

メモリ分散型アレイアクセラレータのための 命令生成手法の開発と評価

楠田 浩平^{†1} 姚 駿^{†1} 中島 康彦^{†1}

我々は、近年多用される科学技術演算やビッグデータ処理、画像処理などといったデータ量の多い演算の速度向上、そして消費電力低減の両立を目的として、演算器群とシングルポートメモリを組にした構造をリング接続した構成のメモリ分散型アレイアクセラレータを提案してきた。しかし、このメモリ分散型アレイアクセラレータの性能を最大限利用するためには、アプリケーション開発者におけるハードウェア構成の十分な理解と専用命令セットの把握が必須であるため、相応の開発コストが必要となる。アプリケーション開発者向けのメモリ分散型アレイアクセラレータ専用コンパイラやライブラリを開発する必要がある。本稿では、既存コンパイラが出力する x86-SIMD 命令列を用いて、メモリ分散型アレイアクセラレータ専用命令列を生成する命令生成手法、メモリ分散型アレイアクセラレータの構造、およびその性能を最大限利用するための命令生成手法におけるチューニングポイントについて述べる。生成された命令列と、ハンドアセンブルされた命令列における必要な命令段数とデータ再利用率を比較し、命令生成手法の有用性を確認した。また、既存のミニコア CPU と速度比較を行った結果、GRAPES ベンチマークにおいて、およそ 3.2 倍程度の実行効率を持つことがわかった。

1. はじめに

現在、科学技術演算やビッグデータ処理、画像処理といった、処理データ量が多く、かつ並列性の高い演算の需要が年々高まっている。これらの演算をより高速に実行するため、GPGPU など多数の演算ユニットを内包したシステムの研究開発が行われている。これらシステムの高効率化・高速化を目指すに当たり、演算器の増強や動作周波数の向上だけでなく、データ転送の高速化やデータ移動・配置の適切化なども極めて重要な要素であり、演算部の高速化などと同等のプライオリティで共に研究が進められている。

我々は、演算の効率化、消費電力の低減を目的とし、単位演算器群にシングルポートメモリを組み合わせた構造をアレイ状に接続した、メモリ分散型アレイアクセラレータ (Energy-aware Multimode Accelerator eXtension : EMAX) を提案している[1], [2]。単位処理に必要なデータを演算器群に対応するシングルポートメモリ (以下ローカルメモリ) に格納し、1 サイクルごとに命令写像をシフトすることにより、内部データパスにおけるデータ移動を最小限に抑えることができ、演算器とローカルメモリを効果的に使用することが高効率な処理を実現している。しかし、EMAX を前述のように高効率に動作させるプログラムを作成するにはハードウェア構成の十分な理解と命令セットの把握が必要となり、チューニングコストに課題があると言える。そこで、既存のコンパイラが出力する x86-SIMD 命令列を入力とする EMAX 専用命令列生成手法を実装し、その性能をハンドアセンブルされた命令列を比較対象とし、評価を行った。

本稿では、2 章においてメモリ分散型アレイアクセラレータ (EMAX) の概要、3 章において実装した命令変換手法の構成と、EMAX 向けのステンシル演算最適化機構について述べ、4 章においてハンドアセンブルされた命令列との段数・速度の比較、ミニコア CPU との速度の比較を行い、評価結果を示す。

2. EMAX の概要

本章では、EMAX の構造とシステム構成について述べ、専用命令セットについて説明する。

2.1 EMAX の構造

EMAX のシステム構成[3]を図 1 に示し、EMAX の演算単位である FU (Function-Unit) の構造を図 2 に示す。EMAX は、複数の FU を 2 次元アレイ状に配置した構造を取る。一行に 4 ユニットの FU が配置され、各段にわたってバスで接続されている。FU は図 2 に示す通り、演算器として EX1, EX2, EAG (アドレス計算用)、ローカルメモリ、EX2-FIFO LMM-FIFO を備える。各 FIFO は行方向の共有バスと接続されており、1 つの FIFO から、行方向 4 つの FU が内包する 8 個の FIFO にデータを転送し、共有することが出来る[4]。また、命令の縦方向シフトに対応するため、最上段の FU と最下段の FU はバスで接合されており、リング構造を取っている。FIFO や縦方向シフト・リング構造の必要性については後述する。EMAX と接続ホストである PC とは USB3.0 にて接続されており、EMAX コンフィギュレーションデータや入出力データの通信を行う。

^{†1} 奈良先端科学技術大学院大学
Nara Institute of Science and Technology

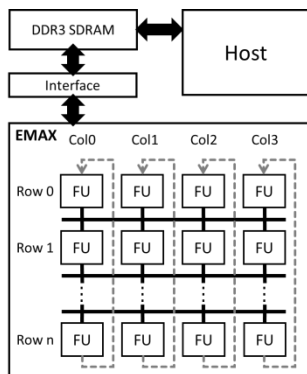


図1 EMAX のシステム構成

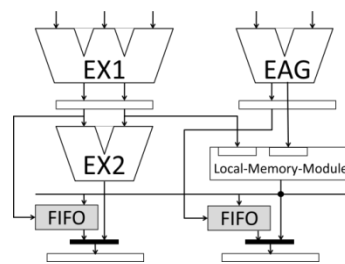


図2 FU の構造

//EMAX2 @row, col, dist [count] ALU_OP RGI & MEM_OP RGI LMM_CONTROL

図3 命令の書式

2.2 EMAX の命令体系

EMAX システムは、プログラムの最内ループ内命令を演算器群に写像し、外部ループを処理するホスト側と連携しながら演算を行う。EMAX 内で最内ループ内命令を高効率に実行するためには、EMAX 専用の命令セットが必要となる。EMAX の命令は、EMAX の論理的構造に近い形で記述することが出来る。具体的な書式を図3に示す。各命令は、写像位置を指定するために列方向位置 row, 行方向位置 col を設定する。また、dist には命令シフト時に写像された命令の段方向移動量を指定する。count には、処理を行う最内ループの繰り返し回数を指定する。ALU_OP には、EX1・EX2 で実行可能な命令を記述する。RGI では、ALU_OP で指定した演算に使用するレジスタの初期化に関する情報を記述する。MEM_OP には EAG で実行可能なロード・ストア命令を記述する。なお、EX1 でもロード命令を実行可能だが、EX2 では FIFO・ローカルメモリからの読み出しが可能なのに対し、EX1 では FIFO からの読み出しのみとなる。LMM_CONTROL には、当該 FU のローカルメモリにデータを転送するための情報を記述する。図4に、C 言語による演算の記述例を、図5に、図4の演算と等価な EMAX 命令列を示す。

3. 命令生成手法の概要

アプリケーション開発者が EMAX を高効率に利用するためには、EMAX のハードウェア構成と命令の特性について十分に理解しておく必要があるため、相応の開発コストが必要となる。開発コストとアプリケーション開発者の負担を少なくするためには、専用コンパイラやライブラリの開発が不可欠である。特に、既存のコンパイラ群を用いて EMAX 専用命令を出力できれば、特別なツールなどの導入や使用法の修得などを省くことができ、アプリケーション

$$B[i] = A[i] + A[i+1] + A[i+2] + A[i+3];$$

図4 C 言語による記述

```
//EMAX2 @0,0,0 [320] add (ri+=,4),r1 Base Addr of A is set to r1
//EMAX2 @1,0,0 [320] &ld (r1, 0), r2 LOAD A
//EMAX2 @1,1,0 [320] &ld (r1, 4), r3 LOAD A+1
//EMAX2 @1,2,0 [320] &ld (r1, 8), r4 LOAD A+2
//EMAX2 @1,3,0 [320] &ld (r1, 12), r5 LOAD A+3

//EMAX2 @2,0,0 [320] fadd (r2, r3), r6 Add ALL
//EMAX2 @2,1,0 [320] fadd (r4, r5), r7
//EMAX2 @2,2,0 [320] add (ri+=,4), r8 Base Addr of B is set to r8
//EMAX2 @3,0,0 [320] fadd (r6, r7), r9 &st r9,(r8,0) Store the result to B
```

図5 EMAX アセンブリによる記述

開発者の負担を最低限に留めることが出来る。

3.1 命令生成手法の構造

提案する命令生成手法は、既存コンパイラが出力する x86-SIMD 命令列を入力とし、EMAX 専用命令列を出力する。x86-SIMD 命令列が出力される条件として、コンパイラにより、元コードの演算に並列性があり、演算イタレーション間に依存性が存在しないことが挙げられる。EMAX のようなアレイ構造を持つ演算機構において、イタレーション間に依存性が存在すると正しいデータフローが成り立たないため、コンパイラによって依存性が存在しないと確認された x86-SIMD 命令を入力として使用することが望ましい。図6に命令生成手法の概要を示す。コンパイラから出力された x86-SIMD 命令列は、Lexer 部にて字句解析に掛けられ、命令種別やレジスタ番号、ベースアドレスなど命令変換に必要な情報を種類ごとに分析し、後段の Parser & Mapper 部に送られる。Parser & Mapper 部では、大きく5つの機能に分けられる。まず、字句解析からのデータを解析し、x86-SIMD 命令を EMAX が行うべき命令単位に分解する。次に、分解された命令列からロード命令のみを抜き出し、アクセスパターンとベースアドレスの関係から、EMAX で再利用可能なロードパターン群を生成し、マッピングを行う。ロード命令は、全ての演算命令におけるデータフローの起点となるため、正しいデータフローを維持するためには演算命令より先にマッピングを行う必要がある。また、ロード命令以外の演算命令において命令数と必要段数の削減を目的とした命令マージを行い、演算命令群を生成する。最後に、最適化されたロードパターンに基づいて、ロード命令を EMAX 命令列としてマッピングし、ロード命令のマッピング終了に続いて、ロード命令以外の演算命令をレジスタレベルのデータフローに基づいてマッピングする。すべての命令のマッピングが終了すると、EMAX の命令フォーマットに従って、マッピングされた命令列が出力される。

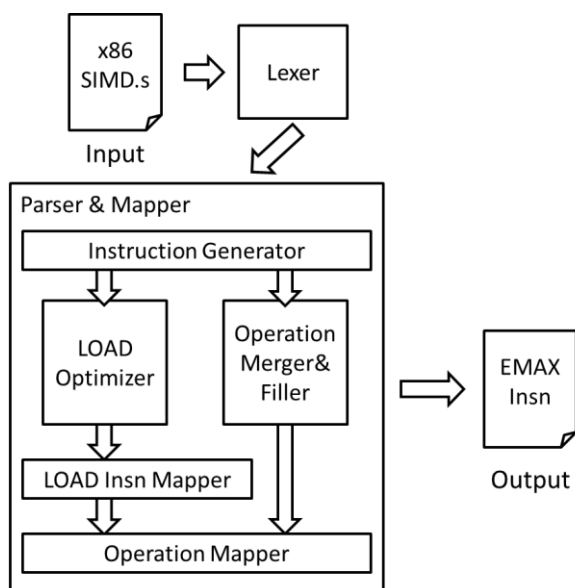


図6 命令変換手法の構成

3.2 SIMD 命令から EMAX 命令への命令変換

入力データは x86-SIMD 命令列であり、EMAX の命令アーキテクチャとは異なるため、それらを変換する仕組みが必要である。変換にあたり、x86-SIMD 命令から直接 EMAX 命令に変換せず、一時的に、より多くの情報を付加することが可能な内部命令に変換することで、後の命令最適化を容易にしている。x86-SIMD 命令から変換された内部命令と、対応する EMAX 命令の例を図7に示す。図7(a)のように、単純な mov 命令であれば、EMAX においても同様の命令が存在するので変換は容易である。しかし、図7(b)のように、命令のソースオペランドがメモリアクセスを伴う場合、EMAX において直接対応する命令は存在しない。従って、x86-SIMD 命令と同等の機能を複数の EMAX 命令を使用して実現することとなる。メモリアクセスを伴うソースオペランドを持つ x86-SIMD 命令を変換する場合は、本来の命令機能を実行する前にロード命令を実行し、どのソースオペランドにもデータが存在している状態にする必要がある。図7(c)のように、既に全てのソースオペランドにデータが存在する場合、その必要はない。

3.3 SIMD レジスタと EMAX レジスタの紐付け

EMAX はアレイ実行を行うため、x86-SIMD 命令を変換する際には、データフローを基に適切なレジスタを逐次マッピングする必要がある。図7(a)の mov 命令は、指定されたアドレスからデータをロードし、指定されたレジスタに格納する。これを EMAX 命令で実現する場合、ロードされたデータは後段に伝搬する必要があるため、新たにレジスタを割り当てる必要がある。更に、アレイ構造のデータフローを確保するために割り当てられたデータとレジスタがどの x86-SIMD レジスタに対応しているかを変換毎に紐付けていく必要がある。

vmmovups +1280(%rdi,%rax,4), %ymm11

	Internal INSN	EMAX
INSN	LOAD	ld
BASE_ADDR	%rdi(gr1)	gr1
SHIFT	1280	1280
REG_DEST	%ymm11(gr2)	gr2

(a) MEM, REG の場合

vaddps +1280(%rdi,%rax,4), %ymm9, %ymm5

	Internal INSN	EMAX
INSN 1	LOAD	ld
BASE_ADDR	%rdi(gr1)	gr1
SHIFT	1280	1280
REG_DEST	gr2	gr2
INSN 2	ADD	fadd
REG_SRC1	gr2	gr2
REG_SRC2	%ymm9(gr3)	gr3
REG_DEST	%ymm5(gr4)	gr4

(b) MEM, REG, REG の場合

vmulps %ymm0, %ymm5, %ymm3

	Internal INSN	EMAX
INSN	MUL	fmul
REG_SRC1	%ymm0(gr1)	gr1
REG_SRC2	%ymm5(gr2)	gr2
REG_DEST	%ymm3(gr3)	gr3

(c) REG, REG, REG の場合

図7 x86-SIMD 命令から内部命令への変換例

3.4 命令シフトと LOAD 命令の最適化

EMAX は一度ロードしたデータ及びストアしたデータを次回以降の演算で再利用することができ、高効率な演算が可能である。しかし、実際に再利用を行うには、ロード命令で指定するベースアドレスからのシフト量をもとにアクセスパターンを解析し、適切なマッピングを行う必要がある。図8に示す5点ステンシル計算におけるデータ読み込み例をもとに、命令シフトによるデータ再利用例を示す。EMAX において、i 回目の実行が終了すると、i+1 回目では命令開始位置を dist で指定した値だけ後段にシフトして実行する。ステンシル計算において、i 回目の実行と i+1 回目の実行における row1 と row2 のローカルメモリにプリフェッチするデータは共通であるので、命令をシフトすることによりデータを再利用することが可能である。命令シフトによる再利用が行える条件として、あるロード命令から row 方向にシフトしたデータをロードする命令、つまりベースアドレスから WD 分移動したアドレスをロードする命令があることが挙げられる。図8の i=k+1 のステンシルにおけるロード A' は、i=k のステンシルにおける A+WD 分移動したアドレス、すなわち C のロード命令によって既にプリフェッチされており、新たにデータをロードする必要が

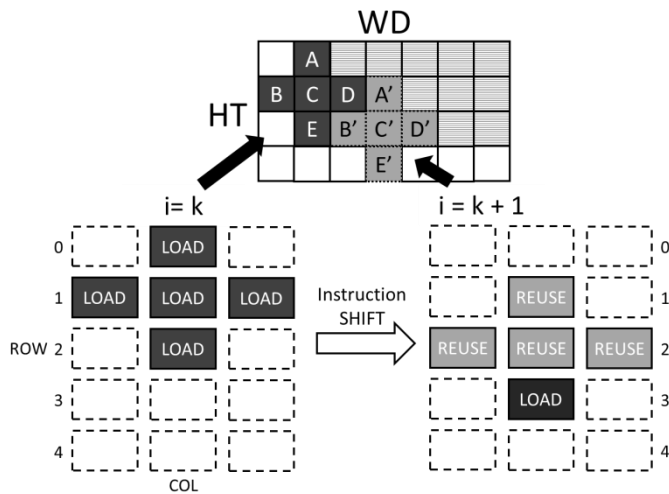


図 8 命令シフトによるデータ再利用例

表 1 FMA3 と等価な命令の組み合わせ

MUL (REG_A, REG_B), REG_TEMP
ADD (REG_TEMP, REG_C), REG_D
FMA3 (REG_A, REG_B, REG_C), REG_D

なく再利用が可能である。これは同ロード命令 C', E'においても同様である。また, EMAX の実行時には最内ループにおいて使用する WD 分のデータがローカルメモリに供給される。C のロード命令に供給された WD 分のデータは, col 方向で隣接するロード命令である B, D の要求するデータとマッチするため, FIFO を通じて再利用が可能である。本最適化機構では, 図 9 のように, row 方向に再利用が行えるロード命令群をベースアドレス別に整理・リスト化し, 命令シフトを行った際に適切に再利用されるよう, EMAX 上において row 方向, かつ同一 col にマッピングを行う。命令をシフトして row が変化した場合でも, 同一 col であれば, 命令移動先の FU 内のローカルメモリに前イタレーションのデータが存在するため, 再利用が可能となる。col 方向再利用が可能なロード命令は, row 方向再利用可能リスト内のアドレスが隣接するロード命令に付随する形で格納を行い, 同 FU もしくは同 col 上に存在する空き FU にマッピングを行う。ロード命令は他の命令におけるデータフローの起点となるため, 演算命令などよりも先に EMAX 命令列にマッピングを行う。

3.5 演算命令のマージによる段数削減と演算マッピング

EMAX に実装されている FMA3(Fused Multiply Add)命令は, 近年の科学技術演算で多用される命令の一つである。EMAX がターゲットとする演算において, FMA3 は多用されると考えられるため, この命令と同等の, 他の命令の組み合わせをマージすることで演算に使用する段数及び命令数を削減する事が可能となる。表 1 に FMA3 と等価な命令

	0	1	2	3	4
LOAD set 0	LOAD i	LOAD i+WD	LOAD i+2WD LOAD i+2WD-4 LOAD i+2WD+4	LOAD i+3WD	LOAD i+4WD
LOAD set 1	LOAD j	LOAD j+WD	LOAD j+2WD	LOAD j+3WD	EMPTY

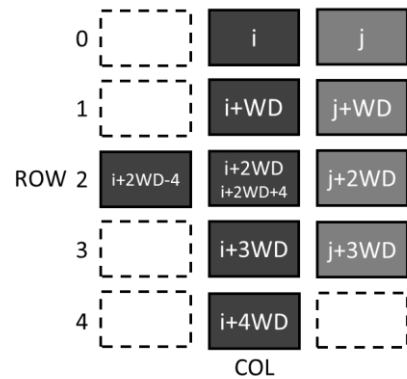


図 9 再利用可能なロード命令のマッピング

の組み合わせを示す。マージ機構の内部は大きく分けて 2 重ループとなっている。外側ループは MUL 命令の探索を行う。内側ループでは, 外側ループにより探索された MUL 命令を起点とし, データフロー方向へ ADD 命令を探索する。ADD 命令は, ソースレジスタとして外側ループにより探索された MUL 命令のデスティネーションレジスタを持つ必要がある。外側・内側ループにより MUL 命令と ADD 命令の組み合わせを発見した場合でも, 命令間のデータフローに依存関係がある場合, マージが不可能となるため, これを解消可能か調べる必要がある。ADD 命令のソースレジスタのうち, 表 1 の ADD 命令における REG_C にあたる MUL 命令のターゲットレジスタとは異なるレジスタが, ADD 命令と MUL 命令間の他の命令によって生成されているならば, データフローを遡ることになるため, そのままマージは行えない。しかし, この依存を発生させる命令と MUL 命令のデータフロー上の位置を交換することができれば, マージは可能となる。この場合でも, 依存を発生させている命令と MUL 命令の間に更に依存を発生させる命令を探索し, さらに依存が存在する場合, 本機構はマージを中止する。依存関係がない場合, もしくは命令位置の交換により依存を解消した場合は, MUL 命令を FMA3 命令に置換し, ADD 命令は消去される。全てのマージが終了すると, EMAX 命令列へマッピングを行う。

内部命令を先頭から 1 つずつ読み出し, 命令の種別ごとにマッピングに必要なレジスタ情報をキーとして EMAX 命令列を走査する。マッピングしようとする命令の全てのソースレジスタが, EMAX 命令列上の命令のデスティネーションレジスタとして存在すればマッピングが可能となり, マッピング可能な FU の探索を行う。EMAX は, 生成されたデータの同一段での利用が不可能であるため, 最後に確認されたデスティネーションレジスタの段以降からマッピングが可能である。以後は, 既に命令がマッピングされて

```
vmovups    0(%rbx), %ymm1
vmovups    4(%rbx,4), %ymm0
vmulps     -1280(%rdi,%rax,4), %ymm1, %ymm4
vmovups    -4(%rdi,%rax,4), %ymm2
vmfmaddd231ps -409600(%rdi,%rax,4), %ymm1, %ymm4
vaddps     4(%rdi,%rax,4), %ymm2, %ymm3
vmfmaddd231ps 0(%rdi,%rax,4), %ymm0, %ymm4
vmfmaddd231ps 1280(%rdi,%rax,4), %ymm1, %ymm4
vmfmaddd231ps 409600(%rdi,%rax,4), %ymm1, %ymm4
vmfmaddd231ps %ymm1, %ymm3, %ymm4
vmovups    %ymm4, 0(%rsi,%rax,4)
addq       $8, %rax
cmpq       $320, %rax
jb         ..B1.2
```

図 10 Jacobi 法の x86-SIMD 命令列

```
//EMAX2 @0,0,1 [320] add (ri+=4),r1
//EMAX2 @0,1,1 [320] add (ri+=4),r4

//EMAX2 @1,0,1 [320]          & ld (r1,0),r2
//EMAX2 @1,1,1 [320]          & ld (r1,4),r3
//EMAX2 @1,2,1 [320]          & ld (r4,-409600),r8
//EMAX2 @1,3,1 [320]          & ld (r4,409600),r16

//EMAX2 @2,0,1 [320]          & ld (r4,-1280),r5

//EMAX2 @3,0,1 [320] ld (r4,-4),r7
//EMAX2 @3,1,1 [320]          & ld (r4,0),r12
//EMAX2 @3,2,1 [320] fmul (r2,r5),r6
//EMAX2 @3,3,1 [320]          & ld (r4,4),r10

//EMAX2 @4,0,1 [320] fma3 (r2,r8,r6),r9
//EMAX2 @4,1,1 [320] fadd (r7,r10),r11
//EMAX2 @4,2,1 [320]          & ld (r4,1280),r14

//EMAX2 @5,0,1 [320] fma3 (r3,r12,r9),r13

//EMAX2 @6,0,1 [320] fma3 (r2,r14,r13),r15

//EMAX2 @7,0,1 [320] fma3 (r2,r16,r15),r17
//EMAX2 @7,1,1 [320] add (ri+=4),r20

//EMAX2 @8,0,1 [320] fma3 (r11,r2,r17),r19 & st r19(r20+=,0)
```

図 11 Jacobi 法の命令生成結果

いない空き FU を探索し、マッピングを行う。この機構により、EMAX 命令列の段数を最小限に抑えつつ、正しいデータフローを確保することが可能である。

3.6 EMAX 命令の出力

LOAD 命令の最適化と演算命令マージが終了すると、EMAX 命令列として出力する。内部命令に格納された各種レジスタやシフト値などのデータを、命令種別ごとに変化する構文規則に合わせ、出力を行う。図 10 に入力となる x86-SIMD 命令列を示し、図 11 に命令生成手法によって生成された EMAX 命令列を示す。

4. 性能評価と考察

本章では、作成した命令生成手法の評価を行う。ベンチマークプログラムをハンドアセンブルしたものと、本命令生成手法で生成したものを段数とデータ再利用率で比較する。また、変換元の x86-SIMD 命令列と変換後の EMAX 命令列の実行速度を比較する。段数及びデータ再利用率比

表 2 SIMD 命令列生成のためのコンパイラと
コンパイルオプション

Compiler	Compile Option
icc 14.0.1	-S -xCORE-AVX2
clang 3.4	-S -O3 -mavx

表 3 ハンドアセンブルされた
ベンチマークの詳細

	#of ALL	# of LOAD	# of FP	# of STORE
Jacobi	16	7	8	1
FD6	49	25	23	1
GRAPES	59	37	21	1

較には EMAX のクロックレベルシミュレータを用い、x86-SIMD 命令列との速度比較には、理想的なインターフェース及び動作クロックを仮定した EMAX 実行環境を想定した。

4.1 評価対象と項目

評価に用いるベンチマークは、Jacobi、FD6（連立方程式ソルバ）、GRAPES（大気シミュレータ）である。それぞれのベンチマークは、再利用可能なロード命令、再利用不可能なロード命令、演算命令、ストア命令のそれぞれが含まれている。段数・再利用率評価の基準となるハンドアセンブルされたベンチマークの詳細を表 3 に示す。これらベンチマークをハンドアセンブルして作成した EMAX 命令列と、命令生成手法によって生成した EMAX 命令列の段数・データ再利用率を比較する。命令生成手法の入力となる x86-SIMD 命令列によって出力される EMAX 命令列も変化するため、Intel-icc コンパイラと LLVM-toolchain の clang コンパイラ[6]の 2 つを比較対象とした。各コンパイラのバージョンと、コンパイルオプションについては表 2 に示す。なお、Intel-icc コンパイラ環境においては、コンパイラの持つベクトライズ能力を最大限引き出すため、ベンチマーク中の最内ループに依存性認識回避のための #pragma ivdep を記述し、より命令生成手法が得意とする x86-SIMD 命令列を出力可能な条件としている。また、x86-SIMD 命令との比較における、x86-SIMD 命令側の実行に使用するコンパイラやそのオプションについて、表 4 に示す。EMAX における理想的なインターフェースとして、PCI-Express x16 を仮定し、動作周波数を 400MHz とした。

4.2 評価結果

実際に、命令生成手法に x86-SIMD 命令列を入力し、出力された EMAX 命令列とハンドアセンブルされた EMAX の命令列を比較した。段数評価については図 12 に、データ再利用率については図 13 にデータを示す。x86-SIMD 命令列との速度比較については図 14 に示し、結果を考察する。

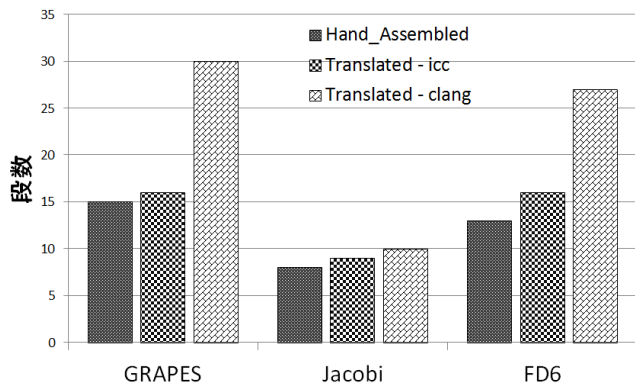


図 12 段数比較

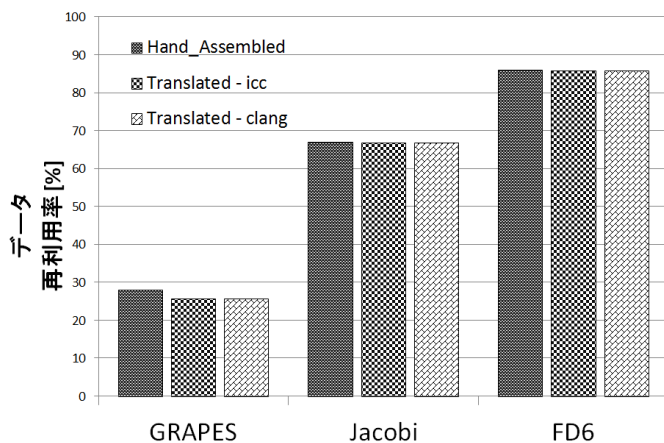


図 13 データ再利用率比較

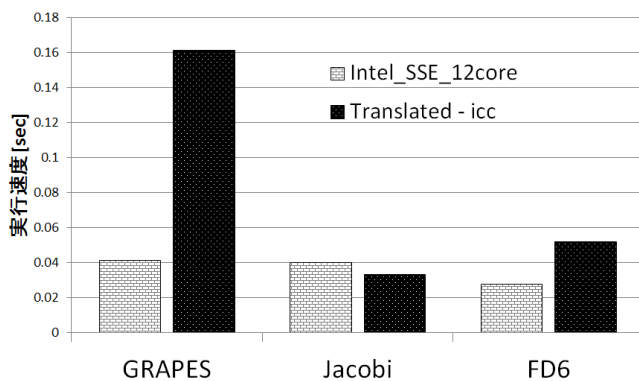


図 14 実行速度比較

表 4 x86-SIMD 命令側実行環境

Compiler	icc 14.0.1
Option	-xAVX
Machine	Intel Xeon E5-2620 x 2
	DDR3-1600 32GB

4.2.1 段数比較

Intel-icc が出力する x86-SIMD 命令列を入力として用いた命令変換手法の出力段数は、ハンドアセンブルされた命令列と比べても遜色なく、現在の EMAX の最大段数である 16 段以内に全て収まっている。一方、clang により生成された x86-SIMD 命令列を使用した場合には、GRAPES では 30 段、FD6 では 27 段となっている。通常のハンドアセンブルの場合、ロードしたデータに対して演算を行い、生成されたデータに対して、更にロードを行ったデータを使用して逐次演算を行う事が多い。しかし、本命令生成手法は入力として、EMAX 命令列とはレジスタの概念が異なる x86-SIMD 命令列を使用しているため、命令列実行中に生成した中間のデータを結果データのストア時まで徐々に演算しつつ使用する可能性が高く、演算中の全てのデータにおいて、時間的依存性があると言える。clang により生成された x86-SIMD 命令列はこれに当てはまり、命令列で使用する中間データの演算が終わると、1 段に 1 命令ずつ結果のマージを行う命令がマッピングされ、最終的に段数使用効率が比較的低い EMAX 命令列を出力する結果となる。Intel-icc の場合、コンパイルオプションにて指定した AVX2[6]命令における FMA3 命令が、演算で発生した中間データを逐次マージしていくため、ほぼハンドアセンブルに近い演算手順となっている。よってハンドアセンブルに近い段数の EMAX 命令列が出力できたと考えられる。本命令生成手法は、x86-SIMD 命令列を出力するコンパイラであれば使用可能であり、本結果のように出力された EMAX 命令列を評価し、目的に応じ適切なコンパイラをアプリケーション開発者が選択することが可能である。また、x86-SIMD 命令列を中間言語として使用することで、変換元の言語を問わないという特徴も備える。

4.2.2 データ再利用率比較

データ再利用率評価について、命令生成手法による EMAX 命令の再利用率は、ハンドアセンブルとほぼ同等な結果となった。本命令生成手法では性能面において、ロード命令の配置最適化によるデータ再利用と、それによるデータ転送量の削減による高速化を期待している。よって、ハンドアセンブルに勝るとも劣らないデータ再利用率を自動化で実現できた意義は大きい。現在、ロード及びストア命令において、データ再利用を明示的に指示する LMM_CONTROL は自動生成を行わないが、これら命令のベースアドレスとシフト量は管理、把握されており、アクセスパターンを基に LMM_CONTROL を生成することは可能である。これにより、アプリケーション開発者の負担を更に低減することが可能になる。

4.2.3 実行速度比較と演算効率

x86-SIMD 命令列と命令生成手法の実行速度比較では、特に GRAPES において差が見られる。そこで、GRAPES を動作させた EMAX のピーク性能を求めると、37 浮動小数点命令で 400MHz 動作であるため、ピーク性能は

$$36 \times 0.4[GHz] = 14.4[GFLOPS]$$

となる。実性能はシミュレータからのデータより 13.1[GFLOPS]であるため、演算効率を求めると、

$$\frac{13.1}{14.4} * 100 = 90.9[\%]$$

となる。一方、x86-SIMD 命令を実行するメニコアマシンは 8 並列 SIMD 演算、6Core*2CPU で 2.1GHz 動作であるため、ピーク性能は

$$8 \times 6 \times 2 \times 2.1[GHz] = 201.6[GFLOPS]$$

となる。実性能はシミュレータからのデータより計算を行った結果、56.5[GFLOPS]であるため、演算効率を求めると

$$\frac{56.5}{201.6} * 100 = 28.0[\%]$$

となる。実行時間で性能を比較すると、EMAX とメニコアマシンの間におよそ 4 倍の差があるが、演算効率を基準として比較すると、EMAX はメニコアマシンと比べて、およそ 3.2 倍優れていることを確認した。今後は、高い演算効率を生かして更なる省電力化や、並列接続による高性能化が見込める。

5. おわりに

本稿では、メモリ分散型アレイアクセラレータ向け命令生成手法及びその命令マッピング方式を提案した。x86-SIMD 命令列を入力とし、自動命令生成を行ったものと、それと等価であるハンドアセンブルされたコードを段数、データ再利用率において比較したところ、それぞれにおいて遜色ない結果を得た。また、メニコア CPU と比較して、およそ 3.2 倍程度の実行効率を持つことを確認し、これを活かしてさらなる省電力化や高性能化が見込めることがわかった。本稿では、ロード命令の配置最適化によるデータ再利用に重点を置いて命令生成の自動化を提案したが、実際の EMAX ハードウェアに関する内部バス制約なども命令生成アルゴリズムに取り入れ、更に実用的な命令生成手法を確立させていく予定である。

謝辞

本稿の執筆にご協力いただいた皆様、また、命令マージアルゴリズムの改善にアドバイスを頂いた方々に、謹んで感謝の意を表する。

本研究の一部は科学研究費補助金（基盤(A)24240005、萌芽 24650020、若手(B)23700060）、及び、半導体理工学研究センター（超低電圧で稼働できる耐エラープロセッサの製

造性を向上させる手法）による。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、シノブシス株式会社及び日本ケイデンス者の協力で行われたものである。

参考文献

- 1) 王昊, 姚駿, 中島康彦: "GCC の vectorizer を利用した演算器アレイ向け命令変換手法", 研究報告計算機アーキテクチャ(ARC), 2013-ARC-203 No.9, Feb. (2013)
- 2) 関賀, 姚駿, 中島康彦: "リング接続を利用しデータ移動を最小限にするアクセラレータの提案", 研究報告システムLSI設計技術(SLDM) SIG Technical Reports, 2013-SLDM-159, Vol.17, pp.1-6, Jan. (2013)
- 3) 藤原知広, 原裕子, 姚駿, 中島康彦: "リング型アレイアクセラレータのマクロパイプライン化による性能見込み", 研究報告計算機アーキテクチャ(ARC), 2013-ARC-206, No.14, pp1-6, Jul. (2013)
- 4) 稲垣慶和, 原祐子, 姚駿, 中島康彦: "リング型アレイアクセラレータ向け演算ライブラリの実装と性能評価", 研究報告計算機アーキテクチャ(ARC), 2013-ARC-206, No.1, pp.1-6, Jul. (2013)
- 5) clang: a C language family frontend for LLVM
<http://clang.llvm.org/>
- 6) Intel Instruction Set Architecture Extensions
<http://software.intel.com/en-us/intel-isa-extensions>