GPU を用いた N-Queens 問題の求解

田中 慶悟　　藤本 典幸
{mu301008@edu, fujimoto@mi.s}.osakafu-u.ac.jp
大阪府立大学　大学院理学系研究科　情報数理科学専攻

近年，汎用計算ができるようになった GPU 上で CUDA を用いて，Somers の高速な N-Queens 問題求解アルゴリズムをさらに高速化する手法を提案する．提案手法は N-Queens 問題を Somers のアルゴリズムで計算可能かつ独立な部分問題の集合に CPU 上で分割し，生成した部分問題を GPU の VRAM 上へと転送し，各スレッドへ動的に割り当て，効率よく並列計算を行う．評価実験を行ったところ，NVIDIA GeForce GTX480 と 2.93 GHz Intel Core i3 CPU を用いた場合，提案手法は Somers のアルゴリズムと比べ N＝19 で 24.5 倍高速であった．また，GPU を用いた Feinbube らの既存手法に比べ，提案手法は 2 倍高速であった．

Solving the N-Queens Problem on a GPU

Keigo TANAKA and Noriyuki FUJIMOTO
Department of Mathematics and Information Sciences,
Graduate School of Science, Osaka Prefecture University

In recent years, GPUs have acquired the ability to perform general purpose computation. In this paper, for CUDA GPUs, we propose a method to parallelize Somers' fast sequential algorithm to solve the N-Queens problem. The proposed method partitions a given instance of the N-Queens problem into sub-problems which can be computed in parallel. Then, the sub-problems are sent to VRAM on a GPU so that each sub-problem is dynamically allocated to a thread. Experimental results on an NVIDIA GeForce GTX480 and a 2.93 GHz Intel Core i3 CPU show that the proposed method solves the 19-Queens problem 24.5 times faster than Somers' sequential algorithm and that the proposed method is twice faster than Feinbube et al.'s existing method for CUDA GPUs.

1. はじめに

N－Queens 問題とは，N×N のチェス盤上で，互いに攻撃し合わないような N 個のクイーンの配置の総数を求める問題である．

N-Queens 問題を解く高速な CPU 用アルゴリズムとしては Somers のアルゴリズム[1]が知られている．Somers のアルゴリズムは最上行から最下行まで一行ずつクイーンを配置していき，しらみつぶしに可能な配置を調べていく．このとき，クイーンを配置した各行に対して，前行までのクイーンの右斜め下，下，左斜め下への効き情報とクイーンを置ける列の情報を保持し，これらの情報をビット演算で高速に処理する．Somers のアルゴリズムでは，さらに解の対称性を利用して，最上行の中央までの解を探索，2 倍して解の総数を求める．ただし，N が奇数の際にはクイーンが最上行の中央列に配置される場合とそれ以外で場合分けをする．

最上行から下の行に順にクイーンを置いていくとき，最初の数個のクイーンの配置を入力として，残りのクイーンを配置することにより得られる解の数をカウントする処理を一つのタスクとすると，それぞれのタスクは並列に計算できる．吉瀬らはこのタスク分割手法に基づいて PC クラスタを用いて 24-Queens 問題を解いている[2]．

現在， Somers の N-Queens 問題求解アルゴ

リズムを基として, PC クラスタや P2P グリッド, FPGA を用いた並列処理[2-4]により，N＝26 まで解が求められている．N-Queens 問題では，クイーンの数 N が大きくなるにつれて解の数が膨大となることが知られており, N=27 の解を求めるにはより一層の高速化が求められる．

本論文では，CUDA [5]対応 GPU を用いて，Somers の N-Queens 問題求解アルゴリズムを高速化する手法を提案する．提案手法は，吉瀬らによるタスク分割手法[2]を参考に CPU 上でタスク分割を行い，これらのタスクを GPU 上で並列に処理する．そのとき，共有メモリを効率よく使用し, タスクをスレッドに動的に割り当てることで，高速化を実現する．

以降の論文の構成は次の通りである. 第 2 章で GPU を用いた既存アルゴリズムについて簡単に説明する．第 3 章で提案手法の詳細を示す．第 4 章で評価実験について述べる. 第 5 章でまとめと今後の課題について述べる. スペースの制限のため，本論文では GPU のアーキテクチャおよびプログラミングの概要については述べない. これらに不慣れな読者は文献[5-10]を参照されたい.

2. CUDA を用いた関連研究

我々の知る限り，CUDA による N-Queens 問題求解の既存研究は文献[11,12]のみである．このうち CPU よりも高速な実装を実現しているのは文献[11]のみであるので，本論文では文献[11]の CUDA 実装を比較対象として 4 章で比較実験を行う. 本章では提案手法とのちがいについて明確にするために，文献[11]の CUDA 実装について説明する．

文献[11]では提案手法と同様に Somers の N-Queens 問題求解アルゴリズムをベースとし，同様のタスク分割手法を用いている．しかし，提案手法と文献[11]の手法には以降に述べるちがいがある．

文献[11]では，タスク分割の際，最初に配置するクイーンの個数，つまり，最初にクイーンを配置した行数を depth と呼んでいる．一つのタスクの入力は data[0, 1, …, depth－1]である．ここで data[i]は k ビット目のビットを 1 に，その他のビットを 0 にすることにより，i 行目のクイーンの位置が k 列目であることを表している. そして，タスクの数だけスレッドを起動し，一つのスレッドに対して一つのタスクを割り当てる. 一方，提案手法では，一つのスレッドが複数のタスクを計算する．さらに，その際，スレッド間の負荷バランスを考慮して, 処理するタスクをスレッドに動的に割り当てる.

タスクの入力として，文献[11]では，スレッド毎に depth 個のクイーンの位置情報が必要であり，それを単純に一つの配列上に並べて，GPU の VRAM 上へ転送している．スレッド数はタスク数と等しくなるように定義するので，VRAM に転送するデータ量は，(タスク数)×(depth)×sizeof(int)となる. それに対して, 提案手法では，depth 桁の N 進数を用いることにより，depth 個のクイーンの位置情報を一つの int に圧縮してから，一つの配列にまとめる．そのため，CPU から GPU の VRAM 上へ転送するデータ量は(タスク数)×sizeof(int)となり，提案手法は文献[11]に比べ転送量を 1/depth に削減している．また，この入力データ量削減は，GPU の VRAM から共有メモリへのデータ転送量の削減にもつながる.

文献[11]では，Somers のプログラムと同様にスタックを用いているが, 提案手法ではループを用いて再帰呼び出しを除去している.

最適化として文献[11]では，各タスクが使用するローカル配列を高速な共有メモリ上に配置している．共有メモリは 32 個または 16 個のバンクに分かれており, 複数のスレッドが同時に同じバンクにアクセスするバンク衝突を起こすとアクセス速度が低下するが，文献[11] では，バンク衝突に対する配慮がなく, バンク衝突が生じるアルゴリズムとなっている. これに対して提案手法では共有メモリ上のローカル配列のレイアウトを工夫し，バンク衝突を完全に避けている.

3. 提案手法

3.1 タスクの入力データの符号化

最初の m 行のクイーンの可能な配置の 1 つを m 桁の N 進数 $p_1p_2\cdots p_m$ で表す．すなわち，p_i は i 行目のクイーンを p_i 列目に配置することを表す． $1 \leqq m \leqq N$ であり，未解決の N-Queens 問題で問題サイズが最小であるのは 27-Queens 問題であることを考えると，N は高々30 程度であると仮定してよい．このため 32 ビットプロセッサを用いる場合, 提案手法における各タスクの入力は一つの int となる.

文献[11]のように VRAM に送る情報が，m 行目までのクイーンの右斜め下，下，左斜め下への効き情報とクイーンを置ける列情報でなく, N 進

数に符号化する理由は，CPU と GPU 間および VRAM と共有メモリとの間のデータ転送量を減らすためである．

3.2 タスク分割

Somers のアルゴリズムと同様に，本研究でも N-Queens 問題の解の対称性を利用する．N が偶数ならば，1 行目のクイーンが配置できる列はチェス盤の (N / 2) 列目までとして解の総数をカウントし，その総数の 2 倍の値を N-Queens 問題の解とする．同様に，N が奇数ならば，1 行目のクイーンが配置できる列はチェス盤の ((N－1) / 2) 列目までとした場合と，1 行目のクイーンは((N＋1) / 2) 列目に配置し 2 行目のクイーンが配置できる列はチェス盤の((N－1) / 2) 列目までとした場合の両方の場合を探索する．このようにすると，探索によって発見された任意の解と，チェス盤を左右反転した対称解は，1 行目または 2 行目のクイーンの位置が必ず異なり，クイーンの配置が同じになることはなく，半分の空間の探索で N-Queens 問題を解ける．

これらの条件を満たす m 行のクイーンの可能な配置をもれなく見つけるため，0 から(N^m/2-1) までのすべての数のそれぞれを N 進数とみなしたときに，その N 進数が表すクイーンの配置が可能であるかを判断し，可能であれば，その数をタスクの入力データの集合に加える．

N 進数が表すクイーンの配置が可能であるかの判断は次のように行う．N 進数をもとに各行にクイーンを配置し，Somers のアルゴリズムを参考に前行目までのクイーンの右斜め下，下，左斜め下への効きにクイーンが配置されているかを調べる．これを m 行目まで調べ，各行で，前行までのクイーンの右斜め下，下，左斜め下への効きにクイーンが配置されていなければ，そのクイーンの配置は可能であり，1 行でも効きにクイーンが配置されていれば，そのクイーンの配置は不可能であると判断する．

タスク分割の疑似コードを図 1 に示す．

```
入力：問題サイズ N，最初に配置するクイーンの行数 m，十分な大きさの配列 task
出力：生成されたタスクの数 nTasks，nTasks 個のタスク task[0..nTasks-1]
int genTasks (int N, int m, int task [ ]){
int pos[m], col[m], neg[m];
int nTasks = 0;
for(int number = 0; number < Nm / 2; number++){
  int arrange = number;
  // number に対応するクイーンの配置を復号
  for(int i = m - 1; i >= 0 ; i--){
    pos[i] = col[i] = neg[i] =1<<(arrange%N);
    arrange /= N;
  }
  // クイーンの配置が正しいかどうかチェック
  int i;
  for(i = 1; i < m; i++){
    if(((pos[i-1]>>1)&col[i])||(col[i-1]&
        col[i])||((neg[i-1]<<1)&col[i]))
      break; // 正しくなかった時
    pos[i] |= (pos[i-1]>>1);
    col[i] |= col[i-1];
    neg[i] |= (neg[i-1]<<1);
  }
  if(i == m)
    task [nTasks++] = number; // 正しかった時
 }
return nTasks;
}
```

図 1　タスク分割の疑似コード

3.3 最初に配置するクイーンの行数

N と m により生成されるタスクの個数を表 1 に示す．表 1 に示すように，生成されるタスクの個数は m によって指数的に増大する．このため m を大きくすると，動的にタスクを割り当てる処理にかかる時間が増大し，さらに，タスクの入力データである，クイーンの配置を符号化した N 進数から m 行目までのクイーンの右斜め下，下，左斜め下への効き情報とクイーンを置ける列の情報の復号にかかる時間も増大する．逆に m を小さくすると，生成されるタスクの数が GPU の性能を発揮するのに必要なスレッド数に満たなくなり，全体の実行時間が低下する．m のちがいによる全体の実行時間の差異を測定した予備実験の結果を表 2 に示す．以上の考察から本論文では m = 5 を採用した．

表 1　生成されるタスクの個数

N \ m	3	4	5	6
15	882	6990	44714	231519
16	1118	9844	70906	419408
17	1393	13510	108466	724001
18	1710	18132	160850	1199146
19	2720	23866	232174	1916187

表 2　m による全体の実行時間の差異(s)
(NVIDIA GeForce GTX560 Ti を使用)

N \ m	3	4	5	6
15	0.4743	0.1187	0.1311	0.4117
16	2.2142	0.4864	0.4615	0.7059
17	4.9520	3.0319	2.8333	3.4173
18	33.7653	21.241	20.1843	20.9399
19	227.9136	162.475	156.3656	156.6231

3.4　ホストコード

GPU の kernel 関数を呼び出す CPU のルーチンを図 2 に示す．CPU は最初の m 行の可能なクイーンの配置のすべてを N 進数で主記憶上に求め，これらのタスク情報を一度にまとめて GPU の VRAM へと転送する．

3.5　最適化(バンク衝突の防止)

1 ブロックあたりのスレッド数は 32 の倍数とし，共有メモリなどのリソース消費量を考慮して，1 ブロックのスレッド数が最大となり，かつ，すべてのマルチプロセッサにスレッドブロックが少なくとも 1 つ割り当たるようにブロック数とブロックあたりのスレッド数を設定する．

総スレッド数を一定にして，ブロック数と 1 ブロックあたりのスレッド数を変化させたときの提案手法の実行時間を表 3 に示す．表 3 より総スレッド数が等しければ，実行時間に差がないため，本研究では，共有メモリの消費量から総スレッド数が計算しやすい 1 マルチプロセッサあたり 1 ブロックを採用している．

各スレッドは，スレッド全体における自らのスレッド番号に応じて VRAM 上の配列 d_task から一つ目のタスクが割り当てられ，そのタスクの入力データを用いて m 行目までのクイーンの右斜め下，下，左斜め下への効き情報とクイーンを置ける列の情報をそれぞれ共有メモリ上の別の配列に求める．共有メモリ上の配列を図 3 のようにサイズ N×(1 ブロックあたりのスレッド数)の

表 3　スレッド数とブロック数の違いによる実行時間
(NVIDIA GeForce GTX560 Ti を使用)

N \ (ブロック,スレッド)	(8, 160)	(40, 32)
15	0.1379	0.1286
16	0.4615	0.4595
17	2.8333	2.8319
18	20.1843	20.1781
19	156.3657	156.3528

```
typedef unsigned long long ULL;
ULL nTasks;   // タスクの数
int *task; // タスクの入力（CPU 上）
int *d_task; // タスクの入力(GPU 上)
ULL cnt; // 解の数（CPU 上）
ULL *d_cnt; // 解の数(GPU 上)
unsigned   counter; // 最初に割り当てたタスク数
unsigned *d_counter; // 割当済みタスク数(GPU 上)

// MP：マルチプロセッサの数
// BLKSZ： 1 ブロックあたりのスレッド数
counter = BLKSZ * MP;
nTasks = genTasks (N, m, task);

cudaMalloc((void**) &d_cnt, sizeof(ULL));
cudaMalloc((void**) &d_task,
        sizeof(int)* nTasks);
cudaMalloc((void**) &d_counter,
        sizeof(unsigned));
cudaMemset(d_cnt,0,sizeof(ULL));
cudaMemcpy(d_task,task ,sizeof(int)*
        nTasks,cudaMemcpyHostToDevice);
cudaMemcpy(d_counter, &counter,
        sizeof(unsigned),
        cudaMemcpyHostToDevice);

NQPart<<< MP, BLKSZ >>>(N, m, d_cnt,
        nTasks, d_task, BLKSZ, d_counter);

cudaMemcpy(&cnt,d_cnt, sizeof(ULL),
        cudaMemcpyDeviceToHost);
cudaFree(d_cnt);
cudaFree(d_task);
cudaFree(d_counter);

cnt *= 2;   // 解の対称性により 2 倍する
```

図 2　CPU 側の実装

配列として定義する．ブロック内での自らのスレッド番号を x，1 ブロックあたりのスレッド数を BLKSZ とすると，各スレッドは配列 A の要素集合{A[x＋k×BLKSZ] | 0≦k≦N－1}の部分を使用する．このようにすると，共有メモリは 32 個のバンクに分かれていて，連続するワードは連続するバンクに割り当てられるように構成されているため，1 ブロックあたりのスレッド数を 32 の倍数とすることで，各スレッドはそれぞれの配

```
// pos[ ]：右斜め下への効き
// col[ ]：下への効き
// neg[ ]：左斜め下への効き
// bitmap[ ]:クイーンの置ける列
__shared__ int pos[N*BLKSZ];
__shared__ int col[N*BLKSZ];
__shared__ int neg[N*BLKSZ];
__shared__ int bitmap[N*BLKSZ];
```

図 3　共有メモリ上の配列の定義

列に関して一つのバンクにだけアクセスすることとなり, バンク衝突によって生じる遅延を防げる.

Fermi 以前の GPU アーキテクチャでは, 共有メモリは 16 個のバンクに分かれており, 1 ブロックのスレッド数を 16 の倍数とすれば共有メモリのバンク衝突を防止できる.

GPU アーキテクチャによって共有メモリのバンク数は異なるため, バンク数にあわせて 1 ブロックあたりのスレッド数を設定する必要がある.

3.6　最適化(タスクの動的割当)

生成される個々のタスクの計算時間は, m に関わらず大きく差がある. また各タスクの計算時間を予め見積もることは容易ではない. このため各スレッドに割り当てられる仕事量が可能な限り均一になるように, 各スレッドにタスクを動的に割り当てる.

タスクの動的割り当てを実現するために, 二つ目以降のタスクは, 自分に割り当てられたタスクが終了したとき, 新たに VRAM 上の配列 d_task[]からタスクの入力を取得し計算することとする. すべてのタスクの入力は d_task[]に保持されており, d_task[]にはすべてのスレッドがアクセスできる. 次に計算されるタスクの管理をする変数*d_counter を用意する. この変数*d_counter には, 現在実行されているタスクの d_task[]の添字の最も大きい値より 1 大きい値が代入されている. タスクの総数はわかっているので, 変数*d_counter の値がタスク総数より小さければタスクが残っており, そうでなければ, タスクが残っていないと判断できる. タスクが残っていれば, スレッドが新たにタスクを取得したのち, *d_counter の値を 1 大きくし, タスクが残っていなければ, 終了する.

GPU 側の計算カーネルを図 4 に示す.

```
__global__ void NQPart(int N, int m, ULL* d_cnt,
    ULL nTasks, int* d_task,
    int BLKSZ, unsigned *d_counter){
unsigned anumber =
          blockIdx.x * blockDim.x + threadIdx.x;
           // スレッド全体におけるスレッド番号
unsigned number = threadIdx.x;
           // 1 ブロック内におけるスレッド番号
unsigned cnt = 0;  // cnt：解の数のカウント
unsigned c;        // c：次のタスク情報の番号
int x = number+m*BLKSZ;  //現在の行
int mask = (1 << N) - 1;
// 割り当てられたタスクの入力から,
// bitmap, pos, col, neg を生成
initialize(N, m, number, bitmap, pos, col,
        neg, mask, BLKSZ, d_task, anumber);
for(;;) {
  if (bitmap[x]) {
    unsigned bit = -bitmap[x] & bitmap[x];
    bitmap[x] ^= bit;
    if (x == number + (N-1)*BLKSZ) cnt++;
    else {
      x += BLKSZ;
      pos[x] = (pos[x - BLKSZ] | bit) >> 1;
      col[x] = col[x - BLKSZ] | bit;
      neg[x] = (neg[x - BLKSZ] | bit) << 1;
      bitmap[x] = mask &
                ~(pos[x]) & ~(col[x]) & ~(neg[x]);
  } }
  else {
    x -= BLKSZ;
    if (x < number + m *BLKSZ){
      c = atomicAdd(d_counter, 1);
      if(c < nTasks){
        // 次のタスクの入力を受け取り,
        // bitmap, pos, col, neg を生成
        initialize(N, m, number, bitmap,
          pos, col, neg, mask, BLKSZ, d_task, c);
        x = number+m *BLKSZ;
      }
      else{
        atomicAdd(d_cnt, cnt);
        return;
}}}}}
```

図 4　GPU の計算カーネル

このように動的にタスクを割り当てることにはいくつかのメリットとデメリットがある.

メリットは，1 スレッドが定数個のタスクを静的に割り当て計算する場合，あるスレッドに割り当てられたタスクが早くに終了したとき，そのスレッドは同じスレッドブロック内で同時実行されるその他のスレッドが全て終わるまで待機していることとなるが，動的に割り当てる場合，割り当てるタスクがなくなるまで，すべてのスレッドはタスクの処理を実行することとなり，計算性能の無駄が減少することである．また，動的に割り当てることにより，各スレッドの仕事量を均一に近づけるというメリットもある．

デメリットは，CUDA 対応 GPU ではスレッドは warp 毎に SIMD 実行されるため，スレッドがタスクを割り当てられ，その入力から m 行目までのクイーンの右斜め下，下，左斜め下への効き情報とクイーンを置ける列の情報を復号している間，同時実行されるその他のスレッドはアイドル状態となることである．そのため，タスクの数が大きくなれば，タスクの割り当てにかかる時間も大きくなり，スレッドのアイドル状態の時間の総量が増えることとなる．また，タスクの数は非常に大きいため，すべてのタスクの入力を共有メモリに保持することはできず，VRAM 上に保持しているため，タスクの入力を取得するには，通信速度の遅い VRAM とタスクの数だけ通信しなくてはならないこともデメリットである．

4. 評価実験

CPU は 2.93GHz Intel Core i3，GPU は NVIDIA GeForce GTX280，GTX480，GTX560 Ti，OS は 64bit Windows7 Professional， コンパイラは Visual Studio 2008 Professional，nvcc-3.2(CUDA3.2)，ディスプレイドライバは Ver280.26 を用いた． 共有メモリを GTX280 では 16KB，GTX480 と GTX560Ti では 48KB 用いた．

4.1 N-Queens 問題の求解時間

各 GPU の実験時のブロック数とスレッド数を表 4 に，提案手法の実行時間を表 5 に示す．

N の大きさによって 1 スレッドあたりの共有メモリの消費量が異なるため，1 ブロックあたりのスレッド数は N によっても個別に設定されるべきであるが，本論文では，CPU と GPU の実行時間，CPU に対する GPU の速度向上率を比較することで，最適化(バンク衝突の防止)と最適化(タスクの動的割当)の組み合わせによる効果を明らかにするため，N = 15～19 において，ブロック数とスレッド数をすべて統一し，N =17,18,19 で最適となるように設定した．N = 15,16 のとき，1 ブロックあたりのスレッド数は GTX280 では 64 スレッド，GTX480 と GTX560 では 192 スレッドと設定するのが最適である．

表 4　実験時のブロック数・スレッド数

	ブロック数	スレッド数 / ブロック
GTX280	30	48
GTX480	15	160
GTX560	8	160

表 5　提案手法の実行時間(s)

N	CPU	GTX280	GTX480	GTX560
15	0.877	0.171	0.116	0.131
16	5.816	0.625	0.327	0.461
17	40.773	3.714	1.769	2.833
18	301.139	26.664	12.355	20.184
19	2335.402	206.197	95.062	156.365

表 6　CPU に対する GPU の速度向上率

N	GTX280	GTX480	GTX560
15	5.12	7.56	8.43
16	9.30	17.78	15.31
17	10.97	23.04	17.56
18	11.29	24.37	18.14
19	11.32	24.56	18.18

CPU に対する GPU の速度向上率を表 6 に示す． CPU のプログラムは Somers のホームページ[1]で公開されている Somers のプログラムを使用し，実行時間を計測している．このプログラムはシングルスレッドの C 言語プログラムで，SSE 命令などは用いていない通常の逐次プログラムである．

GTX480 を用いる場合，N = 15 では，GPU は CPU の 7.5 倍程度の速度であり，N = 16 では，GPU は CPU の 17.8 倍程度の速度である．N = 17,18,19 では GPU は CPU の 23.0～24.5 倍程度の速度向上率となっており，N = 17,18,19 と比べ N = 15, 16 では速度向上率が低くなっている．他の GPU でも同様の傾向がみられる．

転送時間，タスク分割に要する時間，タスクの計算時間を表 7，実行時間全体に対するそれぞれの占める時間の割合を表 8 に示す．表 7，表 8 では GTX560 Ti を使用している．表 7 における全体時間には CUDA API ライブラリの初期化時間が含まれておらず，その時間は 0.02(s)程度である．N=17,18,19 に比べ N=15,16 では速度向上率が低い値となっているのは，表 7，表 8 が示すようにタスクの計算時間に比べ，タスク分割に要する時間，転送に要する時間，ライブラリの初期

表 7　各部分に要する時間(s)
(NVIDIA GeForce GTX560 Ti を使用)

N	転送	分割	計算	全体
15	0.025	0.016	0.064	0.124
16	0.026	0.012	0.395	0.454
17	0.034	0.030	2.756	2.840
18	0.031	0.038	20.085	20.175
19	0.030	0.053	156.286	156.388

表 8　実行時間全体に対する各部分の占める時間の割合
(NVIDIA GeForce GTX560 Ti を使用)

N	転送/全体	分割/全体	計算/全体
15	0.2016	0.1290	0.5161
16	0.0572	0.0264	0.8700
17	0.0119	0.0105	0.9704
18	0.0015	0.0018	0.9955
19	0.0002	0.0003	0.9993

表 9　最適化手法の性能比較(s)
(NVIDIA GeForce GTX560 Ti を使用)

N	最適化無	衝突防止	動的割当	衝突防止 動的割当
15	0.180	0.175	0.137	0.131
16	0.983	0.754	0.691	0.461
17	4.950	4.725	3.342	2.833
18	34.957	32.763	23.988	20.184
19	262.922	244.064	186.220	156.365

化時間の割合が大きいからであり, タスクの計算時間の割合が非常に大きい N = 17, 18, 19 では速度向上率はほぼ一定で，N = 20 以降では，一定の速度向上率になると推測できる.

Feinbube ら[11]は N=16 までの N-Queens 問題を GPU を用いて解いている．Feinbube らの既存 GPU プログラム[11]では，16-Queens 問題の求解時間は NVIDIA GeForce GTX275 では 1.6(s)，GTX295 では 1.1(s)となっている．Feinbube らの既存研究 [11]では行われていなかった「共有メモリのバンク衝突の防止」と「タスクの動的割り当て」を行うことにより，本研究では GTX275 とほぼ同性能の GTX280 を用いた場合，　2 倍以上の性能となっている.

4.2　提案する最適化の効果

最適化(バンク衝突の防止)と最適化(タスクの動的割当)の効果を評価する実験の結果を表 9 に示す.

最適化(バンク衝突の防止)では, N = 19 のとき, 最適化無しと最適化(バンク衝突の防止)を比較すると，最適化(バンク衝突の防止)単独での効果として 8%程度の速度向上がみられ，また，最適化(タスクの動的割当)と最適化(バンク衝突の防止)& 最適化(タスクの動的割当)を比較すると, 最適化(タスクの動的割当)と組み合わせることで, 最適化(バンク衝突の防止)の効果として 19%程度の速度向上がみられた.

最適化(タスクの動的割当)では, N = 19 のとき, 最適化無しと最適化(タスクの動的割当)を比較すると，最適化(タスクの動的割当)単独での効果として 41%程度の速度向上がみられた．また，最適化(バンク衝突の防止)と最適化(バンク衝突の防止)& 最適化(タスクの動的割当)を比較すると，最適化(バンク衝突の防止)と組み合わせることで，最適化(タスクの動的割当)の効果として 56%程度の速度向上がみられた．最適化(タスクの動的割当)ではデメリットもあるが，実験結果から, メリットのほうがより大きいと確認できる.

最適化無しと最適化(バンク衝突の防止)& 最適化(タスクの動的割当)を比較すると，N ＝ 19 のとき，最適化(バンク衝突の防止)& 最適化(タスクの動的割当)の効果として 68%程度の速度向上がみられた.

最適化(バンク衝突の防止)と最適化(タスクの動的割当)はそれぞれで速度向上がみられ，組み合わせると, さらに速度が向上したことから有効であると考えられる.

5.　まとめ

GPU を用いて N-Queens 問題を高速に解く手法を提案した．提案手法は，GPU を用いた既存手法に対して 2 倍，CPU を用いた Somers の手法に対して 24.5 倍高速である.

表 7 より N = 17 を求めるのに N = 16 の約 6.3 倍の時間, N = 18 を求めるのに N = 17 の約 7 倍の時間，N=19 を求めるのに N=18 の約 7.7 倍の時間が必要である. これらの増加率を参考に N = 20 以降も求めるのに必要な時間が増加していくと仮定すると，N = 27 を求めるのには，約 529.5 年必要となる. これはとても現実的な時間ではなく，N = 27 を求めるのには，さらなる高速化が必要となる.

今後は，GPU 上でも有効な枝刈りや，共有メモリの節約によるスレッド数の増加，GPU カーネルのさらなる効率化によって, さらなる速度の向上を目指す.

参　考　文　献

[1] J. Somers, The N Queens Problem: a Study in Optimization, http://www.jsomers.com/nqueen_demo/nqueens.html .

[2] 吉瀬謙二,片桐孝洋,本多弘樹,弓場敏嗣,PCクラスタを用いたN-queens問題の求解,電子情報通信学会論文誌 D-I, J87-D-I(12), pp.1145-1148, 2004.

[3] Inria Sophia Antipolis, nQueens: n=25, http://www-sop.inria.fr/oasis/ProActive2/apps/nqueens25.html .

[4] R. G. Spallek et al., Queens@TUD, http://queens.inf.tu-dresden.de/ .

[5] NVIDIA Corp., NVIDIA CUDA C Programming Guide, http://developer.nvidia.com/nvidia-gpu-computing-documentation, 2011.

[6] 青木尊之, 額田彰, はじめてのCUDAプログラミング, 工学社, 2009.

[7] D. B. Kirk and W. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Morgan Kaufmann, 2010.

[8] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, NVIDIA Tesla: A Unified Graphics and Computing Architecture, IEEE Micro, Vol.28, No.2, pp.39-55, 2008.

[9] J. Sanders and E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley Professional, 2010.

[10] M. Garland and D. B. Kirk, Understanding Throughput-Oriented Architectures, Communications of the ACM, Vol.53, No.11, pp.58-66, 2010.

[11] F. Feinbube, B. Rabe, V. L. Martin, and A. Polze, NQueens on CUDA: Optimization Issues, International Symposium on Parallel and Distributed Computing (ISPDC), pp.63-70, 2010.

[12] V. Pamplona, n-Queens Problem: A Comparison Between CPU and GPU using C++ and Cuda (Presentation), Universidade Federal do Rio Grande do Sul, 2008.