列目が mm5_500.cu，二列目が mm6_500.cu の実行時間データを表す（単位は秒）．これらの平均を取り，棒グラフとして比較したのが chart500.png である．なお，平均を取る際に外れ値を除外した． 同様に chart800.png を作成した．また，mm5_1000.cu と mm6_1000.cu の実行結果を比較すると，行列積の結果が一致しなかったため，これらを比較するデータを作成することができなかった．

3 考察

chart500.png を分析したところ，約 7.5 倍の差があることがわかる．ここから，キャッシュミスによるオーバーヘッドは行列積の計算全体のルーチンの約 7.5 倍を占めると考えられる．

同様に chart800.png を分析したところ，~~約 85 倍~~約 8.5 倍の差があった．行列サイズを大きくすると~~差が縮まるというのは奇妙である．~~**(2021 年 8 月 4 日訂正) 差が拡大したことがわかる（データの集計を取る際，平均を誤って合計で計算したため，この部分だけデータが本来の 10 倍になっていました）．**

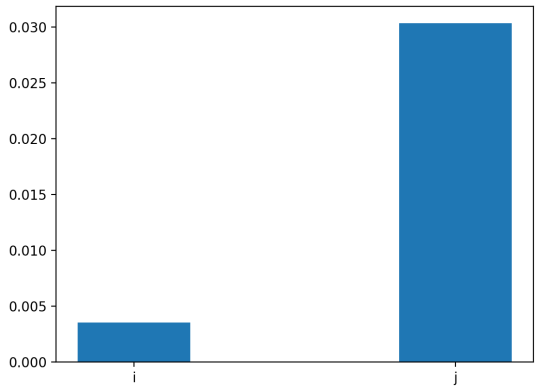


図 2: chart800.png

mm5_1000.cu と mm6_1000.cu の実行結果を比較したときに行列積の結果が一致しなかった理由は，デバイスが許容可能なスレッド数を超えて計算を行ってしまったためであると考えられる．

4 コメント

~~本稿では，行列サイズを大きくすると差が縮まるという奇妙な結果が出ました．~~
もし本稿で間違っているかもしれない箇所に気づかれた場合は，こちらまで連絡をしていただけると幸いです．haruyama.ryo@k.mbox.nagoya-u.ac.jp

5 付録

本稿で使用したプログラムとデータ，グラフを付録として添付する．

```
OPT = -O3 -arch=sm_70 -Xcompiler "-fopenmp"
all:
  gcc -O3 -fopenmp -o mm1 mm1.c
  nvcc ${OPT} -o mm2 mm2.cu
  nvcc ${OPT} -o mm3 mm3.cu
  nvcc ${OPT} -o mm4 mm4.cu
  nvcc ${OPT} -o mm5 mm5.cu
  nvcc ${OPT} -o mm6 mm6.cu

cu:
  nvcc ${OPT} -o mm5_1000.bin mm5_1000.cu
  nvcc ${OPT} -o mm5_2000.bin mm5_2000.cu
  nvcc ${OPT} -o mm6_1000.bin mm6_1000.cu
  nvcc ${OPT} -o mm6_2000.bin mm6_2000.cu

cum:
  nvcc ${OPT} -o mm5_500.bin mm5_500.cu
```

```
  nvcc ${OPT} -o mm5_800.bin mm5_800.cu
  nvcc ${OPT} -o mm6_500.bin mm6_500.cu
  nvcc ${OPT} -o mm6_800.bin mm6_800.cu

clean:
  -/bin/rm ./*.o ./*~
  -/bin/rm ./mm1 ./mm2 ./mm3 ./mm4 ./mm5 ./mm6
  -/bin/rm *.bin *log
```

▲ Makefile

```
// -*- c++ -*-
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <omp.h>

#define frand() (double)(rand()%100)/10.0

__global__ void gpukernel
(int n, double *A, double *B, double *C){
  int i, j, k;
  i = blockIdx.x;
  j = threadIdx.x;
  for(k=0; k<n; k++){
  C[i*n+j] += A[i*n+k] * B[k*n+j];
  }
}

double *A, *B, *C;
double *d_A, *d_B, *d_C; // device memory
int N = 500;
int main(int argc, char **argv){
  int x, y;
  A = (double*)malloc(sizeof(double)*N*N);
  B = (double*)malloc(sizeof(double)*N*N);
  C = (double*)malloc(sizeof(double)*N*N);
  for(y=0; y<N; y++){
    for(x=0; x<N; x++){
      A[y*N+x] = frand();
      B[y*N+x] = frand();
      C[y*N+x] = 0.0;
    }
  }
  cudaMalloc((void**)&d_A, sizeof(double)*N*N);
  cudaMalloc((void**)&d_B, sizeof(double)*N*N);
  cudaMalloc((void**)&d_C, sizeof(double)*N*N);
  cudaMemcpy(d_A, A, sizeof(double)*N*N, cudaMemcpyHostToDevice)
    ;
  cudaMemcpy(d_B, B, sizeof(double)*N*N, cudaMemcpyHostToDevice)
    ;
  cudaMemcpy(d_C, C, sizeof(double)*N*N, cudaMemcpyHostToDevice)
    ;

  double d1, d2;
  cudaDeviceSynchronize();
  d1 = omp_get_wtime();
  gpukernel<<<N,N>>>(N, d_A, d_B, d_C);
  cudaDeviceSynchronize();
  d2 = omp_get_wtime();
  printf("gpukernel %f sec\n", d2-d1);

```

```
  cudaMemcpy(C, d_C, sizeof(double)*N*N, cudaMemcpyDeviceToHost)
    ;
  cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
  // A
  printf("A\n");
  for(y=0; y<N; y++){
    for(x=0; x<N; x++){
      printf(" %.2f", A[y*N+x]);
    }
  printf("\n");
  }
  // B
  printf("B\n");
  for(y=0; y<N; y++){
    for(x=0; x<N; x++){
      printf(" %.2f", B[y*N+x]);
    }
  printf("\n");
  }
  // C
  printf("C\n");
  for(y=0; y<N; y++){
    for(x=0; x<N; x++){
      printf(" %.2f", C[y*N+x]);
    }
  printf("\n");
  }
  free(A); free(B); free(C);
  return 0;
}
```

▲ mm5_500.cu

```
// -*- c++ -*-
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <omp.h>

#define frand() (double)(rand()%100)/10.0

__global__ void gpukernel
(int n, double *A, double *B, double *C){
  int i, j, k;
  i = blockIdx.x;
  j = threadIdx.x;
  for(k=0; k<n; k++){
  C[i*n+j] += A[i*n+k] * B[k*n+j];
  }
}

double *A, *B, *C;
double *d_A, *d_B, *d_C; // device memory
int N = 800;
int main(int argc, char **argv){
  int x, y;
  A = (double*)malloc(sizeof(double)*N*N);
  B = (double*)malloc(sizeof(double)*N*N);
  C = (double*)malloc(sizeof(double)*N*N);
  for(y=0; y<N; y++){
    for(x=0; x<N; x++){
      A[y*N+x] = frand();
```

```
      B[y*N+x] = frand();
      C[y*N+x] = 0.0;
    }
  }
  cudaMalloc((void**)&d_A, sizeof(double)*N*N);
  cudaMalloc((void**)&d_B, sizeof(double)*N*N);
  cudaMalloc((void**)&d_C, sizeof(double)*N*N);
  cudaMemcpy(d_A, A, sizeof(double)*N*N, cudaMemcpyHostToDevice)
    ;
  cudaMemcpy(d_B, B, sizeof(double)*N*N, cudaMemcpyHostToDevice)
    ;
  cudaMemcpy(d_C, C, sizeof(double)*N*N, cudaMemcpyHostToDevice)
    ;

  double d1, d2;
  cudaDeviceSynchronize();
  d1 = omp_get_wtime();
  gpukernel<<<N,N>>>(N, d_A, d_B, d_C);
  cudaDeviceSynchronize();
  d2 = omp_get_wtime();
  printf("gpukernel %f sec\n", d2-d1);

  cudaMemcpy(C, d_C, sizeof(double)*N*N, cudaMemcpyDeviceToHost)
    ;
  cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
  // A
  printf("A\n");
  for(y=0; y<N; y++){
    for(x=0; x<N; x++){
      printf(" %.2f", A[y*N+x]);
    }
  printf("\n");
  }
  // B
  printf("B\n");
  for(y=0; y<N; y++){
    for(x=0; x<N; x++){
      printf(" %.2f", B[y*N+x]);
    }
  printf("\n");
  }
  // C
  printf("C\n");
  for(y=0; y<N; y++){
    for(x=0; x<N; x++){
      printf(" %.2f", C[y*N+x]);
    }
  printf("\n");
  }
  free(A); free(B); free(C);
  return 0;
}
```

▲ mm5_800.cu

```
// -*- c++ -*-
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <omp.h>

#define frand() (double)(rand()%100)/10.0
```

```
9
10 __global__ void gpukernel
11 (int n, double *A, double *B, double *C){
12   int i, j, k;
13   i = blockIdx.x;
14   j = threadIdx.x;
15   for(k=0; k<n; k++){
16   C[i*n+j] += A[i*n+k] * B[k*n+j];
17   }
18 }
19
20 double *A, *B, *C;
21 double *d_A, *d_B, *d_C; // device memory
22 int N = 1000;
23 int main(int argc, char **argv){
24   int x, y;
25   A = (double*)malloc(sizeof(double)*N*N);
26   B = (double*)malloc(sizeof(double)*N*N);
27   C = (double*)malloc(sizeof(double)*N*N);
28   for(y=0; y<N; y++){
29     for(x=0; x<N; x++){
30       A[y*N+x] = frand();
31       B[y*N+x] = frand();
32       C[y*N+x] = 0.0;
33     }
34   }
35   cudaMalloc((void**)&d_A, sizeof(double)*N*N);
36   cudaMalloc((void**)&d_B, sizeof(double)*N*N);
37   cudaMalloc((void**)&d_C, sizeof(double)*N*N);
38   cudaMemcpy(d_A, A, sizeof(double)*N*N, cudaMemcpyHostToDevice)
       ;
39   cudaMemcpy(d_B, B, sizeof(double)*N*N, cudaMemcpyHostToDevice)
       ;
40   cudaMemcpy(d_C, C, sizeof(double)*N*N, cudaMemcpyHostToDevice)
       ;
41
42   double d1, d2;
43   cudaDeviceSynchronize();
44   d1 = omp_get_wtime();
45   gpukernel<<<N,N>>>(N, d_A, d_B, d_C);
46   cudaDeviceSynchronize();
47   d2 = omp_get_wtime();
48   printf("gpukernel %f sec\n", d2-d1);
49
50   cudaMemcpy(C, d_C, sizeof(double)*N*N, cudaMemcpyDeviceToHost)
       ;
51   cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
52   // A
53   printf("A\n");
54   for(y=0; y<N; y++){
55     for(x=0; x<N; x++){
56       printf(" %.2f", A[y*N+x]);
57     }
58   printf("\n");
59   }
60   // B
61   printf("B\n");
62   for(y=0; y<N; y++){
63     for(x=0; x<N; x++){
64       printf(" %.2f", B[y*N+x]);
65     }
66   printf("\n");
```

```
  }
  // C
  printf("C\n");
  for(y=0; y<N; y++){
    for(x=0; x<N; x++){
      printf(" %.2f", C[y*N+x]);
    }
  printf("\n");
  }
  free(A); free(B); free(C);
  return 0;
}
```

▲ mm5_1000.cu

```
// -*- c++ -*-
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <omp.h>

#define frand() (double)(rand()%100)/10.0

__global__ void gpukernel
(int n, double *A, double *B, double *C){
  int i, j, k;
  i = threadIdx.x;
  j = blockIdx.x;
  for(k=0; k<n; k++){
  C[i*n+j] += A[i*n+k] * B[k*n+j];
  }
}

double *A, *B, *C;
double *d_A, *d_B, *d_C; // device memory
int N = 500;
int main(int argc, char **argv){
  int x, y;
  A = (double*)malloc(sizeof(double)*N*N);
  B = (double*)malloc(sizeof(double)*N*N);
  C = (double*)malloc(sizeof(double)*N*N);
  for(y=0; y<N; y++){
    for(x=0; x<N; x++){
      A[y*N+x] = frand();
      B[y*N+x] = frand();
      C[y*N+x] = 0.0;
    }
  }
  cudaMalloc((void**)&d_A, sizeof(double)*N*N);
  cudaMalloc((void**)&d_B, sizeof(double)*N*N);
  cudaMalloc((void**)&d_C, sizeof(double)*N*N);
  cudaMemcpy(d_A, A, sizeof(double)*N*N, cudaMemcpyHostToDevice)
    ;
  cudaMemcpy(d_B, B, sizeof(double)*N*N, cudaMemcpyHostToDevice)
    ;
  cudaMemcpy(d_C, C, sizeof(double)*N*N, cudaMemcpyHostToDevice)
    ;

  double d1, d2;
  cudaDeviceSynchronize();
  d1 = omp_get_wtime();
  gpukernel<<<N,N>>>(N, d_A, d_B, d_C);
```

```
  cudaDeviceSynchronize();
  d2 = omp_get_wtime();
  printf("gpukernel %f sec\n", d2-d1);

  cudaMemcpy(C, d_C, sizeof(double)*N*N, cudaMemcpyDeviceToHost)
    ;
  cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
  // A
  printf("A\n");
  for(y=0; y<N; y++){
    for(x=0; x<N; x++){
      printf(" %.2f", A[y*N+x]);
    }
  printf("\n");
  }
  // B
  printf("B\n");
  for(y=0; y<N; y++){
    for(x=0; x<N; x++){
      printf(" %.2f", B[y*N+x]);
    }
  printf("\n");
  }
  // C
  printf("C\n");
  for(y=0; y<N; y++){
    for(x=0; x<N; x++){
      printf(" %.2f", C[y*N+x]);
    }
  printf("\n");
  }
  free(A); free(B); free(C);
  return 0;
}
```

▲ mm6_500.cu

```
// -*- c++ -*-
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <omp.h>

#define frand() (double)(rand()%100)/10.0

__global__ void gpukernel
(int n, double *A, double *B, double *C){
  int i, j, k;
  i = threadIdx.x;
  j = blockIdx.x;
  for(k=0; k<n; k++){
  C[i*n+j] += A[i*n+k] * B[k*n+j];
  }
}

double *A, *B, *C;
double *d_A, *d_B, *d_C; // device memory
int N = 800;
int main(int argc, char **argv){
  int x, y;
  A = (double*)malloc(sizeof(double)*N*N);
  B = (double*)malloc(sizeof(double)*N*N);
```

```
  C = (double*)malloc(sizeof(double)*N*N);
  for(y=0; y<N; y++){
    for(x=0; x<N; x++){
      A[y*N+x] = frand();
      B[y*N+x] = frand();
      C[y*N+x] = 0.0;
    }
  }
  cudaMalloc((void**)&d_A, sizeof(double)*N*N);
  cudaMalloc((void**)&d_B, sizeof(double)*N*N);
  cudaMalloc((void**)&d_C, sizeof(double)*N*N);
  cudaMemcpy(d_A, A, sizeof(double)*N*N, cudaMemcpyHostToDevice)
    ;
  cudaMemcpy(d_B, B, sizeof(double)*N*N, cudaMemcpyHostToDevice)
    ;
  cudaMemcpy(d_C, C, sizeof(double)*N*N, cudaMemcpyHostToDevice)
    ;

  double d1, d2;
  cudaDeviceSynchronize();
  d1 = omp_get_wtime();
  gpukernel<<<N,N>>>(N, d_A, d_B, d_C);
  cudaDeviceSynchronize();
  d2 = omp_get_wtime();
  printf("gpukernel %f sec\n", d2-d1);

  cudaMemcpy(C, d_C, sizeof(double)*N*N, cudaMemcpyDeviceToHost)
    ;
  cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
  // A
  printf("A\n");
  for(y=0; y<N; y++){
    for(x=0; x<N; x++){
      printf(" %.2f", A[y*N+x]);
    }
  printf("\n");
  }
  // B
  printf("B\n");
  for(y=0; y<N; y++){
    for(x=0; x<N; x++){
      printf(" %.2f", B[y*N+x]);
    }
  printf("\n");
  }
  // C
  printf("C\n");
  for(y=0; y<N; y++){
    for(x=0; x<N; x++){
      printf(" %.2f", C[y*N+x]);
    }
  printf("\n");
  }
  free(A); free(B); free(C);
  return 0;
}
```

▲ mm6_800.cu

```
// -*- c++ -*-
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

```
#include <cuda_runtime.h>
#include <omp.h>

#define frand() (double)(rand()%100)/10.0

__global__ void gpukernel
(int n, double *A, double *B, double *C){
  int i, j, k;
  i = blockIdx.x;
  j = threadIdx.x;
  for(k=0; k<n; k++){
  C[i*n+j] += A[i*n+k] * B[k*n+j];
  }
}

double *A, *B, *C;
double *d_A, *d_B, *d_C; // device memory
int N = 1000;
int main(int argc, char **argv){
  int x, y;
  A = (double*)malloc(sizeof(double)*N*N);
  B = (double*)malloc(sizeof(double)*N*N);
  C = (double*)malloc(sizeof(double)*N*N);
  for(y=0; y<N; y++){
    for(x=0; x<N; x++){
      A[y*N+x] = frand();
      B[y*N+x] = frand();
      C[y*N+x] = 0.0;
    }
  }
  cudaMalloc((void**)&d_A, sizeof(double)*N*N);
  cudaMalloc((void**)&d_B, sizeof(double)*N*N);
  cudaMalloc((void**)&d_C, sizeof(double)*N*N);
  cudaMemcpy(d_A, A, sizeof(double)*N*N, cudaMemcpyHostToDevice)
    ;
  cudaMemcpy(d_B, B, sizeof(double)*N*N, cudaMemcpyHostToDevice)
    ;
  cudaMemcpy(d_C, C, sizeof(double)*N*N, cudaMemcpyHostToDevice)
    ;

  double d1, d2;
  cudaDeviceSynchronize();
  d1 = omp_get_wtime();
  gpukernel<<<N,N>>>(N, d_A, d_B, d_C);
  cudaDeviceSynchronize();
  d2 = omp_get_wtime();
  printf("gpukernel %f sec\n", d2-d1);

  cudaMemcpy(C, d_C, sizeof(double)*N*N, cudaMemcpyDeviceToHost)
    ;
  cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
  // A
  printf("A\n");
  for(y=0; y<N; y++){
    for(x=0; x<N; x++){
      printf(" %.2f", A[y*N+x]);
    }
  printf("\n");
  }
  // B
  printf("B\n");
  for(y=0; y<N; y++){
```

```
    for(x=0; x<N; x++){
      printf(" %.2f", B[y*N+x]);
    }
  printf("\n");
  }
  // C
  printf("C\n");
  for(y=0; y<N; y++){
    for(x=0; x<N; x++){
      printf(" %.2f", C[y*N+x]);
    }
  printf("\n");
  }
  free(A); free(B); free(C);
  return 0;
}
```

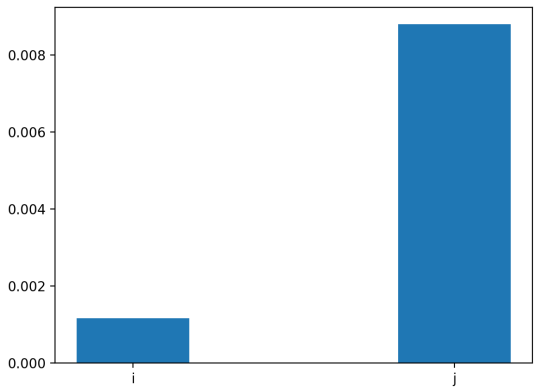
▲ mm6_1000.cu

```
0.001809 0.008751
0.001097 0.008815
0.001112 0.008826
0.001122 0.008762
0.001116 0.008774
0.001104 0.008846
0.001114 0.008823
0.001106 0.008804
0.001054 0.008764
0.001070 0.008790
```

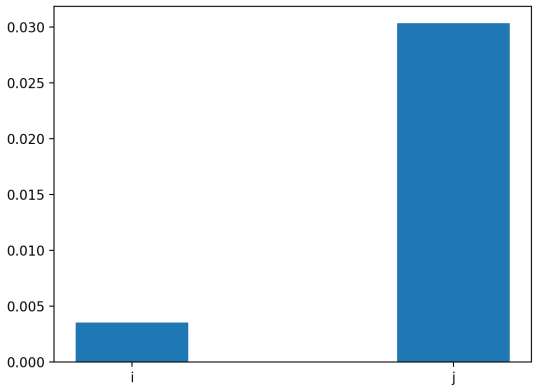
▲ data500

```
0.003600 0.030425
0.003565 0.030280
0.003557 0.030434
0.003553 0.030243
0.003553 0.030204
0.003560 0.030379
0.003560 0.030358
0.003552 0.030212
0.003556 0.030264
0.003562 0.030283
```

▲ data800



▲ chart500.png



▲ chart800.png