Technical Reports on Information and
Computer Science from Kochi
Vol. 4 (2012), No. 8

GPGPU を用いた迷路配線の並列アルゴリズムの一手法

A Parallel Algorithm for Maze-Routing on GPGPUSoC

藤井 良弥 1） 中井 駿介 2） 寺田翔太 2） 村岡 道明 3） 豊永 昌彦 3）

Yoshiya Fujii 1） Shunsuke Nakai 2） Shota Terada 2）Michiaki Muraoka 3） Masahiko Toyonaga 3）

1）高知大学理学部 2）高知大学大学院理学専攻 3)高知大学 情報講座

Information Science Division, Faculty of Science, Kochi University

あらまし

本研究は，短 TAT 設計に向けた VLSI レイアウト設計の自動配線について，並列迷路配線法（1 層，多層）の提案とその効果の検証をおこなったものである．提案する並列迷路配線法は，GPGPU におけるスレッドを，各検索点に割り当てて同時に候補点を検索する．新たな検索点リストの作成においては，並列化したオフセット処理をおこなっている．

実験によりそれらの高速化の効果を調べたところ，並列 1 層配線法では，サイズ 20～100 グリッド四方の配線領域で最大 8.3 倍，並列多層配線法ではサイズ 20～40 グリッド四方の配線領域で最大 12.7 倍の高速化が確認された．

キーワード：短 TAT，GPGPU，迷路配線法，スレッド

1. はじめに

半導体製造プロセスの微細化に伴い，電子機器は小型化，低消費電力，高性能化および多機能化が進んでいる．特に，ここ 10 年内では，CPU コアを複数個搭載したマルチコアやメニーコアといった大規模複雑な CPU が登場している．このような大規模システムを 1 チップで実現するには，VLSI 設計フローにおける上流技術はもとより，レイアウト設計など物理設計についても新たな技術が求められる．

近年，多数のグラフィック処理用コアで構成されている GPU(Graphics Processing Unit)を汎用並列計算機として利用する技術 GPGPU(General Purpose GPU)が提唱され，大規模化への CAD システムの解決策として注目されてきている[3-7]．例えば，文献[4]では論理回路シミュレーションの新手法として，回路分割で各回路動作の独立性を高めて GPGPU を利用するアルゴリズムを提案しており，プロセッサコアである OpenSPARC レベルの論理回路シミュレーションを高速化している．VLSI の回路最適化におけるゲートサイジングとその動作しきい値割り当てにおいても，GPGPU を利用した並列化により従来手法の 56 倍の高速化する方法が提案されている[5]．レイアウト設計分野では，論理設計とフロアプランの 2 つの設計段階（論理から配置）まで一貫して並列処理で最適解を求めて，先進技術と比較して 11 倍高速化する方法が提案されている[6]．また，グラフ計算手法として迷路配線法に転用可能な幅優先探索アルゴリズムについても並列化アルゴリズムが提案されており，GPGPU に対して階層的なカーネル割り付けることで，最新 CPU による処理と比較して 10 倍以上の高速化が報告されている[3]．

一般に並列処理を用いた高速化アルゴリズムでは，問題を独立要素に分割可能かどうかでその性能が大きく変わる．文献[5]，[6]のようにもともと問題が独立している場合は，比較的容易に 2 桁以上の高速化が実現できるが，文献[3]のように，続く処理が先行処理に依存する問題では，高い高速性が望めない．

本研究は，レイアウト設計における配線アルゴリズム，すなわち始点から終点までを連続して検索して配線経路を求める（すなわち，先行処理が続く処理に影響を与える）迷路配線法について，GPGPU を用いた並列処理アルゴリズムの一手法の提案をするものである．ここで用いる GPGPU ソフトウェア環境は，NVIDIA 社の GeForce（あるいは Tesla）で利用できる CUDA 環境を用いている．

本研究では，まず並列 1 層迷路配線法として実装方法を提案し，その後，実用化のために並列多層配線法に拡張する提案をする．並列迷路配線法は，GPGPU を迷路配線法の検索点リストの各点にスレッドを割り当てて並列に候補点の検索をして高速化を図る．検索点リスト作成では，処理効率を上げるためオフセット処理の並列化も行う．並列多層迷路配線法では，CUDA の制限から 5 層，配線領域 40×40 グリッドまでとした．

並列 1 層迷路配線法に関して，評価実験を行い，障害物を使用した配線モデルでは単一スレッドに比べて並列スレッドでは処理スピードが最大 8.3 倍となることを確認した．また並列多層迷路配線法を，ランダムに配置した 2 端子ネットの端子で評価実験を行い，単一スレッドに比べて並列スレッドでは処理スピードが最大 12.7 倍となることを確認した．

本論文の以降の構成は，第 2 章で GPGPU のハード面，ソフト面について説明する．第 3 章では，1 層迷路配線法

について提案する並列化を説明する．第 4 章は，前述のアルゴリズムを多層化し，より実際的な設計での評価をおこなう．第 5 章にまとめを述べる．

2. GPGPU(CUDA)とは

(1) GPGPU

GPU(Graphic Processing Unit)は多数のプロセッサを持つ画像専門処理部品で，これらのプロセッサを並列処理に利用したものをGPGPUという．汎用CPUに比べてスピードは落ちるが，多数のプロセッサを用いることで並列処理を行い，高速化を実現できる．主にゲームコンピューティングに使用されたり，組み込みシステムに使用されていたVDP(Video Display Processor)の代わりに使用されたりしている．並列処理にはスーパーコンピュータがあるが，数百万円以上の費用がかかり非常に高価である．一方でGPGPUは2万円程度と比較的手軽であり，PCにグラフィックカードを取り付けるだけで簡単で使用することができる．

GPGPU を利用するためには，特化したソフトウェア環境が必要である．このソフトウェア環境としては，CUDA や OpenCL などがある．OpenCL とは，様々なプロセッサの GPU を利用する GPGPU ソフトウェア環境を指す．本研究では，CUDA に基づく並列計算を検討していく．

(2) GPGPU のハード構成

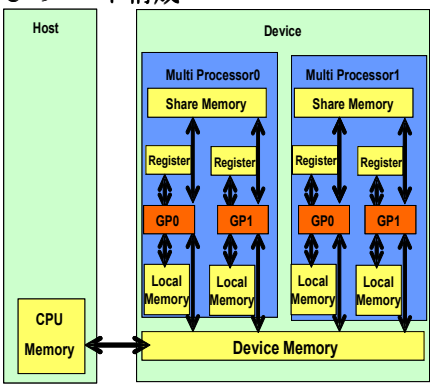


図 2.1 GPGPU のハード構成

GPGPU は，PC 本体 Host(CPU)側の処理と Device(GPU)側の処理で構成されている[1]．Device 側の構成は，Device Memory，Local Memory，Register，Share Memory となっている．Device Memory とは，カーネルの入出力データが置かれるメモリで，外部メモリであり，大容量である．CPU 上のメモリとやりとりをおこなうのはこの部分である．Local Memory とは，GPU チップ外のデバイスメモリに配置され，Register に一度ロードしてからのみ利用可能である．主にローカル変数の退避領域として利用する．400 から 600 サイクルと動作が非常に低速である．Register とは，GPU 内に実装され，カーネル関数のローカル変数を保持するメモリである．この Register の動作は非常に高速である．Share Memory とは，Device の Multi Processor 内で共有されるメモリである．16KB 程度で小容量であるが，Register 並みに動作が高速である．

(3) GPGPU のソフト構成

GPGPU のソフト構成を以下で説明する．

ここではソフトウェア環境 CUDA について説明する．CUDA は Thread，Block，Grid から形成される．Thread とは 1 つの処理過程を示し，複数の Thread をまとめたものを Block と呼ぶ．1Block に最大 512 までの Thread を定義できる．1 つの Block 内では同期命令によって同期処理を行うことができるが，Block 間での同期処理はできない．複数の Block をまとめたものを Grid と呼び，1 つの Grid 内に最大 65535 までの Block を定義できる．これらの Thread，Block，Grid にさせる処理を kernel と呼ぶ．

図 2.2 は GPGPU の構成を示す．Host における矢印は処理の流れを示している．Host 側で Kernel が呼び出されると，Device 側で Kernel を Thread，Block に割り当てて実行する．

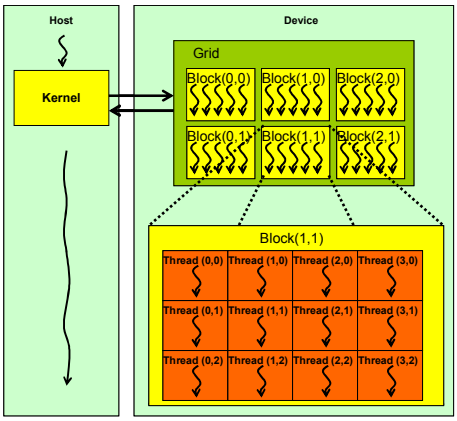


図 2.2 GPGPU のソフト構成

(4) NVIDIA Compute Visual Profiler

CUDA を用いてプログラムを作成するには，プログラムの特性を調べ，遅延の原因となっている部分を確認しながら，スレッドやブロックの使用を考慮した最適なコーディングする必要がある．特性とは，(1)プログラム全体の実行時間の内訳，(2)カーネルにおけるレジスタやメモリの使用状況，(3)カーネルの命令数などがある．NVIDIA Compute Visual Profiler ではこれらの定量値を調べることができる．

3. 並列アルゴリズムによる 1 層迷路配線法

(1) 並列 1 層迷路配線法

データ量が多くなって，短 TAT 化が必要となってきている．そのために GPGPU を使用した並列配線法の検討を行う．並列多層配線法の研究の前段階として，迷路配線法による並列 1 層迷路配線法の説明をする．

並列迷路配線法において，マップメモリについては同様に用いる．一方で並列化にあたり，検索点がそれぞれ同一方向同時に検索を行うようにする．そのため候補点リスト L2 に同時に書き込めるように変更する必要がある．

並列 1 層迷路配線法のアルゴリズムを以下に示す．

［並列 1 層迷路配線法］
Step0. 検索点リスト L1 と候補点リスト L2 を用意する
Step1. 始点(S)をリスト L1 に入れる
Step2. もし L1=φなら Step6 へ，そうでなければリスト L1 の点p全てについて，候補点リスト L2 の領域を独立に確保し，スレッドを割付けて，以下の処理を並列に繰り返す
Step2.1. 隣接する上下左右の点が配線可能なら，L2 に追加する
Step2.2. 終点なら Step4 へ行く
Step3. リスト L2 にオフセット処理を行い，リスト L1 にコピー，リスト L2＝φとして Step2 へ
Step4. トレースバックコードに従って配線形状を計算し出力する
Step5. 終了
Step6. 未配線で終了
図 3.1 並列 1 層迷路配線法のアルゴリズム

まず，Step0 で検索点リスト L1 と候補点リスト L2 を生成する(L1=L2=φ)，次に Step1 で，検索点リスト L1 に始点(S)を登録する．次の Step2 では，検索点リスト L1 の点 p 全てに候補点リスト L2 の領域を独立に確保し，スレッドを割り付けて，以下 Step2.1～Step2.2 までを繰り返す．なおこのとき L1=φなら Step6 へ行き，終了する．

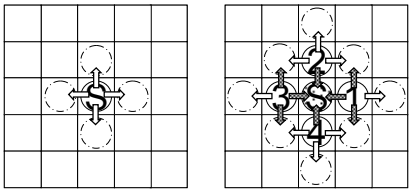
リスト L1 の点 p の全ての点に対して，Step2.1 で上下左右に隣接するグリッドを同一方向同時に検索し，配線可能なら，そのグリッドを候補点リスト L2 に点として追加し，そのマップにトレースバックコードを記入する．もし隣接するグリッドが配線不可能で，かつ終点(T)であれば次の Step2.2 で Step4 へ行く．

Step3 は，検索点 L1 が検索した候補点リスト L2 についてオフセット処理(以降で説明)を行い，新たに検索点リストとしてリスト L1 に全要素点をコピーし，L2 を空(L2=φ)にして，再び Step2 に戻る処理である．

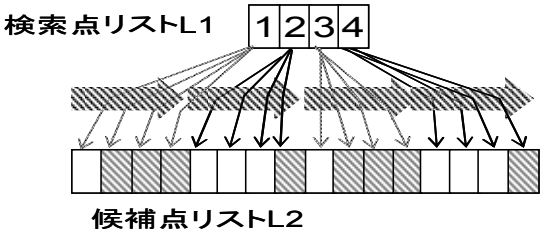
Step4 は，終点(T)を発見した場合の処理で，終点からトレースバックコードを使って始点(S)までの配線経路(座標)求め，出力する処理である．

Step5 は，配線経路を出力した場合の終了処理である．また，Step6 は，配線経路が見つからなかった場合の終了処理である．

本手法では，並列処理で異なる検索点による候補点が L2 に上書きされてデータ欠損しないために，候補点リスト L2 の領域を独立に確保する必要がある．これを，図 3.2 で説明する．図 3.2 では，リスト L1 の要素数が 4 個，｜L1｜＝4 の例を示す．図 3.2(a)，図 3.2(b)では，破線で描かれた円は検索点リストの点から隣接点を検索した際に配線可能であるグリッドを示している．図 3.2(a)は始点(S)で検索を行ったときの様子を示し，図 3.2(b)は図 3.2(a)の検索された点 1～4 で検索を行ったときを示している．図 3.2(c)は図 3.2(b)の検索点リスト L1 と候補点リスト L2 の様子を示している．図 3.2(c)では，候補点リスト L2 において斜線の引かれた点は配線できないグリッドを表している．太線はそれぞれの検索点から並列に検索を行っている様子を示している．並列 1 層迷路配線における候補点数は検索点リストを N とした場合，上下左右の 4 方向を同一方向同時に検索するために候補点リストは 4N の領域を確保すればよい．



(a)始点(S)からの検索 (b)検索点 1～4 からの検索



(c) 検索点リスト L1 と候補点リスト L2 の例
図 3.2 GPGPU を用いた並列 1 層迷路配線法の例

候補点リスト L2 の領域を独立に確保した領域は，使われない場合も生じる．そのため候補点リスト L2 の不要部を圧縮するオフセット処理を行う．

図 3.2(c)では配線可能な候補点数は 8 個であるが，リスト L2 は候補点数が 16 個となっている．検索を繰り返すにつれて，始点終点間の距離が長い場合は，無駄な領域が膨大になり，利用可能なメモリ領域を圧迫してしまう．そのためオフセット処理が必要となる．

オフセット処理を図 3.3 に示す．有効な候補点のみのリストを次検索点リスト L3 と呼ぶこととする．オフセット処理によりリストの位置が定まると，次検索点リスト L3 は有効な候補点ごとにスレッドを割り付けて，並列に書き込みをすることで処理時間の短縮を行える．

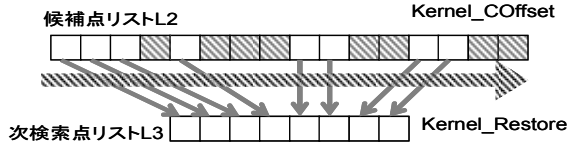


図 3.3 候補点リスト L2 のオフセット処理

提案手法ではこのオフセット処理の並列化による高速化を図る．図 3.4 は本手法のオフセット処理である．候補点リスト L2 について直接オフセット処理を行うのではなく，検索点リスト L1 を利用してリスト L2 の有効な候補点を次検索点リスト L3 に書き出す，書き出し先頭番地を計算する．この方法により，オフセット処理を行うのは検索点リストの点数分だけでよく，計算対象は図 3.3 におけるオフセット処理と比較して 4 分の 1 となる．本手法のオフセット処理では更に高速化を目指して，図 3.5 に示すように検索

点リスト L1 を 2 分割して，オフセット処理計算の並列化を行う．

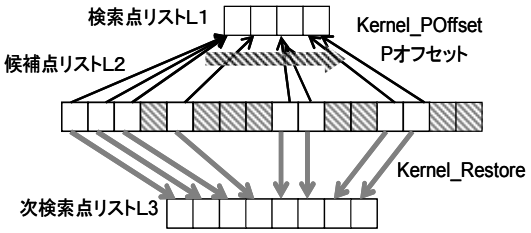


図 3.4　改良したオフセット処理

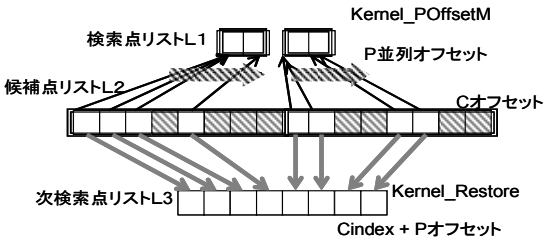


図 3.5　オフセット処理の並列化

1 層迷路配線法のアルゴリズムの動作例について図 3.6 に示したサイズ 7×7 グリッドの配線領域に始点終点を定義した配線問題を使って説明する．なお以下，図 3.6(b)から図 3.6(f)では，検索点リスト L1 の点を青色，候補点リスト L2 の点を黄色の円でそれぞれ示す．また検索方向を細矢印，トレースバックコードを太矢印で示すものとする．

まず Step0 では検索点リスト L1 と検索点リスト L2 を用意し，次に Step1 で，リスト L1 に始点(S)の座標(x_s,y_s)を登録する．この段階の様子を図 3.6(a)に示す．

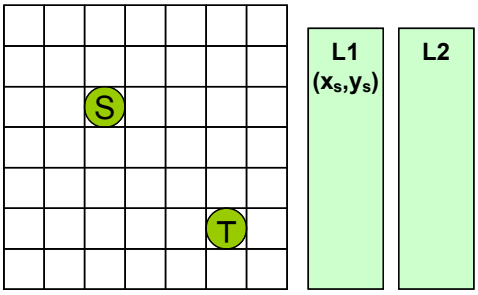


図 3.6(a) Step1 後の配線モデルとリスト

次に Step2 では，検索点リスト L1 の点 p 全てに候補点リスト L2 の領域を独立に確保し，スレッドを割り付けて，Step2.1 および Step2.2 を繰り返す，すなわち Step2.1 でリスト L1 の検索点 p(x_s,y_s)に隣接する上下左右のグリッドの検索をして，配線可能なグリッドである P1(x_1,y_1)，P2(x_2,y_2)，P3(x_3,y_3)，P4(x_4,y_4)をリスト L2 に登録する．このとき，グリッド上の P1，P2，P3，P4 にトレースバックコードを書く．Step2.2 の判定は偽となるため無視する．以上で Step2 が終わる．この段階の様子を図 3.6(b)に示す．

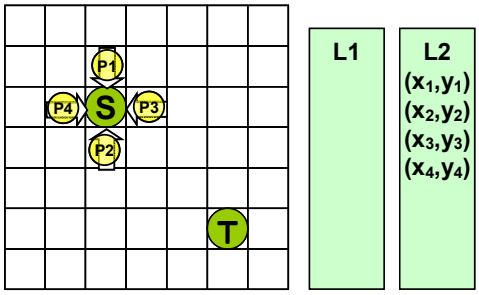


図 3.6(b) Step2 後の配線モデルとリスト

次に Step3 でリスト L2 のオフセット処理を行い，リスト L1 にコピーし，リスト L2 を空 (L2= ϕ)にして，再び Step2 に戻る．この段階の様子を図 3.6(c)に示す．

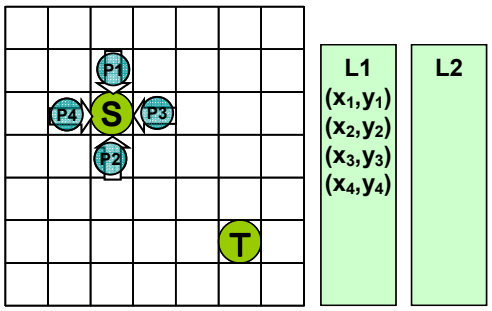


図 3.6(c) Step3 後の配線モデルとリスト

再び Step2 では，検索点リスト L1 の点 p 全てに候補点リスト L2 の領域を独立に確保し，スレッドを割り付けて，Step2.1 および Step2.2 を繰り返す．

Step2.1 ではリスト L1 の検索点 P1~P4 にリスト L2 の領域を独立に確保し，上方向を同時に検索してリスト L2 に登録し，グリッドにトレースバックコードを書く．この様子を図 3.6(d)に示す．繰り返しとして，リスト L1 の検索点 P1~P4 で，下方向を同時に検索してリスト L2 に追加登録し，グリッドにトレースバックコードを書く．この様子を図 3.6(e)に示す．繰り返しとして，リスト L1 の検索点 P1~P4 で左方向，右方向についても同時に検索して，リスト L2 にそれぞれ追加登録し，グリッドにトレースバックコードを書く．

以上で，得られた L2 にオフセット処理を行う．オフセット処理後の様子を図 3.6(f)に示す．オフセット処理を行ったリスト L2 を Step3 でリスト L1 にコピーして，再び Step2 に戻る．これらを繰り返して Step2 で終点を検索して Step4 でトレースバックコードに従って，終点(T)から始点(S)までの経路データを出力して Step5 で終了する．

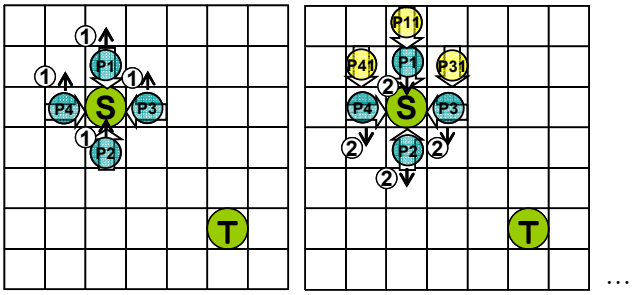


図 3.6(c)上方向の検索　　図 3.6(d)下方向の検索

図 3.6(e)オフセット処理後の配線モデル

4.　並列アルゴリズムによる多層迷路配線法

(1)　並列多層迷路配線法

1 層配線では，多くの端子を配線する場合，先に配線を行われたネットが，新たな配線を行う際の障害物となり，配線不可能になってしまうため配線できない可能性が高い．そこでより配線確率を高めた実用的な多層配線法への拡張を考える．

ここではHV ルール利用して，1 層配線法を拡張して多層配線法を説明する．GPGPU は SIMD (Simple Instruction Multiple Data)構成であり，これを生かすために，最上層と最下層をプロテクトする．そのため奇数番目の層での検索，偶数番目の層での検索の 2 種類だけ考慮する．

並列多層迷路配線法のアルゴリズムを以下に示す．

[並列迷路配線法(多層)]
Step0．検索点リスト L1 と候補点リスト L2 を用意する
Step1．始点(S)をリスト L1 に入れる
Step2．もし L1=φなら Step6 へ，そうでなければリスト L1 の点p全てについて，候補点リスト L2 の領域を独立に確保し，スレッドを割付けて，以下の処理を並列に繰り返す
**　Step2.1．スレッドの点が奇数番目の層なら，隣接する上下と上層下層を検索**
**　　Step2.1.1．配線可能なら，L2 に追加する**
**　　Step2.1.2．終点なら Step4 へ**
**　Step2.2．スレッドの点が偶数番目の層なら，隣接する左右と上層下層を検索**
**　　Step2.2.1．配線可能なら，L2 に追加する**
**　　Step2.2.2．終点なら Step4 へ行く**
Step3．リスト L2 にオフセット処理を行い，リスト L1 にコピー，リスト L2＝φとして Step2 へ
Step4．トレースバックコードに従って配線形状を計算し出力する
Step5．終了
Step6．未配線として終了

図 4.1　並列迷路配線法(多層)のアルゴリズム

まず，Step0 で検索点リスト L1 と候補点リスト L2 を生成する(L1=L2=φ)，次に Step1 で，検索点リスト L1 に始点(S)を登録する．次の Step2 では，検索点リスト L1 の点 p 全てに候補点リスト L2 の領域を独立に確保し，スレッドを割り付けて，以下 Step2.1～Step2.2 までを繰り返す．なおこのとき L1=φなら Step6 へ行き，終了する．

リスト L1 の点 p の全ての点に対して，Step2.1 は点 p が奇数番目の層の場合，Step2.1.1 で上下と上層下層に隣接するグリッドを同一方向同時に検索し，配線可能なら，そのグリッドを候補点リスト L2 に点として追加し，そのマップにトレースバックコードを記入する．もし隣接するグリッドが配線不可能で，かつ終点(T)であれば次の Step2.1.2 で Step4 へ行く．

また，Step2.2 は点 p が偶数番目の層の場合，Step2.2.1 で左右と上層下層に隣接するグリッドを同一方向同時に検索し，配線可能なら，そのグリッドを候補点リスト L2 に点として追加し，そのマップにトレースバックコードを記入する．もし隣接するグリッドが配線不可能で，かつ終点(T)であれば次の Step2.2.2 で Step4 へ行く．

Step3 は，検索点 L1 が検索した候補点リスト L2 についてオフセット処理(以降で説明)を行い，新たに検索点リストとしてリスト L1 に全要素点をコピーし，L2 を空(L2=φ)にして，再び Step2 に戻る処理である．

Step4 は，終点(T)を発見した場合の処理で，終点からトレースバックコードを使って始点(S)までの配線経路(座標)求め，出力する処理である．

Step5 は，配線経路を出力した場合の終了処理である．また，Step6 は，配線経路が見つからなかった場合の終了処理である．

(2)　実験

提案手法をプログラム開発環境 MS Visual C++ 2008 Express Edition と NVIDIA 社の CUDA でC言語により実装した．

使用した使用したマシンのスペックを表 4.1 に示す．以降では，グラフィックカード名で説明する．

配線領域のサイズとして，サイズ 20×20，30×30，40×40 グリッドを使用し，サイズ依存性を調べる．また，配線済み経路に対する依存性を調べるため，それぞれに関して 10，20，30 の 2 端子ネット数で実験を行う．また端子については，配線領域にランダムに配置した 2 端子ネットを使用する．

GeForce GT 220(48 コア)，GeForce 9400 GT(16 コア)，GeForce GTX 480(480 コア)での結果を表 4.2 から表

4.10 に示す. 単位は ms である.

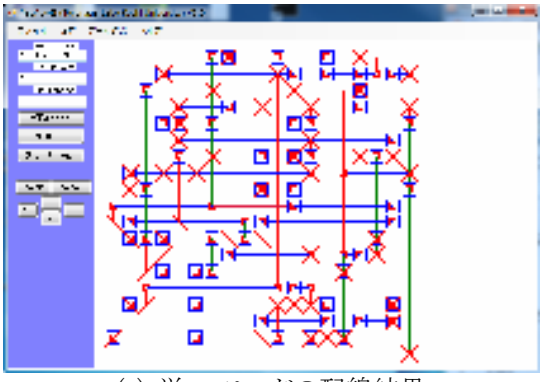
20×20 グリッドにおける実験結果を表 4.2 から表 4.4 に示す.

GeForce GT 220(48 コア)では, 2 端子ネット数 10 の 5 層配線において, 単一スレッドでは 114.8ms, 並列スレッドでは 22.5ms から単一スレッドに比べて並列スレッドでは処理スピードが 5.1 倍となることが確認された. また並列オフセット処理を加えた並列スレッド Tm2 では 14.9ms となり処理スピードが 7.7 倍となることが確認された.

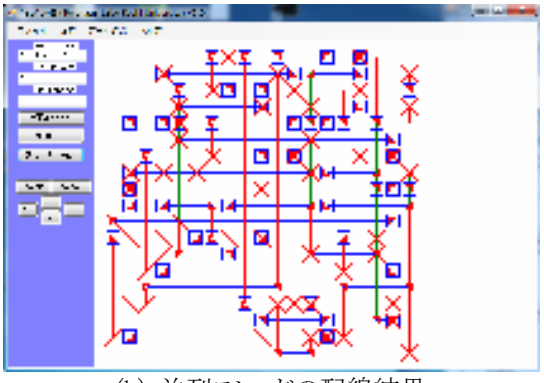
GeForce 9400 GT(16 コア)では, 2 端子ネット数 10 の 5 層配線において, 単一スレッドでは 99.2ms, 並列スレッドでは 29.9ms から単一スレッドに比べて並列スレッドでは処理スピードが 3.3 倍となることが確認された. また並列オフセット処理を加えた並列スレッド Tm2 では 19.2ms となり処理スピードが 5.2 倍となることが確認された.

GeForce GTX 480(480 コア)では, 単一スレッドでは 31.1ms, 並列スレッドでは 21.4ms から単一スレッドに比べて並列スレッドでは処理スピードが 1.4 倍となることが確認された. また並列オフセット処理を加えた並列スレッド Tm2 では 21.4ms となり処理スピードが 1.5 倍となることが確認された.

また 2 端子ネット数 30 の場合, 全ての結果について 2 層配線において未配線が発見された. ここで参考として, 20×20 グリッド, 2 端子ネット数 30 での 5 層配線の配線結果について, 図 4.2 に示す. 図 4.2 について, 1 層目が赤, 2 層目が青, 3 層目が緑, 4 層目が紫, 5 層目がオレンジとなっている.



(a) 単一スレッドの配線結果



(b) 並列スレッドの配線結果

図 4.2 20×20 サイズ 2 端子ネット数 30 の 5 層配線結果

次に 30×30 グリッドにおける実験結果を表 4.5 から表 4.7 に示す.

GeForce GT 220(48 コア)では, 2 端子ネット数 10 の 5 層配線において, 単一スレッドでは 270.3ms, 並列スレッドでは 42.4ms から, 単一スレッドに比べて並列スレッドでは処理スピードが 6.4 倍となることが確認された. また並列オフセット処理を加えた並列スレッド Tm2 では 24.0ms となり処理スピードが 11.3 倍となることが確認された.

GeForce 9400 GT(16 コア)では, 2 端子ネット数 10 の 5 層配線において, 単一スレッドでは 232.7ms, 並列スレッドでは 65.4ms から, 単一スレッドに比べて並列スレッドでは処理スピードが 3.6 倍となることが確認された. また並列オフセット処理を加えた並列スレッド Tm2 では 30.2ms となり処理スピードが 7.7 倍となることが確認された.

GeForce GTX 480(480 コア)では, 2 端子ネット数 10 の 5 層配線において, 単一スレッドでは 54.3ms, 並列スレッドでは 28.8ms から, 単一スレッドに比べて並列スレッドでは処理スピードが 1.9 倍となることが確認された. また並列オフセット処理を加えた並列スレッド Tm2 では 25.5ms となり処理スピードが 7.7 倍となることが確認された.

最後に 40×40 グリッドにおける実験結果を表 4.8 から表 4.10 に示す.

GeForce GT 220(48 コア)では, 2 端子ネット数 10 の 4 層配線において, 単一スレッドでは 371.7ms, 並列スレッドでは 53.4ms から, 単一スレッドに比べて並列スレッドでは処理スピードが 7.0 倍となることが確認された. また並列オフセット処理を加えた並列スレッド Tm2 では 29.2ms となり処理スピードが 12.7 倍となることが確認された.

GeForce 9400 GT(16 コア)では, 2 端子ネット数 10 の 4 層配線において, 単一スレッドでは 316.6ms, 並列スレッドでは 68.0ms から, 単一スレッドに比べて並列スレッドでは処理スピードが 4.7 倍となることが確認された. また並列オフセット処理を加えた並列スレッド Tm2 では 37.0ms となり処理スピードが 8.6 倍となることが確認された.

GeForce GTX 480(480 コア)では, 2 端子ネット数 10 の 4 層配線において, 単一スレッドでは 68.1ms, 並列スレッドでは 32.4ms から, 単一スレッドに比べて並列スレッドでは処理スピードが 2.1 倍となることが確認された. また並列オフセット処理を加えた並列スレッド Tm2 では 28.1ms となり処理スピードが 2.4 倍となることが確認された.

なお, 2 端子ネット数 10 以外の 4 層以上の配線では, スレッド数の制限を越えたため結果が得られなかった.

(3) 考察

表 4.2 から表 4.10 から, 並列化により単一スレッドでの迷路配線法と比べて, 処理スピードが最大 12.7 倍となることが確認された. 層が増えるにつれて並列化の効果が確認された. ネット数に関しては, ネット数が少ないほど並列化の効果が確認された. 配線領域に関しては, 配線領域が広い配線モデルでは並列化の効果が確認された.

20×20 グリッド, 2 端子ネット数 30 の 2 層配線で未配線があったが, これは単に配線経路が見つからなかったためだと考えられる. また, 40×40 グリッドにおける問題は検索点の数が同期処理の行えるスレッドの総数 512 を超えてしまったためだと考えられる. 図 4.4 は 40×40 グリッド, 2 端子ネット数 30 の GPU 多層並列配線について, 3 層配線と 5 層配線について示している. 図 4.4 について, 1 層目が赤, 2 層目が青, 3 層目が緑, 4 層目が紫, 5 層目がオレンジとなっている. 黒く引かれた配線は同一の 2 端子ネットを示している. これにより 5 層配線において最短経路を通過していないのが確認できる.

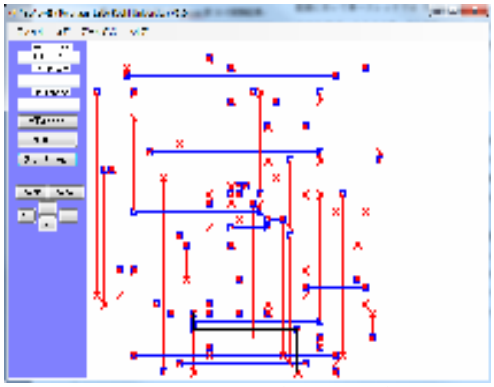


図 4.4(a) 3 層配線での配線結果

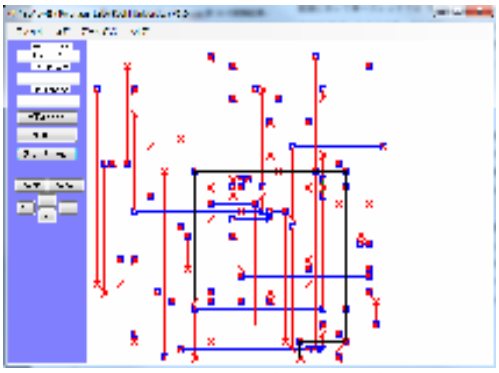


図 4.4(b) 5 層配線での配線結果

5. まとめ

本論文では, GPGPU を用いた並列迷路配線法の検討した. まず並列 1 層配線法について, 実装方法の提案と障害物を使用した配線モデルでの評価実験を行った. その後, 実用化のために並列 1 層配線法を並列多層配線法に拡張した. 並列多層配線法についても同様に実装方法の提案をし, 複数の 2 端子ネットをランダムに配置した配線モデルでの評価実験を行った.

GPGPU の利用として, 迷路配線法において, 検索点リストの点を同数のスレッドで並列に検索して高速化を行った. また候補点リストの効率化のために, オフセット処理についても並列化することで高速化を行った.

並列 1 層配線法では, 20×20, 100×100 グリッドの配線領域に障害物を与えた配線モデルで, 単一スレッドでの迷路配線法に比べて, 処理スピードが最大 8.3 倍となることが確認された. 一方で障害物が多いほど高速化の効果が減少した.

並列多層配線法では, 1 辺 20 から 40 グリッドの配線領域に, ランダムに 10 から 30 までの 2 端子ネットを与えた配線モデルで, 単一スレッドに比べて処理スピードが最大 12.7 倍となることが確認された. 一方で配線領域や検索層を増やすことでスレッド数の制限を越えたため結果が得られなかった.

今後の課題としては, 大規模化に向けたアルゴリズムの改良が挙げられる.

謝辞

本研究を科学研究費補助金(課題番号 22500049)として助成いただいた独立行政法人日本学術振興会に感謝いたします.

参考文献

[1] NVIDIA CUDA Programming Guide Version 1.0.

[2] C. Y. Lee, “An algorithm for path connections and its application,” IRE Trans. Electron. Comput., pp. 346-365(1961).

[3] Lijuan Luo, Martin Wong, Wen-mei Hwu, ″Effective GPU Implementation of Breadth-First Search,″ dac2010, pp52-55(2010).

[4] Debapriya Chatterjee, Andrew DeOrio and Valeria Bertacco, ″GCS: High-Performance Gate-Level Simulation with GP-GPUs,″ Proc of Design, Aotomation, and Test in Europe - DATE, pp.1332-1337(2009).

[5] Yifang Liu, Jiang Hu,“GPU-Based Parallelization for Fast Circuit Optimization,” dac2009, pp943-946(2009).

[6] James Williamson, Yinghai Lu, Li Shang, Hai Zhou, Xuan Zen, ″Parallel Cross-Layer Optimization of High-Level Synthesis and Physical Design,″ aspdac11, p467-472(2011).

[7] 中井駿介, 藤井良弥, 寺田翔太, 村岡道明, 豊永昌彦 ”GPGPU を用いた迷路配線実装の一手法,” DA シンポジウム 2011.

表 4.1 実験環境

グラフィックカード	GeForce GT 220	GeForce 9400 GT	GeForce GTX 480
CPU	Intel Core i5-760 2.8GHz	Intel Core i7-950 3.07GHz	Intel Core 2 Duo E600 3.33GHz
メインメモリ	4GB	2.99GB	3.0GB
CUDA	version 3.2	version 2.3	version 2.3
CUDAプロセッサコア	48	16	480
グラフィッククロック(MHz)	625MHz	550MHz	700MHz
プロセッサクロック(MHz)	1360MHz	1400MHz	1401MHz
メモリクロック(MHz)	790	400	1848
標準メモリ設定	1 GB DDR3	512MHz	1536 GB GDDR5
メモリインターフェース幅	128-bit	128-bit	384-bit
メモリバンド幅(GB/sec)	25.3	12.8	177.4

表 4.2 GeForce GT 220(48 コア)での 20×20 グリッドにおける実験結果

	20*20 10pinpair rand1 (ms)					20*20 20pinpair rand1 (ms)					20*20 30pinpair rand1 (ms)				
layer	Single	Multi	Multi2	S/M	S/M2	Single	Multi	Multi2	S/M	S/M2	Single	Multi	Multi2	S/M	S/M2
2	41.6	22.6	16.2	1.8	2.6	74.5	56.1	35.4	1.3	2.1	—	—	—		
3	63.5	25.3	17.3	2.5	3.7	125.3	61.6	37.2	2.0	3.4	152.7	63.6	56.6	2.4	2.7
4	82.7	27.7	18.3	3.0	4.5	171.2	67.1	39.5	2.6	4.3	227.0	71.8	59.6	3.2	3.8
5	99.2	29.9	19.2	3.3	5.2	212.1	72.5	41.5	2.9	5.1	289.1	80.1	62.9	3.6	4.6

表 4.3 GeForce 9400 GT(16 コア)での 20×20 グリッドにおける実験結果

	20*20 10pinpair rand1 (ms)					20*20 20pinpair rand1 (ms)					20*20 30pinpair rand1 (ms)				
layer	Single	Multi	Multi2	S/M	S/M2	Single	Multi	Multi2	S/M	S/M2	Single	Multi	Multi2	S/M	S/M2
2	47.7	14.9	12.3	3.2	3.9	84	31	27	2.7	3.1	—	—	—		
3	72.9	17.7	13.3	4.1	5.5	143	37	29	3.9	5.0	174.02	53.88	43.20	3.2	4
4	95.4	20.3	14.1	4.7	6.8	197	43	30	4.6	6.5	261.14	62.79	45.73	4.2	5.7
5	114.8	22.5	14.9	5.1	7.7	245	49	32	5.0	7.6	333.41	71.33	48.41	4.7	6.9

表 4.4 GeForce GTX 480(480 コア)での 20×20 グリッドにおける実験結果

	20*20 10pinpair rand1 (ms)					20*20 20pinpair rand1 (ms)					20*20 30pinpair rand1 (ms)				
layer	Single	Multi	Multi2	S/M	S/M2	Single	Multi	Multi2	S/M	S/M2	Single	Multi	Multi2	S/M	S/M2
2	21.6	20.4	20.2	1.1	1.1	41.8	41.2	41.0	1.0	1.0	—	—	—		
3	25.2	21.4	20.7	1.2	1.2	50.2	43.6	42.4	1.1	1.2	69.5	64.7	63.5	1.1	1.1
4	28.3	22.1	21.1	1.3	1.3	57.8	45.3	43.3	1.3	1.3	81.9	67.5	64.9	1.2	1.3
5	31.1	22.6	21.4	1.4	1.5	64.8	46.8	44.0	1.4	1.5	92.6	69.9	66.1	1.3	1.4

表 4.5 GeForce GT 220(48 コア)での 30×30 グリッドにおける実験結果

	30*30 10pinpair rand1 (ms)					30*30 20pinpair rand1 (ms)					30*30 30pinpair rand1 (ms)				
layer	Single	Multi	Multi2	S/M	S/M2	Single	Multi	Multi2	S/M	S/M2	Single	Multi	Multi2	S/M	S/M2
2	112.5	25.3	18.3	4.5	6.1	201.8	49.2	37.3	4.1	5.4	268.8	70.8	56.6	3.8	4.8
3	171.2	31.6	20.4	5.4	8.4	317.0	61.9	41.5	5.1	7.6	444.3	89.0	61.8	5	7.2
4	221.2	37.6	22.5	5.9	9.9	421.3	73.6	45.3	5.7	9.3	603.8	106.7	67.9	5.7	8.9
5	270.3	42.4	24.0	6.4	11.3	522.5	84.0	48.7	6.2	10.7	756.8	122.4	73.1	6.2	10.4

表 4.6 GeForce 9400 GT(16 コア)での 30×30 グリッドにおける実験結果

	30*30 10pinpair rand1 (ms)					30*30 20pinpair rand1 (ms)					30*30 30pinpair rand1 (ms)				
layer	Single	Multi	Multi2	S/M	S/M2	Single	Multi	Multi2	S/M	S/M2	Single	Multi	Multi2	S/M	S/M2
2	95.2	49.1	23.5	1.9	4.1	173.4	101.5	47.9	1.7	3.6	233.5	101.9	72.4	2.3	3.2
3	143.6	55.3	26.0	2.6	5.5	270.9	113.9	52.9	2.4	5.1	381.3	118.6	78.4	3.2	4.9
4	189.4	61.1	28.3	3.1	6.7	363.2	125.2	57.7	2.9	6.3	518.9	135.8	85.5	3.8	6.1
5	232.7	65.4	30.2	3.6	7.7	449.3	134.7	61.8	3.3	7.3	648.1	151.3	91.9	4.3	7.1

表 4.7 GeForce GTX 480(480 コア)での 30×30 グリッドにおける実験結果

	30*30 10pinpair rand1 (ms)					30*30 20pinpair rand1 (ms)					30*30 30pinpair rand1 (ms)				
layer	Single	Multi	Multi2	S/M	S/M2	Single	Multi	Multi2	S/M	S/M2	Single	Multi	Multi2	S/M	S/M2
2	30.7	24.1	23.1	1.3	1.3	58.2	48.2	46.6	1.2	1.3	82.8	71.3	69.5	1.2	1.2
3	39.2	26.0	24.1	1.5	1.6	75.1	51.9	48.6	1.4	1.5	108.2	76.9	72.4	1.4	1.5
4	47.0	27.5	24.9	1.7	1.9	90.9	54.9	50.0	1.7	1.8	131.7	81.6	74.7	1.6	1.8
5	54.3	28.8	25.5	1.9	2.1	105.7	57.5	51.2	1.8	2.1	154.2	85.7	76.7	1.8	2.0

表 4.8 GeForce GT 220(48 コア)での 40×40 グリッドにおける実験結果

	40*40 10pinpair rand1 (ms)					40*40 20pinpair rand1 (ms)					40*40 30pinpair rand1 (ms)				
layer	Single	Multi	Multi2	S/M	S/M2	Single	Multi	Multi2	S/M	S/M2	Single	Multi	Multi2	S/M	S/M2
2	185.7	35.3	23.2	5.3	8.0	321.0	65.0	44.4	4.9	7.2	489.2	101.1	70.1	4.8	7.0
3	281.3	45.4	26.5	6.2	10.6	495.9	83.1	50.3	6	9.9	764.3	131.1	80.4	5.8	9.5
4	371.7	53.4	29.2	7.0	12.7	658.8	—	—			1023.1	—	—		
5	456.3	—	—			812.1	—	—			1271.8	—	—		

表 4.9 GeForce 9400 GT(16 コア)での 40×40 グリッドにおける実験結果

	40*40 10pinpair rand1 (ms)					40*40 20pinpair rand1 (ms)					40*40 30pinpair rand1 (ms)				
layer	Single	Multi	Multi2	S/M	S/M2	Single	Multi	Multi2	S/M	S/M2	Single	Multi	Multi2	S/M	S/M2
2	158.9	50.6	29.7	3.1	5.4	276.6	112.6	57.1	2.5	4.8	419.6	157.8	89.6	2.7	4.7
3	240.2	60.5	33.7	4.0	7.1	424.8	130.5	64.3	3.3	6.6	658.4	187.2	101.4	3.5	6.5
4	316.6	68.0	37.0	4.7	8.6	564.3	—	—			880.1	—	—		
5	390.2	—	—			696.3	—	—			1092.7	—	—		

表 4.10 GeForce GTX 480(480 コア)での 40×40 グリッドにおける実験結果

	40*40 10pinpair rand1 (ms)					40*40 20pinpair rand1 (ms)					40*40 30pinpair rand1 (ms)				
layer	Single	Multi	Multi2	S/M	S/M2	Single	Multi	Multi2	S/M	S/M2	Single	Multi	Multi2	S/M	S/M2
2	41.4	27.7	25.7	1.5	1.6	54.2	31.6	28.4	1.7	1.9	82.0	49.0	44.1	1.7	1.9
3	55.3	30.4	27.0	1.8	2.0	79.7	36.7	31.0	2.2	2.6	123.1	57.1	48.3	2.2	2.6
4	68.1	32.4	28.1	2.1	2.4	103.4	—	—			160.6	—	—		
5	80.5	—	—			125.6	—	—			196.8	—	—		