

GPU 計算を用いた並列進化計算による二次割当て問題の一解法 とその解析

Solving Quadratic Assignment Problems by GPU Computation and Its Analysis

筒井茂義^{1*} 藤本典幸²
Shigeyoshi Tsutsui¹ Noriyuki Fujimoto²

¹ 阪南大学大学

¹ Hannan University

² 大阪府立大学

² Osaka Prefecture University

Abstract: This paper proposes an evolutionary algorithm for solving QAPs with parallel independent run on GPUs and gives a statistical analysis on how the speedup can be attained with this model. With the proposed model, we achieved a GPU computation performance that is nearly proportional to the number of equipped multi-processors (MPs) in the GPUs. We explain these computational results by performing statistical analysis. Regarding performance comparison to CPU computations, GPU computation with a single GPU and two GPUs showed a speedup of x4.4 and x7.9 on average, respectively.

1 はじめに

進化計算の高性能化の手法として、大規模な並列計算機やネットワークで結合された多数の計算機を利用する並列進化計算の研究 [Cantú-Paz 00] が活発に行われてきた。今後、進化計算の利用がより多くの分野に広がって行くためには、多くのエンドユーザがより安価な計算機環境において実現できる高性能な並列進化計算の実現が重要になるとと思われる。

昨今、いわゆるグラフィックボードとして一般の PC に用いられている画像処理用演算装置 (Graphics Processing Unit, GPU) を用いて、流体力学、医療用画像分析、統計データ処理などの科学技術計算の並列計算を実現する GPU 計算 (GPU Computation) の研究が注目され、CPU に対して数 10~100 倍の高速化が図られたという有望な結果が報告されている [NVIDIA 09b, Fujimoto 08, 青木 09]。しかしながら、GPU 計算を適用した進化計算の研究に関しては GP への適用を中心に試みられているが、ハードウェアの制限等からまだあまり進んでいないのが現状である。昨年度開催された GECCO-2009 では、進化計算への GPU 計算の適用に関するワークショップ 2009 Computational Intel-

ligence on Consumer Games and Graphics Hardware (CIGPU-2009) が開催され、2010 年には、WCCI (バルセロナ) にあわせて CIGPU-2010 が開催される。

筆者らは先に、GPU 計算を用いた並列進化計算により 2 次割当て問題 (Quadratic Assignment Problem, 以下 QAP) を解く手法について提案した [Tsutsui 09]。CPU による実行よりも高速化が図られることを確認したが、高速化がどのようにして実現出来ているかの議論は十分行えていなかった。

本研究では、GPU 計算による並列進化計算のモデルとして「独立並列計算」に単純化したモデルを提案し、その実行結果並びに解析結果について述べる。独立並列計算による進化計算の基本的な考え方は以下の通りである。 p 個のサブ集団を異なる乱数シードを使って独立して並列実行させる状況を考える。これらのサブ集団はそれぞれ異なる乱数系列を用いるので解を見つける時間はそれぞれ異なる。あるサブ集団が最初に解を見つけたとき、他のすべてのサブ集団の実行も中断し、この時間を独立並列計算における解を見つける時間と考え、この時間は明らかに各サブ集団が解を見つける時間の平均値よりも短くなる。このように独立並列計算は、単一の集団が解を見つける時間よりも見かけ上短い時間で解を見つけることになる。この考え方を GPU 計算による並列計算に取り入れようとするものである。

*連絡先: 阪南大学経営情報学部
〒580-8502 大阪府松原市天美東5-4-33
E-mail: tsutsui@hannan-u.ac.jp

以下では、先ず2章において、GPU計算による進化計算の先行研究を簡単に紹介する。次に3章において、GPU計算の計算モデルを簡単に述べる。4章では、独立並列計算による進化モデルを詳しく述べる。5章では、実験ならびに解析の結果について議論する。最後に6章では、結果のまとめと今後の課題について述べ、本研究のまとめとする。

2 GPU計算による進化計算の関連研究

進化計算へのGPU計算の研究は始まったばかりであるが、GP (genetic programming) への応用を中心にいくつかの研究がある [Banzhaf 08]。Langdonらは、GPにおける関数評価をGPUで行うシステムを提案している [Langdon 08a]。また、[Langdon 08b]では、乳癌の予測のためのGPをGPU計算で実現するシステムを報告している。Robilliardらは、GPで生成されたプログラム(個体)をGPUにより、高速に評価する方法を提案している [Robilliard 08]。Wilsonらは、ビデオゲームマシンを用いたGPを実現している [Wilson 08]。これらのGPへの応用では、GPの個体評価をGPUで高速に評価しようとするものである。

GP以外の進化計算への応用研究は少ないが、その中で先駆けとしてFokらはGeForceFX 6800を用いて並列EP (evolutionary programming) (EP)を実現した結果を報告している [Fok 07]。並列EPのCPUに対するスピードアップ比は、 $\times 1.25 \sim \times 5.02$ である。Claytonらは、ニューラルモデルの評価と進化に応用する分散適応型遺伝的アルゴリズム (distributed adaptive genetic algorithm, DAGA) をGPUに実装している [Clayton 08]。Wongは並列MOEAをCUDA環境を用いて実現している [Wong 09]。CPUに対するスピードアップ比は $\times 5.62 \sim \times 10.75$ である。

これらの事例では、進化計算自体をGPUに実装しているが、Maitreらは、CPUとGPUのハイブリッド型モデルを提案している [Maitre 09]。このアプローチでは進化計算はCPUに実装し、評価関数の計算をGPUで高速に実行する。材料科学の実問題への応用でCPUに対するスピードアップ比として $\times 60$ が得られたことを報告している。

進化計算の応用では個体の評価に要する時間が大きいので、Maitreらの方法は現実的なアプローチであり、またGPでのGPU計算を用いる研究が多く見られるのも同様な視点である。しかし一方、進化計算自体を高速にする必要がある問題も多く存在する。その代表例はスケジューリング問題や割当て問題などの組み合わせ最適化問題である。このような問題では、個体の評価の高速化とともにローカルサーチを含めた進化計算自体

の高速化が重要となる。筆者らの先の研究 [Tsutsui 09] および本研究は、進化計算アルゴリズムの高速化を狙いとするものである。

3 GPU計算の概要

GPU計算では、NVIDIA社がGPU計算向けに提供しているC言語の統合開発環境CUDA (Compute Unified Device Architecture) が最もよく利用されており、ここでは、CUDAによる並列計算モデルを基にGPU計算の概要を簡単に述べる [NVIDIA 09a]。

図1は、GPU計算に用いられる代表的なGPUの一つであるNVIDIA GeForce GTX285のアーキテクチャである。GPU内におけるプロセッサは、スレッドプロセッサ (thread processor, 以下TP) と呼ばれ、8個のTPが一つのグループであるマルチプロセッサ (Multi-Processor, MP) を構成している。MP内のTPは、16KBの高速共有メモリ (shared memory, 以下SM) を介してデータを共有することができる。一方、MP間のデータ共有はVRAMを介して行われる。VRAMはGPUのメインメモリであり、プログラムはVRAMに格納される。VRAM上のデータに対してはキャッシュメモリ機能がないので、VRAMを介したデータ共有は効率的な方法と言えない。したがって、SMを効率良く使用することが大切となる。

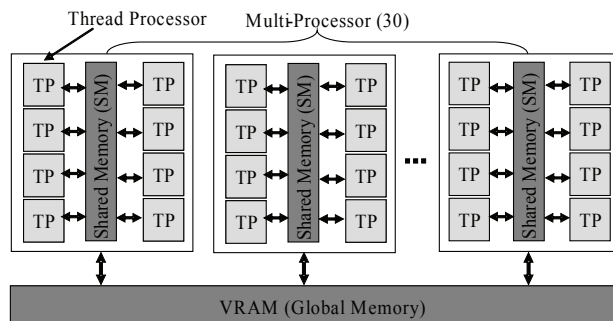


図1: GPUのアーキテクチャ (GeForce GTX285)

CUDAのプログラミングモデルは、基本的にマルチスレッドプログラミングである。図2にCUDAのプログラミングモデルを示す。CUDAプログラムでは、スレッドはグリッド (grid) とブロック (block) の2階層構成をとる。ブロックは、スレッドの集合であり、1次元、2次元、または3次元構成をとることができる。一方、グリッドはブロックの集合であり、1次元または2次元構成をとることができる。各スレッドは、カーネル関数 (kernel function) に記述された同じコードを実行する。スレッドのスケジューリングは、ハードウェアにより自動的に行われる。カーネル関数は、通常のデータ引数の他、グリッドとブロックの定義を引

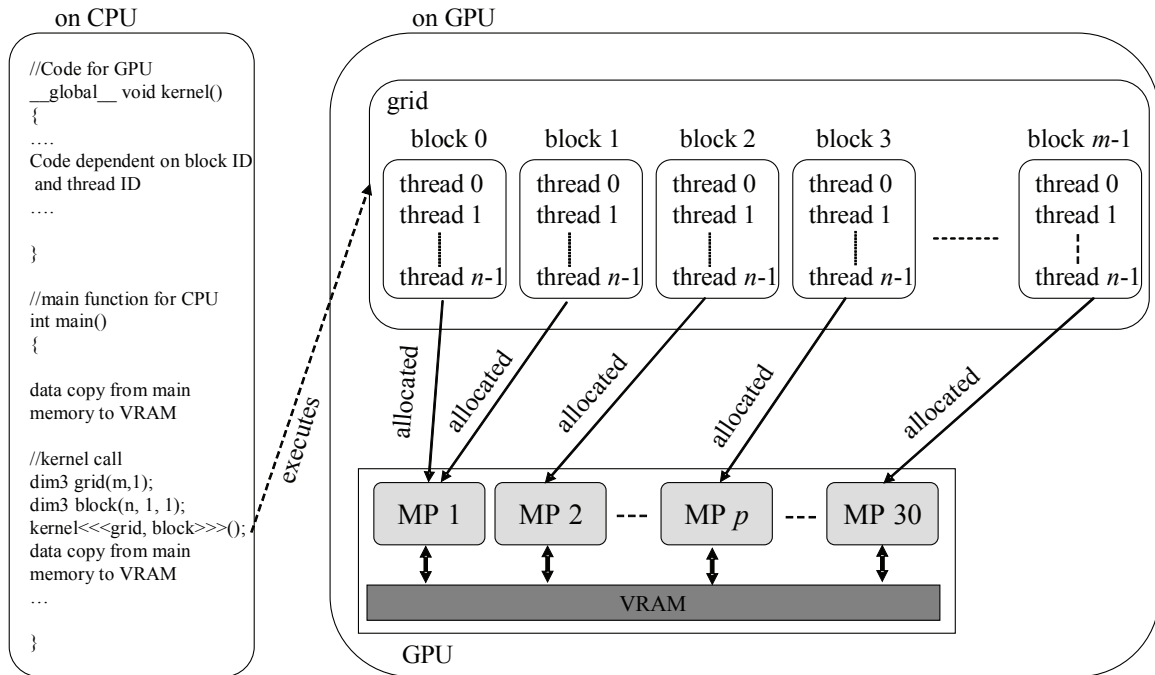


図 2: CUDA プログラミングモデルの概要

数として持ち、カーネル関数がコールされると、グリッドとブロックの定義にしたがってスレッドが生成され、それらのスレッドが一斉に実行を開始する。

VRAM のバンド幅は 159 GB/秒と高速である。しかしながら、VRAM へのアクセスに際しては、100~125 命令時間分の遅延が発生する。しかしこの遅延は、十分な数のスレッドに順序よくアクセスさせることができれば、見かけ上隠すことができる。MP のリソースである SM やレジスタ（ローカル変数はレジスタに割当てられる）は、一つの MP に同時に割当てられるブロック間で分割して使われる。したがって、一つの MP に割当てられるブロック数は、これらの制限によって決まり、同時に実行されるスレッド数もこのブロックの割当て状況に依存する。

VRAM には、コンスタントメモリ（*constant memory*）と呼ばれる 64KB の読み出し専用メモリ領域がある。コンスタントメモリへのアクセスは、各 MP に対して 8KB のキャッシュメモリを持っているので、各 MP 間で共有される定数データを格納するのに適している。

4 GPU を用いた独立並列計算による進化計算のモデル

本章では QAP の解法に用いる GPU を用いた独立並列計算による進化計算のモデルについて詳しく述べよう。

4.1 2 次割当て問題 (QAP)

ここでは、2 次割当て問題 (Quadratic Assignment Problem, QAP) について簡単に触れておく。QAP は、 L 個の施設を、 L 箇所の場所にコスト関数が最小になるように割当てるとい問題であり、企業における事業所の最適配置やビル内の部門の最適配置を決めるのに用いられる。コスト関数 $cost(\phi)$ は以下のように定義される。

$$cost(\phi) = \sum_{i=1}^L \sum_{j=1}^L f_{ij} d_{\phi(i)\phi(j)}. \quad (1)$$

ここで、 d_{ij} は場所 i と j 間の距離を表す距離マトリックスであり、 f_{ij} は、施設 i と施設 j 間の関係の強さを表すフロアマトリックスであり、 ϕ は割当て状態を示す順序である。QAP では、コスト関数 $cost(\phi)$ が 2 次形式の非線形問題となっているため、組み合わせ最適化問題の中では最も困難な問題の一つである [Sahni 76]。

4.2 世代交代モデル

本研究で用いる世代交代モデルを図 3 に示す。集団サイズを N で表すと、このモデルでは、サイズ N の集団 P と同じくサイズ N の作業用の集団 W の二つを用いる。集団 P は、現集団の個体を保持し、 W は、新しく生成された個体を一時的に保管する作業用の領域である。進化プロセスは以下のように行われる。

- Step 1 集団を初期化し, 世代カウンター $t \leftarrow 0$ とする .
- Step 2 P 内の各個体 I_i を評価する .
- Step 3 P 内の各個体 I_i に対して, 交叉のためのパートナー I_j を P 内の個体からランダムに選ぶ ($i \neq j$) . このペア (I_i, I_j) に対して交叉オペレータを適用して一つの子個体 I'_i を作業集団 W の位置 i に生成する .
- Step 4 各個体 I'_i に対して, 突然変異率 p_m で突然変異オペレータを適用する .
- Step 5 W 内の各個体 I'_i を評価する .
- Step 6 各位置 i 毎に, I_i と I'_i を比較し, I'_i の方が I_i よりもよい場合には I'_i を次の世代の個体とするため, I_i と I'_i とを入れ替える .
- Step 7 $t \leftarrow t + 1$.
- Step 8 終了条件を満たせばアルゴリズムを終了し, 満たしていなければ Step 3 へ戻る .

一般に交叉オペレータは2つの親個体から2つの子個体を生成するが, Step 3 では1個体のみを生成する . 2つの親個体を用いる交叉において一つの子個体しか生成しない手法は, Whitley らによる GENITOR アルゴリズムの設計でも用いられている [Whitley 89] .

Step 6 の比較操作は, サイズ2のトーナメント選択のようにになっているが, この比較は同一のインデックス i 間で行われる . 子個体 I'_i は, I_i 個体を片親として生成されている (もう一方の親はランダムに選ばれたものである) ため, I'_i と I_i とは, 染色体レベルでの類似度が高い . このように類似度の高い個体間での選択は, Mahfoud によって提案されている deterministic crowding 法 [Mahfoud 95] と同様, 集団の多様性維持に効果がある .

並列化の視点から見ると図3の世代交代モデルでは, 各個体の処理が, 他の個体の処理と独立に実行でき, 各個体の処理を一つのスレッドとして容易に実装することができる . 本研究では, 4.3 節で述べるように, 1個体の処理を各ブロックにおける1スレッドとして実装している .

交叉オペレータに関しては, Order Crossover (OX) [Oliver 87] と Partially Mapped Crossover (PMX) [Goldberg 89] とを予備実験した結果 PMX が優れており, 本研究では PMX を用いている . QAP では, 評価関数は, TSP のように染色体におけるノードの順序に依存するのではなく, ノードの位置に依存すると考えられるので QAP における PMX の優位性はある程度予測できることである .

突然変異オペレータには, 染色体におけるランダムな2点のノード値を交換する swap オペレータを用いた . 突然変異率 p_m は, 突然変異を適用する個体を選ぶ確率である . なお, 進化計算を用いる QAP の解法では, ローカルサーチを用いるのが一般的である [Stützle 00, Maniezzo 99, Tsutsui 08] . しかし本研究の主目的の一つは, GPU を用いた並列進化計算の効果をより明確にすることであるので, 本研究ではローカルサーチを適用していない .

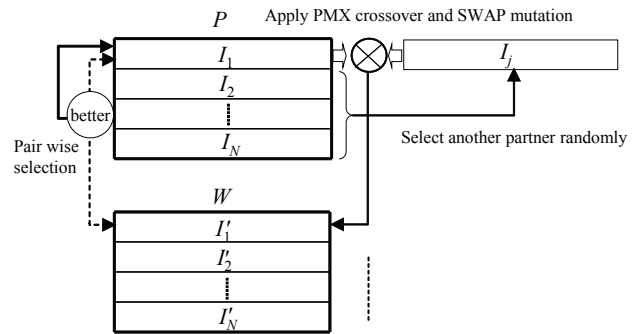


図3: 世代交代モデル

4.3 独立並列計算による進化モデルの実装

本研究では, 3章で述べた GeForce GTX285 GPU を2個用いる . このGPUは, 16KBのSMを共有する8個のTPからなるMPを30個有している . TPによるスレッド処理機能を有効に活用するために, 一つのMPには, できるだけ多くのスレッドを立ち上げ, 並列実行させることが必要である . 4.2 節で述べた世代交代モデルからなる集団を各MP実装する . ここでは, この集団をサブ集団と呼ぶ . このサブ集団を各MP間で独立に並列実行させるモデルを考える . したがって, 本研究で用いるモデルでは, MP内でのスレッドの並列実行と, 各MP間での並列実行の2階層並列処理モデルとなる .

本研究では個体におけるストリングには, *unsigned char* 型の配列を用いる . これにより, 問題サイズの最大が255になるという制限が生じるが, QAPでは, 大きな問題でも高々150程度であるので特に問題は生じない . QAPの問題サイズを L とすると, 各個体のストリングには LB (バイト) 要する . また各個体の評価値を記憶するために *int* 値が必要であり $4B$ を要する . さらに各個体毎に必要な乱数発生に $4B$ 必要となる . また後ほど述べるリスタート戦略 (4.4 節参照) の制御のために3つの *int* 値の領域 $12B$ が必要となる .

このように, 図3のモデルを実装するのに必要なメモリ量はバイト数で, $2N(L+4) + 4N + 12 = 2NL + 12N + 12$ となる . ただし, N は, サブ集団のサイズであ

る．本研究では $N = 128$ を用いているので，この値は $256L + 1548$ である．SM は 16KB であるので，この設定での最大問題は， $L = 57$ となる．なお， $N = 64$ に設定した場合には， L は最大 121 まで可能である．CUDA プログラミングの設定の観点からは，1 ブロックのスレッド数を 128 に設定し，ブロック数の設定を 1GPU あたり 30 と設定することになる．QAP の距離マトリクス d_{ij} およびフロマトリクス f_{ij} は 64KB の容量を持つ constant メモリに格納できるように *unsigned short* 型で記憶させ，高速にコスト関数の計算ができるように工夫した．

各 MP で実行されるブロック（サブ集団）はそれぞれ独立に並列実行される．あるブロックにおいて，知られている最適解からの誤差が指定された値以下の解（以下では，許容解と呼ぶ）が見つかるとうアルゴリズムの終了となるが，これを実現するため，各 MP から参照できる VRAM 上にフラッグ「Foundflag」を設ける．Foundflag の初期値は 0 とし，ある MP で許容解が見つかり 1 を設定する．他の MP におけるサブ集団のスレッドは各世代毎に Foundflag を参照し，もし Foundflag が設定されていることを検出すると直ちにスレッドを終了する．このようにして，独立実行しているサブ集団は，何れかの MP で許容解が見つかり実行を終了し，アルゴリズム全体が終了する．

4.4 リスタート戦略の実装

4.3 節で設定したサブ集団のサイズ $N = 128$ は，QAP を解くにはやや小さい．集団サイズが小さい場合には一般に初期収束を起こし局所解にトラップされる．初期収束により，局所解にトラップされるのを避けるために本研究ではリスタート戦略をサブ集団のアルゴリズムに組み込んでいる．

進化計算においてはリスタート戦略は，進化が停滞した際に局所解から集団を脱出させる手段としてよく用いられる手法である [Mathias 97, Eshelman 91, Tsutsui 97]．ブロック内のサブ集団は個々の個体がスレッドとして実行されているので，この実行環境下で効率よくリスタート戦略を実現するために，本研究では以下のような方法を用いた．サブ集団における現世代の最良解を f_{c-best} とする．各世代において評価関数値が f_{c-best} に等しい個体の数を数え，それを B_{count} とする．もし， B_{count} が $N \times B_{rate}$ よりも大きくなったら，そのサブ集団は局所解にトラップされたと判断し，そのサブ集団を再初期化する．この際，エリート解を保存する方法もあるが，エリート解保存戦略は再度同じ局所解にトラップされやすいことが予備実験で分かり，本研究ではすべての個体を再初期化することにした．ここでは， $B_{rate} = 0.6$ を用いた．

先に述べたように各個体はブロック内のスレッドとして独立して実行されているので， B_{count} を数えるためには，排他制御と同期を伴った処理が必要である．ここでは，SM でそれらの機能を実現できるアトミック命令 (*atomic instructions*) を用いて実現した．なお，この処理は，オーバーヘッドの原因になるので， B_{count} のチェックは 50 世代おきに行うことにした．

リスタート戦略の効果を表 1 に示す．この表の結果は，1 ブロックでの実行を CPU (Intel Core i7 965 3.2 GHz) でエミュレーションした結果である．突然変異率には $p_m = 0.1$ を用いている．この表から，リスタート戦略により 1 つのブロック (1 つのサブ集団) の実行においてすべての問題で，許容解が得られている．リスタート戦略を用いない場合は，多くの実行において初期収束を起こして解が見つかっていないことが分かる．

リスタート戦略を組込んだ独立並列計算における進化モデルの実装の CUDA 環境におけるカーネル関数の疑似コードを図 4 に示す．

表 1: リスタート戦略の効果

QAP Instances	no restart				restart			
	#LdO#	T_{avg}	Min	Max	#LdO#	T_{avg}	Min	Max
tai25b	6	0.07	0.05	0.10	30	0.07	0.05	3.77
kra30a	1	0.27	0.27	0.27	30	10.03	0.27	61.13
kra30b	0	-	-	-	30	25.20	0.45	92.24
tai30b	0	-	-	-	30	3.96	0.39	12.53
kra32	0	-	-	-	30	10.70	0.33	59.52
tai35b	0	-	-	-	30	29.69	2.53	137.72
ste36b	0	-	-	-	30	17.00	0.73	45.10
tai40b	1	0.30	0.30	0.30	30	6.96	0.30	20.96
tai50b	0	-	-	-	30	48.07	1.12	147.37

#OPT: Number of success run in 30 runs

T_{avg} : Average time to find acceptable solutions in success runs in second

5 実験結果とその解析的検討

5.1 実験条件

本研究で用いた計算機はインテル社の Core i7 965 (3.2 GHz) プロセッサと NVIDIA 社の GeForce GTX285 を 2 個搭載した PC である．OS は Windows XP Professional でグラフィックスドライバは，NVIDIA graphics driver Version 195.62 である．CUDA プログラムのコンパイルには，Microsoft Visual Studio 2008 Professional Edition (最適化オプションは/O2) および CUDA 2.3 SDK を用いた．また，2 つの GPU は，Win32 API によるスレッドプログラミングにより並列実行させる．

テスト問題には QAPLIB ベンチマークライブラリ [qap 09] の問題を用いる．QAPLIB のテスト問題は，つぎの 4 つのクラス，すなわち，1) ランダム生成問題，2) グリッド距離ベースランダム問題，3) 実問題，4)

```

//Code for GPU
__global__ void kernel()
{
    get seed of random number generator for threadIdx.x from VRAM;
    t=0;
    initialize this subpopulation in this block;
    evaluate;
    for(int t=1; t<MaxGeneration; t++){
        __syncthreads();
        reset memory contents related restart;
        Evolutionary Code for threadIdx.x, blockIdx.x;
        ---
        if (this thread (individual) found an acceptable solution)
            set FoundFlag in VRAM to 1;
            write the solution to VRAM;
            return;
        else{
            if(FoundFlag in VRAM ==1)//an acceptable solution has found in some other bckc
                return;
        }
        if(t%50==0){
            atomicMin(fc_best, perf);//get minimum value at s_minValue in SM
            __syncthreads();
            if(perf == *fc_best)//if this thread have minimum functional value
                atomicAdd(B_count, 1);//add 1 to B_count in SM
            __syncthreads();
            if(threadIdx.x==0){//check restart condition by thread with id 0
                if(*B_count > (int)(POOL_SIZE*B_rate+0.5))//satisfy?
                    *restart_flag = 1;//inform other thread that the restart condition has satisfied
            }
            __syncthreads();
            if(*restart_flag == 1){//restart?
                initialize string of this thread;
                evaluate;
                __syncthreads();
            }
        }
        else
            __syncthreads();
    }
}

//main function for CPU
int main()
{
    ....
    data copy from main memory to VRAM
    //kernel call
    dim3 grid(p,1);
    dim3 block(128, 1, 1);
    kernel<<<grid, block, SM_size>>>();
    copy from main memory to VRAM
    ....
}

```

図 4: 独立並列計算による進化モデルの実装の CUDA 環境におけるカーネル関数の擬似コード

実問題風に生成した問題，に分けられる [Stützle 00] .
 ここでの実験では，クラス 3) および 4) から問題サイズが 25 ~ 50 の以下の 9 個の問題，すなわち，tai25b, kra30a, kra30b, tai30b, kra32, tai35b, ste36b, tai40, and tai50b を用いる .

実験は各問題に対して 30 回とし，評価は決めた解が得られるまでの時間で行う . 決められた解は，tai50b を除いて知られている最適解 (誤差が 0 の解) とし，tai50b に関しては，知られている最適解からの誤差が 0.06% 以内の解とした . 時間は 30 回の実験の平均値 $T_{p,avg}$ で示す . ここで， p は，MP の数 (この設定では CUDA のブロック数) である . すべての実験で $p_m = 0.1$ を用いる .

5.2 単一ブロックでの実行における時間分布

ここでは，GPU 計算を 1 ブロック ($p = 1$) のみを用いて，一つのサブ集団で実行した場合の解を得る時間がどのように分布しているかを実験する . 分布を精度良く得るためにここでは実験回数を 100 回とした . 100 回の実験における時間の分布を図 5 に示す . 同図において各*印は，100 回の実行における各回の時間を示しており，黒い四角は平均時間を ($T_{1,avg}$) を示している .

図 5 の分布が一つのサブ集団で実行した場合の解を得る時間の分布をよく近似していると仮定すると， $p(p > 1)$ 個のブロックを並列に実行した場合，一回の実験における時間は，1 つのブロックで p 回実行したときの最小値であるので，その平均値である $T_{p,avg}$ は，明らかに 1 ブロックによる実行の平均値 $T_{1,avg}$ よりも小さくなる . このように， p 個のブロックで独立並列計算すると，ブロックを 1 つしか用いない場合よりも早く解が得られることになる . すなわち， $p > 1$ に対して $T_{p,avg} < T_{1,avg}$ である .

5.3 独立並列計算による進化計算の実験結果

本研究で用いる PC には，2 つの GPU が実装されているので，ここでは 1 つの GPU を用いて 30 ブロックを独立並列計算させる実験 ($p = 30$) と 2 つの GPU を用いて 60 ブロックを独立並列計算させる実験 2 つを行った結果について述べる . CUDA のプログラミングモデルでは，3 章で述べたように，実在する MP 数以上のブロックを実行させることができるが，4.3 節で述べた実験条件の設定では，実在する MP 数以上のブロックを実行してもハードウェアリソース (レジスタや SM) の不足から，実際はそれらはシーケンシャルに実行されるので，性能低下をもたらしてしまう . 表 2 に結果をまとめる .

同表において $gain_p$ は， $T_{1,avg}/T_{p,avg}$ ，すなわち， p ブロックの独立並列計算によって解を得た時間の，1 ブロックを使用した場合に対する高速化の倍率を示している . なお， $p = 30, 60$ に対しては， $T_{p,avg}$ は実験回数である 30 回の平均であるが， $T_{1,avg}$ には，5.2 節の図 5 の結果を用いており，これは 100 回の実験の結果である . この結果を見ると，たとえば tai25b では， $gain_{30} = 9.56$ ， $gain_{60} = 10.85$ となり， $gain_p$ は比較的小さい値となっている . 一方，kra30a の結果を見ると， $gain_{30} = 25.43$ ， $gain_{60} = 49.10$ となっており，ほぼ p に比例する値が得られている .

$gain_p$ の値は，問題によって異なっているが， $p = 30$ に対しては，[10, 35] の範囲にあり，また $p = 60$ に対しては，[10, 70] の範囲にあり，tai25b, tai40b を除いて概ね p に比例した結果となっている . tai25b および tai40b では， $gain$ が小さい値となっているが，これらの問題では， $T_{1,avg}$ の標準偏差の値は他の問題の標準偏差よりも相対的に小さく，独立並列計算の効果が小さくなっていることが原因と考えられる .

5.4 独立並列計算の統計的推定による解析

本節では，5.3 節で述べた結果の統計的推定による解析について述べる . 1 ブロックを使う場合の実行時間の確率密度関数を $f(t)$ とし， $f(t)$ の確率分布関数を $F(t)$ としよう . すなわち，

$$F(t) = \int_0^t f(t)dt. \quad (2)$$

ここで，4.3 節で述べた独立並列計算を p 個のブロックを用いて行った場合を考え，その場合の実行時間の確率密度関数，確率分布関数および平均実行時間をそれぞれ， $g(p, t)$ ， $G(p, t)$ および $M(p)$ で表わそう . これらは以下のように容易に $F(t)$ から以下のようにして得られる .

時間 t において， p 個の何れのブロックも解を発見していない確率 $\tilde{G}(p, t)$ は，独立実行としているので同時確率から

$$\tilde{G}(p, t) = (1 - F(t))^p \quad (3)$$

である . したがって， $G(p, t)$ は，

$$G(p, t) = 1 - \tilde{G}(p, t) = 1 - (1 - F(t))^p \quad (4)$$

であり， $g(t)$ は，

$$g(p, t) = \frac{dG(p, t)}{dt}, \quad (5)$$

$M(p)$ は

$$M(p) = \int_0^{\infty} t \cdot g(p, t)dt \quad (6)$$

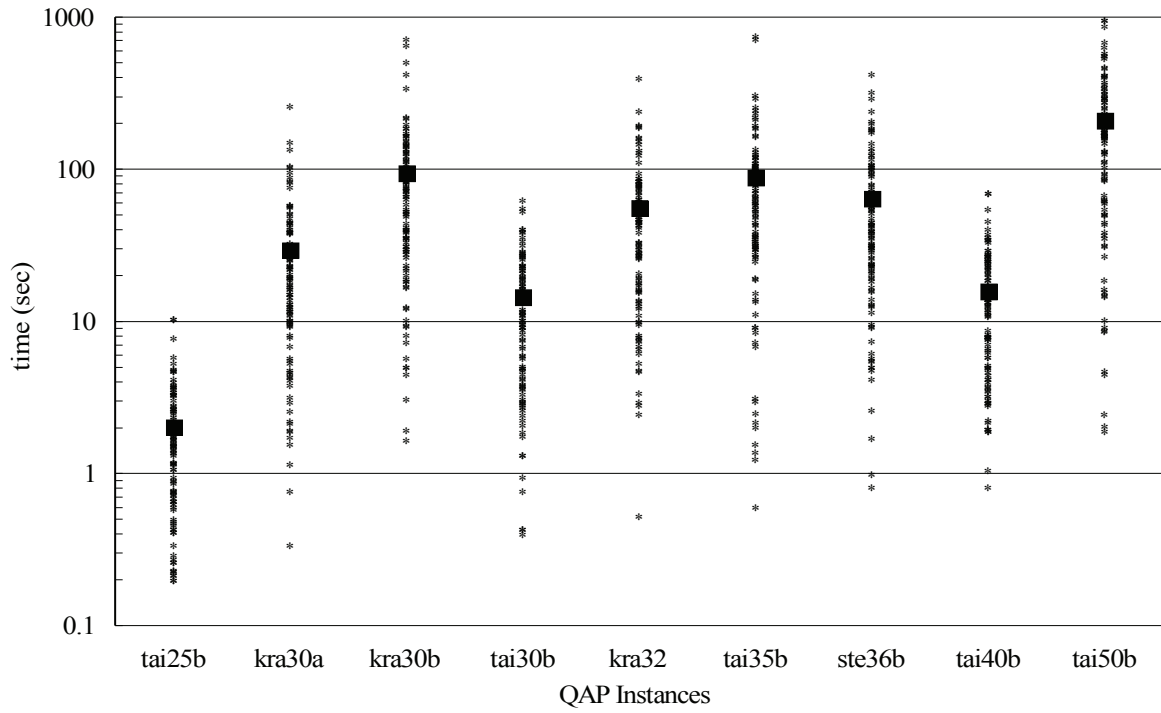


図 5: 単一ブロックでの実行における時間分布

表 2: 独立並列計算による進化計算の結果とその統計的推定

GPU	Instances	tai25b					kra30a					kra30b				
	No of blocks p	GPU			Γ		GPU			Γ		GPU			Γ	
		$T_{p,avg}$	σ	$gain_p$	$M(P)$	Δ_p	$T_{p,avg}$	σ	$gain_p$	$M(P)$	Δ_p	$T_{p,avg}$	σ	$gain_p$	$M(P)$	Δ_p
GPU \times 1	1	2.02	1.89	1.00	-	-	34.25	36.79	1.00	-	-	113.69	113.36	1.00	-	-
	30	0.21	0.06	9.56	0.03	0.18	1.35	1.23	25.43	0.79	0.56	3.17	2.52	35.85	2.92	0.25
GPU \times 2	60	0.19	0.02	10.85	0.00	0.18	0.70	0.47	49.10	0.34	0.36	1.63	1.48	69.59	1.21	0.42
GPU	Instances	tai30b					kra32					tai35b				
	No of blocks p	GPU			Γ		GPU			Γ		GPU			Γ	
		$T_{p,avg}$	σ	$gain_p$	$M(P)$	Δ_p	$T_{p,avg}$	σ	$gain_p$	$M(P)$	Δ_p	$T_{p,avg}$	σ	$gain_p$	$M(P)$	Δ_p
GPU \times 1	1	14.31	13.64	1.00	-	-	56.18	61.11	1.00	-	-	92.08	75.90	1.00	-	-
	30	0.71	0.46	20.24	0.45	0.25	2.13	1.80	26.35	1.78	0.35	3.67	3.56	25.12	3.25	0.41
GPU \times 2	60	0.46	0.16	30.88	0.16	0.30	1.12	0.93	50.11	0.83	0.29	1.65	1.38	55.68	1.66	-0.01
GPU	Instances	ste36b					tai40b					tai50b				
	No of blocks p	GPU			Γ		GPU			Γ		GPU			Γ	
		$T_{p,avg}$	σ	$gain_p$	$M(P)$	Δ_p	$T_{p,avg}$	σ	$gain_p$	$M(P)$	Δ_p	$T_{p,avg}$	σ	$gain_p$	$M(P)$	Δ_p
GPU \times 1	1	70.82	73.23	1.00	-	-	19.07	16.43	1.00	-	-	212.55	203.64	1.00	-	-
	30	2.57	1.89	27.54	1.63	0.94	1.15	0.62	16.56	0.46	0.69	8.75	8.65	24.29	6.19	2.56
GPU \times 2	60	1.35	0.77	52.40	0.66	0.70	0.90	0.16	21.27	0.12	0.78	4.28	3.84	49.62	2.72	1.56

σ : standard deviation

Γ : Results with Gamma distribution estimation

Values of $T_{p,avg}$ and $M(P)$ are in second

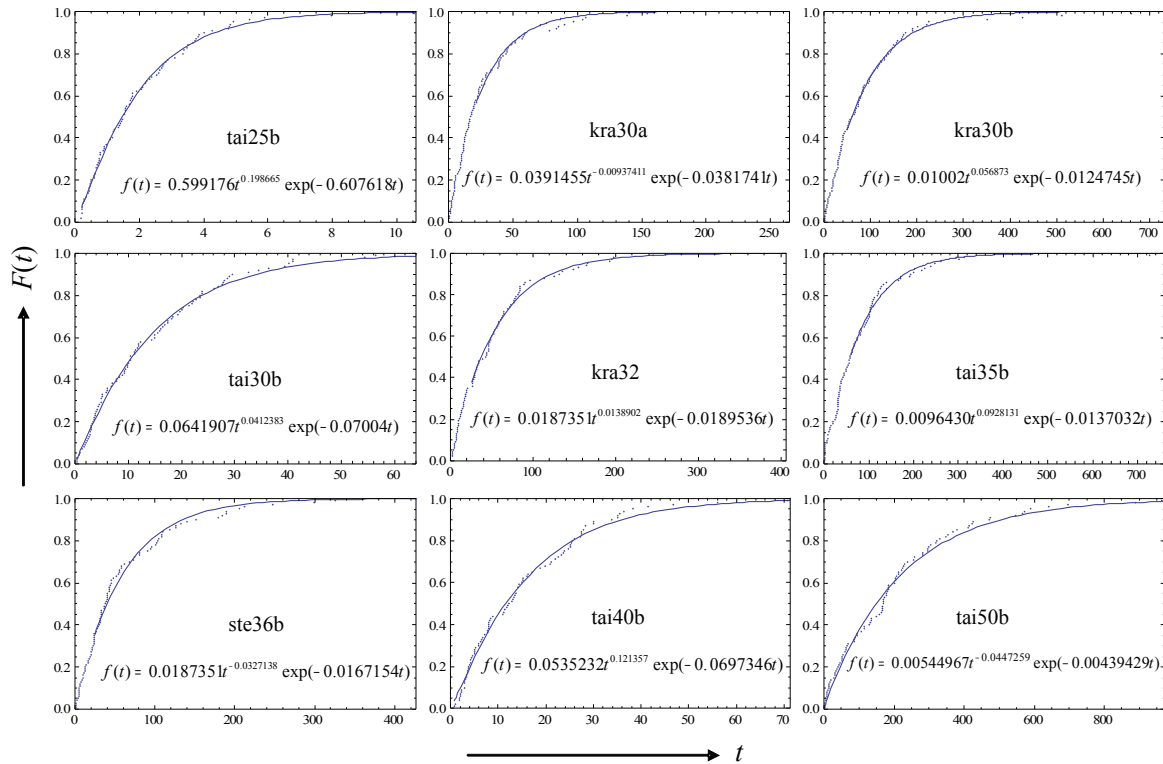


図 6: ガンマ分布による分布推定

として得られる。

図 5 の分布を最小二乗法を用いてパラメトリック推定を行った。正規分布およびガンマ分布で推定した結果、ガンマ分布による推定が図 6 に示すようによくヒットするという結果が得られた（図 5 の縦軸は対数尺度になっているのでやや判別しにくいだが、この分布は、正規分布のように平均値に対して対称ではなく t の小さい方に偏っている）。

式 (6) を図 6 で示した推定に当てはめ、 $p = 30$ および $p = 60$ に対して $M(p)$ を求めた。この結果も表 2 に示した。この結果を見ると、 $M(p)$ は GPU 計算で得られた結果 ($T_{p,avg}$) とほぼ等しいことが分かる。しかし、 $M(p)$ の値は tai35b ($p = 60$) を除いて GPU 計算による結果よりもやや小さくなっていることが分かる。この差を Δ_p で示すと、 Δ_p の値は問題によって異なるが、大きい問題ほど Δ_p は大きくなる傾向にある。

式 (4) は、独立並列計算によるオーバーヘッドはないということを前提にしているが、GPU 計算では、 Δ_p に相当する遅延が起こっている可能性がある。GPU の詳細な性能情報は分からないが、QAP のデータ（距離マトリックスとフローマトリック）を格納しているコンスタントメモリへのアクセス競合による遅延が一つの原因として考えられる。コンスタントメモリは各 MP 単位に 8KB のキャッシュメモリが備えられているが、それでもアクセス競合による遅延は十分考えられる。一

方、分布推定における誤差も考えられる。これらは今後の検討課題の一つである。

以上述べたように、この解析結果は、GPU による計算とのいくつかの誤差 (Δ_p) が存在しているが、概ね独立並列計算により高速化が得られる理論的なバックグラウンドを与えている。

5.5 CPU 計算との比較

ここでは、GPU 計算による高速化を確認するために、同じ問題を CPU 計算で行った結果との比較結果について述べる。CPU 計算の実行環境は、5.1 節で述べたもので、CPU はインテル社の Core i7 965 (3.2 GHz) である。

CPU 計算に用いる世代交代モデルは、4.2 節で述べたものと同じとし、また突然変異やリスタート戦略も同じである。ただし、CPU 計算では B_{count} (4.4 参照) の計算は容易であるので収束状況の判定は各世代で行う。CPU 計算では 2 つの実験を行った。一つは集団サイズを 64, 128, 256, 512, 1024, および 2048 と変化させてチューニングしたもの (Model I) と集団サイズを GPU 計算と同じ 128 に固定したもの (Model II) である。

結果は、表 3 にまとめている。tai50b では 1GPU および 2GPU の場合、Model I に対してスピードアップ

表 3: GPU 計算と CPU 計算の比較

QAP instances	GPU Computation		CPU Computation			speedup to Model I		speedup to Model II	
	GPU×1 ($T_{30,avg}$)	GPU×2 ($T_{60,avg}$)	Model I		Model II	GPU×1	GPU×2	GPU×1	GPU×2
			CPU (T_{avg})	Population Size	CPU (T_{avg}) with Population size 128				
tai25b	0.21	0.19	0.82	128	0.82	3.9	4.4	3.9	4.4
kra30a	1.35	0.70	6.64	1024	21.45	4.9	9.5	15.9	30.7
kra30b	3.17	1.63	25.20	128	25.20	7.9	15.4	7.9	15.4
tai30b	0.71	0.46	2.05	512	3.96	2.9	4.4	5.6	8.5
kra32	2.13	1.12	10.70	128	10.70	5.0	9.5	5.0	9.5
tai35b	3.67	1.65	12.16	512	29.64	3.3	7.4	8.1	17.9
ste36b	2.57	1.35	15.07	256	16.99	5.9	11.1	6.6	12.6
tai40b	1.15	0.90	4.44	512	6.98	3.9	5.0	6.1	7.8
tai50b	8.75	4.28	18.76	512	48.07	2.1	4.4	5.5	11.2

Values of $T_{30,avg}$, $T_{60,avg}$ and $M(P)$ are in second

比はそれぞれ, $\times 3.9$ および $\times 4.4$ である. kra30b の場合はそれらはそれぞれ $\times 7.9$, $\times 15.4$ となった. 用いた問題で平均してみると, 1GPU, 2GPU の場合, Model I に対して $\times 4.4$ and $\times 7.9$ である. Model II に対しては, 当然スピードアップ比は大きな値を示している.

さて, ここでさらなる高速化の可能性と限界について考察してみよう. 本研究では進化計算モデルを CUDA の 1 ブロックに実装し, その独立並列計算を行っている. 5.3, 5.4 節で見たように計算速度はブロック数にほぼ比例した高速化が得られるという結果を示している. したがって, ここで問題となるのは, 1 つのブロックでの処理の高速化が重要となる.

```
//cut1 are cut2 are random number in [0,L-1]. cut1 < cut2 is assumed.
//unsigned char *parent1, *parent2 are strings of the parents.
//unsigned char *child is a new string to be generated.
for (int j = 0; j < L; j++)
  child[j] = parent1[j];
for (int i = cut1; i < cut2; i++){
  for (int j = 0; j < L; j++){
    if (parent2[i] == child[j]){
      unsigned char tmp = child[i]; child[i] = child[j]; child[j] = tmp;
      break;
    }
  }
}
```

図 7: 本研究で用いている PMX オペレータのコード

図 7 は本研究で用いた PMX オペレータのコードである. ここで $cut1$, $cut2$ は交叉ポイントである. 交叉ポイントはランダムに乱数により生成され, スレッド間で異なっている. この結果, ブロック内における各スレッドは異なったループ回数 (for ループ) となり, さらに問題の性質上分岐命令 (if 分岐) の条件はスレッド間で全く異なる. GPU では一つの MP 内においては, 32 スレッド単位に SIMD 風に行われるので, スレッド間での処理の流れが異なるプログラムでは, 多くの待ち合わせ時間が発生してしまう.

このようなランダム性に起因する処理の遅延を避ける方法については, 各スレッド間で同じ交叉ポイントを用いる方法なども考えられるが, SIMD 環境で高速に処理できるオペレータの設計を考える必要がある. これについては今後の検討課題である.

6 むすび

GPU 計算を用いて独立並列計算による進化計算により 2 次割当て問題 (QAP) を解く方法を提案し, 実験を行った結果とその解析について述べた. この結果, 本方式では, ほぼ GPU の MP 数に比例した速度が得られることを示すことができた. また, その解析的裏付けを示した.

このモデルでは, 実行時間の標準偏差が大きいほど効果的な速度が得られる. 進化計算では従来から Exploration と Exploitation のバランスをとった進化計算のモデルが良いとされてきたが, このモデルではむしろ Exploration と同時に Exploitation をより強くした進化計算モデルが有利であることを示唆している.

CPU 計算に対するスピードアップ比では, CPU 計算をチューニングしたモデル (Model I) に対しては, 1GPU で $\times 4.4$, 2GPU で $\times 7.9$ である. GPU 計算と同じ集団サイズとした場合には, 1GPU で $\times 7.2$, 2GPU で $\times 13.1$ という結果を得た.

本研究に関連して今後に残された問題として, 以下のような研究課題があり, 今後取り組んでいく予定である.

- (1) SIMD 環境に適した新しい進化計算オペレータの提案
- (2) Exploration と同時に Exploitation をより強くした進化計算モデルの提案

- (3) 統計解析モデルの精度の向上とそのための実験データの収集
- (4) より大きな問題への適用とローカルサーチの組み込み手法の提案

なお、次期 GPU として開発されているコードネーム Fermi では、ハードウェアの機能の大幅な強化が図られており、これらの検討課題にはかなり応えられるものと期待できる。

謝辞

本論文の一部は、文部科学省科学研究補助金（基盤研究 C）課題番号 19500199 の一環として行われた。

参考文献

- [Banzhaf 08] Banzhaf, W., Harding, S., Langdon, W., and Wilson, G.: *Accelerating Genetic Programming through Graphics Processing Units*, Springer (2008)
- [Cantú-Paz 00] Cantú-Paz, E.: *Efficient and Accurate Parallel Genetic Algorithms*, Kuwer Academic Publishers (2000)
- [Clayton 08] Clayton, T. F., Patel, L. N., Leng, G., Murray, A. F., and Lindsay, I. A. B.: Rapid evaluation and evolution of neural models using graphics card hardware, in *Proc. of the 10th Conf. on Genetic and Evolutionary Computation* (2008)
- [Eshelman 91] Eshelman, L. J.: The CHC adaptive search algorithm: how to have safe search when engaging in nontraditional genetic recombination, in *Foundations of Genetic Algorithms*, Morgan Kaufmann Publishers (1991)
- [Fok 07] Fok, K., Wong, T., and Wong, M.: Evolutionary Computing on Consumer-Level Graphics Hardware, *IEEE Intelligent Systems*, Vol. 22, No. 2 (2007)
- [Fujimoto 08] Fujimoto, N.: Dense Matrix-Vector Multiplication on the CUDA Architecture, *Parallel Processing Letters*, Vol. 18, No. 4 (2008)
- [Goldberg 89] Goldberg, D.: *Genetic algorithms in search, optimization and machine learning*, Addison-Wesley publishing company (1989)
- [Langdon 08a] Langdon, W. and Banzhaf, W.: A SIMD Interpreter for Genetic Programming on GPU Graphics Cards, in *Proc. of the 11th European Conf. on Genetic Programming*, Springer (2008)
- [Langdon 08b] Langdon, W. and Harrison, A.: GP on SPMD parallel graphics hardware for mega Bioinformatics data mining, *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, Vol. 12, No. 12 (2008)
- [Mahfoud 95] Mahfoud, S.: A Comparison of Parallel and Sequential Niching Methods, in *Proc. of the Six International Conf. on Genetic Algorithms*, Morgan Kaufmann (1995)
- [Maitre 09] Maitre, O., Baumes, L. A., Lachiche, N., Corma, A., and Collet, P.: Coarse Grain Parallelization of Evolutionary Algorithms on GPGPU Cards with EASEA, in *Proc. of the 2009 Genetic and Evolutionary Computation Conf. (GECCO-2009)*, Morgan Kaufmann Publishers (2009)
- [Maniezzo 99] Maniezzo, V. and Colorni, A.: The ant system applied to the quadratic assignment problem, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 11, No. 5 (1999)
- [Mathias 97] Mathias, K. E. and Whitley, L. D.: Changing representation during search: a comparative study of delta coding, *Evolutionary Computation*, Vol. 2, No. 3 (1997)
- [NVIDIA 09a] NVIDIA, (2009), <http://www.nvidia.com/page/home.html>
- [NVIDIA 09b] NVIDIA, : *CUDA Programming Guide 2.1* (2009)
- [Oliver 87] Oliver, I., Smith, D., and Holland, J.: A study of permutation crossover operators on the travel salesman problem, in *Proc. of the 2nd International Conf. on Genetic Algorithms*, L. Erlbaum Associates Inc. (1987)
- [qap 09] QAPLIB - A Quadratic Assignment Problem Library (2009), <http://www.seas.upenn.edu/qaplib>
- [Robilliard 08] Robilliard, D., Marion-Poty, V., and Fonlupt, C.: Population Parallel GP on the G80 GPU, in *Proc. of the 11th European Conf. on Genetic Programming*, Springer (2008)
- [Sahni 76] Sahni, S. and Gonzalez, T.: P-complete approximation problems, *Journal of the ACM*, Vol. 23, (1976)
- [Stützle 00] Stützle, T. and Hoos, H.: Max-min ant system, *Future Generation Computer Systems*, Vol. 16, No. 9 (2000)
- [Tsutsui 97] Tsutsui, S., Ghosh, A., Corne, D., and Fujimoto, Y.: A Real Coded Genetic Algorithm with an Explorer and an Exploiter Populations, in *Proc. of the 7th International Conf. on Genetic Algorithm (ICGA97)*, Morgan Kaufmann Publishers (1997)
- [Tsutsui 08] Tsutsui, S.: Parallel Ant Colony Optimization for the Quadratic Assignment Problems with Symmetric Multi Processing, in *Proc. of the Sixth International Conf. on Ant Colony Optimization and Swarm Intelligence*, Springer (2008)
- [Tsutsui 09] Tsutsui, S. and Fujimoto, N.: Solving quadratic assignment problems by genetic algorithms with GPU computation: a case study, in *Proc. of the Genetic and Evolutionary Computation Conf. GECCO (Companion)*, ACM (2009)
- [Whitley 89] Whitley, D., Starkweather, T., and Fuquay, D.: Scheduling problems and traveling salesman problem: The genetic edge recombination operator, in *Proc. of the 3rd International Conf. on Genetic Algorithms*, Morgan Kaufmann (1989)
- [Wilson 08] Wilson, G. and Banzhaf, W.: Linear Genetic Programming GPGPU on Microsoft's Xbox 360, in *Proc. of the IEEE Congress on Evolutionary Computation 2008 (CEC'08)* (2008)
- [Wong 09] Wong, M. L.: Parallel multi-objective evolutionary algorithms on graphics processing units, in *Proc. of the Genetic and Evolutionary Computation Conf. GECCO (Companion)*, ACM (2009)
- [青木 09] 青木 尊之: フル GPU による CFD アプリケーション, *情報処理*, Vol. 50, No. 2, pp. 107-115 (2009)