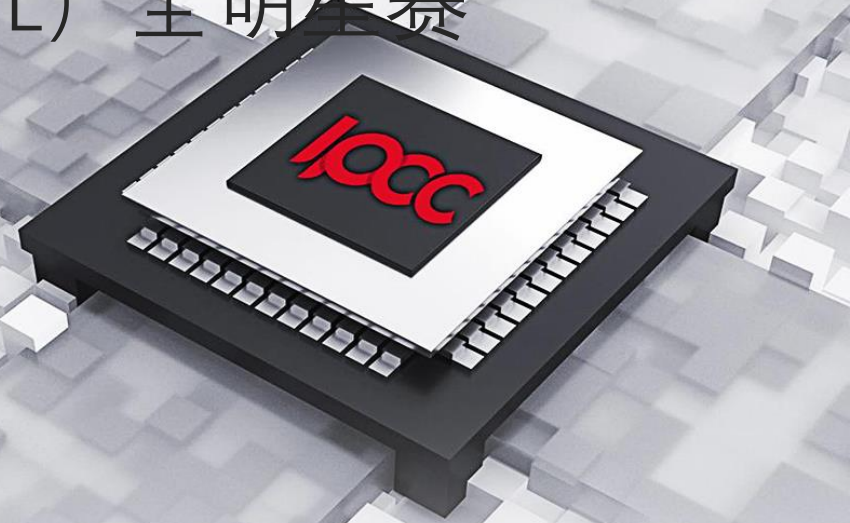
Association for Computing Machinery

ACM中国 国际并行计算挑战赛

AMD

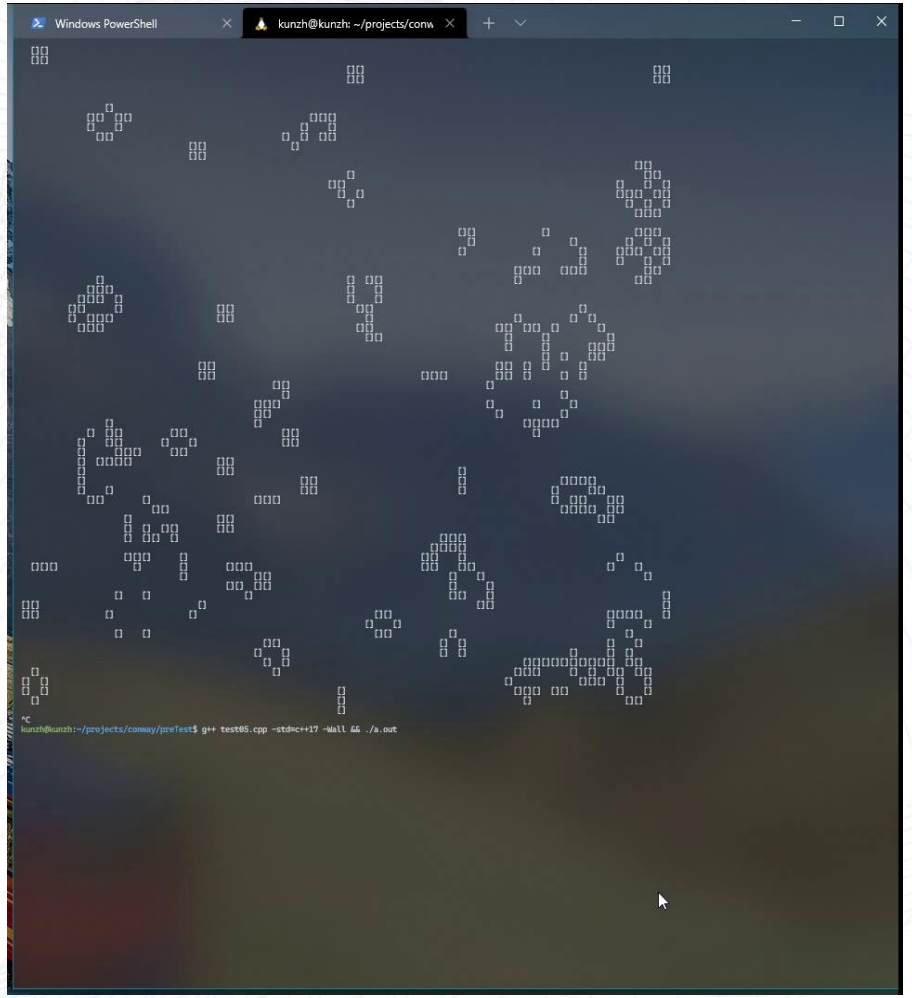
全国大学生高性能计算超级联赛（HPC-PL）全明星赛

暨第三届ACM中国-国际并行计算挑战赛开幕赛



汇报人：吴智琨　2022年5月22日

目录 CONTENTS



02.应用程序运行的硬件环境

CPU	Intel 10875H	AMD EPYC 7452	Intel Xeon Glod 8180
Core(s) per socket	8	32	28
Thread(s) per core	2	1	2
Sockets (numa)	1	2	2
Frequency	2.3 ~ 5.1(4.3 all) GHz	2.35 ~ 3.35 GHz	2.5 ~ 3,8 GHz
L1d/L1i cache	256KB/256KB	32KB/32KB	32KB/32KB
L2 cache	2MB	512KB	1MB
L3 cache	16MB	16MB	38MB
AVX2-GFLOPS	< 700	2406.4*2	2240*2
stream	24.1GB/s	244.9GB/s	137.4GB/s
Max bandwidth	45.8 GB/s	400 GB/s	250 GB/s
	开发/编译平台	运行平台	参考参数

开发/编译平台：INTEL i7-10875H
信息简述：8核16线程，开发及初步调优使用

运行平台：AMD EPYC 7452
信息简述：NUMA架构，32核x2，无超线程，支持fma, avx2，比赛实际运行平台

02.应用程序运行的软件环境


```
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 20.04.2 LTS
Release:        20.04
Codename:       focal
```

```
LSB Version:    :core-4.1-amd64:core-4.1-noarch:cxx-4.1-amd64:cxx-4.1-noarch:desktop-4.1-amd64:desktop-4.1-noarch:languages-4.1-amd64:languages-4.1-noarch:printing-4.1-amd64:printing-4.1-noarch
Distributor ID: CentOS
Description:    CentOS Linux release 7.9.2009 (Core)
Release:        7.9.2009
Codename:       Core
```

开发/编译平台：
系统：WSL2 Ubuntu 20.04 LTS
Gcc：9.3.0
Glibc：2.31

运行平台：
系统：CentOS Linux 7.9
Gcc：4.8.5 -> 10.2.0 (module load)
Glibc：2.17

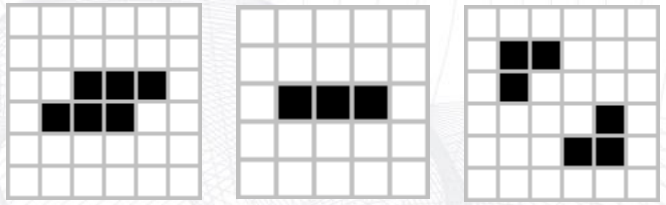
03. 应用程序简介

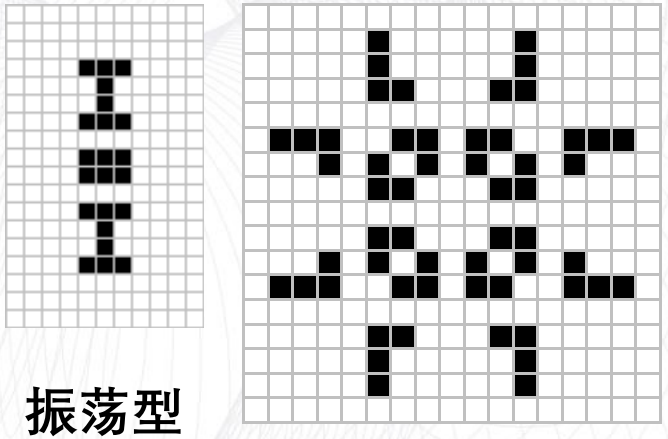
生命游戏（Game of Life），全称为 John Conway's Game of Life，是英国数学家约翰·康威在 1970 年代所发明的一种**元胞自动机**。

所谓元胞自动机其实是一种离散的状态机，即无数个独立的格子，每个格子处于某种状态，然后所有格子按照预先设定好的规律进行状态演化。格子们可以是任意维度、任意形状、按任意规律排布的。而生命游戏就是最简单的元胞自动机之一：在二维平面上的方格子（细胞），**每个细胞有两种状态：存活或死亡**，而下一回合的状态完全受它周围 8 个细胞的状态而定。**按照以下三条规则进行演化**：

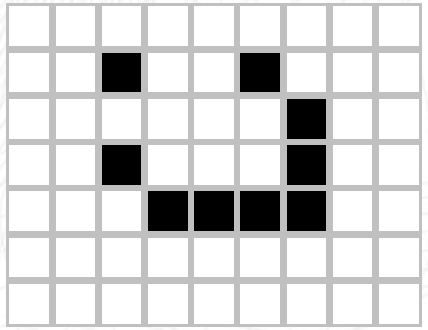
1. 活细胞周围的细胞数如果**小于 2 个或多于 3 个则会死亡**；（离群或过度竞争导致死亡）
2. 活细胞周围如果**有 2 或 3 个细胞可以继续存活**；（正常生存）
3. 死细胞（空格）周围如果恰好**有 3 个细胞则会诞生新的活细胞**。（繁殖）

这三条规则简称 **B3/S23**。





振荡型



整体移动型

03. 应用程序简介——原始结构

readPattern

gettimeofday (&start, NULL);

runConwayLifeGame(iterations)

output

gettimeofday (&end, NULL);

```
for (int i = 1; i < N - 1; ++i) {
  for (int j = 1; j < N - 1; ++j) {
    // 主要区域，计数周边有多少个 1
    int cnt = 0;
    for (int k = 0; k < 8; ++k) {
      int ii = i + dy[k];   int jj = j + dx[k];
      if (a[ii][jj] == 1)     cnt++;
    }
    // 存活，周边有 2 或 3 个，则继续存活
    if (a[i][j] == 1) {
      if (cnt == 2 || cnt == 3)  tmp[i][j] = 1;
      else                tmp[i][j] = 0;
    }
    // 死亡，周边有 3 个，则复活
    else {
      if (cnt == 3)   tmp[i][j] = 1;
      else                tmp[i][j] = 0;
    }
  }
}
```

03. 应用程序简介——优化思路

矩阵分块 -> L1/L2 Cache 匹配（未使用）

多线程/多进程并行化（没消除 barrier)

异步通信（未使用）

向量化及 blendv 代替 if（未使用）

bitMap（变形版本）

就地运算

局部多次迭代迭代

查表

识别死亡分块：若一个区域及其周边宽度为n的区域上一轮未更新，则接下来n次迭代也不会更新（未使用）

hashlife（未使用）

```
for (int i = 1; i < N - 1; ++i) {
  for (int j = 1; j < N - 1; ++j) {
    // 主要区域，计数周边有多少个 1
    int cnt = 0;
    for (int k = 0; k < 8; ++k) {
      int ii = i + dy[k];   int jj = j + dx[k];
      if (a[ii][jj] == 1)    cnt++;
    }
    // 存活，周边有 2 或 3 个，则继续存活
    if (a[i][j] == 1) {
      if (cnt == 2 || cnt == 3)  tmp[i][j] = 1;
      else                tmp[i][j] = 0;
    }
    // 死亡，周边有 3 个，则复活
    else {
      if (cnt == 3)   tmp[i][j] = 1;
      else                tmp[i][j] = 0;
    }
  }
}
```

03. 应用程序简介——优化思路

矩阵分块 -> L1/L2 Cache 匹配（未使用）

多线程/多进程并行化（没消除 barrier)

异步通信（未使用）

向量化及 blendv 代替 if（未使用）

bitMap（变形版本）

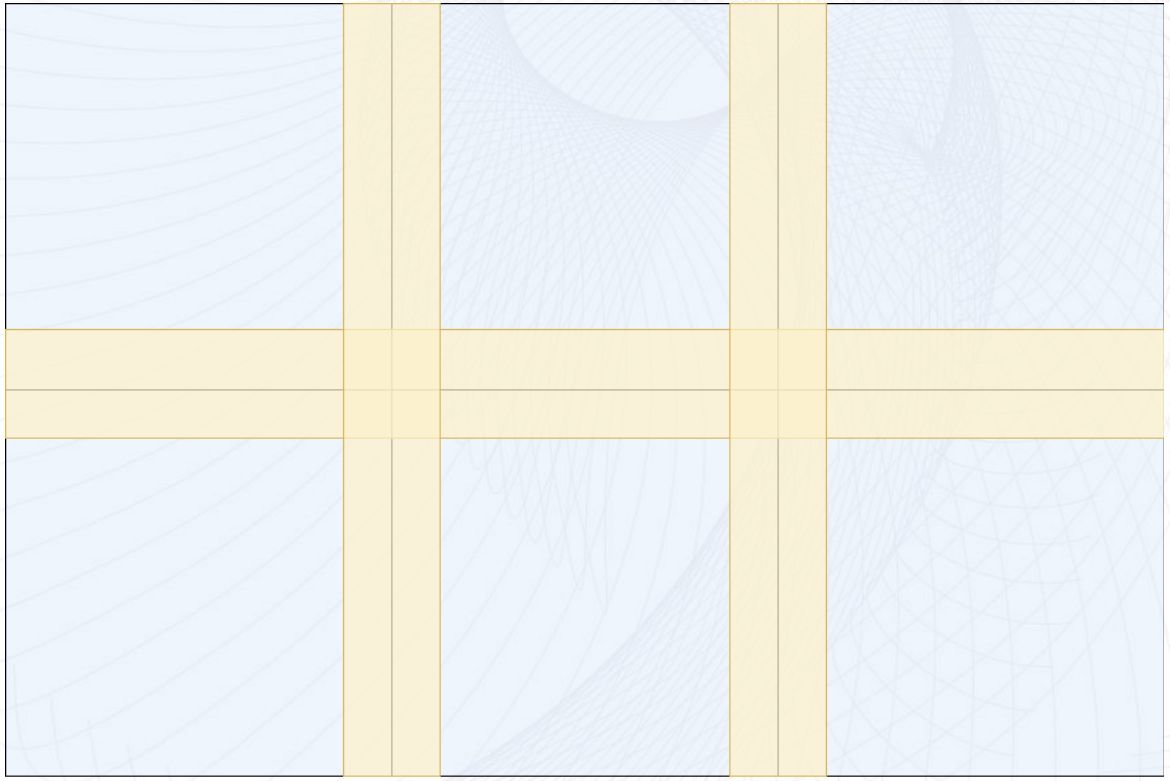
就地运算

局部多次迭代迭代

查表

识别死亡分块：若一个区域及其周边宽度为n的区域上一轮未更新，则接下来n次迭代也不会更新（未使用）

hashlife（未使用）



03. 应用程序简介——优化结构

readPattern

gettimeofday (&start, NULL);

runConwayLifeGame(1)

convert to bitArray

---------->

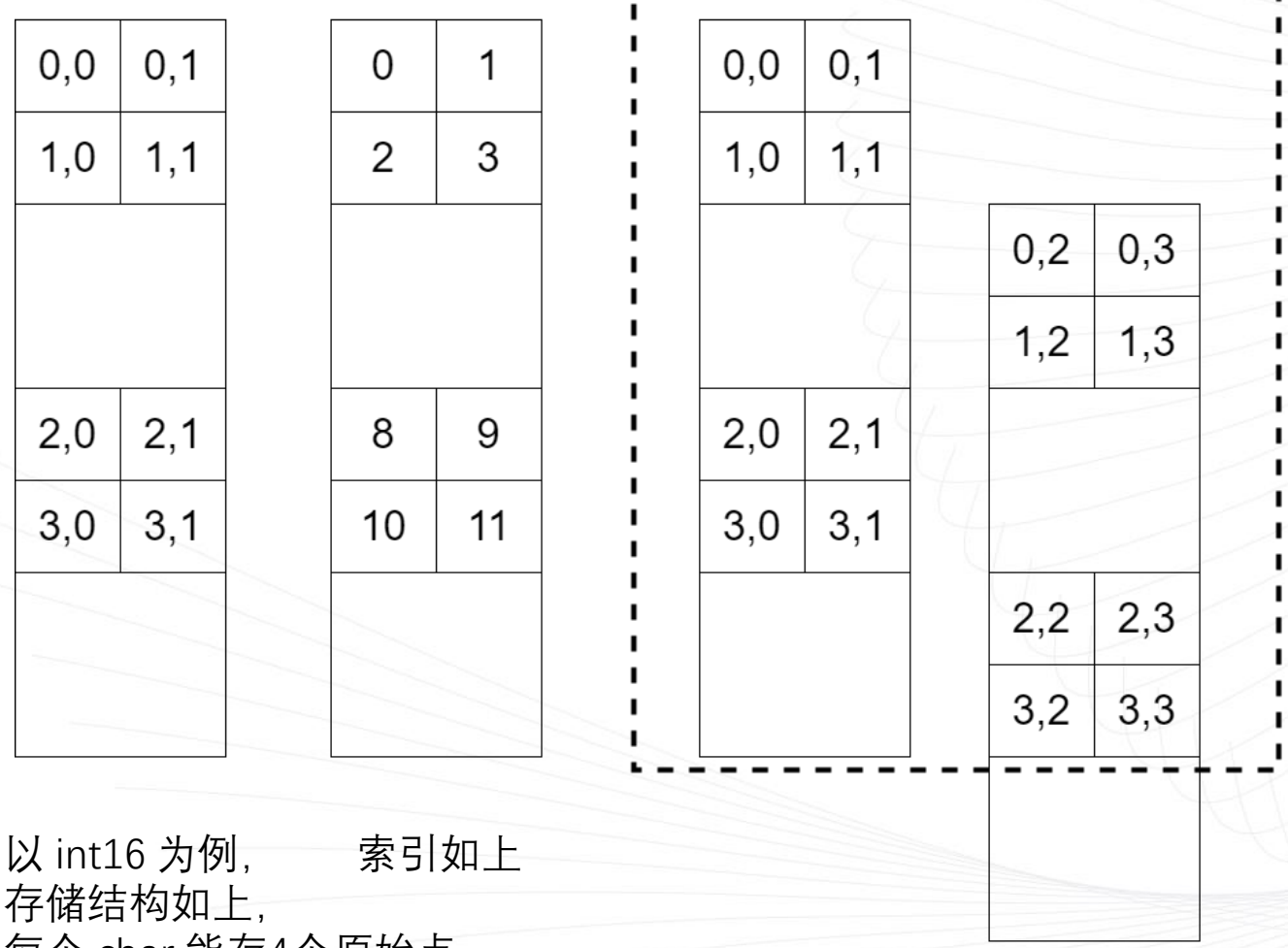
convert from bitArray

output

gettimeofday (&end, NULL);

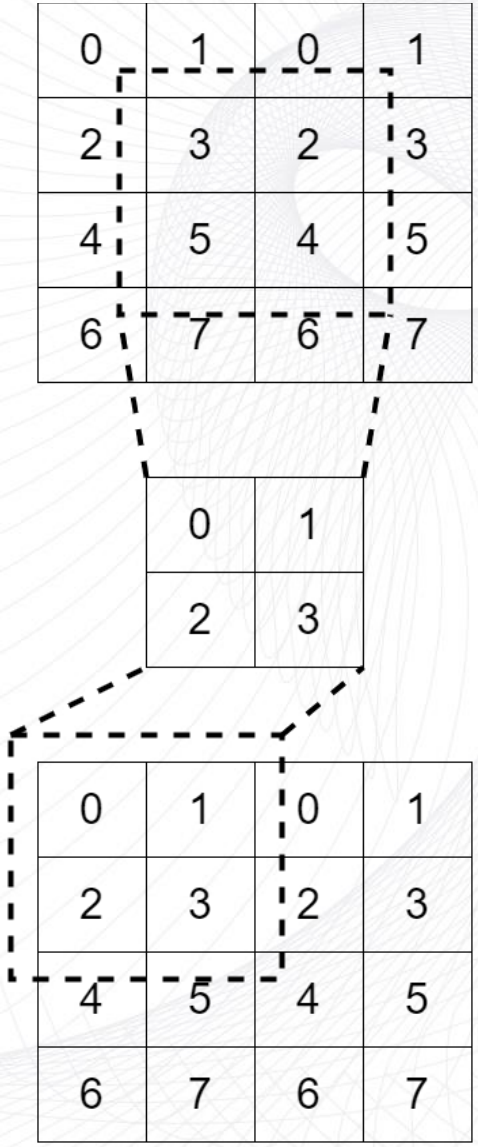
```
for (size_t st = 0; st < steps_times; st++) {
#pragma omp parallel for collapse(2)
   for (int i = 0; i < grid_len; i++)
     for (int j = 0; j < grid_len; j++)
        copy edge
#pragma omp parallel for collapse(2)
   for (int i = 0; i < grid_len; i++)
     for (int j = 0; j < grid_len; j++)
        kernel
}

template <bool is_left, bool is_up,
                bool is_right, bool is_down, size_t smax>
void kernel(...) {
  for (size_t s = 1; s <= smax; s++)
    for (size_t i = 1; i <= height - 2; i++)
      for (size_t j = 0; j < width - s; j++)
        mCount0...   // cal a uint64 (32 points)
  copy with offset
  return;
}
```

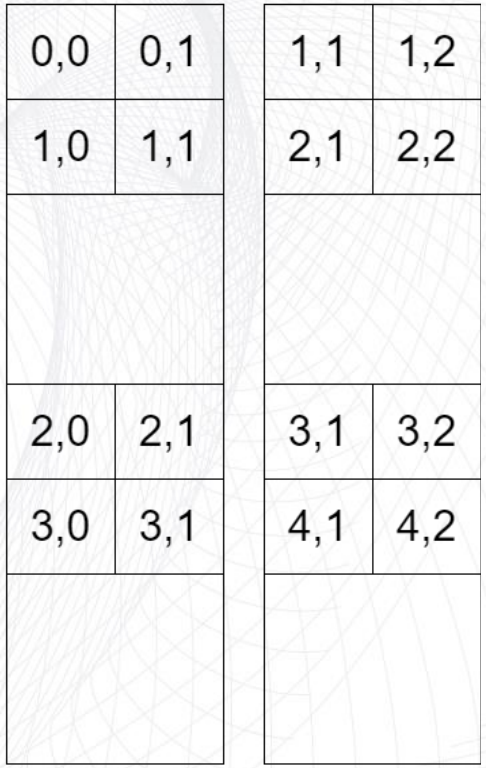
04.优化方法——局部数据结构



以 int16 为例，
存储结构如上，
每个 char 能存4个原始点

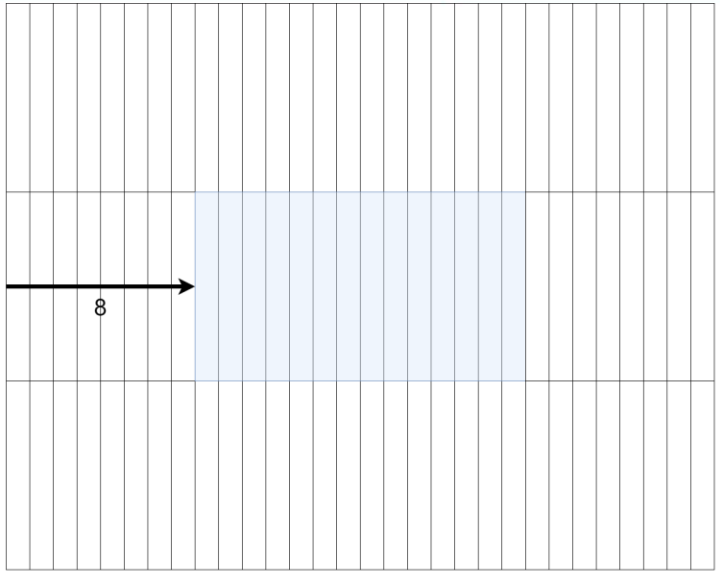
索引如上



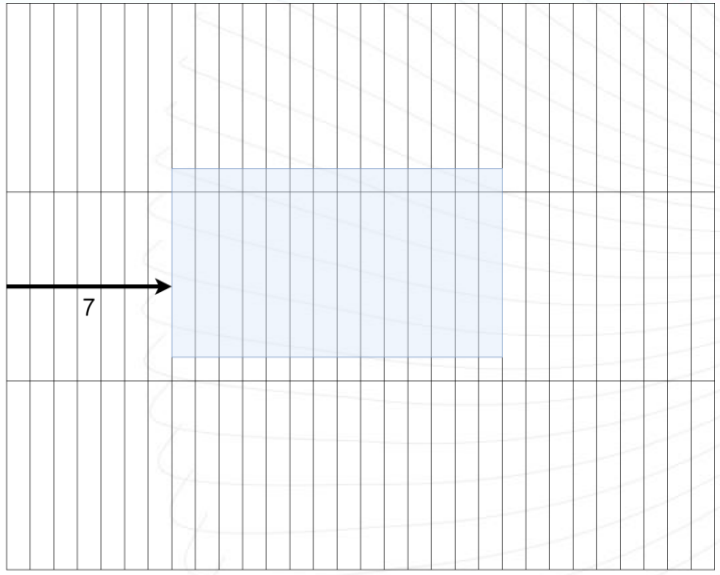


每次由16个点计算得到4个点
计算方式为查表
每次均为就地计算，
会造成往左上方的**偏移**！！

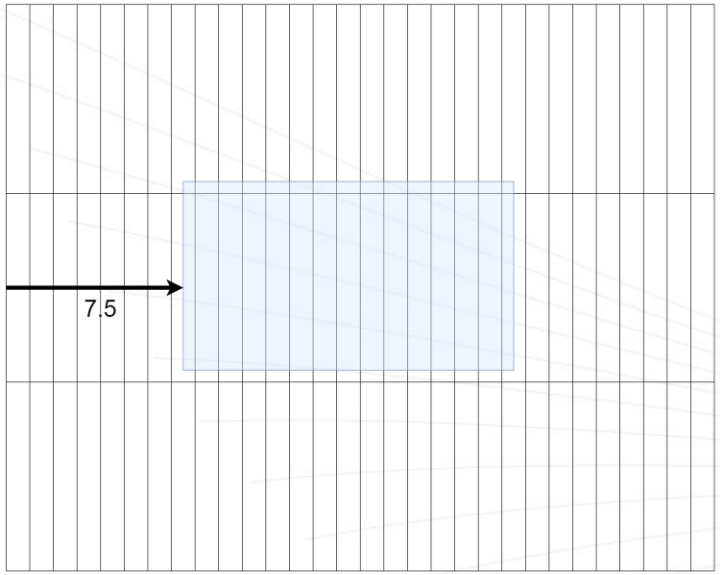
04.优化方法——分块数据结构



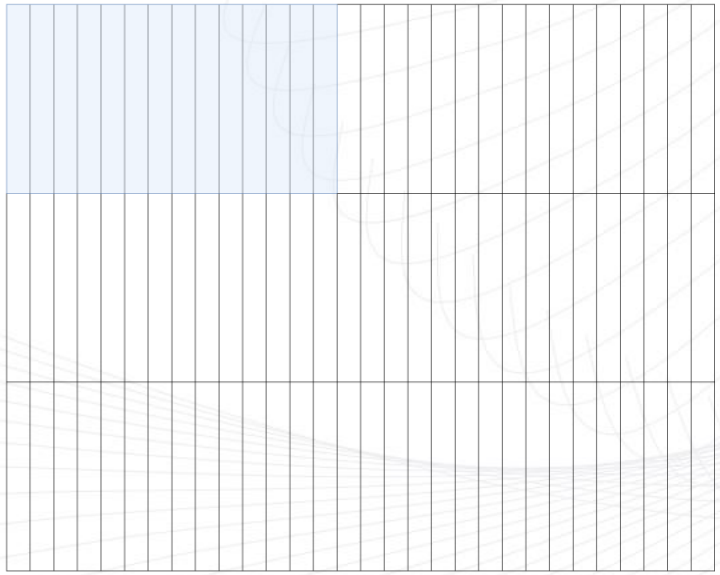
0 iter



2 iter

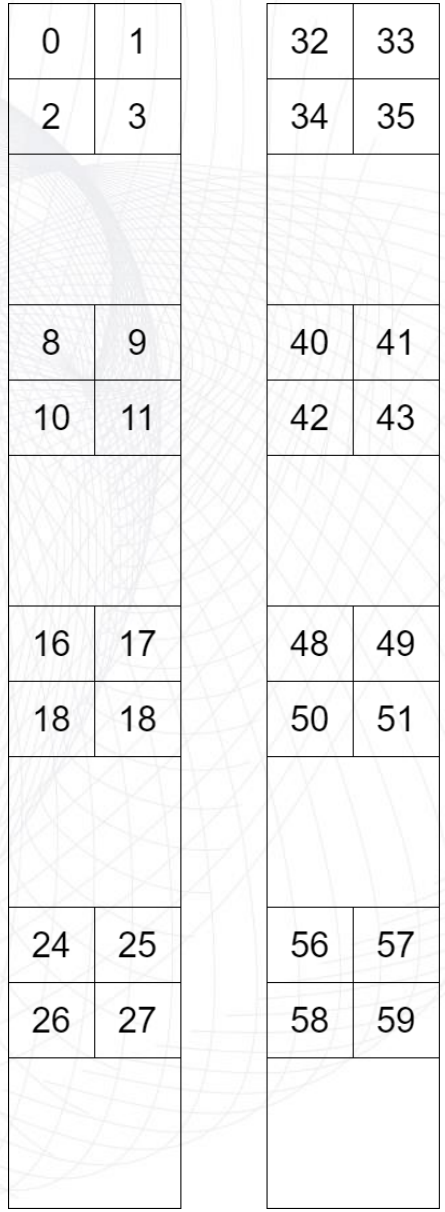


1 iter



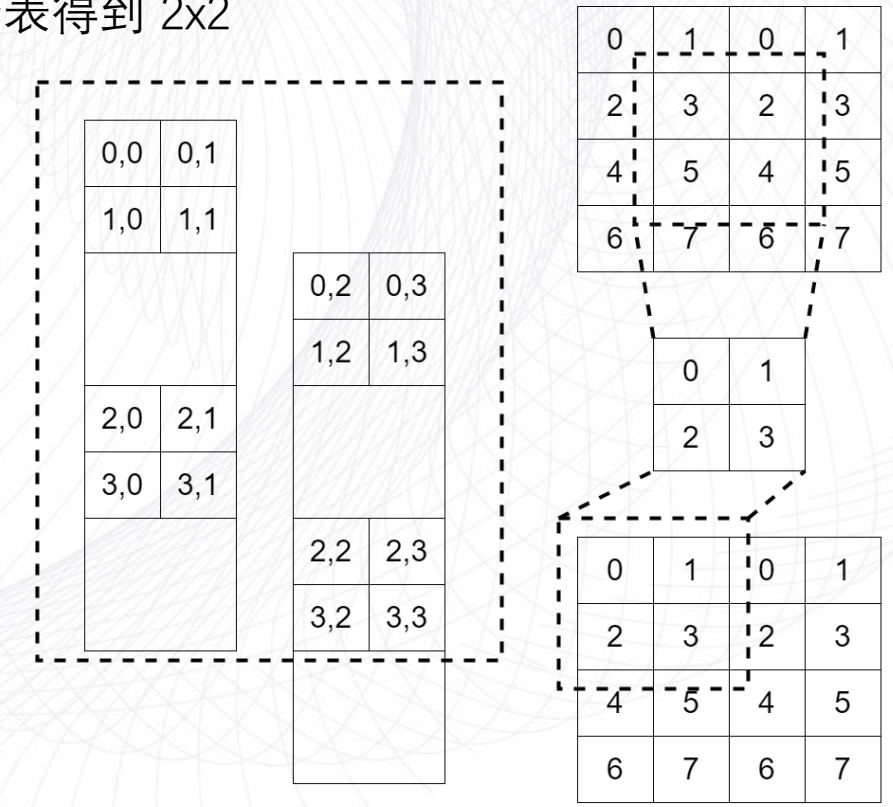
16 iter

- 每个分块类似左图，由若干个uint64组成
- 每次迭代时，由于数据结构与就地运算的原因，会造成往左上方的偏移。
- 同时，为了每一块都能迭代多轮，以便运用 **L1 Cache**
- 每个分块的上下左右各有相当于 **16 个点**的区域，该区域即是为了读取周边分块的数据，也是供迭代进行偏移的空间
- **16次迭代**完成后，如右下角，需要手动**拷贝**为左上角的状态



04.优化方法——核心计算


```
inline u_int64_t mCount0(u_int64_t **bitArray0, u_int8_t patternMap[], size_t i,
                         size_t j) {
  u_int64_t tmp = bitArray0[i][j] | (bitArray0[i][j + 1] << 4);          // 错位拼接
  unsigned char cans[8] = {0};
  for (size_t ss = 0; ss < (sizeof(u_int64_t) - 1); ss++) {
    cans[ss] = patternMap[tmp & 0xFFFF];        // 每次取低16位，即 4x4 查表得到 2x2
    tmp = tmp >> 8;                             // 平移8位，即计算两行后
  }
  //  tmp 只剩低8位了 （原来的56:64）
  u_int16_t tmp1 =
      bitArray0[i + 1][j] | (bitArray0[i + 1][j + 1] << 4); // 取低8位
  cans[7] = patternMap[(tmp | (tmp1 << 8)) & 0xFFFF];
  u_int64_t *pans = (u_int64_t *)cans;
  return *pans;
}
```



04.优化方法——核心计算


```
template <bool is_left, bool is_up, bool is_right, bool
is_down, size_t smax>
void forwardN_bound(u_int64_t **bitArray0, u_int8_t
patternMap[], size_t width,
                    size_t height, size_t bwidth, size_t bheight) {
  for (size_t s = 1; s <= smax; s++) {
    { // 最上面一行
      size_t i = 0;
      size_t jstart = 0;
      DEAL_LEFT_UP
      for (size_t j = jstart; j < width - s; j++) {
        bitArray0[i][j] = mCount0(bitArray0, patternMap, i, j);
        if constexpr (is_up)
          bitArray0[i][j] = clear_up(bitArray0[i][j], s);
        DEAL_RIGHT
      }
    }
```

- 其他有点用的技术：模板元
- 边界检查的 if else 在实际代码中会造成流水线问题，其实只有边上的分块需要边界检查，这是运用模板元技术与编译期分支，可以为不同的边界情况单独生成代码。配合宏可以做到边界处理良好、性能良好且减少代码量。

```
    for (size_t i = 1; i <= height - 2; i++) {
      // 主要部分
      size_t jstart = 0;
      DEAL_LEFT
      for (size_t j = jstart; j < width - s; j++) {
        bitArray0[i][j] = mCount0(bitArray0, patternMap, i, j);
        DEAL_RIGHT
      }
      if constexpr (is_down) {
        if ((i + 1) * 16 > 16 - s + bheight) {
          u_int64_t tmp;
          if ((16 - bheight % 16 + s) % 2 == 0) {
            tmp = 0x0F0F0F0F0F0F0F0F;
          } else {
            tmp = 0x030F0F0F0F0F0F0F;
          }
          if ((16 - bheight % 16 + s) == 16)
            tmp = 0;
          tmp >>= ((16 - bheight % 16 + s) % 16 / 2 * 8);
          for (size_t j = 0; j < width; j++) {
            bitArray0[i][j] &= tmp;
          }
          break;
        };
```

05.程序运行结果

AMD EPYC 7452		
	单位：ms	加速比
Base	109712	1.00
Base -O3	8966	12.24
Optmize	3448	31.82
Optmize -O3	712	154.10
Optmize -O3 parallel 64	41	2675.90