Vol.2011-HPC-130 No.9
2011/7/27

インタプリタ型汎用生体シミュレータ insilicoSim の GPU による高速化

奥山倫弘[†1,†4] 置田真生[†1] 安部武志[†2]
浅井義之[†2] 野村泰伸[†3] 萩原兼一[†1]

*insilico*Sim は多数の多様な常微分方程式から構成されるヘテロな生体モデルを扱う汎用生体シミュレータである．多様なモデルを扱うために，このシミュレータはシミュレーションに関わる数式を表す独自形式のバイトコード用インタプリタとして動作する．本稿ではこのインタプリタを GPU（Graphics Processing Unit）に実装し，シミュレーションを高速化する手法を述べる．数式間のデータ依存に基づくレベルスケジューリングを用い，相互依存のない数式を並列計算する．スレッド間での処理の分岐を避けるために，類似した式を同じワープのスレッドに割り当てる．また，ワープごとにバイトコードを単一化し，メモリ参照を削減する．結果，約 4000 個の式を含むモデルを CPU での単一コアと比べ 13.6 倍高速にシミュレーションできた．

Accelerating Interpreter of General Biophysical Simulatior 'insilicoSim' on the GPU

Tomohiro Okuyama,[†1,†4] Masao Okita,[†1]
Takeshi Abe,[†2] Yoshiyuki Asai,[†2] Taishin Nomura[†3]
and Kenichi Hagihara[†1]

*insilico*Sim is a general biophysical simulator for heterogeneous biophysical models that are consists of many and manifold ordinary differential equations. In order to simulate a variety of models, this simulator operates as an interpreter for an internal byte code representation of simulation related mathematical expressions (MEs). This paper describes an acceleration method of this simulator by implementing its interpreter on the graphics processing unit (GPU). We use level scheduling under the constraint of data dependence among MEs to compute independent MEs in parallel. Our method assigns similar MEs to threads in the same warp to reduce divergent branches. We also reduce memory access by unifying byte codes interpreted by a warp. Our method simulates a model containing 4000 MEs 13.6X faster than that on a core of the CPU.

1. はじめに

近年，生物学分野において生体機能の数理モデル化が広く研究されている．これらの研究では，生体の数理モデルに基づくシミュレーションにより，生体機能の理解が試みられている．生体モデル記述言語である ISML[1),2)] は神経細胞や心筋細胞などの特定領域のモデル化に特化しない，汎用の生体モデル記述言語である．この言語は，細胞内小器官，細胞および臓器など複数の階層を含むマルチスケールなモデルを記述できる．これらのモデルは互いに依存する複数種の常微分方程式（ODEs：Ordinary Defferential Equations）およびそれらが参照する数式から構成される．

*insilico*Sim[3)] は ISML モデルを扱う汎用生体シミュレータであり，モデル中の ODEs を連立させた初期値問題を数値的に解くことにより，時間発展型のシミュレーションを行う．また，このシミュレータは，ISML の拡張などに伴う機能追加を容易にするために拡張性を重視している．多様な ODEs への対応および高い拡張性を実現するために，*insilico*Sim は ODE を独自形式のバイトコードに変換し，解釈実行するインタプリタとして動作する．

生体モデルを用いた研究の進展に伴い，細胞などの生体部品を多数接続した大規模なモデルが作成されている．これらのモデルはシミュレーションが長時間におよぶため，その高速化の要求がある．そこで，文献 3) は MPI を用いた並列化により，大規模なモデルにおいて 8 コアのマルチコア CPU 上で 3.5 倍の速度向上を達成する．一方，初期化および通信時間を除いた計算の速度向上は 6 倍であり，通信が性能ボトルネックの一つと指摘している．

そこで，本研究では *insilico*Sim を共有メモリ型の並列化により高速化することを目指し，CUDA（Compute Unified Device Architecture）[4)] を用いて GPU（Graphics Processing Unit）による高速化を図る．多様なモデルを扱うために，入力モデルに応じて GPU 上で

†1 大阪大学大学院情報科学研究科コンピュータサイエンス専攻
Department of Computer Science, Graduate School of Information Science and Technology, Osaka University
†2 沖縄科学技術研究基盤整備機構 オープンバイオロジーユニット
Open Biology Unit, Okinawa Institute of Science and Technology
†3 大阪大学大学院基礎工学研究科機能創成専攻
Department of Mechanical Science and Bioengineering, Graduate School of Engineering Science, Osaka University
†4 日本学術振興会特別研究員 DC
Research Fellow of the Japan Society for the Promotion of Science

シミュレーションする ODEs を変更する仕組みが必要である．モデルは多様な ODE を含んでおり，ODE ごとに処理を分岐する必要がある．この時，GPU においては分岐に伴う CUDA スレッド間の divergent branch[4] を避ける必要がある．また，*insilico*Sim の特徴である拡張性を維持することが望ましい．

提案手法は *insilico*Sim のインタプリタを GPU 上に移植することにより，多様な ODE への対応および拡張性を保ちつつ高速化を図る．*insilico*Sim のバイトコードは 1 つの ODE および数式ごとに 1 つ生成される．そこで，GPU では 1 つのバイトコードを 1 つの CUDA スレッドにより解釈実行する．モデル中の式には互いに参照依存があり，この依存に従いながら並列計算するために，レベルスケジューリングを用いる．また，多数の細胞を接続したモデルにおいては，各細胞の構造は同様であるから，モデル中には類似する ODE が多い．この特徴に着目し，類似した式をまとめ，SIMD 型の計算機である GPU における演算効率を向上させる．さらに，類似した式のバイトコードを単一化し，メモリ参照量を削減する．

2. 関 連 研 究

GPU を用いて ODEs の初期値問題を高速に解く既存研究として，文献 5) および文献 6) が挙げられる．これらの研究では，ODE が 10 個以下の比較的小規模な ODEs について，初期値を変えた多数のインスタンスを生成し，それぞれのインスタンスについてシミュレーションする．1 インスタンスを 1 つの CUDA スレッドが担当し，多数のインスタンスを GPU 上で並列計算することにより，GPU の高い演算性能を活用する．

特に，文献 5) は生体モデル記述言語 SBML を用いて記述された生体モデルのシミュレーションを対象に，SBML から CUDA カーネルを生成するトランスレータを提案している．NVIDIA GeForce GTX 280 を用い，CPU に比べ 59.3 倍の速度向上を達成する．

本研究は，大規模なモデルに 1 つの初期値を与えた，1 つの問題インスタンスの高速化を目的とする．そのために，モデル中の数式や ODE を多数の CUDA スレッドにより並列計算する．文献 3) は，ODE の多い大規模モデルにおいてはトランスレータによる変換および生成されたコードのコンパイル時間がシミュレーション時間よりも長いと指摘している．したがって，モデルの変更およびシミュレーションを繰り返す実験およびモデル開発用途において利便性が低い．文献 3) はこの問題を解決する手法として，コードの変換時間が短いインタプリタ方式を用いる．同様の理由により，本研究においてもインタプリタを用いる．

文献 7) は，CUDA スレッドおよび CUDA スレッドが担当するデータの対応付けを動的に変更し，divergent branch を避ける手法を提案している．本研究では，事前に式を表すバイトコードおよび CUDA スレッドの対応付けを調整し，ワープ[4] 内のスレッドが処理するバイトコードを単一化することにより，divergent branch の削減を図る．

3. insilicoSim の概要

*insilico*Sim[3] は生体モデル記述言語 ISML[1] により記述されたモデルを扱う汎用生体シミュレータである．ISML は XML を基にした記述言語であり，細胞などの生体部品をモジュールと呼ばれる単位で記述する．モジュールは別のモジュールを含むカプセル化およびモジュール間同士の相互依存関係を定義できる．それぞれ，細胞および細胞内小器官のモデル化や，細胞間の相互作用のモデル化などに用いられる．

各モジュールにはシミュレーションにおける時刻 t を独立変数とする ODE，ODE が参照する数式・定数および ODE の初期条件が複数記述されている．以降，単純な数式および ODE をまとめて式と呼ぶ．モジュール間の依存関係を定義することにより，異なるモジュールの式を互いに参照できる．

*insilico*Sim における処理の流れは，初期化および ODEs の数値計算に別けられる（**図 1**）．まず，初期化では生体モデルを受け取り，モデル中に含まれる式，変数および定数を抽出し，式の依存関係を表す依存グラフを作成する．依存グラフ $G = (V, E)$ は式 $f \in V$ を頂点，式間の依存関係 $\langle f, g \rangle \in E$ を辺とする有向グラフである．辺 $\langle f, g \rangle$ は式 g が式 f に依存することを示す．ここで，g が参照する変数の値を f が決めるとき，g が f に依存するとする．MPI による並列計算を行う場合，依存グラフを分割し計算ノードに部分グラフを割り当てる．ノードへの計算割り当て後，各ノードは割り当てられた依存グラフから式の計算順序をスケジューリングする．このスケジューリングには依存グラフのトポロジカルソートを用いる．ただし，ODE により値が決まる変数は数値計算上，1 時刻前の値を参照することから，ODE を参照する依存は無視する．図 1 ではこの依存を破線で示す．その後，1 つの式につき 1 つのバイトコードを生成する．

以上の初期化後，初期化時に決定したスケジューリングに基づいて，式を表すバイトコードをインタプリタにより順に計算する．1 時刻分の計算後，時刻を進め，ユーザが指定したステップ数分の計算を繰り返し，モデル中の変数の時間変化を出力する．数値解法としては，Euler 法および Runge-Kutta 法を用いる．

4. GPU を用いた insilicoSim インタプリタ

提案手法は，*insilico*Sim のシミュレーション実行部であるインタプリタを GPU を用い

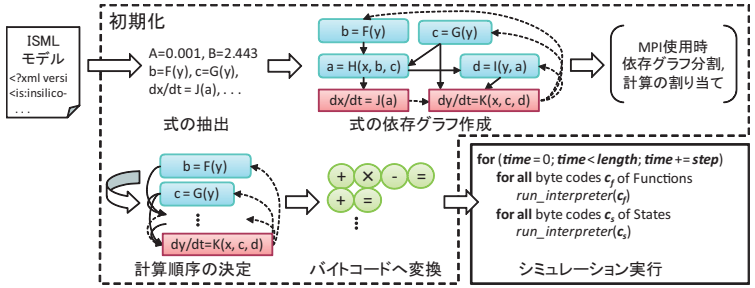


図 1 *insilico*Sim[3] の処理概要

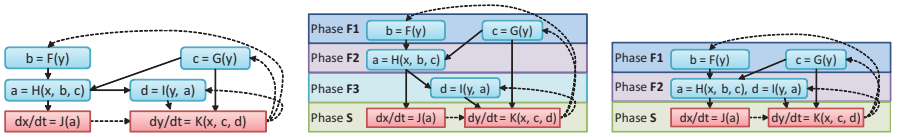
て高速化する．そのために，式の計算順をスケジューリングする方法をレベルスケジューリングに変更し，同一レベルにあるすべての式を GPU 上で並列計算する．以降，レベルスケジューリングにより分けられたレベルをフェーズと呼ぶ．同一フェーズに含まる式は並列計算可能であるが，それらは計算内容が相異なるものを含む．したがって，単純に CUDA スレッドへ処理を割り当てた場合，divergent branch が生じ GPU での演算効率が低下する．そこで，各フェーズのバイトコードを類似順に並び替え，divergent branch を削減する．また，ワープ内の全スレッドが同一構造の式を処理する場合，それらのバイトコードを単一化し，メモリ参照量を削減する．この並び替えおよび単一化は初期化時に行う．

4.1 レベルスケジューリングおよびフェーズの統合

互いに依存のない式を GPU 上で同時に計算するために，レベルスケジューリングにより式の計算順序を決める．**図 2** に例を示す．提案手法は数式および ODE の計算に異なるカーネルを用いており，すべての数式の計算後に ODE を計算するフェーズを設ける．

提案手法はフェーズごとに，GPU において CUDA カーネル[4] を 1 回起動する．モデルにより，フェーズ内の式が少ない場合があり，そのようなフェーズでは GPU 上での計算時間に比べカーネル起動にかかる時間が相対的に長い．このカーネル起動時間を削減するために，フェーズを統合しカーネルの起動回数を削減する．そのために，次の条件を満たす式を，より前のフェーズに移動する．

まず，式 f について，f が依存する式の集合を $D(f)$ および f を含むフェーズを $P(f)$ とする．f が依存するすべての式 $g \in D(f)$ のうち，最後に計算される式を e とする．$D(f) \cap P(e) = \{e\}$，すなわち，f が依存する e 以外の式が e を含むフェーズ $P(e)$ に無いとき，f を $P(e)$ に移動する．そして，e の後に f を計算する．以上の式の移動操作を，2



(a) 式の依存関係　(b) レベルスケジューリング　(c) 複数フェーズの統合

図 2 計算順序の決定

番目のフェーズ以降にスケジュールされたすべての式に施す．その後，フェーズ中のすべての式を移動できたフェーズを削除する．なお，式 f を式 e の後に移動した場合，e のバイトコードに f のバイトコードを連結する．これにより，e の後に f を計算する．

例えば，**図 2(b)** においてフェーズ F3 の $d = I(y, a)$ は，フェーズ F2 で値が決まる変数 a を参照している．F2 には a の他に $I(y, a)$ が参照する変数がないことから，d の計算を F2 に移し，a の後に続けて計算する．一方，F2 の $a = H(x, b, c)$ は F1 において並列計算される b および c に依存する．したがって，一貫性を保つために a は F1 へ移動できない．

4.2 バイトコードの並び替えおよび単一化

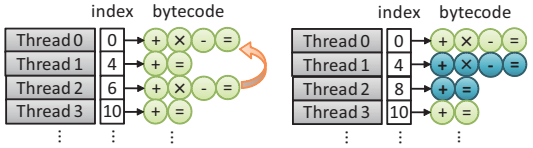
GPU 上では各 CUDA スレッドが 1 つのバイトコードを解釈実行し，フェーズ内の全バイトコードを並行計算する．ワープを構成する 32 個の CUDA スレッドが計算内容の異なるバイトコードを担当する場合，divergent branch が生じ GPU の性能を引き出せない．

そこで，**図 3** に示すように，各フェーズにおいて計算するバイトコードを並び替え，CUDA スレッド間の divergent branch を削減する．まず，フェーズ内のバイトコードを長い順にソートする．次に，それらを長さが同じバイトコードの集合 $B_1, B_2, \cdots B_n$ に分け，各 B_i 内のバイトコードを類似度に基づいて並び替える．

バイトコードの類似度は，バイトコード中のオペコード列をベクトルとみなしたときのベクトル間の距離で定める．本研究ではこの距離を単純な置換数，すなわち，2 つのオペコード列内の同じ位置に異なるオペコードがある数で定める．B_i から任意に 1 つのバイトコード $b_{i,1} \in B_i$ を取り出し，それを B_i において先頭に配置するバイトコードとする．次に，$b_{i,1}$ とその他すべてのバイトコード $b_{i,x} \in B_i - \{b_{i,1}\}$ の距離を調べ，距離が最も近いバイトコードを 2 番目に配置する．これを $b_{i,2}$ とし，以降 $b_{i,j}(j = 3, \cdots |B_i|)$ を順に決める．

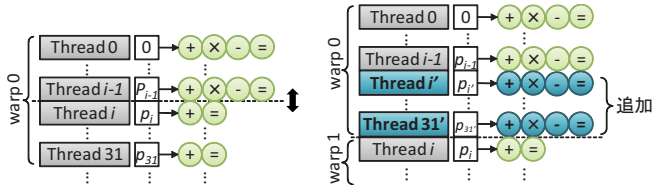
類似したバイトコードの数がワープサイズ[4] よりも小さい場合，並び替え後も divergent branch が発生する．そこで，**図 4** に示すように，冗長なスレッドを追加し，ワープ内の全スレッドに同じオペコード列を持つバイトコードを担当させる．

Vol.2011-HPC-130 No.9
2011/7/27



(a) 並び替え前　(b) 並び替え後

図 3　式の類似度に基づく並び替え



(a) divergent branch が生じる場合　(b) 冗長なスレッドの追加

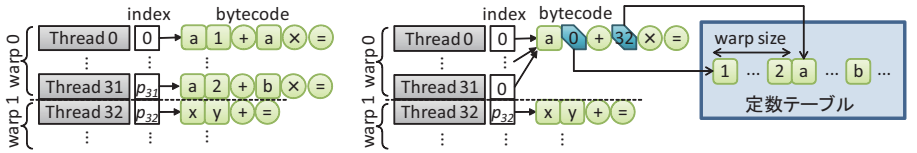
図 4　冗長なスレッドによる divergent branch の回避

加えて，ワープ内の CUDA スレッドが担当するバイトコードを単一化し，バイトコード参照によるメモリアクセスを削減する（**図 5**）．バイトコード間で異なるオペランドが存在する場合，そのオペランドをバイトコード外の定数テーブルに格納し，その定数テーブルを間接参照するようにバイトコードを修正する．

4.3　バイトコードの詳細

*insilico*Sim のバイトコードは，スタックマシンでの実行を想定しており，式を解析した構文木を後順序でバイト列に変換する．既存の CPU 版[3] では約 40 種のオペコードを持つ．例えば，スタックおよびメモリへデータを入出力する命令として，定数および変数値をスタックへプッシュする `CONSTANT` および `VARIABLE` や，スタック上にある計算結果を変数へ代入する `ASSIGN` がある．また，スタック上の値に対する算術演算，三角関数，比較，論理演算命令および条件付き代入向けの条件分岐命令などがある．特徴として，加算および乗算命令は 2 つ以上のオペランドをとることができ，和や積を効率よく扱える．このバイトコードは式の計算に特化しており，現在扱えるデータ型は倍精度浮動小数点型のみである．

GPU 版では，バイトコードを単一化するために，スタックへのプッシュおよび計算結果の出力に使われる代入命令を拡張する．その他の命令は CPU 版と同様である．前述の通



(a) 単一化前　(b) 単一化後

図 5　バイトコードの単一化

り，バイトコードを単一化するとき，スレッド間で異なる定数および変数オペランドは定数テーブルへワープごとに連続させて書き出す．そして，定数は定数テーブルから，変数は定数テーブルに記録されたアドレスを基に実際の値を間接参照する．これらの動作をするために，定数および変数用に `CONSTANT_IND` および `VARIABLE_IND` 命令を追加する．バイトコードを単一化するときに元の `CONSTANT` および `VARIABLE` をこれらの間接参照命令および定数テーブルの参照位置に置き換える．同様に，結果の出力先を定数テーブルから取得する代入命令 `ASSIGN_IND` を追加する．これらは GPU 向けのバイトコードの単一化処理および GPU 上で動作するインタプリタを実装したカーネル内部でのみ使用され，シミュレータのその他の部分およびモデルへの変更を生じさせない．

4.4　インタプリタの実装

図 6 にインタプリタ部を実装したカーネルの擬似コードを示す．`update_functions()` カーネルが数式を計算し，`update_odes_euler()` カーネルが Euler 法に基づいて ODEs を計算する．これらのカーネルをフェーズごとに GPU で起動する．なお，数式および ODEs は別々のフェーズで計算することから，両カーネルを同時に起動することはない．両カーネルの引数 `index` は各 CUDA スレッドが担当するバイトコードの先頭アドレスを格納した配列であり，引数 `codes` はバイトコードを格納した領域である．`consts` は定数テーブル，`input` および `output` はそれぞれ各フェーズにおける変数値の入力および出力用配列である．

`update_functions()` において数式を計算する時，計算結果の代入命令 `ASSIGN` は単に出力先配列へ値を書き出す．一方，`update_odes_euler()` において ODEs を計算するとき，`ASSIGN` は Euler 法に基づいて差分を加算する．両カーネルにおいて `ASSIGN` を除く処理は共通である．そこで，テンプレートを用いて共通部分を担当する `update()` 関数を記述し，重複した処理の記述を避ける．Runge-Kutta 法においては，ODEs の計算に 3 種類のカーネルを用いるが，同様にテンプレートにより重複コードを少なくできる．

`update()` 関数において，計算の途中結果を積むスタックにはローカルメモリ[4] を用いる．大多数の CUDA スレッドが担当する式において，オペランドは比較的少ない．ゆえに，必要なスタック長は短くてよいと考えられるからである．

オペランド中の定数および変数が異なるバイトコードを単一化した場合，それらを間接参照する．インタプリタはバイトコードから定数テーブル内の参照位置を取り出し，ワープ内でのスレッド番号を加算してスレッドごとに異なる定数および変数のアドレスを定数テーブル `consts` から取り出す．

CPU は 1 ステップの計算における各フェーズについて，`update_functions()` または `update_odes_euler()` カーネルを起動する．全フェーズの計算後，必要に応じて GPU での計算結果をメインメモリに転送する．そして，シミュレーションのステップを 1 つ進めて各フェーズの計算を繰り返す．

5. 評価実験

提案手法による速度向上を確認するために，5 つの生体モデルについてシミュレーションの実行時間を測定した．提案手法（GPU 版）の実験環境は，CPU が Intel Xeon X5472 3.0 GHz，メモリは 8 GB，GPU は NVIDIA GeForce GTX 480 である．OS は Windows 7 x64 SP1，開発環境は Visual Studio 2010 および CUDA 4.0 である．ビデオドライバはバージョン 270.81 を用いた．既存の CPU 版[3] の実行時間は，CPU に Intel Core i7 930 2.8 GHz を搭載し，12 GB のメモリを持つ計算機上で測定した．OS および開発環境は GPU 版と同様である．なお，CPU および GPU 上でのシミュレーションにおいて，モデル中の変数は全て倍精度浮動小数点数を用いている．

表 1 に実験に用いた生体モデルを示す．Luo-Rudy は文献 8) の心筋細胞モデルを ISML で記述したものである．LR-Ring および LR-Line はそれぞれリング状に 80 個および直線状に 100 個の Luo-Rudy を接続している．また，Wang および Rybak はそれぞれ文献 9) および文献 10) の神経細胞モデルに基づいた ISML モデルである．表 1 に示すように，提案手法は GPU 上で数千の式を含む大規模なモデルのシミュレーションを実行できる．

5.1 シミュレーションの実行時間

図 7 に，表 1 のモデルを Euler 法により 1000,000 ステップ（シミュレーション時間 10 秒，刻み幅 0.01 ミリ秒）計算する実行時間を示す．GPU 版はバイトコードの並び替え（Reorder），冗長スレッドの追加（Redundant），バイトコードの単一化（Unify）およびフェーズの統合（Merge）の 4 手法すべてを適用し，10,000 ステップごとに計算結果をメインメモリへ転送する．GPU 版のスタックサイズは 32 とした．計算にかかる時間のみを比較するために，CPU 版および GPU 版ともに計算結果のファイル出力は行っていない．

```
template<class AssignT>
__device__ void update(index, codes, consts, input, output) {
  stack[STACK_SIZE];
  *sp = stack - 1;
  id = blockIdx.x * blockDim.x + threadIdx.x;
  id_in_warp = threadIdx.x % WARP_SIZE;
  pc = index[id];
  while((inst = read_int(pc, codes)) != END) {
    switch(get_opcode(inst)) {
    case CONSTANT: /* push a constant value */
      push(sp, read_double(pc, codes));
    case CONSTANT_IND: /* push a constant value */
      adr = read_int(pc, codes) + id_in_warp * sizeof(double);
      push(sp, consts[adr]);
    case VARIABLE: /* push a variable value */
      push(sp, input[read_int(pc, codes)]);
    ...
    case PLUS_NARY: /* add or sum */
      n = get_num_ops(inst, pc, codes);
      sp -= n-1;  v = 0;
      for(i=0; i<n; i++) v += sp[i];
      push(sp, v);
    ...
    case ASSIGN: /* assign a value to a variable */
      AssignT::assign(read_int(pc, codes), pop(sp), input, output);
    ...
} } }
__constant__ double *delta; /* an array of step sizes */
struct AssignFunction {
  __device__ static void assign(pos, value, in, out) { out[pos] = value; }
}
struct AssignEuler {
  __device__ static void assign(pos, value, in, out) { out[pos] = in[pos] + value * delta[pos]; }
}
__global__ void update_functions(index, codes, consts, input, output) {
  update<AssignFunction>(index, codes, consts, input, output); }
__global__ void update_odes_euler(index, codes, consts, input, output) {
  update<AssignEuler>(index, codes, consts, input, output); }
```

図 6 インタプリタを実装した CUDA カーネルの擬似コード

Luo-Rudy は小規模なモデルであり，GPU を用いる処理のオーバヘッドが相対的に大きく，CPU でのシミュレーションよりも実行時間が 2.7 倍長くなっている．一方，1000 以上の式を含むその他 4 つのモデルでは，4.06〜13.6 倍の高速化を達成する．

表 2 にデータ出力を含む全体の実行時間を示す．シミュレーション時間を 1 秒および刻み幅を 0.01 ミリ秒とし，100,000 ステップのシミュレーションを実行した．全ステップにおけ

表 1 性能評価に用いるモデル

モデル	Luo-Rudy[8]	Wang[9]	Rybak[10]	LR-Ring	LR-Line
数式の数	31	1200	1362	2647	3298
ODE の数	8	400	480	640	800

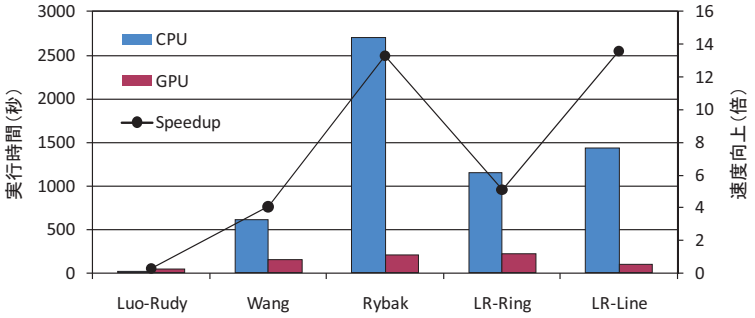


図 7 Euler 法による 1000,000 ステップのシミュレーション実行時間（単位：秒）

表 2 データ出力を含む 100,000 ステップの実行時間（単位：秒）. () 内はインタプリタの実行時間が占める割合

モデル	Euler 法		Runge-Kutta 法	
	CPU	GPU	CPU	GPU
Luo-Rudy	1.66 (96.8%)	5.68 (96.7%)	6.17 (99.2%)	97.0 (99.9%)
Wang	69.3 (94.5%)	25.4 (63.0%)	262 (98.6%)	142 (96.5%)
Rybak	295 (99.1%)	26.0 (83.9%)	1060 (99.8%)	137 (97.3%)
LR-Ring	125 (99.1%)	63.2 (38.9%)	492 (99.8%)	192 (99.0%)
LR-Line	159 (97.4%)	84.7 (13.3%)	621 (99.3%)	116 (95.0%)

表 3 Euler 法使用時のビデオメモリ使用量

モデル	Luo-Rudy[8]	Wang[9]	Rybak[10]	LR-Ring	LR-Line
バイトコード	3.05 KB	8.09 KB	13.7 KB	13.2 KB	14.6 KB
定数テーブル	2.38 KB	83.5 KB	91.3 KB	89.8 KB	101 KB
変数など	3.64 KB	77.1 KB	77.9 KB	139 KB	169 KB
計算結果	4.43 MB	229 MB	229 MB	410 MB	496 MB
(1 ステップ分)	0.453 KB	23.5 KB	23.5 KB	42.0 KB	50.8 KB
合計	4.43 MB	230 MB	230 MB	410 MB	496 MB

る全変数の値を出力しており，ファイルアクセスが性能ボトルネックとなる．そこで，データ出力用の CPU スレッドを作成しデータ出力およびインタプリタの実行を異なる CPU スレッドで実行した．出力用 CPU スレッドには 10,000 ステップごとに計算結果を渡す．データ出力用 CPU スレッドは，受け取った計算結果をそのままバイナリファイルとして書き出す．出力を渡す間隔は実験的に決定した．

CPU 版では，Euler 法および Runge-Kutta 法ともに，データ出力時間がほぼ隠蔽され，実行時間の 94%以上がインタプリタの実行時間である．一方，GPU 版では Euler 法を用いるとき，インタプリタの実行よりも出力にかかる時間が長い．特に，LR-Line ではインタプリタの実行時間は実行時間全体の 13%である．ただし，実用上はユーザが興味のある変数のみを出力すればよく，また複数ステップに 1 度の出力で十分な場合もある．このような場合にはファイル出力は性能ボトルネックとならないことが多い．

なお，10,000 ステップの計算結果を格納するために，LR-Line では約 500 メガバイトのビデオメモリを使用する．**表 3** にビデオメモリ使用量の内訳を示す．計算結果用の領域を除く使用メモリ量は，今回用いたモデルにおいて数百キロバイトである．したがって，出力する変数の削減などにより，計算結果用の領域を縮小するとより式の多いモデルを扱える．

5.2 提案手法の効果

図 8 に GPU 版に適用した 4 手法の効果を示す．GPU からメインメモリへの計算結果の転送を除き，GPU を用いた計算時間のみを計測した．図左端の Base は 4 手法をいずれも適用しない場合であり，他はそれぞれ左隣から適用手法を 1 つずつ加えている．

バイトコードの並び替え処理（Reorder）は，Base に比べ計算時間を 7〜56%増加させる．Reorder は最長のバイトコードを先頭の CUDA スレッドに割り当て，類似したバイトコードを以降の CUDA スレッドに割り当てる．したがって，スレッド ID が小さな CUDA スレッドに長いバイトコードを割り当てる傾向がある．これにより，ワープ間で負荷の偏りが生じ，計算時間が増加すると考えられる．

Reorder に加え，冗長なスレッドの追加（Redundant）により，Luo-Rudy において計算時間が 48%短縮される．Luo-Rudy は類似した式が少なく，異なるバイトコードが同一ワープ内で実行されており，分岐が divergent branch となる割合が多い．実際に，CUDA Visual Profiler[4] を用いてシミュレーションの最初の 100 ステップについて，数式計算カーネルにおける divergent branch の割合を測定した．divergent branch の割合は Reorder において約 8%であり，Redundant により約 0.2%に減少する．一方，他のモデルには類似し

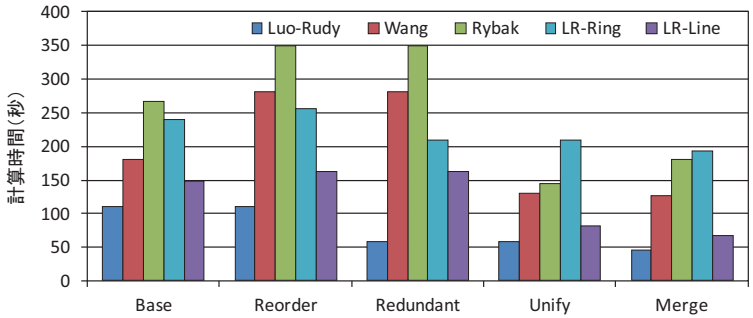


図 8 各手法の効果（Euler 法，100,000 ステップ）

表 4 バイトコードおよび定数テーブルの使用メモリ量（単位：キロバイト）

モデル	Base	Unify
Luo-Rudy	4.0	5.4
Wang	210.9	91.6
Rybak	291.3	100.6
LR-Ring	313.5	104.1
LR-Line	387.1	117.2

た式が多数あり，Redundant の効果が顕著には表れない．

図 8 において，バイトコードの単一化（Unify）による性能向上への寄与が最も高く，Base に比べ計算時間を 12〜48%短縮する．**表 4** にバイトコードおよび定数テーブルの使用メモリ量を示す．定数テーブルを追加したために，Luo-Rudy は Base よりも Unify のメモリ使用量が多い．一方，その他のモデルにおいては 1/2〜1/3 に削減される．特に，同じ細胞を多数接続するモデルほど削減率が高い．Redundant の場合と同様にプロファイラを用いてビデオメモリ参照命令数および実際にビデオメモリ参照リクエストが発行された数を調べた．Redundant により CUDA スレッドが増加することから，参照命令数は Base に比べ Luo-Rudy では 751%，その他のモデルにおいては 26〜41%増加する．Luo-Rudy での命令数増加率が高い原因は，Redundant により追加される冗長スレッド数が相対的に多く，スレッド数が Base の 10 倍以上に増加するからである．一方，参照リクエスト数は Base に比べ 42〜76%削減されており，バイトコードの単一化によりメモリ参照量を削減できている．

フェーズの統合（Merge）は Rybak 以外のモデルにおいては Unify までを適用したより

表 5 フェーズ中のバイトコード数（上段）および種類数（下段）

モデル	Base	Merge
Luo-Rudy	22 → 6 → 2 → 1 → 8 14 → 6 → 2 → 1 → 3	22 → 2 → 8 16 → 2 → 3
Wang	1000 → 100 → 100 → 400 7 → 1 → 1 → 3	1000 → 100 → 400 7 → 1 → 3
Rybak	842 → 280 → 40 → 480 → 200 8 → 5 → 1 → 5 → 4	842 → 40 → 480 → 200 11 → 1 → 5 → 4
LR-Ring	1924 → 562 → 81 → 80 → 640 16 → 9 → 2 → 1 → 4	1924 → 322 → 1 → 640 18 → 6 → 1 → 4
LR-Line	2399 → 699 → 100 → 100 → 800 15 → 7 → 1 → 1 → 3	2399 → 399 → 80 0 17 → 4 → 3

も計算時間を 2〜19%短縮する．一方，Rybak では計算時間が 24%長くなる．Rybak では数式を計算する最初のフェーズにおいて，Merge 前はバイトコード中の平均オペコード数が 9.3，最大 16 であるのに対し，統合後は平均 13.6，最大 81 であった．つまり，統合により負荷が突出して高いバイトコードが生じる．1 つの式に依存する複数の式の計算が，1 つのバイトコードに連結されたために生じると考えられる．この負荷の不均衡により，Rybak の計算時間が増大したと考えられる．

表 5 にレベルスケジューリング後に，各フェーズが含むバイトコードの数および種類数を示す．Base の場合，バイトコード数は各フェーズで計算する式の数と同じである．種類数は，バイトコードのオペコード列が一致する場合に同種であるとみなして計数した．Merge により，Luo-Rudy および LR-Line では 2，その他のモデルでは 1 フェーズ削減されており，実モデルにおいてフェーズ統合できることが分かる．一方，Merge により，各フェーズにおけるバイトコードの種類は増加しうる．

提案手法は，GPU でのインタプリタが使用するスタックをローカルメモリに配置している．**図 9** にスタックを共有メモリ[4] に配置した場合の性能を示す．*insilico*Sim のバイトコードには，和や積など任意個数のオペランドをとる命令がある．最大スタック長以下で計算できるよう，これらの命令においてオペランド数が多い場合は 2 回以上の演算に適宜分割している．今回用いたモデルが必要とするスタック長は，分割しない場合最大 100 程度であり，最も分割した場合は 7 である．図 9 より，LR-Line を除き，ローカルメモリおよび共有メモリのいずれでも性能に大きな差はない．これは，Fermi 世代の GPU におけるローカルメモリに対するキャッシュの効果であると考えられる．

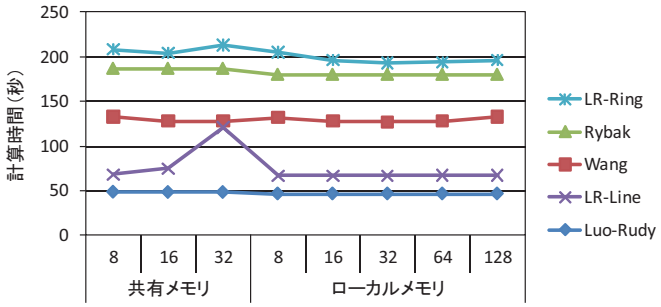


図 9 スタックの配置先による性能変動（Euler 法，100,000 ステップ）

表 6 CPU 版に対する GPU 版の誤差の数および計算結果全体に占める割合

モデル	Euler 法		Runge-Kutta 法	
Luo-Rudy	0	(0%)	0	(0%)
Wang	1	(6×10^{-7}%)	0	(0%)
Rybak	6	(4×10^{-6}%)	9	(5×10^{-6}%)
LR-Ring	200125	(0.06%)	171614	(0.05%)
LR-Line	80858	(0.02%)	206	(5×10^{-5}%)

5.3 GPU を用いた計算による誤差

最後に，GPU での計算誤差について述べる．**表 6** に，CPU および GPU 版の計算結果のうち，異なる数値の数およびそれらが計算結果全体に占める割合を示す．ステップ数は 100,000 ステップである．LR-Ring を Euler 法により計算した場合，最も誤差のある数値が多く，結果全体の 0.06%において CPU および GPU 版に差がみられた．その多くは，絶対値が 10^{-14} 以下の数である．この時の最大の絶対誤差は 0.1 であり，CPU および GPU での計算結果がそれぞれ 13433.9 および 13433.8 であった．最大の相対誤差は 2.04×10^{9} であり，CPU および GPU での値は 2.06×10^{-24} および -4.2×10^{-15} であった．

6. ま と め

本研究では，汎用生体シミュレータである *insilico*Sim を高速化するために，多数の ODEs を並列計算するインタプリタを GPU 上に実装した．GPU 上で並列処理するために，レベルスケジューリングを用いて数式および ODEs の計算順を決定する．そして，同じレベルのすべての式を 1 回のカーネル実行で計算する．多数の細胞を相互接続したモデルにおいては，類似した式が多数存在することに着目し，類似した式のバイトコードをワープごとに単一化する．この単一化により，SIMD 型の計算機である GPU における演算効率の向上およびメモリ参照量の削減を図った．結果，CPU の単一コアを用いる場合に比べ，最大 13.6 倍高速であった．今後の課題は，バイトコードのチューニングおよび GPU クラスタ環境を用いたより大規模なモデルに対するシミュレーションを可能にすることである．

謝辞 本研究の一部は特別研究員奨励費（23・5860），科学研究費補助金（基盤研究（B）2330007），科学研究費補助金（若手（B）23700036）および大阪大学グローバル COE プログラム *in silico* medicine の補助による．

参 考 文 献

1) Asai, Y., Suzuki, Y., Kido, Y., Oka, H., Heien, E., Nakanishi, M., Urai, T., Hagihara, K., Kurachi, Y. and Nomura, T.: Specifications of insilicoML 1.0: A Multilevel Biophysical Model Description Language, *J. Physiological Sciences*, Vol.58, No.7, pp.447–458 (2008).
2) Physiome.jp, `http://www.physiome.jp/` (2011).
3) Heien, E.M., Okita, M., Asai, Y., Nomura, T. and Hagihara, K.: insilicoSim: an Extendable Engine for Parallel Heterogeneous Biophysical Simulations, *Proc. 3rd Int'l Conf. Simulation Tools and Techniques (SIMUTools '10)*, pp.78:1–78:10 (2010).
4) NVIDIA Corporation: CUDA Programming Guide Version 4.0 (2011).
5) Ackermann, J., Baecher, P., Franzel, T., Goesele, M. and Hamacher, K.: Massively-Parallel Simulation of Biochemical Systems (2009).
6) Herrmann, F., Silberholz, J., Bellone, M., Guerberoff, G. and Tiglio, M.: Integrating post-Newtonian equations on graphics processing units, *Classical and Quantum Gravity*, Vol.27, No.3 (2010). 12 pages.
7) Zhang, E.Z., Jiang, Y., Guo, Z., Tian, K. and Shen, X.: On-the-Fly Elimination of Dynamic Irregularities for GPU Computing, *SIGPLAN Not.*, Vol.46, No.3, pp. 369–380 (2011).
8) Luo, C.-H. and Rudy, Y.: A Model of the Ventricular Cardiac Action Potential. Depolarization, Repolarization, and Their Interaction, *Circulation Research*, Vol.68, No.6, pp.1501–1526 (1991).
9) Wang, X.-J. and Buzsáki, G.: Gamma Oscillation by Synaptic Inhibition in a Hippocampal Interneuronal Network Model, *J. Neuroscience*, Vol.16, No.20, pp. 6402–6413 (1996).
10) Rybak, I.A., Shevtsova, N.A., Lafreniere-Roula, M. and McCrea, D.A.: Modelling spinal circuitry involved in locomotor pattern generation: insights from deletions during fictive locomotion, *J. Physiology*, Vol.577, No.2, pp.617–639 (2006).