ハイパフォーマンス
コンピューティング 56－2
（１９９５．６．１）

ジェットパイプラインのためのコンパイル技術に関する一検討

佐々木 毅人, 仲池 卓也, 片平 昌幸, 沈 紅, 小林 広明, 中村 維男

東北大学大学院 情報科学研究科

命令レベルの並列性を利用し、ベクトル演算機能を有するジェットパイプラインは高速演算を可能にする。しかし、その性能を充分に活用するためには、コンパイラによって充分な最適化が行われなければならない。ベクトル化や並列化は、各々単独では、ベクトル計算機やVLIW計算機用コンパイラでさまざまな手法が用いられている。しかし、ジェットパイプラインではその両方の組合せにより高い演算性能を実現することを目的としているため、ベクトル化および並列化を組み合わせた命令スケジューリングについて検討する必要がある。本稿では、そのアプローチについて述べ、シミュレーションによりその効果を確認する。

A Study on A Compile Technique for Jetpipeline

Takehito Sasaki, Takuya Nakaike, Masayuki Katahira, Hong Shen, Hiroaki Kobayashi, and Tadao Nakamura

Graduate School of Information Science, Tohoku University

To achieve high computation power, we have proposed the Jetpipeline architecture that utilizes ILP (Instruction Level Parallelism) including vector operations in addition to scalar operations. In the Jetpipeline architecture, a compiler has an important role because it exploits ILP from operations In this paper, we present a compile technique for Jetpipeline based on both parallelizaiton for scalar operations and vectorization for vector operations. The proposed compile technique is examined through simulation experiments.

1 はじめに

コンピュータアーキテクチャには、常により高い演算処理能力が求められている。現在、命令レベル並列性を利用した、スーパスカラアーキテクチャ[1]やVLIWアーキテクチャ[2]、あるいはデータの連続的な処理を高速化したベクトル演算器などが高い演算性能を示している。しかし、その性能向上度は次第にピークへ近付いており、大幅な向上は難しいと考えられる。

また、高速演算性能を実現するためには、ハードウェアが高速な演算機能を有していることだけではなく、そこで用いられるコンパイラを中心とするソフトウェアが実行時の効率を高められることが重要である。このことは、ベクトル演算方式のスーパコンピュータにおける自動ベクトル化コンパイラやスーパスカラプロセッサ専用コンパイラが、それぞれ実行時の効率を高める上で重要な役割を果たしていることからもわかる。

我々は、次世代アーキテクチャへの一つのアプローチとして、並列演算機能とベクトル演算機能の融合による高速演算処理を実現するジェットパイプラインアーキテクチャを提案している[3] [4]。ジェットパイプラインは、ハードウェアとしてベクトル演算と並列演算の両方を実現し、高速な演算性能を有する。一方で、ハードウェアの演算効率を左右する命令スケジューリングは全てコンパイラで行う[5] [6] [7]。

本稿では、ジェットパイプラインのための命令スケジューリング手法について考察する。ジェットパイプラインのためのコンパイラは、プログラム中のループのような部分に対してベクトル化を行う。その結果、ジェットパイプラインの目的コードにはスカラ命令とベクトル命令が混在する。ジェットパイプラインのスカラ演算器とベクトル演算器は、共通の命令パイプラインにより制御されるので、目的コード中の命令レベル並列性を利用するためには、スカラ命令とベクトル命令の区別なく並列化を行う。

第2節では、対象となるジェットパイプラインアーキテクチャについて述べる。第3節では、ベクトル化および並列化のためのコンパイル技術について述べる。第4節では、シミュレーションによる性能評価を行う。第5節はまとめである。

2 ジェットパイプラインアーキテクチャ

ジェットパイプラインは、同一の機能を持つ複数の命令パイプラインにより同時に命令をフェッチし、スカラ演算ユニットおよびベクトル演算ユニットを並列に駆動することができる。そのため、ベクトル演算による連続的な演算と、スーパスカラやVLIWのような並列化による命令の同時実行といった特徴を併せ持ち、より高速な演算の実行を可能にする。ジェットパイプラインの構成を図1に示す。

ジェットパイプラインの各命令パイプラインは、命令フェッチ、命令デコードの2つのセグメントで構成される。命令パイプラインは、全て同期して命令フェッチを行うので、目的コードは予め命令パイプラインの数に合わせられていなければならない。スーパスカラのようなアーキテクチャでは、動的に命令スケジューリングを行うが、ジェットパイプラインでは、コンパイラにより静的にスケジューリングされた目的コードを用いる。従って、ジェットパイプラインは命令スケジューリングのためのハードウェアを持たない。

各命令パイプラインでデコードされた命令は、命令の種類に応じてスカラユニットもしくはベクトルユニットで処理される。スカラユニットは、命令パイプラインと1対1に対応する整数演算用ALU、スカラレジスタファイルで構成される。ベクトルユニットは、整数演算用と浮動小数点演算用のベクトルパイプライン、およびベクトルレジスタファイルで構成される。

ジェットパイプラインは、スカラユニットにおいても、ベクトルユニットにおいてもパイプラインを用いることにより演算効率の向上を図っている。ジェットパイプラインでは、パイプラインの演算効率低下の一因であるデータハザードを解消するためにフォワーディング(forwarding)を行っている。ただし、浮動小数点演算命令と整数演算命令では、実行時間に隔たりがあるためフォワーディングによりデータが次の命令へ転送されるまでの時間が異なる。これを解消するために、コンパイラは実行時間を考慮したタイミングの調整を行わなければならない。また、ベクトル演算命令とスカラ演算命令の実行時間も大きく異なるので、これらのタイミング調整もコンパイラが行う。

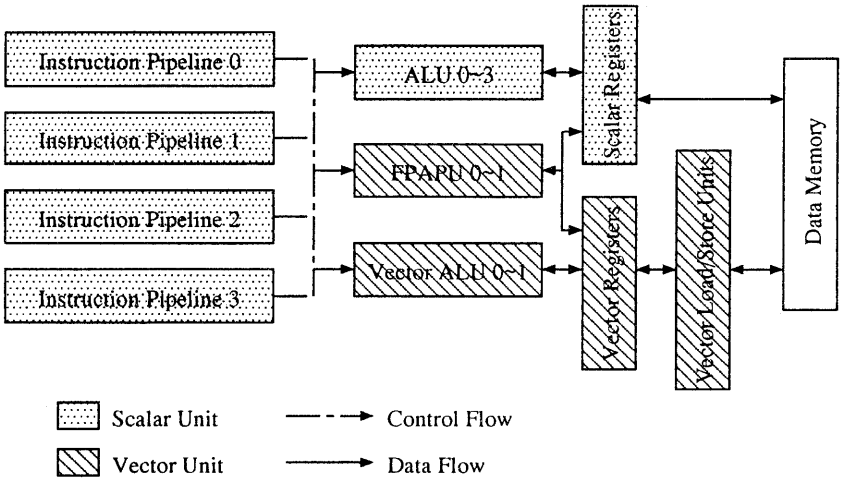

図 1: ジェットパイプラインのブロック図

3 ジェットパイプラインのためのコンパイル技術

第 2節で述べたように、ジェットパイプラインは動的な命令スケジューリング機構を持たず、コンパイラによる静的命令スケジューリングを前提としている。よって、ジェットパイプラインのためのコンパイラは、高級言語で記述されたプログラムをジェットパイプラインの命令セットへ翻訳することだけでなく、ハードウェアの実行効率を高めるために最適な並列化命令スケジューリングも行う [5] [6]。

ジェットパイプラインのためのコンパイラ（以下、ジェットパイプライン・コンパイラ）の特徴は、ループのような同一演算が繰り返し行われる構造に対して、ベクトル命令を生成する点にある。併せて、プログラムの持つ命令レベル並列性を利用した命令スケジューリングを行う。

具体例として次のようなプログラムを考える

```
for (k = 0 ; k < 1000 ; k++) {
  x[k] =  q + y[k]
       * (r * a[k] + t * b[k]);
}
```

ジェットパイプライン・コンパイラでは、はじめにこのようなループのベクトル化可能性について調べる。このループプログラムは繰り返しの間に依存関係を持たないので、ベクトル化可能である。

ベクトル化可能なループと判断されると、ベクトル命令を生成可能な形式に変換される。

```
N = 1000 / LENGTH;
for (k = 0 ; k <= N ; k++) {
  j = k * LENGTH;
  for (i = 0; i < LENGTH; i++) {
    x[j+i] =  q + y[j+i]
           * (r * a[j+i]
           +  t * b[j+i]);
}
```

ここで LENGTH と示された定数はベクトル長である。ジェットパイプラインの一つのベクトルレジスタはスカラデータ 128 要素を格納できるので、ベクトル長は LENGTH = 128 となる。

変形されたループをジェットパイプラインのアセンブリ言語 [8] に翻訳すると、スカラ命令とベクトル命令が混在する次のようなコードになる。

```
    ldi 1000,%r4
    ldi LENGTH,%r5
    div %r4,%r5,%r4
    ldi 0,%r6       ;set k=0
L1:
    mul %r6,%r5,%r7
```

```
vbcast r,%v1
vlir %v4,adr(a),%r7
vmul %v1,%v4,%v4
vbcast t,%v2
vlir %v5,adr(b),%r7
vmul %v2,%v5,%v5
vadd %v4,%v5,%v4
vlir %v5,adr(y),%r7
vmul %v4,%v5,%v4
vbcast q,%v3
vadd %v3,%v4,%v4
vsir %v4,adr(x),%r7
addi 1,%r6,%r6
cmp %r4,%r6
jc ge,L1
nop
```

更に、このようなコードに内在する命令レベル並列性を利用して並列化することによって、ジェットパイプラインの目的コードが生成される(付録A参照)。以下では、ジェットパイプラインのための目的コードを生成する過程で重要な2つの手法、並列化とベクトル化の詳細について述べる。

3.1 ベクトル化

先の例で示したように、ジェットパイプライン・コンパイラでは、繰り返し間の依存関係がないループに対してベクトル化を行う。ベクトル化を行うことによって、スカラユニットに比べて高いデータ処理能力を有するベクトルユニットを有効に活用することができる。

ベクトル化は、コンパイルの比較的早い段階で行われる。これは、ベクトル化が繰り返し間に依存関係のないループに対して行われることと、ループ形態の変更を伴うためである。ここでいうループ形態の変更とは、先の例でも示したように、繰り返し数の多いループをベクトル長に合わせて多重化したり、あるいは、繰り返し数の少ない多重ループを一重化することによって繰り返し数をベクトル長に近付けることなどを指す。

ループ形態の変更を行うためには、コンパイル時にループの繰り返し数が予め決定されていなければならない。しかし、繰り返し中に条件分岐を含むループは、繰り返し回数が実行時に変更されることがある。このようなループをベクトル化した場合には、ベクトルマスクレジスタ(Vecotor Mask Register)を用いることによって、実行時に決定された分岐方向をベクトル演算に反映することが可能である。

以上のような操作は、すべてコンパイラにより自動的に行われるので、ジェットパイプラインコンパイラで行われるベクトル化は自動ベクトル化方式である。従って、ベクトル命令を生成したり、ベクトル化率を高めたりするためにソースプログラムの記述を変更する必要は全くない。

自動ベクトル化方式は、スーパコンピュータのためのコンパイラなどで広く用いられている。スーパコンピュータのような大型機では、ベクトル化率が高いほど実行時間の短縮につながる。そのため、ループの形態を大幅に変更し、繰り返し間に依存関係がある回帰型の演算などにおいてもベクトル化ができるようにコンパイラが設計されている。しかし、ベクトル化率を高めるためにループの形態を大幅に変更することは、複雑なデータフロー解析を必要とするなど、コンパイル・コストの増加につながる。

ジェットパイプライン・コンパイラでは、高度なベクトル化によってベクトル化率を高めるというアプローチは採用していない。ジェットパイプラインは、ベクトル演算のみによって高速化を達成しているのではなく、命令レベル並列性を利用することによっても演算処理量を増加させることができる。そのため、命令レベル並列性を増加させる手法を組み合わせて用いることは、極めて有効であると考えられる。

ジェットパイプライン・コンパイラでは、ベクトル命令を発行する際に命令レベル並列性を増加させる手法として、ループ・アンローリングを用いる。先の例にループ・アンローリングを適用すると以下のようになる。

```
N = 1000 / LENGTH;
for (k = 0 ; k <= N ; k++) {
  j = k * LENGTH;
  for (i = 0; i < 2*LENGTH; i+=2) {
    x[j+i] =  q + y[j+i]
           * (r * a[j+i]
           +  t * b[j+i]);
    x[j+i+1] =  q + y[j+i+1]
             * (r * a[j+i+1]
             +  t * b[j+i+1]);
}
```

ループ・アンローリングによって、一つの繰り返し中で実行される計算が増している。それぞれの式は互いに依存関係を持たないので、並列実行可能なベクトル命令として翻訳される。

3.2 並列化

第2節で述べたように、ジェットパイプラインのスカラユニット、およびベクトルユニットを構成する各演算器は、すべて4本の命令パイプラインでデコードされた命令を元に動作する。

並列化の対象となる命令列は、先の例で示した通り、ジェットパイプラインの命令セットに基づくアセンブリ言語へ翻訳されたものである。アセンブリは、ベクトル演算命令とスカラ演算命令が混在しているので、ベクトル演算命令とスカラ演算命令の区別無く並列化を行う必要がある。

並列化の過程は大きく2つに分けられる。はじめに、並列実行可能な命令を基本ブロックから選択する。続いて、選択された並列実行可能な命令を各命令パイプラインで同時にフェッチできるよう整列する。並列実行可能な命令の選択には、ディスパッチスタック [9] を用いる。ディスパッチスタックを用いた手法は、レジスタ間の依存関係のみに着目するため、命令の種類を問わない。従って、スカラ命令とベクトル命令の区別なく並列実行可能な命令を選択することができる。

3.2.1 ディスパッチスタックを用いた並列実行可能な命令の選択

ディスパッチスタック [9] を用いて並列実行可能な命令を選択する手順は、以下の通りである。はじめに、命令を基本ブロック毎にをディスパッチスタックに読み込む。次に、ディスパッチスタック内に読み込まれた命令のレジスタについて、それぞれのレジスタがどのような依存関係に拘束されているかを調べ、拘束されている場合にはスタック内のフラグを立てる。全くフラグの立っていない命令は、他の命令と無関係であり、並列実行可能な命令としてスタックから取り出すことができる。

以上の操作をディスパッチスタック内の命令がすべて取り出されるまで繰り返す。スタック内の命令がすべて取り出されると、次の基本ブロックをスタックに読み込み、同様の操作を繰り返す。

ジェットパイプライン・コンパイラでは、ディスパッチスタックをソフトウェアによって実現する。そのため、スタックサイズをほぼ無限とすることが可能であり。基本ブロックのすべての命令から並列実行可能な命令を選択することができる。ディスパッチスタックは、通常ハードウェアによって実現される。その場合、ハードウェア量に限りがあるため、スタックサイズに上限を設けなければならない。ジェットパイプライン・コンパイラでは、このような限界をソフトウェアで行うことで解消している。従って、基本ブロックが大きい場合には、ハードウェアで実現した場合よりも並列実行可能な命令を多数選択できる確率が高く、プログラムの持つ並列性を充分に活用できるという利点がある。

3.2.2 並列実行可能な命令の整列

ジェットパイプライン・コンパイラでは、ディスパッチスタックから一度に取り出すことのできる命令数に、特に制限を設けない。従って、ディスパッチスタックから取り出された命令を命令パイプラインの数に合うよう整列する必要がある。スタックから取り出された命令の数が、命令パイプラインよりも少ない場合にはノン・オペレーション命令を加える。逆に、命令パイプライン数よりも多い場合には、2サイクルに分けて命令フェッチされるように分割する。

また、これ以外にも実行時に効率の低下を招く原因となるリソース競合などを避けるために、目的コードの整列時にいくつかの制約を設けておく。

1. **リソース制約**: 浮動小数点命令およびベクトル命令については、同時に使用できるハードウェア・リソースに限りがある。リソース数を越える場合には、複数のサイクルで命令フェッチされるように分割する。
2. **実行遅延制約**: 命令の種類により実行に必要とするパイプラインステージ数が異なるために、実行時間の違いを生じる。これを吸収するために、ノン・オペレーション命令を挿入するなどのタイミング調整をコンパイラが行う。

リソース制約は、命令パイプラインのリソー

ス数による制限が主であるが、それに加えて、ベクトル演算器もしくは浮動小数点演算パイプラインのリソース数による制限もある。これらはそれぞれ2個ずつしか用意されていないので、同時にフェッチ可能なベクトル演算命令もしくは浮動小数点演算命令の数を制限する。並列実行する命令の組合せに制限を設ける場合、ハードウェアで行うにはリザベーションステーションのような特殊な回路を必要とするが、ジェットパイプラインではソフトウェアで行うので、コンパイラの変更だけで済む。よって、ハードウェア量の増加を防ぐことができる上、実行時に空いているリソースを調べるなどの無駄な処理を必要としないため、効率を高めることができる。

第2節で述べたように、スカラユニットにおけるパイプラインでは、通常のパイプラインと同様にフォワーディングによるデータハザードの解消を行っている。しかし、スカラの浮動小数点演算命令はフォワーディングを行った場合でも、整数演算命令に比べて実行時間が長いため、この遅延を考慮した命令の配置を行う必要がある。

具体的には、遅延を生じる命令とその結果を必要とする命令との間に、遅延を生じる命令と無関係な命令を挿入する。これにより、全命令に対する有効な命令の割合の低下を防止する。挿入できる命令が少ない場合にはノン・オペレーション命令を挿入する。

4 シミュレーションによる性能評価

第3節で述べた手法の有効性を確認するために、ベンチマークプログラムを用いたシミュレーションを行った。ジェットパイプライン・シミュレータは、SPARC Station 上に C 言語で構築されている。ベンチマークプログラムには、リバモア・ループからいくつかのループ (Kernel 1, 3, 5, 7, 9) を用いた。

シミュレーションを行うために、ベンチマークプログラムを GNU C コンパイラにより SPARC Station のアセンブリに翻訳する。これをジェットパイプラインの命令セットに変換し、命令スケジューリングを行った。命令スケジューリングは次の5通りで行った。括弧内は、スケジューリング方法と図2に示したグラフの凡例との対応を示す。図2は、シミュレーション結果である。

1. 並列化なし (Sequential)
2. スカラ命令のみのコードを並列化 (Dispatch Stack)
3. スカラ命令のみのコードを並列化 (Software Pipelining [10])
4. ベクトル命令を含むコードを並列化 (Vector)
5. ベクトル命令を含むコードを並列化 (Vector and Unrolling)

ベクトル命令を用いることの効果は、スケジューリング方法 2,3,4 の結果を比較することで確認できる。例えば、kernel 1 ではベクトル命令を用いることにより処理時間が短縮されている。一方、kernel 5 ではベクトル化の効果は薄く、実行時間が大幅に増加している。

kernel 1 と kernel 5 のプログラム構造の違いに着目すると、kernel 1 は繰り返し間に依存を持たないベクトル化に適したループであるのに対し、kernel 5 は繰り返し間に依存を持ち、ベクトル化にあまり適さないループである。そのため、kernel 1 ではベクトル化の効果が非常によく現れているのに対して、kernel 5 ではベクトル化を行ったために、繰り返し間の依存関係を解消するためのオーバヘッドが大きく現れている。

次に、ベクトル化の際に並列化を意識した場合の効果を確認する。ジェットパイプライン・コンパイラでは、ベクトル化の際にループアンローリングを併用し、並列実行可能なベクトル命令を増加させる。図2からわかるように、ベクトル化と並列化の組合せであるスケジューリング方法5は、ループの性質に依存せずに実行時間を短縮されている。

以上のように、ジェットパイプラインでは、ベクトル演算を並列実行することによって、ループが依存関係を持つかどうかとは無関係に実行時間を短縮することが可能であることがシミュレーションにより確認された。従って、ジェットパイプラインのためのコンパイラでは、ベクトル化を行う際には、ベクトル化率を高めることよりも並列実行可能なベクトル命令を増やすことに重点を置いた方が効果的であるといえる。

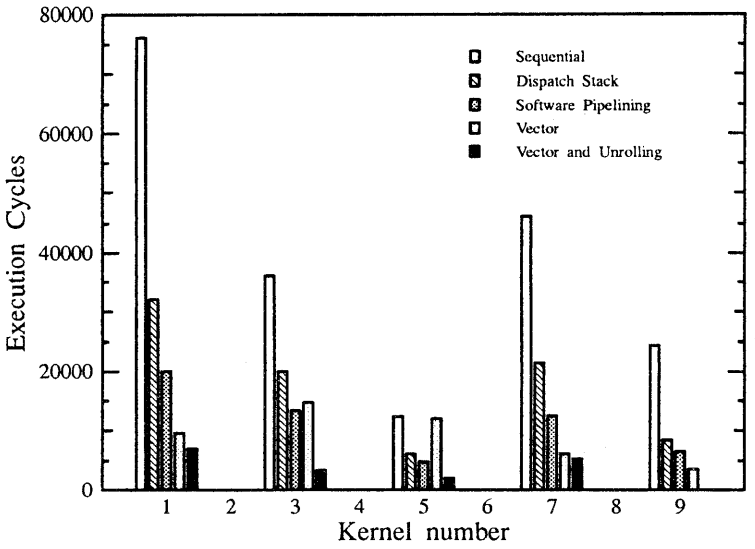

図 2: シミュレーション結果

5 おわりに

本稿では、ジェットパイプラインアーキテクチャのためのコンパイラにおけるベクトル化および並列化の手法について述べ、それらの組合せによって生成された目的コードがジェットパイプライン上で効率良く実行されていることをシミュレーションによって確認した。

ジェットパイプライン・コンパイラは、ベクトル化の際にベクトル化率を高めることよりも、ループアンローリングによって並列実行可能なベクトル命令を増やすことに重点をおいていることが、スーパコンピュータ向けコンパイラなどで用いられているベクトル化手法と異なる。

ベクトル化と並列化はトレードオフの対象である。それぞれ相互に補完しあっているので、今後はさまざまなプログラムによるシミュレーションを通して最適なバランスを発見する必要がある。

現在、ジェットパイプライン・コンパイラは、ベクトル化部分と並列化部分とに分けてインプリメント中である。並列化部分はほとんど完成しているが、ベクトル化部分は未完成の部分が多い。今後は、ベクトル化と並列化の割合を変えたシミュレーションを行っていく予定である。

コンパイラは、ハードウェアの多くの部分に依存するので、ハードウェアに関する詳細な設計が欠かせない。今後は、ソフトウェア的な面だけでなく、CAD などを用いたハードウェアのシミュレーションも行っていく予定である。

参考文献

[1] マイクジョンソン, 村上 和彰監訳. スーパースカラ・プロセッサ. 日経 BP 出版センター, 1994.

[2] J. A. Fisher. The VLIW machine: A multiprocessor for compiling scientific code. *IEEE Computer*, Vol. 17, No. 7, pp. 45–53, July 1984.

[3] 中村維男, 片平昌幸. ジェットパイプラインを用いた次世代スーパコンピュータ. 日本機会学会第 5 回計算力学研究講演会論文集, pp. 365–366. 日本機械学会, 1992.

[4] Masayuki Katahira, Hong Shen, Hiroaki Kobayashi, and Tadao Nakamura. Jet-pipeline: A hybrid pipeline architecture

for instruction-level parallelism. In *Proceedings of the High Performance Computing Conference '94*, pp. 317–323. National Supercomputing Research Centre, National University of Singapore, September 1994.

[5] 佐々木毅人, 片平昌幸, 小林広明, 中村維男. ジェットパイプラインのための並列化に関する一検討. 東北支部第 29 期総会・講演会講演論文集, No. 941-1, pp. 112–114. 日本機械学会, March 1994.

[6] 佐々木毅人, 片平昌幸, 小林広明, 中村維男. ジェットパイプラインのための命令スケジューリングに関する一検討. 電子情報通信学会 秋期大会 講演論文集, p. 83. 電子情報通信学会, June 1994.

[7] Masayuki Katahira, Takehito Sasaki, Hong Shen, Hiroaki Kobayashi, and Tadao Nakamura. Software pipelining for jetpipeline architecture. In *Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks*, pp. 127–134, December 1994.

[8] 片平昌幸. 命令レベル並列処理方式とその応用に関する研究. PhD thesis, 東北大学, 工学研究科 機械工学専攻, 1995.

[9] Ramon D. Acosta, Jacob Kjelstrup, and H. C. Torng. An instruction issuing approach to enhancing performance in multiple functional unit processors. *IEEE Transaction on Computers*, Vol. C-35, No. 9, pp. 815–828, September 1986.

[10] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceeding of the SIGPLAN '88 Conference on Programing Languages Design and Implementation*, pp. 318–328. ACM, June 1988.

付録

A 目的コード

```
        ldi 1000,%r4
        ldi LENGTH,%r5
        ldi 0,%r6
        nop

        div %r4,%r5,%r4
        nop
        nop
        nop

L1:     mul %r6,%r5,%r7
        vbcast r,%v1
        vbcast t,%v2
        vbcast q,%v3

        vlir %v4,adr(a),%r7
        vlir %v5,adr(b),%r7
        nop
        nop

        vmul %v1,%v4,%v4
        vmul %v2,%v5,%v5
        nop
        nop

        vadd %v4,%v5,%v4
        nop
        nop
        nop

        vlir %v5,adr(y),%r7
        nop
        nop
        nop

        vmul %v4,%v5,%v4
        nop
        nop
        nop

        vadd %v3,%v4,%v4
        nop
        nop
        nop

        vsir %v4,adr(x),%r7
        addi 1,%r6,%r6
        nop
        nop

        cmp %r4,%r6
        nop
        nop
        nop

        jc ge,L1
        nop
        nop
        nop

        nop
        nop
        nop
        nop
```