RC-001

カスタム計算機におけるメモリレイアウト最適化のための チューニングツールの開発

Development of a tuning tool of memory layout on custom computers

新垣 誠 † 喜屋武 克樹 † 平井 裕介 ‡ 仲宗根 宏貴 ‡ 長名 保範 †
Makoto Arakaki Katsuki Kyan Yusuke Hirai Hiroki Nakasone Yasunori Osana

1 はじめに

科学技術計算や、Web 上の大量のデータの処理などの大規模計算をより効率的に行うために、FPGA をはじめとする再構成型デバイスを用いてカスタム計算機を作る手法が注目されている。マイクロプロセッサに比べて低消費電力・高性能なシステムをコンパクトに実現できる可能性がカスタム計算機の魅力である。FPGA 上の回路設計や、FPGA と外部のインタフェイスの設計など、依然としてさまざまな困難も多い。

専用の演算ハードウェアを構築する際のひとつの大きな問題として、メモリアクセスが挙げられる。FPGA ベースの計算機でも、データを保持するためのオフチップメモリとして用いられるのは SDRAM であり、実際の設計においては SDRAM へのアクセスがボトルネックとなって性能が制限されてしまうこともありうる。マイクロプロセッサを用いたシステムでは古くからキャッシュがメモリアクセスの性能改善に用いられてきたが、カスタム計算機ではマイクロプロセッサのようなキャッシュ機構の実装は一般に困難である。

そこで本研究では、FPGA 上に特定の計算のための専用演算パイプラインを実装することを想定して、その際のメモリアクセスパターンの分析とチューニングを行い、SDRAM のバンド幅を有効に利用することを可能にするツールの実装とその初期評価について述べる。

2 ハードウェアモデルと本研究の目的

2.1 ハードウェアモデル

本研究で想定するハードウェアの構成を図 1 に示す。FPGA には 1 チャネルまたは複数の SDRAM あるいは SDRAM モジュールが接続されており、FPGA 上にはそれぞれの SDRAM インタフェイスを駆動するためのコントローラが実装されている。コントローラは Xilinx などの FPGA ベンダのツールで提供される FIFO ベースのものを想定しており、FPGA 上の計算エンジンは、このコントローラを経由してメモリアクセスを行う、ごく一般的な構成である。なお、各チャネルのメモリのデータ幅やアドレス幅は異なっていてもよいものとする。

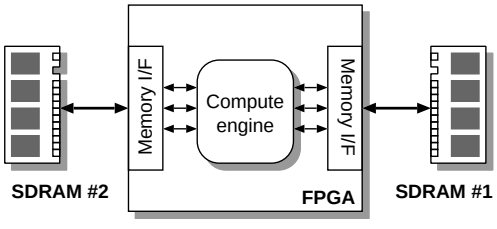

図 1 想定するハードウェアの構成

2.2 SDRAM とカスタム計算機

本稿で述べる結果は DDR2 SDRAM を用いた場合のものであるが、本研究での実装は特に SDRAM の種類を限定するものではない。SDRAM はいずれも、

- メモリセルを複数のバンクに分割して個別に動作させる
- 一度のコマンドで連続する列 (column) アドレスのバースト転送を行う

ことによって高速化を行っているため、同一のバンク、あるいは同一の行 (row) アドレスへのアクセスが連続した場合には比較的短いレイテンシでアクセスが可能だが、行アドレスが切り替わる場合には大きなレイテンシが必要になる。また、一度のアクセスで一定のワード数がバースト転送されるため、これをなるべく全部使うようにしないと、無駄な転送サイクルが発生することになり、実効バンド幅が低下する。このような問題は、マイクロプロセッサにおいてはキャッシュによって緩和されるが、回路規模の大きくなりがちなキャッシュを FPGA ベースのカスタム計算機に実装することは難しい。

2.3 本研究の目的

したがって、計算エンジンの回路がどのような順序でメモリアクセスを行うかによって SDRAM の実効バンド幅が大きく変化するため、これを最適化することは非常に重要である [1]。メモリアクセスのパターンはメモリ上のデータ配置と、その参照順によって決定づけられるが、いったん HDL によるハードウェアの設計ができあがってしまうと変更することは容易ではない上に、大容量の SDRAM の HDL モデルを含む RTL シミュレーションは所要時間や必要メモリサイズなどさまざまな困難を伴う。

そこで、本研究では、アルゴリズムをハードウェア化する前の、ソフトウェアによるプロトタイプの時点で、プログラム中にメモリモデルを埋め込み、メモリアクセスに所要するクロックサイクル数を推定する手法を提案・実装した。ソフトウェアであればデータへのアクセス順や一旦読み出したデータの再利用は容易に記述と変更が可能であり、オリジナルのアルゴリズムのソフトウェア実装をそのまま用いることができるので、バグの混入も防ぐことができる。また、RTL のシミュレーションと異なり、より高い抽象度のモデルを用いる

† 琉球大学工学部
‡ 琉球大学大学院理工学研究科

表 1　DDR SDRAM のタイミングパラメータ

名称	値		名称	値	
	(ns)	(clk)		(ns)	(clk)
CL		4	tRP	15.0	2
AL		0	tWTR	7.5	1
RL		4	tWR	15.0	2
WL		3	tWPST	4.8	1
tCCD	15.0	2	tRTP	7.5	1
tRCD	15.0	2			

ことで、短い時間で結果を得ることが可能である。

これらの仕組みは C++ のテンプレートクラスとして用意されており、さまざまなデータ型やメモリレイアウト、あるいはメモリ構成に柔軟に対応することができる。

3　基本メモリモデルの構築

本研究で実装したメモリモデルは 2 階層にわかれており、メモリのタイミングモデルだけを提供する基本メモリモデル、データの保持までを行う配列モデルから構成される。本節ではまず基本メモリモデルについて述べる。

3.1　予備評価

メモリの挙動をシミュレートするための C++ クラスを実装するにあたり、

- FPGA: Xilinx Virtex-5 XC5VLX50T-1[2]
- SDRAM: Micron MT47H32M16BT-37E[3] (125MHz)

を前提とする RTL シミュレーションを実施した。16bit・32M word の SDRAM で、アドレス構成は Bank 2bit, Row 13bit, Column 10bit である。メモリコントローラは Xilinx の Memory Interface Generator (MIG) で生成し、Xilinx ISE 14.7 で論理合成・配置配線を行った。メモリモデルは Micron から提供される DDR2 SDRAM の Verilog モデルを用い、タイミングパラメータは表 1 のように設定した。

SDRAM のリード・ライトアクセスの最後に CAS コマンドで列アドレスを指定することで行われるが、CAS 発行前に行アドレスの指定やプリチャージなどのコマンドが必要になる場合もある。これは具体的には、(1) 対象のバンクが以前のアクセスでオープンされているか、(2) バンクがオープンならば、そのバンクで直前にアクセスされたのが対象の行アドレスと同じか、(3) 前のアクセスの read/write と今回のアクセスの read/write の組み合わせ、の 3 つの要素による。これに必要なクロックサイクル数は、基本的に表 1 に示されるタイミングパラメータで決定されるものの、メモリコントローラの実装によってはそれよりも長いサイクル数が必要な場合もありうる。

そこで、DDR2 SDRAM の Verilog モデルと、メモリコントローラの RTL 記述を用いたシミュレーションで、前のアクセスでの CAS の発行から次のアクセスの CAS の発行までに必要な最小の間隔を調査した結果が表 2 である。DDR2-SDRAM では標準が 4 ワードバーストであり、これには 2 クロックサイクルが必要なため、同一バンク・同一行アドレスでの Read から Read と Write から Write が最短の 2 クロックとなっており、そのほかのアクセスの場合にはそれぞれ所定のオーバーヘッドが必要であることが確認できた。また、これらは、いずれもタイミングパラメータから算出される最短の時間であった。

表 2　CAS コマンドの最小発行間隔

コマンド		バンクの	行	CAS
ひとつ前	現在	状態	アドレス	Interval
Read	Read	クローズ	Any	5
Read	Read	アクティブ	同一	2
Read	Read	アクティブ	異なる	10
Write	Read	クローズ	Any	8
Write	Read	アクティブ	同一	8
Write	Read	アクティブ	異なる	14
Read	Write	クローズ	Any	5
Read	Write	アクティブ	同一	5
Read	Write	アクティブ	異なる	8
Write	Write	クローズ	Any	5
Write	Write	アクティブ	同一	2
Write	Write	アクティブ	異なる	14

また、Xilinx のメモリコントローラでは FIFO 経由でコマンドやデータの書き込み・読み出しを行うが、リセット後の最初のコマンドが Read ならばそこから 15 クロック後にデータが Read Data FIFO から出力され、Write ならば 14 クロック後に CAS がアサートされることが確認された。

次節では、以上のデータに基づいて構築した基本メモリモデルについて述べる。

3.2　実装

図 1 に示したように、複数のメモリチャネルをもつシステムをシミュレートできるようにするために、メモリの各種パラメータや状態は C++ クラスとしてカプセル化しておき、プログラム内に複数のインスタンスを持つことができるように実装した。このクラスを `sdram_model` と呼ぶ。このクラスでは SDRAM のタイミングモデルだけを提供し、データを保持することはしない。データは別クラスで管理することで、より柔軟な使い方ができるようになっている。

`sdram_model` クラスには表 2 に示すパラメータのほかに、Bank, Row, Column の各アドレスの幅を指定することができる。デフォルトでは Micron MT47H32M16BT のアドレスの構成と、表 2 のパラメータが設定されている。

このクラスの主なメンバ関数は以下の 4 つで、指定した Read および Write 動作が行われる時刻 (システム初期化時からのクロックサイクル数) を求めることができる。

- `void setup_address (int` *bank*`, int` *row*`, int` *col*`);`
  bank, row, col で指定したアドレス構成で `sdram_model`

を初期化する。

- `int write (int` *address*`);`
  直前までの動作に続いて *address* への書き込みを行い、CAS が発行される時間を返す。
- `int read (int` *address*`);`
  直前までの動作に続いて *address* からの読み出しを行い、CAS が発行される時間を返す。
- `int read (int` *address*`, int` *time*`);`
  time で指定した時刻に新たに *address* からの読み出しを行い、CAS が発行される時間を返す。

Read では時刻を指定してコマンドを発行できるようになっているが、これはメモリコントローラへのコマンド入力は必ずしも連続して行われるわけではなく、アイドルになる時間もある、ということを考慮して設定されているものである。Read の場合には、たとえばポインタをメモリから読み出す場合のように、コマンドの完了時刻が次のコマンドの入力の時刻に影響を与えることがあるため、時刻を指定するオプションが用意されている。Write の場合にはこのようなことを考慮しなくてよいため、時刻指定のオプションは設けていない。

`read()` や `write()` の関数群は、内部的には `req_clk()` という所要クロック算出のための関数で表 2 のようなデータを参照して所要時間を算出し、加えてバンクや列アドレスの情報を保存している。

3.3 検証

ランダムに生成したアドレスへの、ランダムな read/write の列を生成し、これにかかる時間を Cadence NC-Verilog を用いた RTL シミュレーションと、`sdram_model` によるソフトウェアでの計算でそれぞれ計算した。1000 回分のアクセスを入力列として与え、RTL シミュレーションでは 11,530 クロック、`sdram_model` では 11,560 クロックという結果を得た。30 クロックの違いは、`sdram_model` ではメモリコントローラ内の FIFO の詳細な動作を考慮しないためと考えられるが、差は 0.26% であり、`sdram_model` は実用上充分な精度をもつといえる。

また、RTL でのシミュレーションには数十秒を要するものの、本プログラムでは通常の PC でも 1 秒以下で実行が可能であり、大幅な高速化が可能である。

4 配列モデル

メモリのタイミングモデルは前節で述べた `sdram_model` によってシミュレートされるが、実際に数値計算などを行うプログラムをこれのテストベンチとして用いるには、プログラム中の配列アクセスからメモリのアドレスを計算し、`sdram_model` に渡す必要がある。

このために用いるのが配列モデルのクラスで、現在は 1 次元と 2 次元の任意の型の配列を扱うことのできる `mem2d` クラスが実装されている。必要があれば 3 次元以上の配列に拡張するのも容易である。

`sdram_class` と `mem2d` がどのようにメモリチャネルと配列に対応するかを表したのが図 2 である。物理的なメモリ

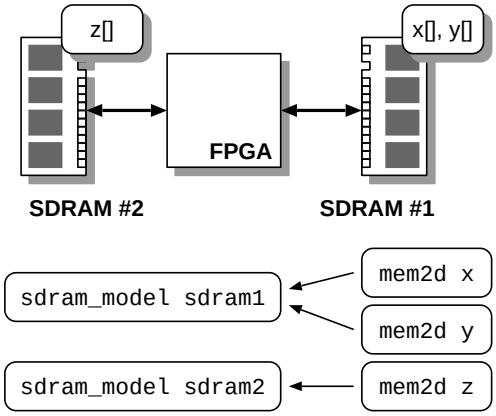

図 2　メモリ・配列と各クラスの対応

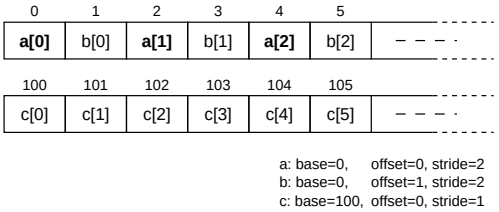

図 3　ベース・オフセット・ストライド

チャネルには 1 対 1 対応で `sdram_class` のインスタンスを置き、これが各チャネルのタイミングモデルを保持する。メモリに配置されるデータは、プログラム中では通常配列として表現されるが、これを `mem2d` クラスのインスタンスで置き換え、それぞれ対応する `sdram_class` と紐付けることにより、プログラム中でのデータへのアクセスをもとに `sdram_class` によるタイミング計算を行うことができる。

4.1 メモリへの配置モデル

`mem2d` では、配列のインデックスからメモリのアドレスを計算し、これを `sdram_address` に渡すことを行っている。図 2 で示したように、ひとつのメモリには複数の配列を配置できるので、それぞれの配列がそのメモリのどこにどのように配置するか、ということを指定できるようにしておく必要がある。

そこで、`mem2d` では Base, Offset, Stride、という 3 つの値を導入している。1 次元配列の場合には、

$$\text{Address} = \text{Base} + \text{Offset} + \text{Index} \times \text{Stride}$$

という式を用いて配列のインデックスからメモリのアドレスを求める。2 次元配列の場合には、C や C++ と同じように 1 次元に展開してからこの式を適用する。ここで用いられるアドレスはバイトアドレスではなく、メモリの物理的なアドレス (Bank, Row, Column をビット連結したもの) であるので、配列を連続したメモリ空間に配置する場合の Stride は必ずしも 1 ではなく、たとえば 16bit 幅のメモリに 32bit の整数を連続して配置するならば Stride は 2 である。

Base, Offset, Stride を適切に用いると、図 3 のように、配列を交互にインターリーブして配置したり、バンクごとに別々

の配列を配置することが可能である。

4.2 データの保持

sdram_model はデータを保持していないので、mem2d は内部に一次元配列を持っており、mem2d のインスタンス生成時にこの型を指定することができる。型指定は C++ のテンプレートクラスの仕組みを用いているので、任意のデータ型を用いることができる。

4.3 実装

ここでは mem2d のインタフェイスを、配列の型やサイズと Base, Offset, Stride の設定を行うための仕組みと、データアクセスのための仕組みに分けて説明する。まず、設定を行うための仕組みとしては、主に以下のコンストラクタと関数を用いる。

- mem2d<*type*> *instance* (*size*);
- mem2d<*type*> *instance* (*size_x, size_y*);
  コンストラクタでは 4.2 で述べた mem2d の保持するデータの型と、1 次元配列あるいは 2 次元配列のサイズを与える。これによって必要なデータの配列のメモリ領域が確保される。
- setup_model (*sdram_model, base, offset, stride*);
  setup_model() によって mem2d インスタンスと sdram_model インスタンスの紐付けを行い、また Base, Offset, Stride の各パラメータを与える。

データアクセスの仕組みには大きく分けて 3 種類のアクセス方式が用意されている。

- 通常アクセス: sdram_model へのアクセスとデータの読み書きの双方を行う。Read と Write の両方がある。
  - *type* read (*x* [, *y*]);
  - void write (*x* [, *y*], *value*);
- ブロッキングアクセス: sdram_model へのアクセスを行い、次のアクセスが発行されるのはこのアクセスが完了した後になるように時刻の調整を行う。主にポインタ読みだしのために用いるため、Read のみ実装されている。
  - *type* read_b (*x* [, *y*]);
- サイレント転送: sdram_model へのアクセスを行わずデータの読み書きだけを行う。主に初期値の書き込みやデバッグ用のアクセスなどに使用する。
  - *type* read_silent (*x* [, *y*]);
  - void write_silent (*x* [, *y*], *value*);

なお、現在の実装では配列参照演算子 [] によるデータへのアクセスはできず、これらの関数群を使用する必要がある。

以上の関数群を使えば、実際に FPGA がアクセスするのと同じ順序で mem2d にアクセスすることで、一定の処理が完了するまでの間にメモリアクセスに所要するトータルの時間を求めることができる。

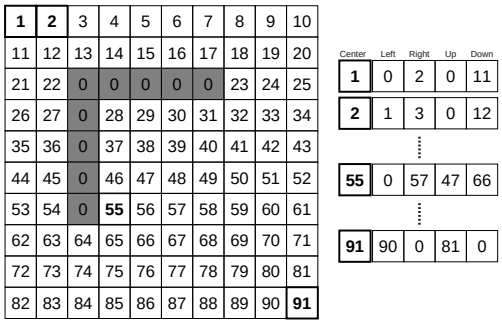

図 4　セル情報の取り扱い

5 評価

この節では、これまでで述べてきた仕組みを使って、C++ を使って記述された、2 次元の拡散方程式を解くプログラムをベースにメモリアクセスにかかる時間の推定と改善を行った結果を述べる。

5.1 プログラムの概要

このプログラムでは図 4 (a) に示すように、10×10 の正方格子で、左上に逆 L 字型の壁のある 2 次元空間において、拡散方程式の時間発展を 1 次の陽解法で計算する。各セルは 2 種類の物理量 (たとえば別々の物質の濃度) を保持しており、これがそれぞれ独立に拡散係数 k にしたがって拡散するものとする。

具体的には、ある時間刻みにおける着目セルの物理量 X_t と 4 近傍セルの物理量 X_l, X_r, X_u, X_d から、次の時間刻みにおける着目セルの物理量 X_{t+1} を、拡散係数 k を用いて

$$X_{t+1} = X_t + k(X_l + X_r + X_u + X_d - 4X_t) \quad (1)$$

として計算する。2 種類の物理量は独立に拡散するものとしているので、それぞれ個別に計算する。

5.2 データ構造

空間内の各セルには図 4 (a) のように通し番号を振るが、壁や周辺など境界になるところには番号を割り当てない。

この番号を用いて、図 4 (b) にあるようなセル間の隣接状態を表す配列を作成する。10×10 のセルの外縁部や壁に接している部分には 0 番を割り当てておき、この配列を参照して各セルの物理量にアクセスすれば、式 1 を順次解いて次の時間刻みにおける各セルの物理量を求めることができる。

プログラム内には、以下の 5 つの主要な配列がある。

- **N**: 図 4 (b) に対応する隣接情報の配列
- $\mathbf{X}_1, \mathbf{Y}_1, \mathbf{X}_2, \mathbf{Y}_2$: 図 4 (a) に対応する物理量の配列

物理量の配列が 4 セットあるのは、2 種類の物理量の拡散を解くため、ある時間刻みにおいてはたとえば $\mathbf{X}_1$ と $\mathbf{Y}_1$ が現在の物理量、$\mathbf{X}_2$ と $\mathbf{Y}_2$ が次の時間刻みでの物理量を持っており、この 2 セットの配列は時間刻みごとに入れ替わることになる。

物理量は倍精度浮動小数点型、隣接情報は整数型の配列として実装した。

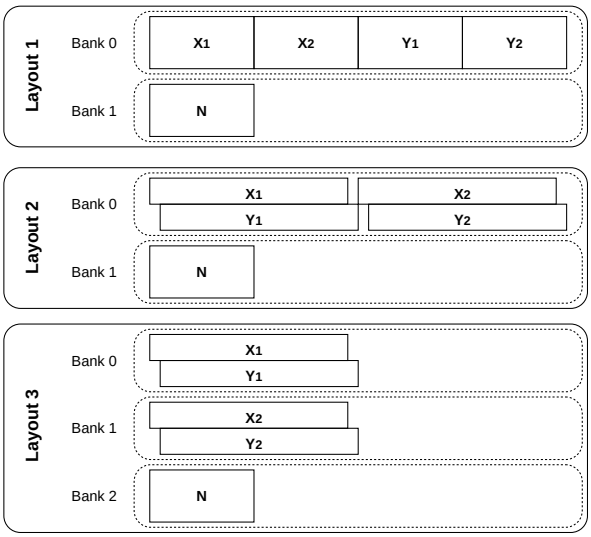

図 5　配列のメモリ配置

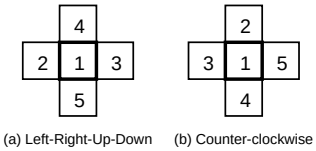

図 6　セルの参照順

5.3　性能評価

以上で述べたような拡散方程式を解くプログラムを C++ で記述し、$\mathbf{N}, \mathbf{X}_1, \mathbf{Y}_1, \mathbf{X}_2, \mathbf{Y}_2$ の各配列を mem2d を用いるように書き直して性能評価を行った。

メモリへのデータ配置は図 5 に示したように、

1. バンク 0 に $\mathbf{X}_1, \mathbf{Y}_1, \mathbf{X}_2, \mathbf{Y}_2$ をそれぞれまとめて配置、バンク 1 に $\mathbf{N}$ を配置
2. バンク 0 に $\mathbf{X}_1$ と $\mathbf{Y}_1$、$\mathbf{X}_2$ と $\mathbf{Y}_2$ をそれぞれインターリーブして配置、バンク 1 に $\mathbf{N}$ を配置
3. バンク 0 に $\mathbf{X}_1$ と $\mathbf{Y}_1$ をバンク 1 に $\mathbf{X}_2$ と $\mathbf{Y}_2$ をそれぞれインターリーブして配置、バンク 2 に $\mathbf{N}$ を配置

の 3 通りで評価した。インターリーブした場合はふたつの配列の同じセルが連続アクセスされるのでバーストサイクルが有効に利用できる可能性があり、さらに 2 つのセットを別々のバンクに配置することで Read と Write を行うバンクを分離できるので効率が上がることが期待できる。

また、周辺にセルを参照する順は図 6 のように、左右上下の順でアクセスするものと、反時計回りにアクセスするものの 2 通りで評価した。前者の場合は注目セルとその左右の 3 セルが隣接するため、バーストサイクルがより有効に利用できると考えられる。

また、ふたつの物理量についての計算も、X についてすべてのセルの計算をおこなってから Y についてすべての計算を行う場合と、X と Y の同じセルについて交互に計算を進める方法を評価した。これも、後者ではバーストサイクルの有効利用ができると予想された。

なお、この評価を行う際、メモリレイアウトの変更は mem2d のパラメータ (Base, Offset, Stride) を変更するだけでよいが、セルの参照順序と計算順序の変更にはソースコードの変更が必要である。

これらの組み合わせについて、計算終了までのメモリアクセスに要するクロックサイクル数をまとめたものが表 3 である。データを 3 つのバンクに分散し、セルの参照順の連続化や計算順のインターリーブを行うことで、一番遅いケースの約 6 倍の実効性能を得られると確認された。

5.4　考察

本手法では、ソフトウェアによる数値計算アルゴリズムの実装をテストベンチとして用い、配列へのアクセスからメモリのタイミングモデルを駆動して、ハードウェアによる実装で同様のメモリアクセスを行った場合にメモリアクセスに所要する時間を算出する方法を提案・実装した。このモデルを導入する際にはソフトウェアの実装には多少の手を加えなければならず、また、メモリアクセスの順序がハードウェアと同じになるように注意が必要であるが、RTL 記述による HDL でのシミュレーションから性能を推定するよりもはるかに簡単に行うことができ、メモリレイアウトの変更についてはパラメータの変更だけで済む点が大きな長所である。

実際には FPGA 内部の演算処理とメモリアクセスは平行して行われるので、演算処理の時間について考慮しないこの手法での精度には限界があるが、メモリアクセスのオーバーヘッドが性能上の懸念事項である場合には、充分有効な性能指標になりうると考えている。また、演算パイプラインの遅延が静的に決まる場合には、これを考慮した実装は可能であるが、現在のところこれを容易に行うための枠組みの実装は行っていない。

6　まとめ

従来、FPGA を用いた計算機でメモリ周りのチューニングを実施するためには、文献 [1] のように、システム設計の段階で綿密な計画をたててから実装を行う必要があった。しかし、これは複雑なアルゴリズムの場合には非常な困難を伴い、事前にさまざまなメモリレイアウトを試すということは現実的には困難であった。

本研究で開発されたフレームワークを用いることで、C++ で記述したアルゴリズムをそのままテストベンチとして用い、実行時のメモリアクセスに必要な時間を見積もることが容易に可能となった。また、メモリのレイアウトのパラメータを変更して実行することで、さまざまなレイアウトに容易に変更することも可能である。

今後は、現在の実装には含まれていない、計算そのものの処理にかかる時間を考慮したより正確な FPGA とメモリのタイミングモデルを構築できるシステムについて検討していきたいと考えている。

表 3　メモリアクセスの所要クロック数

計算順	X 全部→ Y 全部		X, Y 交互	
セル参照順	上下左右	反時計回り	上下左右	反時計回り
1	56,450	59,890	118,330	118,330
Layout 2	41,278	41,158	20,648	20,588
3	38,878	40,318	19,948	20,168

謝辞

本研究は JSPS 科研費 基盤研究 (C) 25330067 の助成をうけたものです。

また、本研究は東京大学大規模集積システム設計教育研究センターを通し、日本ケイデンス株式会社の協力で行われたものです。Xilinx ISE は Xilinx University Program によって提供されたものを利用しています。

参考文献

[1] 佐々木徹, 溝口大介, 長嶋雲平. Car-parrinello 計算向け三次元 FFT ロジックの開発. 情報処理学会論文誌. コンピューティングシステム, Vol. 45, No. 11, pp. 313–320, Oct. 2004.

[2] Xilinx Inc. *Virtex-5 FPGA User Guide*, v5.4 edition, Mar. 2012.

[3] Micron Inc. *DDR2 SDRAM 512Mb: x4, x8, x16*, w edition, Apr. 2014.