卒業研究論文

Cell/B.E. 向けフレームワーク NestStep の改良

指導教員　　松尾 啓志 教授
　　津邑 公暁 准教授

名古屋工業大学 工学部 情報工学科
平成 18 年度入学　18115022 番

今井 満寿巳

平成 22 年 2 月 8 日

Cell/B.E. 向けフレームワーク NestStep の改良

今井 満寿巳

内容梗概

近年，プロセッサの発熱量増加問題により，クロック周波数の向上が頭打ちになりつつあり，プロセッサのスカラ処理性能の向上が難しくなってきている．そのため，今日のプロセッサではマルチコア化を進め，並列処理性能を向上させる事で，プロセッサ全体としての処理性能の向上を図っている．また，比較的単純なアーキテクチャで高い性能を出すことができる SIMD 命令を充実させ，ベクトル処理性能を高めるという手法も，最近のプロセッサでは用いられている．その一つに，ヘテロジニアスマルチコアプロセッサの Cell/B.E. があり，高い処理性能を目指した SIMD 命令を持つが，その高い処理性能を生かすためには，高度なプログラミング技術が必要であり，プログラマを育成する妨げとなっている．そのため今日，Cell/B.E. でのプログラミングの煩わしさを少しでも軽減できるようなフレームワークが検討および開発されている．

本論文ではその一つである，Cell/B.E. 向けフレームワーク NestStep に注目した．しかし，NestStep を使用する上で，Cell/B.E. の DMA 転送の制限や，NestStep が SIMD 演算に未対応であることなどの問題点が挙げられる．そこで，この問題点を解消するために，NestStep を改良することで，DMA 転送制限の隠蔽，NestStep の SIMD 対応，さらに SIMD 演算の記述サポートを実現し，その評価を行った．

評価の結果，DMA 転送の制限が隠蔽され，ユーザの負担の削減を実現した．また，NestStep を SIMD 演算に対応させることで，NestStep を用いたプログラムの高速化を実現した．

本研究では SIMD 演算の記述サポートとして，SIMD 演算を使用したコードへの変換を実現したが，変換には制約が伴うものであった．今後の課題として，SIMD 演算を使用したコードへの変換の制約を緩和することが考えられる．

Cell/B.E. 向けフレームワーク NestStep の改良

目次

1 はじめに

近年，プロセッサの発熱量増加問題により，クロック周波数の向上が頭打ちになりつつあり，プロセッサのスカラ処理性能の向上が難しくなってきている．そのため，今日のプロセッサでは，マルチコア化を進め並列処理性能を向上させる事で，プロセッサ全体としての処理性能の向上を図っている．また，比較的単純なアーキテクチャで高い性能を出すことができる SIMD 命令を充実させ，ベクトル処理性能を高めるという手法も，最近のプロセッサでは用いられている．

Cell Broadband Engine(以下 Cell/B.E.)[1] は，ヘテロジニアスマルチコアプロセッサの 1 つであり，SONY，東芝，IBM の 3 社によって共同で開発されたプロセッサである．Cell/B.E. はエンタテイメント向け専用機に搭載することを主な目的として，高い処理性能を目指した SIMD 命令を持つマルチコアプロセッサである．Cell/B.E. は，1 つの汎用プロセッサ PPE(PowerPC Processor Element) と 8 つの演算プロセッサ SPE(Synergistic Processor Element) を，1 チップ上に集約したヘテロジニアスマルチコアプロセッサである．

しかし，その高い処理性能を生かすためには，高度なプログラミング技術が必要であり，Cell/B.E. プログラマを育成する妨げとなっている．そのため，プログラミングの煩わしさを少しでも軽減できるようなフレームワークが検討されている．

本研究では，Cell/B.E. プログラミングに関する技術的障壁を緩和するために，Cell/B.E. のフレームワークである NestStep[2] に注目した．NestStep には Cell/B.E. のプログラミングをサポートする機能があり，その中に DMA 転送の記述をサポートする機能がある．しかし，Cell/B.E. の DMA 転送には，転送データ量に制限があり，NestStep を用いて記述した場合でも，プログラマはその制限を意識してプログラミングをする必要がある．また，NestStep はいくつかのデータ構造を持っているが，それらはベクタ型に対応しておらず，NestStep を用いたプログラムでは SIMD 演算を使用することは困難である．

本論文ではこのような NestStep の問題点を解消するため，NestStep の改良とその評価をする．2 章では NestStep，Cell/B.E. のアーキテクチャ，動作の仕組みについて概略を述べる．3 章では，NestStep の問題点を解消するための，NestStep の改良点について述べる．4 章では改良点の実装内容について述べる．5 章では改良点についての評価を行う．最後に 6 章で本論文全体をまとめる．

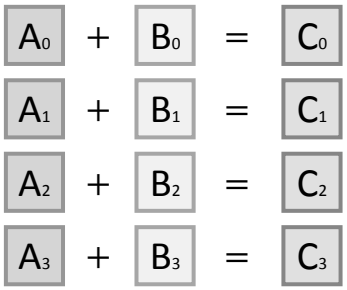

図 1: スカラ演算

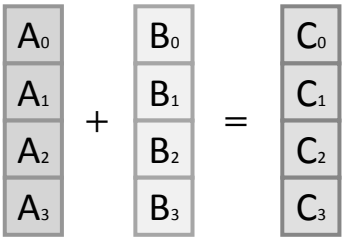

図 2: SIMD 演算

2 背景

本章では，本研究で取り扱ったフレームワーク NestStep，および Cell/B.E. について述べる．

2.1 Cell/B.E.

Cell/B.E. はヘテロジニアスなマルチコア SIMD プロセッサである．本節では Cell/B.E. について述べる前にまず，Cell/B.E. の性能を引き出すために重要な役割を持つ SIMD 演算について述べる．

2.1.1 SIMD 演算

SIMD とは Single Instruction Multiple Data の略で，SIMD 演算とは，1 命令で複数のデータに対して処理をする演算方式である．SIMD 演算に対して，1 命令で 1 つのデータに対して逐次的に処理をする，従来の処理方式をスカラ演算と言う．具体例として，4 つのデータに対する加算処理を例に，スカラ演算と SIMD 演算の違いについて述べる．**図 1** にスカラ演算の処理を，**図 2** に SIMD 演算の処理を示す．従来のスカラ演算では，図 1 のように 4 つの処理結果を求めるために，4 回の加算命令を逐次的に実行する．一方，SIMD 演算では，図 2 のように 1 回の加算命令で 4 つの処理結果を求める．このように，SIMD 演算はスカラ演算と比較して，同じ処理を少ない命令数で実行できる効率的な演算と言える．

しかし，SIMD 演算には制限がある．SIMD 演算では，複数のデータに対する処理を 1 命令で実行できるが，予め命令として定義されたパターンの処理しか実行できない．図 2 のように複数のデータに対して同じ加算処理をするような場合は，SIMD 演算で処理をすることができるが，**図 3** のように，複数のデータに異なった処理をするような場合は，SIMD 演算で処理することができない．

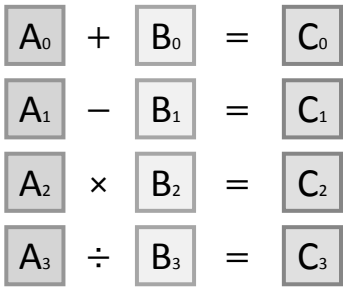

図 3: SIMD 演算できないパターン

また，SIMD 演算を使用する場合，用いるデータはベクタ型のデータでなければならない．従来のプログラミングで使用する int，float，double などのデータ型をスカラ型と言うのに対し，SIMD 演算に用いるデータ型をベクタ型と言う．Cell/B.E. で扱うベクタ型のデータは全て 128 ビット (=16 バイト) 固定長であり，データ型によって 2 から 16 の要素を持つ．各要素はスカラ型のデータであり，ベクタ型のデータは，16 バイト長の配列にスカラ型のデータがひとかたまりに格納されたものと考えられる．SIMD 演算では，このようにデータをひとかたまりで扱うことで，複数のデータに対する処理を，1 命令で実行している．

2.1.2 Cell/B.E. アーキテクチャ

Cell/B.E. は，SONY，東芝，IBM の 3 社により共同開発された，高い処理性能を目指したマルチコア SIMD プロセッサである．Cell/B.E. は，1 つの汎用プロセッサ PPE(PowerPC Processor Element) と 8 つの演算プロセッサ SPE(Synergistic Processor Element) を 1 チップ上に集約したヘテロジニアスマルチコアプロセッサである．Cell/B.E. はシングルスレッド時よりもむしろ，マルチスレッド時に高い性能を発揮するプロセッサであり，9 つのコアをあわせた浮動小数点演算能力は最大時で 200GFLOPS を超える．Cell/B.E. アーキテクチャの概要を**図 4** に示す．

各プロセッサコアは，EIB(Element Interconnect Bus) と呼ばれる高速なバスで接続されている．EIB の転送速度は 204.8GB/秒である．また，EIB はメインメモリや外部入出力デバイスとも接続されている．SPE はそれぞれ 256KB のローカルストア (以下 LS) と呼ばれるスクラッチパッドメモリを持つ．メインメモリへのアクセスは LS を介してのみ行う．また，SPU(Synergistic Processor Unit) とは，SPE の演算処理を行う核となるユニットであり，各 SPU は直接メインメモリや他の SPE 上の LS にアクセスすることはできず，Memory Flow Controller(以下 MFC) と呼ばれるユニットを利用する必要がある．この LS とメモリ間でのデータ転送に擁する時間は非常に大きいため，

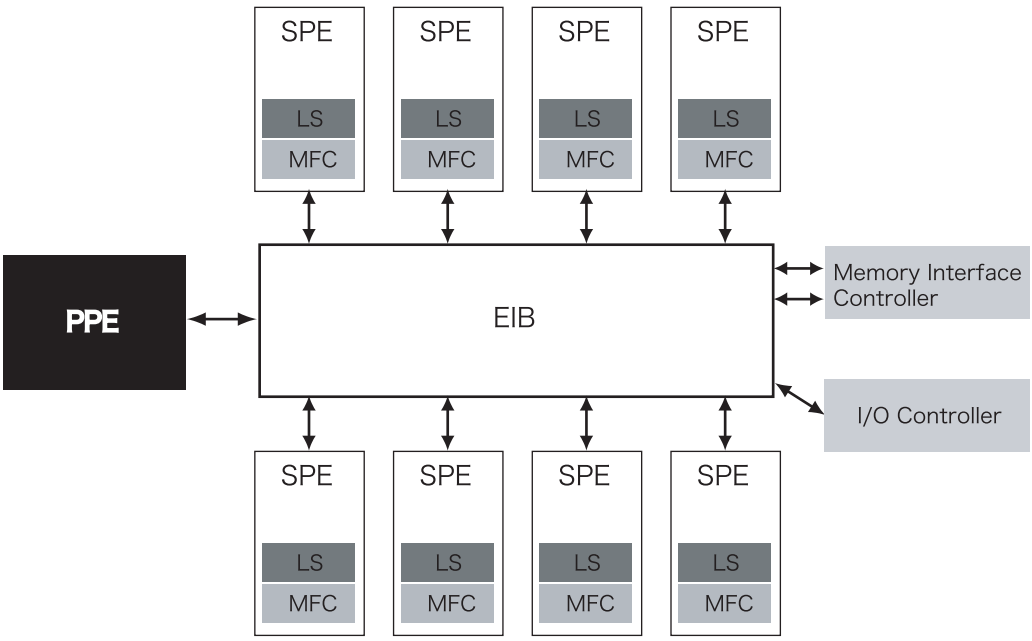

図 4: Cell/B.E. アーキテクチャ

メモリレイテンシを隠蔽する方法として，ダブルバッファリングという手法がよく使用される．これは，LS 上にバッファを 2 つ用意しておき，片方のバッファがメインメモリにアクセスしている裏で，もう片方のバッファで計算を行うという方法である．

また SPE は，128bit の SIMD 演算を行うパイプラインを持ち，LS へのアクセスを行うパイプラインと合わせて，2Way のスーパースカラパイプライン構造となっている．計算のレイテンシは大きいが，128 本の豊富なレジスタを利用して，複数のデータに対する処理を行うことで，計算のレイテンシを隠蔽することが可能である．一方で，Cell/B.E. は独特なアーキテクチャであるが故に，プログラム開発を行う際に注意を払う必要がある．以下にこれを具体的に挙げる．

(1) Cell/B.E. の特徴を行かしたプログラムの開発には，Cell/B.E. に搭載されている，性質の異なる 2 種類のコアで動作するプログラムを，それぞれに対して用意する必要がある．また，それらのプログラムは，協調して動作するように設計する必要があるため，並列分散プログラミングの技術が必要となる．

(2) 複数の SPE を協調動作させるようなプログラムを記述する場合，DMA 転送と呼ばれる，Cell/B.E. で用いられているデータ転送方式や，プロセッサ間で同期をとる場合のメモリシステムの機構などのアーキテクチャの詳細を理解する必要がある．また，Cell/B.E. の種類によっては，搭載されている SPE の個数が異なったり，Cell/B.E. のハードウェア提供ベンダやその上に載せる OS，低レベルライブ

ラリによってSPEを制御する方法が異なるため，開発されるアプリケーションがアーキテクチャや下位システムに依存した移植性の低いものになりやすい．

(3) SPEは，SPU SIMD命令等の組み込み関数を用いることで，高速な演算を実現している．しかし，SPEの性能を引き出すために，コンパイラ等による開発ツールを用いて，自動的に最適化を行い，プログラムの高速化に繋がる箇所を抽出することはまだ困難である．そのため，SPEの性能を引き出すためには，プログラマがSPEを効率的に使用する技術を学習する必要があり，Cell/B.E.を用いたプログラミングの技術障壁は高いと言える．

2.2 NestStep

NestStepは，並列計算用BSPモデルを採用した並列プログラミング言語である．NestStepについて述べる前に，まずBSPモデルについて簡単に述べる．

2.2.1 BSPモデル

BSP(Bulk Synchronous Parallel)[3]モデルとは，1990年にValiantらによって実装されたもので，Oxford大学により提案された並列アーキテクチャのひとつである．これはプログラムを計算，大域通信，バリア同期の3ステップの繰り返しで計算されると考えるモデルである．この3ステップをまとめてsuperstepと呼び，3つのステップは具体的に以下の3つの役割を担う．

(1) 各プロセッサ上で計算を行う段階．各プロセッサは局所変数もしくはリモート変数のコピーのみに対してアクセス可能である．

(2) 次のsuperstepに必要なデータをプロセッサ間で通信する段階．

(3) 各superstepを待ち合わせるバリア同期の段階．

superstepの概念を**図5**に示す．BSPモデルでは，実際に計算を行うプロセッサをworkerと呼び，workerの動作を指示するプロセッサをmasterと呼ぶ．BSPモデルの特徴として，実際のworker間の通信はmasterを中継して行われるが，プログラムの記述時にはmasterを中継することを明記する必要がないことが挙げられる．このworker間の通信部分は，図5の計算部分とバリア同期の間で行われる通信部分にあたる．BSPモデルでは，superstep内で計算に使われた値などを，任意の時刻に，任意のworker同士で自ら通信することはできない．

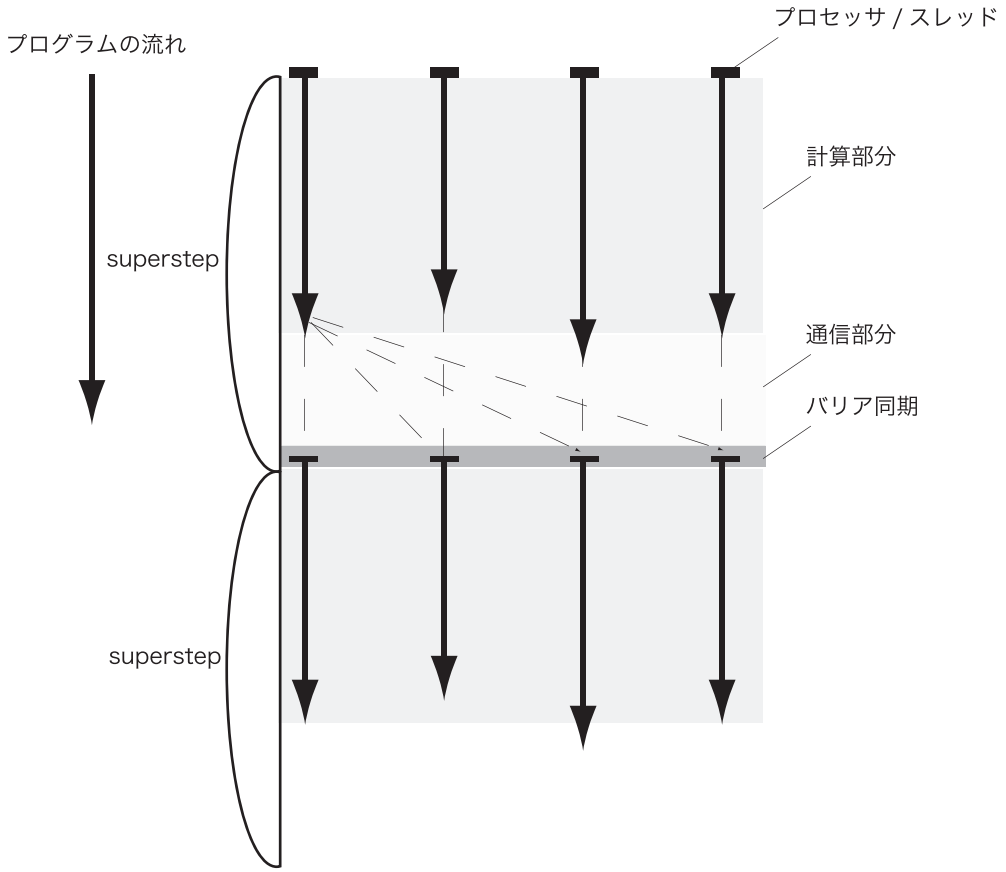

図 5: BSP モデルと superstep

BSP モデルでは，プログラムで記述された superstep の順序通りに，p 個のスレッドが並列計算を行う構造を持つ．図 5 では合計 4 つのプロセッサで superstep を実行している．

BSP モデルにおける superstep での実行時間とプログラムの総実行時間の関係を次に示す．ここで，

- バリア同期にかかる時間（オーバーヘッド）：L
- 通信にかかるデータ転送率：g
- 各プロセッサのプログラム最大実行時間：w
- 各プロセッサの最大通信量：h

とする．

各 superstep の実行時間 t(step) には次の関係がある．

$$t(step) = w + hg + L$$

superstep の実行時間は，計算部分，通信部分，バリア同期にかかる時間の総和で表される．まず，superstep の計算部分にかかる時間は，最も計算に時間がかかったプロセッサの実行時間が適用されるため，図 5 の例では左から 3 番目のプロセッサでのプログラム実行時間が w となる．次に，通信部分にかかる時間は，各プロセッサの最大通信量 h と通信にかかるデータ転送率 g の積で表される．そして，バリア同期の部分にかかる時間は，バリア同期のオーバーヘッド L である．

また，BSP モデルのプログラム全体の実行時間 t(prog) は，上で示した各 superstep の実行時間 t(step) を全て足し合わせたものである．そのため，次のように表すことができる．

$$t(prog) = \sum_{step} t(step)$$

2.2.2 Cell-NestStep-C

NestStep は，前項で述べた並列計算用 BSP モデルを採用した並列プログラミング言語であり，Christoph W. Kessler によって提唱された (1998 - 2000)．1998 年に Java を用いて拡張された NestStep-Java，2000 年に C 言語を用いて拡張された NestStep-C がある．NestStep-C は 2006 年に改訂され，MPI を用いたクラスタ上で動作が可能になった Cluster-NestStep-C も存在する．そして，2007 年に拡張された Cell-NestStep-C により，Cell/B.E. 上での動作が可能になった．NestStep は既存の BSP モデルを拡張することで，階層的なプロセッサ組織概念と superstep の入れ子による動的多層並列処理をサポートしている．ここで，Cell-NestStep-C ライブラリの概念図を**図 6** に示す．Cell-NestStep-C を用いて記述されたプログラム (C もしくは C++で記述可能) を用意し，NestStep ライブラリをリンクさせて Cell/B.E. 用の汎用コンパイラでコンパイルすることで，Cell/B.E. 上で動作が可能になる．

さらに NestStep は，BSP モデルにはなかった，ソフトウェアによる仮想共有メモリを実現しており，分散したメモリ領域ではなく一つの大きなメモリ領域とみなすことによって，superstep の同期を実現している．NestStep のサブセットプロトタイプは Java のようなシーケンシャルベース言語をベースに実装されており，Java や C のような命令型プログラミング言語を拡張させることで設計された．

NestStep は，オブジェクト言語構造や run-time サポートにより，プログラム実行の明確な制御とオブジェクトの共有化を可能にしている．

一方で NestStep では，共有およびプライベートな変数と配列が用意されている．プライベートな変数と配列は，一般的な C 言語のデータ型のように扱うことができる．

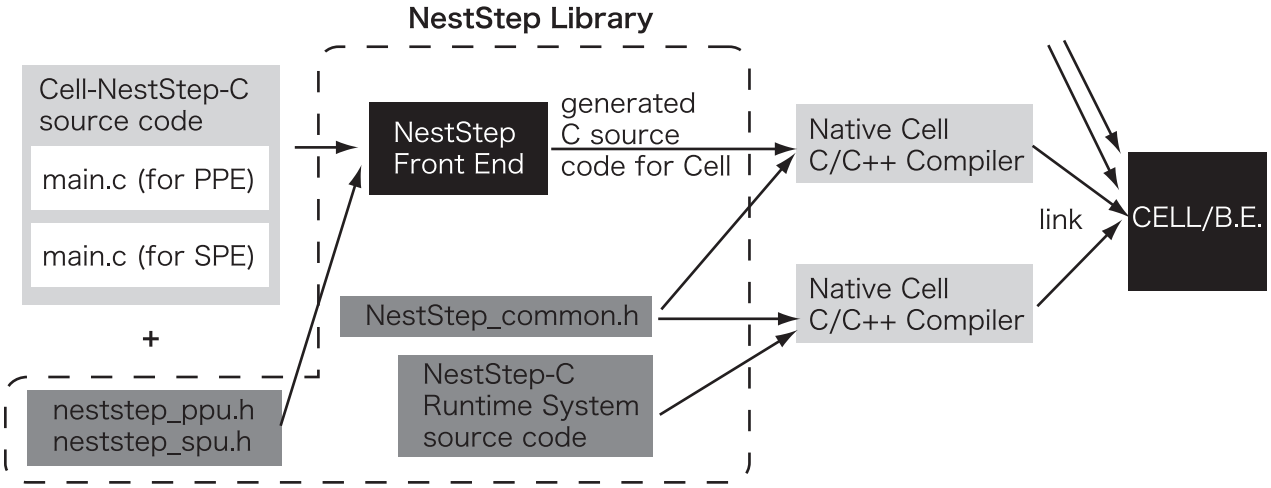

図 6: Cell-NestStep-C ライブラリの概念図

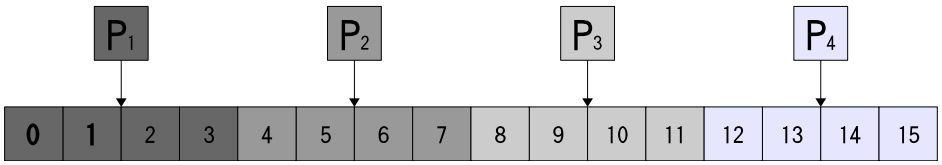

図 7: Block Distributed Array

また，プライベートな変数および配列は，それを扱うプロセッサ側からのみアクセスが可能である．共有変数は，各 superstep ごとに同期を行う．

さらに，NestStep は，block distributed array と cyclic distributed array という 2 種類の分散データ型をサポートしている．block distributed array は，配列をプロセッサ数に分割し，その分割したブロックを各プロセッサに割り当てる．**図 7** は，要素 16 個の配列を 4 つのプロセッサに均等に割り当てる block distributed array の例である．この例では要素 0〜3 はプロセッサ P1，要素 4〜7 はプロセッサ P2 に割り当てられる．cyclic distributed array は，配列をプロセッサ数で分割した block distributed array のブロックをさらに小さなブロックで仕切り，その仕切られたブロックを周期的にプロセッサに割り当てられる．**図 8** では，要素 0，4，8，12 はプロセッサ P1，要素 1，5，9，13 はプロセッサ P2 に割り当てている例である．記述するプログラムの特性に合わせて，2 種類の array を使い分けることで，プログラムの動作がより効率的になる．

2.2.3 問題点

Cell/B.E. では，SPU がメインメモリや他 SPE 上の LS へアクセスすることはできないため，MFC ユニットを用いた DMA 転送によってデータ転送をする必要がある．し

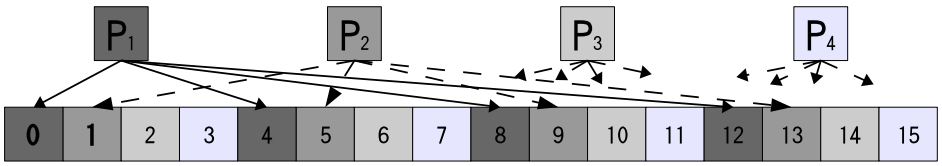

図 8: Cyclic Distributed Array

かし，Cell/B.E. の DMA 転送には，1 度の転送で転送できる最大データサイズが 16KB という制限がある．それより大きなサイズのデータを転送する場合は，複数回の転送をする必要がある．この制限により，ユーザは転送するサイズを意識したプログラミングをする必要がある．NestStep には Cell/B.E. のプログラミングをサポートする機能があり，その中に DMA 転送の記述をサポートする機能がある．しかし，NestStep の機能を用いた場合でも，DMA 転送の制限を無視することはできない．

また，Cell/B.E. は SIMD 演算を用いることで高速な演算を可能にするプロセッサである．SIMD 演算を使用する場合，演算するデータはベクタ型のデータである必要がある．NestStep は，先述の block distributed array や cyclic distributed array といった分散データ型のように，いくつかのデータ構造を持っているが，それらはベクタ型の変数に対応していない．そのため，NestStep のデータ構造を使用したプログラムでは，SIMD 演算を使用することができず，NestStep では Cell/B.E. の性能を引き出すことは難しいと言える．

3 提案

NestStep を用いたプログラミングには，いくつか問題点がある．そこで，その問題点を解消するために，NestStep を改良することを提案する．本章では，NestStep の改良点について述べる．

3.1 DMA 転送制限隠蔽

Cell/B.E. の DMA 転送には転送できるデータのサイズに制限がある．そのため，ユーザはその制限を意識したコードの記述をする必要があり，これはユーザの負担に繋がる．そこで，DMA 転送の制限をユーザが意識することなく，コードを記述できるようににすることを提案する．

NestStep を用いた DMA 転送の具体的なコードの例を**図 9** に示す．図 9 では，要素数 N のデータをメインメモリから LS へ転送している．転送には NestStep で実装されている get 関数を用いている．get 関数はメインメモリから LS への DMA 転送を行う

```
1  for(i = 0; i < N/buffersize; i++){
2    from = i * buffersize;
3    to = (i + 1) * buffersize - 1;
4    get(mmaddr, buffer, from, to);
5    wait_complete(buffer);
6    for(j = 0; j < buffersize; j++){
7      data[from + j] = buffer[j];
8    }
9  }
```

図 9: DMA 転送の例

関数であり，メインメモリのアドレス mmaddr から，LS のアドレス buffer にデータをを格納する．from と to は転送データを指定するものであり，from は転送データの先頭要素の添字を，to は末尾要素の添字を指している．転送はノンブロッキングであり，転送完了を待つためには，NestStep で実装されている wait_complete 関数を用いる．図 9 の例では，転送データ量が 16KB より大きい場合を考慮して，転送を buffer 単位で分割している．buffer の大きさを 16KB 以下にすることで，DMA 転送の制限を守ることができる．このように，既存の NestStep では，16KB を超える DMA 転送をするためには，ユーザが意識的に転送を分割する必要がある．

Cell/B.E. はマルチメディア系の処理に適したプロセッサであり，通常，扱うデータサイズは大きなものとなる．しかし，DMA 転送の制限により，1 度の転送での最大データサイズは 16KB であり，マルチメディア系の処理で扱うデータを転送するには不十分なサイズである．そのため DMA 転送の制限を超えた転送をすることは特別なことではない．16KB より大きなサイズのデータ転送をする場合は，図 9 のように転送を分割する必要がある．しかし，ユーザが本来扱いたいデータのまとまりは 16KB より大きなものであり，これを分割することはユーザの考えとは異なった処理であると言える．また，本来まとめて扱うはずのデータを分割して転送することは，プログラムの可読性を低下させ，バグの温床になる．

そこで，ユーザが DMA 転送の制限を意識することなく記述できるようにすることを提案する．16KB より大きなデータの転送を要求された場合，フレームワーク内で自答的に転送を分割する．これにより，ユーザは意識的に転送を分割する必要はなく，

ユーザの意図するコードの記述が容易になる．

3.2 SIMD 対応

Cell/B.E. の性能を引き出すためには，SIMD 演算を使用する必要があるが，NestStep のデータ構造はベクタ型の変数には対応しておらず，NestStep のデータ構造を用いたプログラムでは，SIMD 演算を使用することができない．そこで，NestStep の各データ構造にベクタ型を対応させることにより，NestStep を用いたプログラムであっても SIMD 演算を使用可能にすることを提案する．また，SPE で SIMD 演算を用いたプログラミングをするには，組み込み関数を使用する必要がある．しかし，組み込み関数を使用すると，コードの可読性が低下し，バグの温床になる．そこで，組み込み関数を使用せずに SIMD 演算を記述することが可能になるように，SIMD 演算を用いたコードの記述のサポートをすることを提案する．

Cell/B.E. は，SIMD 演算を用いることで高速に演算できるプロセッサである．特に SPE は SIMD に特化しており，スカラ型のデータ用のレジスタは存在せず，ベクタ型のデータを扱うレジスタと同じレジスタが使用される．また，スカラ型のデータ用のロード命令，ストア命令，演算命令もなく，スカラ演算は SIMD 命令を用いて実行される．

SPE におけるスカラ演算の流れを**図 10** に示す．図 10 では data1 を入力として演算し，その出力である data2 をメモリに格納する．まず，入力データを Register1 に読み出す．SPE はスカラ型のデータ 1 つのみを読み出すことはできないため，data1 を含む 16 バイトのデータを読み出す．SPE はスカラ型のデータを扱う場合，レジスタの特定の一部分のみに注目する．この特定の位置をプリファード・スカラ・エレメントと呼ぶ．SPE がスカラ演算をする場合，演算に用いるデータはプリファード・スカラ・エレメントに位置する必要がある．そのため，Register1 に読み出したデータをシフトした後，演算する．また，SPE はスカラ型のデータ 1 つのみを格納することはできない．そのため，データ 2 を格納するために，格納先を含む 16 バイトのデータを Register2 に読み出し，その一部分を data2 で置き換え，格納する．このように，SPE におけるスカラ演算は効率的ではない．そのため，Cell/B.E. の性能を引き出すためには，SIMD 演算を使用することが重要である．

SIMD 演算を使用するためには，処理するデータはベクタ型である必要がある．しかし，NestStep のデータ構造で使用できるのは，int，float，double の 3 種類のスカラ型データタイプのみであり，NestStep のデータ構造はベクタ型のデータには対応してい

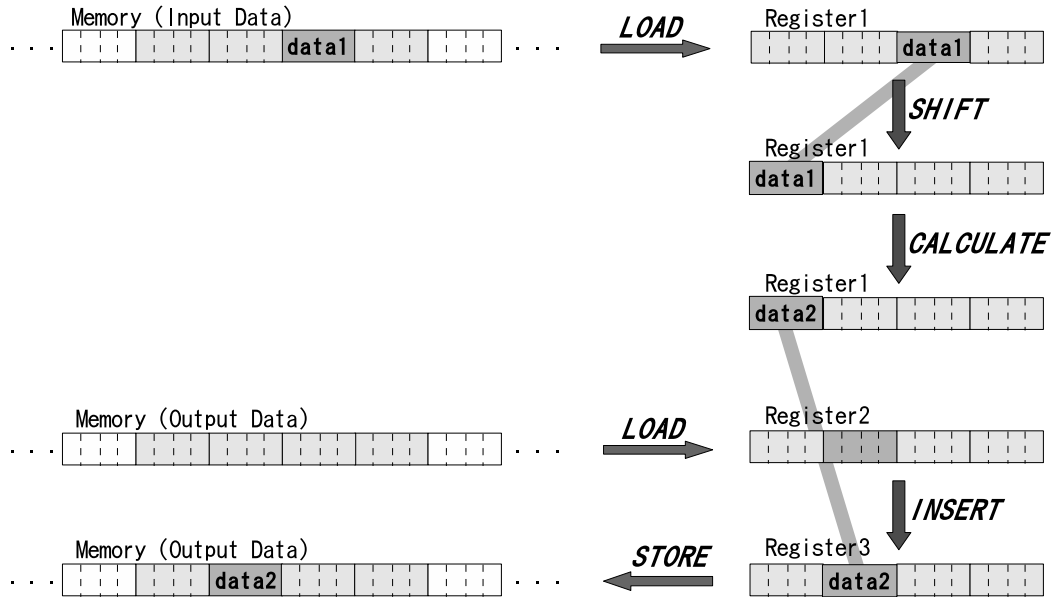

図 10: SPE におけるスカラ演算の流れ

ない．そのため NestStep のデータ構造を用いたプログラムでは SIMD 演算を使用することができない．SIMD 命令を使用しないと SPE での高速な演算は難しく，Cell/B.E. の性能を引き出すことができない．そこで，提案手法では，NestStep のデータ構造に，新たなデータタイプとしてベクタ型を追加することで，NestStep のデータ構造を用いたプログラムであっても，SIMD 演算を使用することを可能にする．NestStep を使用したプログラムで SIMD 演算が使用可能になることで，そのプログラムの高速化が実現できる．

SPE には SPU SIMD 命令という SIMD 命令が実装されている．SPU SIMD 命令の使用例を**図 11** に示す．図 11 ではスカラ演算と SIMD 演算で，$(A + B) - (C + D)$ を計算している．加算には spu_add，減算には spu_sub という，SPE の組み込み関数を使用している (6 行目)．このように SIMD 演算を使用する場合は，演算子ではなく組み込み関数を使用しなければならないので，コードの可読性が低下し，バグの温床になる．そこで，提案手法では SIMD 演算を使用する場合でも，演算子を用いた記述を可能にする．ユーザはベクタ型変数を用いた演算であっても，演算子を用いて記述する．ユーザが記述した演算子を用いたコードを，プリプロセッサを用いて組み込み関数を用いたコードに変換する．これにより，ユーザは SIMD 演算であっても演算子を使用することが可能であり，ユーザの意図するコードの記述が容易になる．

```
1  //(A + B) - (C + D)を計算する
2  //スカラ演算
3  ans_s = (scaA + scaB) - (scaC + scaD);
4
5  //SIMD演算
6  ans_v = spu_sub(spu_add(vecA, vecB), spu_add(vecC, vecD));
```

図 11: SPU SIMD 命令の使用例

4 実装

本章では，NestStep の改良として行った DMA 転送制限の隠蔽，SIMD 対応の具体的な実装内容について述べる．

4.1 DMA 転送分割

NestStep ではメインメモリ-LS 間で DMA 転送のための関数が実装されている．実装されている関数は 3 種類に分類され，それぞれ，メインメモリから LS へ転送する関数を get 関数，LS からメインメモリへ転送する関数を store 関数，転送の完了待ちをする関数を wait_complete 関数と呼ぶ．しかし，この関数を使用した場合でも，ユーザは Cell/B.E. の DMA 転送の制限を意識したプログラミングをする必要がある．そこで，DMA 転送の制限を隠蔽するために，新たな DMA 転送のための関数を提案する．提案する関数は，内部で自動的に DMA 転送を分割することで，ユーザから DMA 転送の制限を隠蔽する．提案する関数も 3 種類に分類される．以下にその詳細を示す．

get_nolimt(mmaddr, lsaddr, lbound, ubound)

get_nolimt はメインメモリから LS へ DMA 転送する関数である．メインメモリのアドレス mmaddr から LS のアドレス lsaddr へ，データを転送する．lbound と ubound は，転送するデータの先頭要素と末尾要素の添字であり，lbound から ubound までの添字のデータが転送される．

store_nolimit(mmaddr, lsaddr, lbound, ubound)

store_nolimt は LS からメインメモリへ DMA 転送する関数である．LS の領域 lsaddr からメインメモリのアドレス mmaddr へ，データを転送する．lbound と ubound は，転送するデータの先頭要素と末尾要素の添字であり，lbound から ubound ま

での添字のデータが転送される．

wait_complete_nolimit(lsaddr, size)

wait_complete_nolimit は，get_nolimt と store_nolimt による DMA 転送の完了を待つ関数である．wait_complete_nolimit は，LS のアドレス lsaddr と転送データ量 size を引数にとる．wait_complete_nolimit は，get_nolimt もしくは store_nolimt で使用した LS のアドレスと，その際の転送データ量 (バイト) を引数で渡すことで，その転送の完了を待つ．

get_nolimt，store_nolimt は，内部で NestStep の既存関数である get，store を呼び出している．転送データ量が 16KB を超えていた場合，get，store を複数回呼び出す．この際，1 回の呼び出しの転送データ量を 16KB 以下にする．これにより，ユーザから DMA 転送の制限を隠蔽する．get，store と get_nolimt，store_nolimt は引数，戻り値は変わらないため，ユーザは提案仮数を既存関数と同じように使用することができる．

Cell/B.E. では DMA 転送はタグで管理されており，NestStep では転送に用いる LS のアドレスがタグに対応付けされている．そのため，wait_complete ではそれを引数として受け取ることで DMA 転送を管理している．wait_complete の使用例を**図 12** に示す．図 12 では 30 要素のデータを，3 回に分割して転送している (1~3 行目)．get を 3 回呼び出したため，その完了待ちには wait_complete を 3 回呼び出している (5~7 行目)．このように wait_complete を呼び出す回数は，get や store を呼び出した回数と等しくなくてはいけない．get_nolimit，store_nolimit は関数内部で複数回 get，store を呼び出している．そのため，wait_complete_nolimit は get_nolimit，store_nolimit が呼び出す get，store と等しい回数だけ wait_complete を呼び出す必要がある．get_nolimit，store_nolimit が get，store を呼び出す回数は，転送データ量によって決定する．そのため，wait_complete_nolimit は転送データ量を引数として受け取る．

Cell/B.E. で DMA 転送を管理するために使用されているタグは，0 から 31 の 32 通り存在する．そのうち 0 と 1 は NestStep で予約されているため，ユーザが DMA 転送に使用できるタグは 30 通りとなる．提案関数で転送を分割する場合，分割した数だけタグも使用する．そのため，提案関数が処理できる転送要求の最大データサイズは 480KB(=30*16KB) である．しかし，SPE の LS のサイズは 256KB であり，256KB を超える DMA 転送をすることは無いと言える．そのため，提案関数を使用することで，Cell/B.E. の DMA 転送制限を隠蔽できる．

```
1  get(mmaddr, lsaddr1, 0, 9); //転送1
2  get(mmaddr, lsaddr2, 10, 19); //転送2
3  get(mmaddr, lsaddr3, 20, 29); //転送3
4
5  wait_complete(lsaddr1); //転送1の完了待ち
6  wait_complete(lsaddr2); //転送2の完了待ち
7  wait_complete(lsaddr3); //転送3の完了待ち
```

図 12: wait_complete の使用例

4.2 データ構造への対応

NestStep のデータ構造はベクタ型変数に対応しておらず，SIMD 演算を使用することができない．そこで，NestStep のデータ構造にベクタ型を対応させ，SIMD 演算を使用できるようにする．NestStep をベクタ型に対応させるために，VEC タイプの追加，VEC 用関数の追加，VEC 用処理の追加を行う．

4.2.1 VEC タイプの追加

NestStep のデータ構造を使用する場合，変数のデータタイプを選択する必要がある．既存データタイプには IVAR，FVAR，DVAR の 3 種類があり，それぞれスカラ型の int，float，double に対応する．そこで，NestStep で SIMD 演算を使用するために，新たなベクタ型に対応するデータタイプを追加する．

追加するデータタイプは VECI と VECF の 2 種類であり，それぞれベクタ型の vector signed int，vector float に対応する．ベクタ型には他にもいくつか型があるが，NestStep がサポートしている int，float，double 以外のベクタ型については本研究では実装しないものとする．また vector double という型もあるが，PPE に実装されている SIMD 命令である VMX 命令では vector double は実装されていないため，本研究では実装しないものとする．

4.2.2 VEC 用処理の追加

NestStep ではデータを処理する場合，そのデータタイプに応じた処理をする．そこで，新たに追加したデータタイプである VECI と VECF に対応する処理を追加する．追加する処理は，メモリに関する処理と演算処理の 2 種類に分類される．

メモリに関する処理は，領域確保やアドレス計算などがある．これらは使用するデー

タ型の大きさによって処理内容が異なる．しかし，追加したVECIとVECFはベクタ型であり，Cell/B.E.では16バイト固定長である．そのため，メモリに関する処理は，VECIとVECFに対応する2つの処理を追加するのではなく，両タイプに対応する1つの処理を追加する．

演算処理では，演算のために変数を使用するデータ型にキャストするため，データタイプによって処理内容が異なる．そのため，VECIとVECFに対応する2つの処理を追加する．また，追加したタイプはベクタ型であるので，演算にはSIMD演算を用いる必要がある．そのため，VECIとVECFに対応する，SIMD演算を用いた演算処理を追加する．

4.2.3 VEC用関数の追加

NestStepのget，store関数はデータタイプ別に実装されている．つまり既存のget，store関数にはIVAR用，FVAR用，DVAR用の3種類の関数がある．そのため，新たに追加したデータタイプであるVECI，VECFに対応する関数を追加する．

get関数，store関数は，使用するデータ型の大きさによって処理内容が異なる．そのため，データタイプ別に関数が実装されている．しかし，Cell/B.E.のベクタ型は型によらず16バイト固定長である．そのためVECI，VECFに対応する2種類の関数を実装するのではなく，VECI，VECFの両タイプに対応する1種類の関数を実装する．

4.3 SIMD記述サポート

SIMD演算を使用する場合は，演算子ではなく，組み込み関数を用いて記述しなければならない．そこで，ユーザには演算子を用いてコードを記述してもらい，そのコードをプリプロセッサを用いて変換する．これにより，SIMD演算であっても，演算子を用いて記述することが可能になる．

SIMD演算を用いたコードに変換する際，コード上の全ての演算をSIMD演算に変換することは不可能である．また，変換可能な演算であっても，その演算をSIMD演算に変換することで高速化されるかどうかの判断は困難である．そこで変換する範囲はユーザが指定することにする．変換範囲の指定はプラグマを用いて実装する．変換範囲の開始地点と終了地点をプラグマを用いて指定することで，その間の範囲を変換する．

変換は範囲内のfor文を対象にする．SIMD演算を使用することによる高速化を図る場合，複数データに同様の演算を行う処理に対してSIMD演算を適用することで，高速化が期待できる．そのようなコードは，多くの場合ループ文を用いて記述される．そ

```
1  #SIMD_BEGIN
2  for(i = 0; i < N; i++){
3    vec_add[i] = vec1[i] + i;
4    vec_mal[i] = vec1[i] * vec2[i];
5    count++;
6  }
7  #SIMD_END
```

図 13: ユーザが記述するコード例

```
 1  for(i = 0; i < N; i+=4){
 2    vec_add[i/4] = spu_add(vec1[i/4],
 3                           (vector signed int){i, i+1, i+2,i+3});
 4    vec_mal[i/4] = spu_convts(spu_madd(spu_convtf(vec1[i/4], 0),
 5                                       spu_convtf(vec2[i/4], 0),
 6                                       spu_splats((float)0)));
 7    count++;
 8    count++;
 9    count++;
10    count++;
11  }
```

図 14: 変換後のコード

のため変換範囲内のループ文を変換することで高速化が期待できる．

変換は次のルールに従って行うこととする．

- for 文のイタレータ変数のストライド幅を 4 倍にする
- スカラ型の変数のみを用いた文をループ中に 4 つ記述する
- ベクタ型の変数を用いた文を SIMD 命令を使用した文に変換する

このルールに基づいた変換の例を**図 13**，**図 14** に示す．図 13 のコードを変換したものが図 14 のコードである．

ユーザが記述するコードは，演算子を用いたコードであり，スカラ演算と SIMD 演

算は同様に記述される．そのため，ループ内でスカラ演算と SIMD 演算の両方が記述される場合がある．しかし，スカラ演算と SIMD 演算では 1 文で処理するデータ数が異なるので，ループで同じ回数の文を実行すると，2 種類の文で処理されるデータ数が等しくならない．そこで，ユーザには処理をするデータ数だけループが回るように記述してもらい，SIMD 演算とスカラ演算で処理されるデータ数が等しくなるように変換する．

NestStep に実装するベクタ型変数は vector signed int，vector float のみであり，両型とも 1 変数内に 4 つのデータを持つ．そのため 1 文で処理するデータ数は 4 つであり，ループ回数を 1/4 にすることで，SIMD 演算で処理するデータの数は，変換前のループ回数と等しくなる．ループ回数を 1/4 にするために，イタレータ変数の変化ストライドを 4 倍にする．図 13 ではイタレータ変数のストライドは 1 であるため (図 13 2 行目)，それを変換した図 14 ではイタレータ変数のストライドは 4 になる (図 14 1 行目)．

ループ中のスカラ型の変数のみを使用した文は，変換後のループの中に，同じ文を 4 回記述する (図 14 7∼10 行目)．これにより，ループ回数が削減された変換後のループでも，変換前のループ回数と等しい数だけ，その文が実行されることになる．

ループ中のベクタ型の変数を用いた文は，SIMD 命令を使用した文に変換する．変換する演算は四則演算のみとする．図 14 では加算と乗算の演算を変換している (図 14 2∼6 行目)．vector signed int の乗算の SIMD 命令は，Cell/B.E. に実装されている SIMD 命令には無いため，vector float にキャストして乗算を行う．キャストには SPU の組み込み関数である spu_convtf と spu_convts を用いる．spu_convtf は vector float にキャストする関数であり，spu_convts は vector signed int にキャストする関数である．vector signed int の乗算を行う際には，2 つのオペランドを vector float にキャストして演算した後，演算結果を vector signed int にキャストして格納する (図 14 4∼6 行目)．int の乗算をスカラ演算で実行する場合と，vector float にキャストして SIMD 演算で実行する場合の速度を**表 1** に示す．表に示されるのは SPE を用いて 1000 個の整数データ同士の乗算を行った場合の実行時間 (ミリ秒) である．SIMD 演算で実行する場合の時間は，キャストのオーバヘッドも含んだ時間である．表を見ると分かるように，キャストのオーバヘッドを考慮しても，SIMD 演算に変換することでの高速化は見込める．また，Cell/B.E. に実装されている SIMD 命令に，除算命令は無いため，逆数を求める演算と乗算を組み合わせることで，除算命令の代わりとする．そのため vector signed int の除算の場合も，vector float にキャストして演算を行う．

ループ中にベクタ型とスカラ型の両データ型が混在している文がある場合 (図 13 3

表 1: 1000 個の整数乗算にかかる時間

	実行時間 (ミリ秒)
スカラ演算	0.168
SIMD 演算	0.039

行目)，スカラ型のデータをパックして生成したベクタ型のデータを用いて，SIMD 演算を適用する (図 14 3 行目)．使用されているスカラ型のデータが定数ならば，生成するベクタ型データに格納するデータは全て同じでよい．しかし，使用されているスカラ型のデータが変数の場合，ループ中でその値が変化することを考慮しなければならない．本研究では簡単化のために，スカラ型変数の値の変化は，変数がイタレータ変数の場合のみ考慮する．ベクタ型を用いた文の中にイタレータ変数が使用されていた場合，図 14 のように，ループ中の変化を考慮してベクタ型データを生成する (図 14 3 行目)．このように変換することで，イタレータ変数の変化にも対応したコードを生成することができる．

このルールに基づいて変換をすると，ユーザが意図したものとは異なる動作をするコードを生成する場合がある．そのような場合を避けるため，変換には次の制約を定める．この制約は変換範囲内の for 文内部に課せられるものである．この制約を守らない場合，変換後のコードの動作は保証されない．

- ループ中に値が変化する変数 (イタレータ変数を除く) を評価する文を記述しない
- ベクタ型変数を用いた文内ではスカラ型変数への代入は行わない
- ループ回数は 4 の倍数であること
- ループ中で分岐命令を使用しない

本研究で上記の制約付きで変換をするが，この制約の緩和は今後の課題とする．

5 評価

本章では DMA 転送隠蔽によるコードの変化，SIMD 対応による速度変化を評価する．

5.1 DMA 転送制限隠蔽によるコードの変化

DMA 転送隠蔽によるコードの変化を**図 15** と**図 16** に示す．図 15 が既存の NestStep を用いて記述したコードであり，図 16 が提案関数を用いて記述した関数である．両図では要素数 N のデータの DMA 転送をしている．

```
1  //データ取得
2  for(i = 0; i < N/buffersize; i++){
3    get(mmaddr, buffer, i*buffersize, (i+1)*buffersize-1);
4    wait_complete(buffer);
5    for(j = 0; j < buffersize; j++){
6      data[i*buffersize + j] = buffer[j];
7    }
8  }
9
10 //データ処理
11 sort(data);
12
13 //データ格納
14 for(i = 0; i < N/buffersize; i++){
15   for(j = 0; j < buffersize; j++){
16     buffer[j] = data[i*buffersize + j];
17   }
18   store(mmaddr, buffer, i*buffersize, (i+1)*buffersize-1);
19   wait_complete(buffer);
20 }
```

図 15: 既存の NestStep を用いた DMA 転送

既存の NestStep を用いて記述した場合に対して，提案関数を用いて記述した場合は，DMA 転送の制限を意識すること無く記述ができている．そして，記述するコードの量の削減もされている．これらのことから，改良によってユーザの負担を削減することに成功したと言える．

5.2 速度比較

既存の NestStep のプログラムと，SIMD 演算を用いたコードに変換したプログラムとの，速度の評価を行った．評価を行った環境を**表 2** に示す．

評価には PLAYSTATION3 を用いた．NestStep ライブラリ (PPU 用，SPU 用) な

```
1  //データ取得
2  get_nolimit(mmaddr, data, 0, N-1);
3  wait_complete_nolimit(data, N*sizeof(int))
4
5  //データ処理
6  sort(data);
7
8  //データ格納
9  store_nolimit(mmaddr, data, 0, N-1);
10 wait_complete_nolimit(data, N*sizeof(int));
```

図 16: 提案関数を用いた DMA 転送

表 2: 評価環境

プラットフォーム	PLAYSTATION3
OS	Fedora 10
CPU	Cell/B.E. 3.2GHz
コンパイラ	gcc 4.1.1
最適化オプション	O0, O3

らびに，使用したテストプログラムを ppu プログラム用コンパイラ ppu-gcc(バージョン 4.1.1)，SPU 用コンパイラ spu-gcc(バージョン 4.1.1)，埋め込み SPU 用コンパイラ ppu-embedspu を用いてコンパイルした．最適化オプションは，SIMD 化による速度向上を評価するために O0 オプションを，より現実的な環境での評価を行うために O3 オプションを使用した．

評価プログラムには，map，pi を実装した，NestStep ライブラリを使用したプログラム (以下 original) と，そのプログラムを SIMD 演算を使用したプログラムに変換したプログラム (以下 trans) を用いた．

O0 オプションでの評価結果を**図 17**, **図 18** に，O3 オプションでの評価結果を**図 19**, **図 20** に示す．テストプログラム map，pi を，SPE を 1 から 6 基使用した場合の実行時間を評価した．評価結果は，original の SPE1 基での実行時間を 1 としたグラフである．

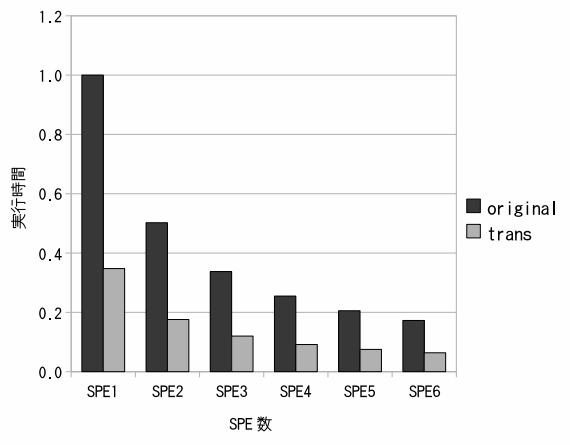

図 17: map O0

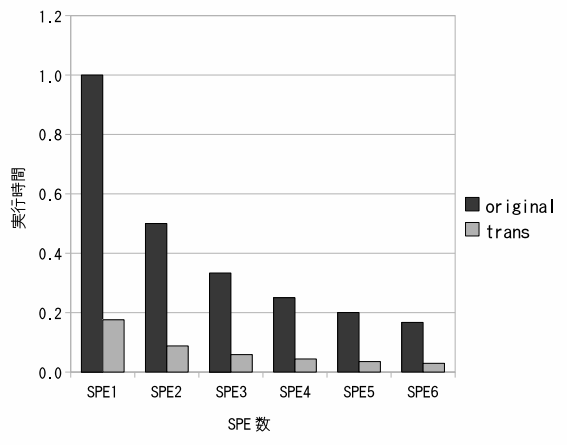

図 18: pi O0

グラフからは，original よりも trans の方が実行速度が速いことが分かる．O0 オプションを使用した場合は，map では約 2.8 倍，pi では約 5.7 倍の速度向上を実現した．O3 オプションを使用した場合は，map では約 2 倍，pi では約 20 倍の速度向上を実現した．また，SPE の数を増加させた場合，original，trans 共に同じ比率で高速になっている．このことから，プログラムの並列性を失うことなく，変換が行われていることが分かる．

map と pi ではプログラム中で扱うデータのサイズが大きく異なっており，map で扱うデータのサイズは，pi で扱うデータのサイズに比べて非常に大きい．PLAYSTATION3 のメモリ容量は，PPE のメインメモリが 256MB，SPE の LS が 256KB であり，メモリ容量が少ない．map と pi で速度向上率が異なる原因は，メモリ容量不足にあると思われるが，詳細な原因の調査は行っていない．

original を trans に変換する際，いくつかコードを変更した．変換するためには，宣言する変数をベクタ型として宣言するなど，変換範囲外のコードを SIMD 演算に対応させる必要があるためである．変換範囲外のコードの変更をすることなく，変換を可能にすることは，今後の課題である．

6 おわりに

本論文では，Cell/B.E. 向けフレームワーク NestStep を使用する際，問題となる点を述べ，それを解消するために，NestStep の改良をした．改良点は 2 つあり，1 つは DMA 転送制限の隠蔽，1 つは NestStep の SIMD 演算への対応である．DMA 転送制限の隠蔽することで，ユーザは DMA 転送の制限を意識しないコードの記述が可能に

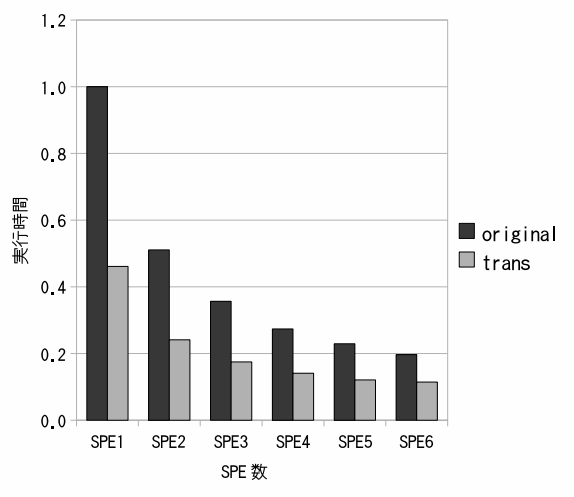

図 19: map O3

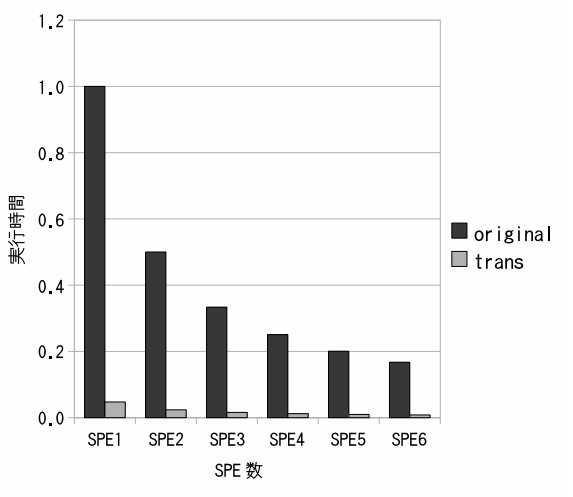

図 20: pi O3

なった．NestStep を SIMD 演算に対応させることで，NestStep を用いたプログラムであっても，SIMD 演算の使用が可能になった．また，既存の NestStep のプログラムを SIMD 演算を用いて記述することで，最大で 20 倍の高速化を実現した．さらに SIMD 演算を用いたコードを，プリプロセッサを用いて生成することで，ユーザへの負担を削減することを実現した．

今後の課題として，SIMD 演算の記述のさらなるサポートが挙げられる．本研究では制約付きで，ユーザに指定された範囲内の for 文のみを，SIMD 演算を用いたコードに変換したが，この変換にはいくつかの制約が課せられる．変換の制約を緩和し，変換可能な範囲を拡張させることで，さらなるユーザの負担の削減が実現できる．

謝辞

本研究のために多大な御尽力を頂き，日頃から熱心な御指導を賜った名古屋工業大学の松尾啓志教授，津邑公暁准教授，齋藤彰一准教授，松井俊浩助教に深く感謝致します．また，本研究の際に多くの助言，協力をして頂いた松尾・津邑研究室および齋藤研究室の方々に深く感謝致します．

参考文献

[1] Sony Computer Entertainment: *Cell Broadband Engine Architecture*, 1.01 edition (2006).

[2] Keβler, C. W.: NestStep: Nested Parallelism and Virtual Shared Memory for the

BSP model (1999).

[3] Valiant, L.: A bridging model for parallel computation, *Communication of the ACM*, Vol. 33, pp. 103–111.