并行计算与 GPU 编程

温家琛：2017211427

秦子涵：2017211434

秦峰：2017211225

提交日期：2020 年 6 月 11 日

一 、FJSP 及 GA 简述

1 .求解问题(Flexible Job-shop Scheduling Problem)

简介：柔性作业车间调度问题是传统 Job-Shop 调度问题的扩展，在传统的 Job-Shop 调度问题中，工件的每道工序只能在一台确定的机床上加工，而在柔性作业车间调度问题中，每道工序可以在多台机床上加工，并且在不同的机床上加工所需时间不同。柔性作业车间调度问题减少了机器约束，扩大了可行解的搜索范围，增加了问题的复杂性。

问题描述：一个加工系统有 m 台机器，要加工 n 种工件。每个工件包含一道或多道工序，工件的工序顺序是预先确定的；每道工序可以在多台不同的机床上加工，工序的加工时间随机床的性能不同而变化。调度目标是为每道工序选择最合适的机器、确定每台机器上各工件工序的最佳加工顺序及开工时间，使系统的某些性能指标达到最优。此外，在加工过程中还需满足以下约束条件:

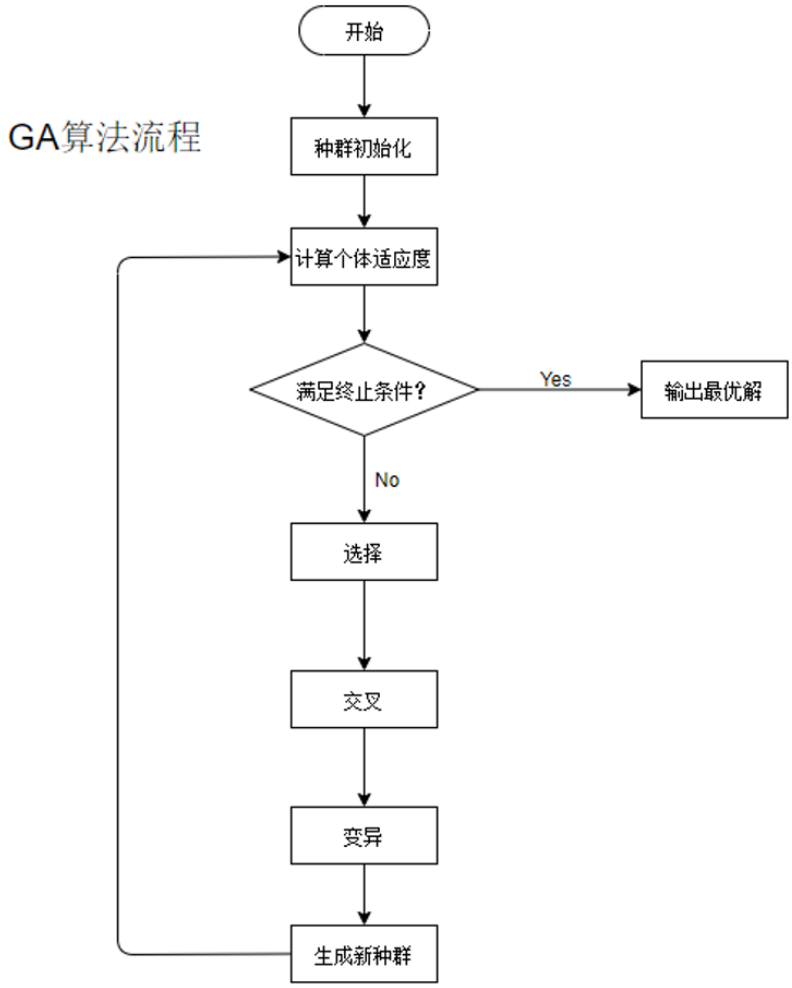
- 同一时刻同一台机器只能加工一个零件
- 每个工件在某一时刻只能在一台机器上加工，不能中途中断每一个操作
- 同一工件的工序之间有先后约束，不同工件的工序之间没有先后约束
- 不同工件具有相同的优先级

优化目标：确定每个机器上工序的加工顺序和每个工序的开工时间，使得最大完工时间 Cmax 最小或者其他指标达到最优

2 算法简介(Genetic Algorithm)

基本思想：生物的进化是通过染色体来实现的，染色体上有着许多控制生物性状的基因，这些基因会在遗传过程中随着染色体的交叉进行重新组合，同时会以一定概率发生变异。遗传算法的基本思路与此类似，可以将待优化问题的求解看作生物努力适应环境的过程，问题的解对应生物种群中的个体，算法的搜索便是种群一代代进化最终形成稳定物种的过程

算法流程图:



二、算法设计

1.GA 算法设计

- step1：编码
- step2：种群初始化
  - 随机初始化个体的遗传信息，并将工序初始化为顺序工序，即执行顺序为工序 1，工序 2……
  - 按照随机初始化的遗传信息改变工序的顺序，以达到随机初始化工序的目标

- 伪代码如下

```
for j <- 1 to tot 1 do
    s[i].a[j] = rand();
    s[i].count[j] = j;


for j <- 1 to n 1 do
    qsort(s[i].count + num[j - 1] + 1, num[j] - num[j - 1], sizeof(int),
 cmp3);
     end
end
```

- step3：计算适应度
  - 通过设计适应度函数来评价个体的优劣程度
  - 实现算法中通过计算每种工序的总时长来评判优劣度，时长越短，优良性越高
  - 适应度函数

$$f(P) = C_{max}$$
$$= \min\{max\ C_i\}$$
$$其中C_i是工件J_i完工的时间\ i \in (1,n)$$

- step4：选择
  - 根据适应度函数所计算的值度量个体的优劣程度，以此决定个体在下一代中是被淘汰还是被遗传。适应度越大的个体，即越优良的个体被选中的机会就越大，而适应度越小的个体被选中的机会就越小
  - 实现算法中采取了计算适应度后，对所有个体依据适应度排序，结果存放在临时数组中，之后选取适应度最大的前 N 个个体作为下一代的父代
  - 伪代码如下

```
sort(fitness);

//初始化最优数组
for i <- 1 to (sizeT + Cross) * 2 1 do
    best[i] <- i;
end

qsort(best + 1, (sizeT + Cross) * 2, sizeof(int), cmp2);

//选取最优个体
for i <- 1 to sizeT 1 do
    tmp[i] <- s[best[i]];
end
```

```
for i <-1 to sizeT 1 do
    s[i] <- tmp[i];
 unit(s[i].a);
end
```

- step5：交叉
  - 在种群中的所有个体中随机选择两个个体配对，并将其作为父代进行交叉操作，在父代基因中随机生成基因点位，使得子代可以分别继承父代的部分基因，以达到基因重组的效果
  - 伪代码如下

```
//随机选取父代
x <- rand() % sizeT + 1;
y <- rand() % sizeT + 1;

//s 数组是个体数组
//a 数组是遗传信息
//count 数组是每个工件工序
for i <- 1 to Cross 1 do
    for j <- 1 to tot 1 do
    s[i + sizeT].a[j] <- s[x].a[j] - s[y].a[j];
    s[i + sizeT].count[j] <- j;

 qsort(s[cur].count + 1, tot, sizeof(int), cmp1);

 for j <- 1 to n 1 do
    qsort(s[cur].count + num[j - 1] + 1, num[j] - num[j - 1],
 sizeof(int), cmp3);
    end
end
```

- step6：变异
  - GA 算法中采用变异是为了模拟自然遗传中适者生存，并不断繁衍的过程，其目的在于改善算法的局部搜索能力和维持群体多样性
  - FJSP 问题中每道工序均可以由多台机器完成，而每台机器完成该道工序的时间因机器性能等因素有所不同，因此变异时随机选取部分工序，然后对选中的工序按照某种策略(比例选择策略，加工时间优先策略等)选择一台机器来完成该工序，并将完成的机器号写入对应的基因串中，以确保得出的结果是可行解
  - 伪代码如下

```
for i <- i to (sizeT + Various) 1 do
    cur <- i + sizeT + Various;
```

```
    for j <- 1 to tot 1 do
        delta[j] <- rand() - (RAND_MAX / 2);
        delta[j] <- delta[j] * (rand() % 2);
end

for j <- 1 to tot 1 do
        s[cur].a[j] <- s[i].a[j] + delta[j];
end

qsort(s[cur].count + 1, tot, sizeof(int), cmp1);

for j <- 1 to n 1 do
        qsort(s[cur].count + num[j - 1] + 1, num[j] - num[j - 1],
sizeof(int), cmp3);
end
end
```

- setp7：生成新种群
  - 当前迭代次数加一，生成新种群，并判断是否应终止迭代，若否，则跳转至 step3

2.PGA 算法设计

- **设计思想**
  - 遗传算法源自自然进化，自然也具有自然进化所固有的并行属性，遗传算法的并行性包括了内在并行性和隐含并行性
  - 内在并行性：遗传算法本身适合大规模并行计算，可以把初始种群分为若干子种群，每个子种群交给一个处理器单独进化，等进化结束后进行统一比较选择
  - 隐含并行性：遗传算法采用种群方式组织搜索，可以同时搜索解空间中的多个区域，并相互交流，这使得它在每次只执行种群规模为 N 的情况下，实际进行了约 O(n^3)次有效计算
  - 并行遗传算法相比于上述串行遗传算法可在很多地方实现并行化，除了初始化部分是纯串行之外，在选择，交叉，变异，适应度评价等多方面存在着并发性
- **并行化策略与设计**
  - 适应度评价函数：个体适应度评价需要占用一定的时间，提升计算个体适应度的效率，找到并行计算个体适应度函数的恰当方式，可以有效提升遗传算法的选择效率

- 产生新种群：PGA 中个体的选择是同时完成的，不同个体的适应度相互独立，在适应度函数评价个体的时候可以采用并行化策略，**同理，交叉，变异等过程也可采用并行化策略**
- 设计主从式模型的 PGA 算法，从原理上并没有对标准遗传算法的框架进行任何改动，不会影响解决具体问题的效果，其算法框架参考 GA 算法框架即可，因此下述设计步骤仅叙述两者不同之处
- PGA 算法在种群初始化时是串行操作，之后可将选择，交叉，变异，适应度评价等进行并行化操作。而在每一代种群之间也采用串行化操作。在 CPU 中进行数据初始化操作，之后将初始化的数据地址拷贝至 GPU，进行并行化操作
- 并行操作时需要通信以达到所有个体的每一次操作的同步

三、调测分析

1.调试日志

- version1：尝试进行 cuda 编程，首先以正确运行为目的进行代码编写
  - 此版本仅对计算每一代所有个体性状时，即计算每个个体的最短工序时间，进行了并行计算
  - 初次尝试 cuda 编程的加速效果，效果不佳，甚至出现了负优化的现象，只是简单的对于最后的计算流程进行了核函数的编写
  - 发现问题：每代遗传样本必须是代与代之间串行；每次进行计算的时候需要由 CPU 向 GPU 传送数据，原本并不复杂的计算经过了内存拷贝过程后反而将计算节约的时间花费在了传递的过程中
- version2：针对 version1 的调试思路进行代码的流程修改
  - 考虑到上一个版本在拷贝中花费了大量的时间，我们思考把数据全部存放在 GPU 内存中，整个遗传算法也放到 GPU 的一个线程中，需要用到数据时直接用传递的指针读数据即可，这样在需要并行计算时再调用其他的 kernel（动态并行），这样就消灭了来回拷贝数据的时间。

  - 将算法整体加入核函数中，但是考虑到遗传样本在代与代之间是串行的，优化空间并不大，相比 CPU 版本慢了整整 20~30 倍
  - 发现问题：__global__中调用其他的__global__，会非常慢，只能调用其他的__device__
- version3：调整算法架构，并将每代样本中进行处理的流程尽量做并行化处理
- version4：查阅了资料并请教了助教老师后有以下优化方向
  - 减少核函数的嵌套，尽量避免动态的并行
  - 内存分配的优化，统一发配
  - 选定了优化内存分配的方案，提前分配 cuda 内存，运行时间得到了高效的优化
  - 弃用 kernel 中的 malloc()函数
- version5：优化代码结构及相关变量命名问题

2.测试数据

（说明：以下数据均来自于助教提供服务器执行结果）

- Mk01： *Mk01.txt*
- Mk02： *Mk02.txt*
- Mk03： *Mk03.txt*
- Mk04： *Mk04.txt*
- Mk05： *Mk05.txt*

3.加速比

- 加速比定义，设 T1 是某算法在串行计算机上的运行时间，Tp 是该算法在 p 个处理器上所构成的并行机上运行的时间，则此算法在该并行机上的加速比 Sp 定义为

$$S_p = \frac{T_1}{T_p}$$

- 服务器可执行文件及测试数据等截图:

```
temp@gpulab-1080ti:~/gtx-iT0801$ ls
main_cu    main_cu_3  main_cu_5  Mk02.txt  Mk04.txt  output.txt
main_cu_2  main_cu_4  Mk01.txt   Mk03.txt  Mk05.txt
```

- Mk01
  - **加速比 = 71.164**(保留三位小数)
  - python：172.38s
  - python 运行截图

```
Finished in 172.38s

\noindent\resizebox{\textwidth}{!}{
\begin{tikzpicture}[x=.5cm, y=1cm]
\begin{ganttchart}{1}{42}
[vgrid, hgrid]{42}
\gantttitle{Flexible Job Shop Solution}{42} \\
\ganttitlelist{1,...,42}{1} \\
```

  - CUDA-C：2.42229s
  - CUDA-C 运行截图

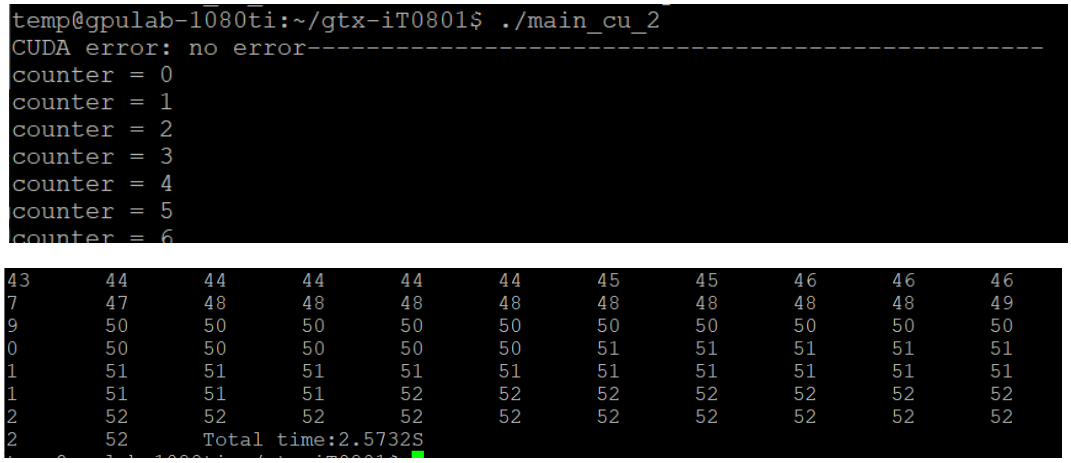
```
temp@gpulab-1080ti:~/gtx-iT0801$ ./main_cu
CUDA error: no error----------------------------------------------------
counter = 0
counter = 1
counter = 2
counter = 3
counter = 4
counter = 5
counter = 6
```

```
49      53      53      53      54      54      54      54      55      55      55      55
6       56      56      56      56      56      56      56      56      56      56      56
6       56      56      56      56      56      56      56      56      56      56      56
6       56      56      56      56      56      56      56      56      56      56      56
6       56      56      56      56      56      56      56      56      56      56      56
6       56      56      56      56      56      56      56      56      56      56      56
6       56      56      56      56      56      56      56      56      56      56      56
6       56      Total time:2.42229S
```

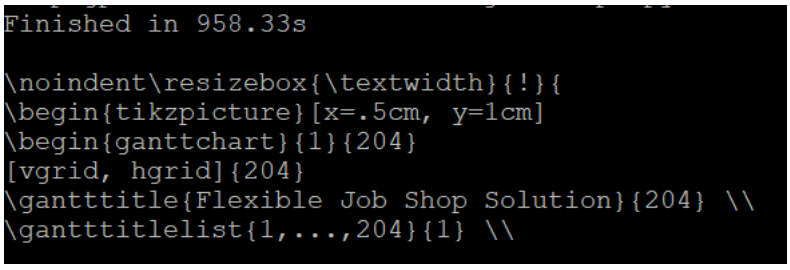
- Mk02
  - **加速比 = 64.624**(保留三位小数)
  - python：166.29s
  - python 运行截图

```
temp@gpulab-1080ti:~/gtx-iT0801$ ./main_cu_2
CUDA error: no error----------------------------------------------------
counter = 0
counter = 1
counter = 2
counter = 3
counter = 4
counter = 5
counter = 6
```

- CUDA-C：2.5732s
- CUDA-C 运行截图



- Mk03
  - **加速比 = 147.419**(保留三位小数)
  - python：958.33s
  - python 运行截图



  - CUDA-C：6.50073s
  - CUDA-C 运行截图

```
temp@gpulab-1080ti:~/gtx-iT0801$ ./main_cu_3
CUDA error: no error------------------------------------------------------
counter = 0
counter = 1
counter = 2
counter = 3
counter = 4
counter = 5
counter = 6
```

```
285     302     306     306     308     310     314     314     317     317     319     323
26      327     329     329     329     330     331     331     333     333     333     333
33      333     333     333     333     333     333     333     333     335     335     335
37      337     338     339     339     340     340     340     340     341     341     342
43      344     344     344     344     345     345     345     345     345     346     347
47      347     347     347     347     347     347     347     347     347     347     347
47      347     347     347     347     348     349     349     349     349     349     349
50      350     Total time:6.50073S
```

- Mk04
  - **加速比 = 111.319**(保留三位小数)
  - python：359.02s
  - python 运行截图

```
temp@gpulab-1080ti:~/flexible-job-shop$ python3 main.py test_data/Brandimarte_Data/Text/Mk04.fjs
Finished in 359.02s

\noindent\resizebox{\textwidth}{!}{
\begin{tikzpicture}[x=.5cm, y=1cm]
\begin{ganttchart}{1}{67}
[vgrid, hgrid]{67}
\ganttttitle{Flexible Job Shop Solution}{67} \\
\ganttttitlelist{1,...,67}{1} \\
```

  - CUDA-C：3.22515s
  - CUDA-C 运行截图

```
temp@gpulab-1080ti:~/gtx-iT0801$ ./main_cu_4
CUDA error: no error------------------------------------------------------
counter = 0
counter = 1
counter = 2
counter = 3
counter = 4
counter = 5
counter = 6
```

```
87      90      92      93      93      94      95      95      95      96
7       98      98      99      99      100     100     101     101     101
03      103     103     103     103     103     103     103     104     104
05      105     106     106     106     106     106     106     106     106
06      106     106     106     106     106     106     106     106     106
06      106     106     106     106     106     106     106     106     106
06      106     106     106     106     106     106     106     106     106
06      106     Total time:3.23515S
```

- Mk05
  - **加速比 = 223.878**(保留三位小数)

- python：896.05s
- python 运行截图

```
temp@gpulab-1080ti:~/flexible-job-shop$ python3 main.py test_data/Brandimarte_Data/Text/Mk05.fjs
Finished in 896.05s

\noindent\resizebox{\textwidth}{!}{
\begin{tikzpicture}[x=.5cm, y=1cm]
\begin{ganttchart}{1}{177}
[vgrid, hgrid]{177}
\ganttitle{Flexible Job Shop Solution}{177} \\
\ganttitlelist{1,...,177}{1} \\
```

- CUDA-C：4.00245s
- CUDA-C 运行截图

```
temp@gpulab-1080ti:~/gtx-iT0801$ ./main_cu_5
CUDA error: no error----------------------------------------------------
counter = 0
counter = 1
counter = 2
counter = 3
counter = 4
counter = 5
counter = 6
```

```
215     221     222     222     225     226     227     228     228     228
30      230     230     230     230     230     230     230     230     230
30      231     231     231     231     231     231     231     231     231
35      235     235     235     235     235     235     235     235     235
35      235     235     235     235     235     235     235     235     235
35      235     235     235     235     235     235     235     235     235
35      235     235     235     235     235     235     235     237     238
38      238     Total time:4.00245S
```

4.解的质量

数据 \ 类型	Python 串行	CUDA C++并行
Mk01	1,…,42	1,…49
Mk02	1,…,28	1,…,40
Mk03	1,…,204	1,…,285
Mk04	1,…67	1,…,87

Mk05	1,…177	1,…,215

四、代码附录

- 串行版本
  - 完整代码： *source.cpp*
- CUDA-C 版本
  - 完整代码： *CUDA-C.cu*
  - 核心算法

```
//遗传算法
void geneticAl(good *d_s, int *d_num, int *num, int *d_n, int *n, int *d_tot, in
t *tot, ob **d_a, ob **a,
   int *d_facx, int *facx, int *d_facy, int *facy, good *d_tmp, good *tmp, int *
d_best, int *best) {
   *tot = num[*n];                                          //tot 记录工序总计
   for (int i = 1; i <= sizeT; i++)                         //循环 sizeT=100 次，初始化 si
zeT 个个体
   {
      for (int j = 1; j <= *tot; j++)                       //该循环让每个个体的 a 中 50 个
工序随机，count 中的第[j]初始化为 j
      {
         s[i].a[j] = rand();
         s[i].count[j] = j;
      }
      unit_c(s[i].a, *tot);                                 //s[i].a 的规范化
      cmper = s[i].a;

      qsort(s[i].count + 1, *tot, sizeof(int), cmp1); //按照 a 大小给 count 排序
      for (int j = 1; j <= *n; j++)                         //给各个物件的工序排序
      {
         qsort(s[i].count + num[j - 1] + 1, num[j] - num[j - 1], sizeof(int), c
mp3);
      }
   }

   cudaMemcpy(d_s, s, MaxPopul * sizeof(good), cudaMemcpyHostToDevice);
```

```
    int critical = sizeT + 1;
    // 执行kernel
    Calculate << < 1, 128 >> > (d_a, d_s, d_facx, d_facy, d_tot, critical);//计算
最初代种群表现性状ans

    cudaError_t error = cudaGetLastError();
    printf("CUDA error: %s----------------------------------------------------\n",
 cudaGetErrorString(error));

    unsigned int *d_rand;
    cudaMalloc((void **)&d_rand, 1024 * sizeof(unsigned int));
    curandGenerator_t gen; //生成随机数变量
    curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_XORWOW);  //步骤1：指定算法
    curandSetPseudoRandomGeneratorSeed(gen, time(NULL));//步骤2：随机数初始化

    int counter = 0;
    while (counter <= MaxGen) {//cpu 中执行循环，调用kernel
        cout << "counter = " << counter << endl;

        //交叉
        curandGenerate(gen, d_rand, 1024);              //步骤三：生成随机数
        critical = Cross + 1;
        cross << < 1, 512 >> > (d_rand, d_n, d_num, d_s, d_tot, critical);
        cudaDeviceSynchronize();//等待同步

        //变异
        curandGenerate(gen, d_rand, 1024);
        critical = sizeT + Cross + 1;
        variation << < 1, 512 >> > (d_rand, d_n, d_num, d_s, d_tot, critical);
        cudaDeviceSynchronize();

        //计算性状ans
        critical = sizeT + Cross * 2 + 1;
        Calculate << < 2, 512 >> > (d_a, d_s, d_facx, d_facy, d_tot, critical);
        cudaDeviceSynchronize();

        //选择遗传
        for (int i = 1; i <= (sizeT + Cross) * 2; i++) {
            best[i] = i;
        }
        cudaMemcpy(s, d_s, MaxPopul * sizeof(good), cudaMemcpyDeviceToHost);
        qsort(best + 1, (sizeT + Cross) * 2, sizeof(int), cmp2);
        cudaMemcpy(d_best, best, MaxPopul * sizeof(int), cudaMemcpyHostToDevice);
        critical = sizeT + Cross + 1;
        genetic_1 << < 1, 128 >> > (d_tmp, d_s, d_best, critical);
        cudaDeviceSynchronize();
        genetic_2 << < 1, 128 >> > (d_s, d_tmp, d_tot, critical);
        cudaDeviceSynchronize();
```

```
        counter++;
    }
    curandDestroyGenerator(gen); //释放指针
    cudaFree(d_rand); //释放 GPU 侧内存空间
}

//计算个体表现型，即工序所需最短时间
__global__ void Calculate(ob **a, good *s, int *facx, int *facy, int *tot, int
n) {
    int x = 0;
    if (n == 101) {
        x = blockDim.x * blockIdx.x + threadIdx.x + 1;
    }
    else {
        x = blockDim.x * blockIdx.x + threadIdx.x + 101;
        n += 100;
    }

    int Max = 0;                                    //Max 最大时间

    if (x < n) {
        int sx[MaxOper];
        int sy[MaxOper];
        int mt[MaxOper];
        int gt[MaxOper];
        memset(sx, 0, sizeof(sx));
        memset(sy, 0, sizeof(sy));
        memset(mt, 0, sizeof(mt));
        memset(gt, 0, sizeof(gt));

        for (int i = 1; i <= *tot; i++)             //该循环就做下面两件事
        {
            sx[s[x].count[i]] = facx[i];            //第几个工序对应属于第几个物件
            sy[s[x].count[i]] = facy[i];            //这个工序对应物件的第几个工序
        }
        for (int i = 1; i <= *tot; i++)
        {
            int tmp = 0;
            int chosen;
            int time = 100000;
            int chosen_time;
            int len = a[sx[i]][sy[i]].count;

            for (int j = 0; j < len; j++)
            {
                if (mt[a[sx[i]][sy[i]].m[j]] < time)
                {
                    time = mt[a[sx[i]][sy[i]].m[j]];
```

```
                chosen = a[sx[i]][sy[i]].m[j];
                chosen_time = j;
            }
        }
        tmp = max(mt[chosen], gt[sx[i]]);
        tmp += a[sx[i]][sy[i]].t[chosen_time];
        Max = max(tmp, Max);
        mt[chosen] = tmp;
        gt[sx[i]] = tmp;
    }
    s[x].ans = Max;
  }
}

__device__ void bubbleSort_1(int *base, int tot, double *cmp) {
  int tmp = 0;
  for (int i = 0; i < tot - 1; i++) {
    for (int j = i; j < tot - 1; j++) {
      if (cmp[base[j]] > cmp[base[j + 1]]) {
        tmp = base[j];
        base[j] = base[j + 1];
        base[j + 1] = tmp;
      }
    }
  }
}

__device__ void bubbleSort_2(int *base, int tot) {
  int tmp = 0;
  for (int i = 0; i < tot - 1; i++) {
    for (int j = i; j < tot - 1; j++) {
      if (base[j] > base[j + 1]) {
        tmp = base[j];
        base[j] = base[j + 1];
        base[j + 1] = tmp;
      }
    }
  }
}

__device__ void bubbleSort_3(int *base, int tot, good *s) {
  int tmp = 0;
  for (int i = 0; i < tot - 1; i++) {
    for (int j = i; j < tot - 1; j++) {
      if (s[base[j]].ans > s[base[j + 1]].ans) {
        tmp = base[j];
        base[j] = base[j + 1];
        base[j + 1] = tmp;
      }
    }
```

```
    }
}

//交叉
__global__ void cross(unsigned int *d_rand, int *n, int *num, good *s, int *tot,
 int critical) {
    int i = blockDim.x * blockIdx.x + threadIdx.x + 1;
    if (i < critical)
    {
        double cmp[MaxOper];
        int x = d_rand[i] % sizeT + 1, y = d_rand[2 * i] % sizeT + 1;
        int cur = i + sizeT;
        s[cur].ans = 0;
        for (int j = 1; j <= *tot; j++)
        {
            s[i + sizeT].a[j] = s[x].a[j] - s[y].a[j];
            s[i + sizeT].count[j] = j;
        }
        unit(s[i + sizeT].a, *tot);

        for (int i = 1; i < *tot; i++) {
            cmp[i] = s[cur].a[i];
        }

        bubbleSort_1(s[cur].count + 1, *tot, cmp);
        for (int j = 1; j <= *n; j++)
        {
            bubbleSort_2(s[cur].count + num[j - 1] + 1, num[j] - num[j - 1]);
        }
    }
}

//变异
__global__ void variation(unsigned int *d_rand, int *n, int *num, good *s, int *
tot, int critical) {
    int i = blockDim.x * blockIdx.x + threadIdx.x + 1;
    if (i < critical)
    {
        double del[MaxOper];
        double cmp[MaxOper];
        memset(del, 0, sizeof(double) * 500);
        int cur = i + sizeT + Cross;

        for (int j = 1; j <= *tot; j++)
        {
            del[j] = (d_rand[i] / 2) * (d_rand[2 * i] % 3);
        }

        unit2(del, tot);
```

```
        for (int j = 1; j <= *tot; j++)
        {
            s[i + sizeT + Cross].a[j] = s[i].a[j] + del[j];
            s[i + sizeT + Cross].count[j] = j;
        }
        for (int i = 1; i < *tot; i++) {
            cmp[i] = s[cur].a[i];
        }
        s[cur].ans = 0;

        bubbleSort_1(s[cur].count + 1, *tot, cmp);
        for (int j = 1; j <= *n; j++) {
            bubbleSort_2(s[cur].count + num[j - 1] + 1, num[j] - num[j - 1]);
        }
    }
}
```