

(19)日本国特許庁(JP)

(12)特許公報(B2)

(11)特許番号

特許第7220914号

(P7220914)

(45)発行日 令和5年2月13日(2023.2.13)

(24)登録日 令和5年2月3日(2023.2.3)

(51)国際特許分類

F I

G 0 6 F 9/50 (2006.01)

G 0 6 F 9/50 1 5 0 C

G 0 6 F 9/52 (2006.01)

G 0 6 F 9/52 1 5 0 Z

G 0 6 F 17/16 (2006.01)

G 0 6 F 17/16 F

請求項の数 18 (全49頁)

(21)出願番号 特願2019-547267(P2019-547267)

(86)(22)出願日 平成30年4月27日(2018.4.27)

(65)公表番号 特表2020-518881(P2020-518881
A)

(43)公表日 令和2年6月25日(2020.6.25)

(86)国際出願番号 PCT/EP2018/060932

(87)国際公開番号 WO2018/197695

(87)国際公開日 平成30年11月1日(2018.11.1)

審査請求日 令和3年4月6日(2021.4.6)

(31)優先権主張番号 102017109239.0

(32)優先日 平成29年4月28日(2017.4.28)

(33)優先権主張国・地域又は機関
ドイツ(DE)

(73)特許権者 519309913

アイエルヌメリクス ゲーエムベーハー
ドイツ連邦共和国、1 0 4 0 7 ベルリ
ン、ダンツィガー ストラッセ 2 1 9

(74)代理人 100104411

弁理士 矢口 太郎

(72)発明者 カッチバック、ハイモ

ドイツ連邦共和国、1 0 1 1 5 ベルリ
ン、ハーバーザートストラッセ 6

審査官 三坂 敏夫

最終頁に続く

(54)【発明の名称】 コンピュータに実装する方法、コンピュータ可読媒体および異種計算システム

(57)【特許請求の範囲】

【請求項1】

コンピュータに実装する方法であって、次の複数のステップ：

異種計算システムの第1の計算カーネル(140-144)を持つ第1の処理ユニット(71)と第2の計算カーネル(150-154)を持つ第2の処理ユニット(72)を初期化するステップであって、

なおそこで第1の計算カーネル(140-144)と第2の計算カーネル(150-154)の各々は、複数の一般データタイプの要素を保存する第1のデータ構造(A)を受け取るように形成されたプログラムセグメント(220)から供給される数値演算を実行するように形成され、このプログラムセグメント(220)は、前記数値演算の出力を生成する演算的仕事量と、前記出力のサイズに関するデータ及び前記出力の構造に関するデータの内の少なくとも1つと、を含む関数メタ情報を有する、初期化するステップと、

上記関数メタ情報と第1データ構造(A)の実行時インスタンス(A1)のデータメタ情報を用いて、

実行時インスタンス(A1)で数値演算を実行するための第1計算カーネル(140-144)を第1処理ユニット(71)で実行するための第1の予測コストを計算し、

そして同じく実行時インスタンス(A1)で数値演算を実行するための第2計算カーネル(150-154)を第2処理ユニット(72)で実行するための第2の予測コストを計算するステップと、

なおそこで上記データメタ情報は、実行時インスタンス(A1)の実行時の大きさの情

報と実行時インスタンス（A 1）の実行時の場所の情報を含む、

およびつぎの2つのステップ：

第1の予測コストが第2の予測コストより低いまたは等しい場合に、実行時インスタンス（A 1）で数値演算を実行するための第1計算カーネル（140-144）を第1処理ユニット（71）で実行するステップ、および

第1の予測コストが第2の予測コストより高い場合に、実行時インスタンス（A 1）で数値演算を実行するための第2計算カーネル（150-154）を第2処理ユニット（72）で実行するステップ

のどちらか一方のステップと、

から構成される方法。

10

【請求項2】

請求項1の方法であって、上記プログラムセグメント（220）は、さらに上記一般データタイプまたはその他の一般データタイプの複数の要素を保存することができる第2のデータ構造を受け取るように形成され、

そこで上記第2のデータ構造の実行時インスタンスのデータメタ情報は第1の予測コストおよび第2の予測コストを数値的に計算するために用いられ、そして第2データ構造情報の上記実行時インスタンスの上記データメタ情報は、第2データ構造情報の上記実行時インスタンスの実行時の大きさ情報と第2データ構造情報の上記実行時インスタンスの実行時の場所情報を含む方法。

20

【請求項3】

請求項1～2のいずれか1つに記載の方法であって、そこで各々のデータメタ情報はさらに上記各々の実行時インスタンス（A 1）の同期状態か非同期状態かを示す情報を含み同期の方法を示す同期オブジェクト、およびまたは上記各々の実行時インスタンス（A 1）の実行時タイプ情報をも含む方法。

【請求項4】

請求項1～3のいずれか1つに記載の方法であって、そこで第1データ構造（A）、第1データ構造（A）の実行時インスタンス（A 1）、第2データ構造の実行時インスタンスと第2データ構造の少なくとも一つが各々の一般データタイプの可変数個の要素を保存することができ、

なおそこで、前記第1データ構造（A）、前記第1データ構造（A）の実行時インスタンス（A 1）、前記第2データ構造の実行時インスタンス、及び前記第2データ構造の少なくとも一つは、コレクション、ベクトル、行列、リスト、キュー、スタック、グラフ、トリーそしてn次元直方のデータ構造の少なくとも一つを保存しおよびまたは含み、

30

そこで、第1データ構造（A）の実行時インスタンス（A 1）と、第2データ構造の実行時インスタンスは画像を含み、

そこで、第1データ構造（A）実行時インスタンス（A 1）と、第2データ構造の実行時インスタンスは測定データを含み、

およびまたは、

そこで数値演算は行列演算である

方法。

40

【請求項5】

請求項1～4のいずれか1つに記載の方法であって、

そこで第1データ構造（A）、第1データ構造（A）の実行時インスタンス（A 1）、第2データ構造の実行時インスタンス及び第2データ構造の内の少なくとも一つが各々の実行時メタ情報を保存することができ、この実行時メタ情報は、実行時大きさ情報、実行時場所情報、実行時同期情報およびまたは実行時タイプ情報を含む

方法。

【請求項6】

請求項1～5のいずれか1つに記載の方法であって、

そこで多項式コスト関数が第1および第2の予測コストを計算するために用いられ、

50

そこで第 1 及び第 2 の予測コストの計算は、計算コストの決定および転送コストの決定の少なくとも 1 つを含み、

およびまたは、

そこで第 1 及び第 2 の予測コストは実行時中およびまたは第 1 処理ユニット (7 1) と第 2 処理ユニット (7 2) の初期化の後に計算される

方法

【請求項 7】

請求項 1 ～ 6 のいずれか 1 つに記載の方法であって、

そこで出力を生成する仕事量は、第 1 の予測コストおよび第 2 の予測コストを計算するために用いられ、

そこで出力を生成する仕事量は、上記第 1 処理ユニット (7 1) 上の一般データタイプの 1 要素への数値演算を実行するための第 1 の数値的仕事量と上記第 2 処理ユニット (7 2) 上の一般データタイプの 1 要素への数値演算を実行するための第 2 の数値的仕事量を含み、

そこで第 1 の数値的仕事量は上記第 1 処理ユニット (7 1) の決定された特性に応じて更新され、

およびまたは、

そこで第 2 の数値的仕事量は上記第 2 処理ユニット (7 2) の決定された特性に応じて更新される

方法

【請求項 8】

請求項 1 ～ 7 のいずれか 1 つに記載の方法であって、

さらに前記第 1 処理ユニット (7 1) および前記第 2 処理ユニット (7 2) を初期化する前に、前記異種計算システム上で制御カーネルを実行するステップを含み、

そこで上記制御カーネルは、第 1 処理ユニット (7 1) および第 2 処理ユニット (7 2) の特性を決定し、

第 1 処理ユニット (7 1) および第 2 処理ユニット (7 2) の決定された特性に従って関数メタ情報を更新し、

第 1 の予想コストおよび第 2 の予想コストを計算し、

第 1 の予想コストおよび第 2 の予想コストに基づいて、それぞれの計算カーネルを実行するための第 1 の処理ユニット (7 1) または第 2 の処理ユニット (7 2) を選択し、およびまたは、

選択された処理ユニット (7 1、7 2) 上でそれぞれの計算カーネルの実行を開始するように形成される方法。

【請求項 9】

請求項 1 ～ 8 のいずれか 1 つに記載の方法であって、さらに次のステップ、

第 1 処理ユニット (7 1) およびまたは第 2 処理ユニット (7 2) の特性を決定するステップ、

第 1 処理ユニット (7 1) の決定された特性およびまたは第 2 処理ユニット (7 2) の決定された特性に従ってそれぞれの関数メタ情報を更新するステップ、

その数値演算を実行するのに適したものとして第 1 処理ユニット (7 1) および第 2 処理ユニット (7 2) を決定するステップ、

実行時に、そして第 1 および第 2 の予想コストを計算する前に、第 1 のデータ構造の次元および第 1 データ構造内の記憶された要素の数のうちの少なくとも一つを決定するステップ、

そして実行時に、かつ第 1 および第 2 の予想コストを計算する前に、第 2 のデータ構造の実行時インスタンスの次元および第 2 のデータ構造の実行時インスタンス内の記憶された要素の数のうちの少なくとも一つ

を決定するステップ、

の少なくとも 1 つを含む方法。

10

20

30

40

50

【請求項 10】

請求項 1～9 のいずれか 1 つに記載の方法であって、さらに、
さらなる第 1 の計算カーネル（140—144）を用いて第 1 の処理ユニット（71）を
初期化し、さらなる第 2 の計算カーネル（150—154）を用いて第 2 の処理
ユニット（72）を初期化するステップ、

そこで、さらなる第 1 の計算カーネル（140—144）およびさらなる第 2 の計算カ
ーネル（150—154）は、第 1 のデータ構造（A）およびまたは一般データタイプあ
るいはもう一つの一般データタイプそして数値演算の出力を表す複数のデータ構造を格納
するさらなるデータ構造を受け取るように構成された引き続くプログラムセグメント（2
21）から導出されるさらなる数値演算を実行するように構成され、

そこで、上記引き続くプログラムセグメント（221）は、さらなる数値演算の出力の
サイズに関するデータを含むさらなる関数メタ情報、さらなる数値演算の出力の構造、お
よびまたはさらなる数値演算の出力を生成するための仕事量を含む、

およびまたは

さらなる関数メタ情報と、さらなるデータ構造のさらなる実行時インスタンスのデータ
メタ情報および第 1 のデータ構造（A）の実行時インスタンス（A1）のデータメタ情報
の少なくとも 1 つを用いて、

実行時インスタンス（A1）およびまたはさらなる実行時インスタンスに対してさらな
る数値演算を実行するために、さらなる第 1 の計算カーネル（140—144）を第 1 の
処理ユニット（71）上で実行する第 1 の予想コストを数值的に計算し、

およびまたは、実行時インスタンス（A1）およびまたはさらなる実行時インスタ
ンスに対してさらなる数値演算を実行するために、第 2 の計算カーネル（150—154）を
第 2 の処理ユニット（72）上で実行する第 2 の予想コストを数值的に計算する
ステップを含み、

そこでさらなる実行時インスタンスのデータメタ情報は、さらなる実行時インスタ
ンスの実行時サイズ情報およびまたはさらなる実行時インスタンスの実行時場所情報を含む

方法

【請求項 11】

請求項 10 に記載の方法であって、さらに

そこで前記数値演算は、第 1 のデータ構造（A）およびまたは第 2 のデータ構造に対す
る計算命令の縮小セットの命令であり、

およびまたは、そこで前記さらなる数値演算は、第 1 のデータ構造（A）およびまたは
さらなるデータ構造に対する計算命令の縮小セットの命令である

方法。

【請求項 12】

請求項 1～11 のいずれか 1 つに記載の方法であって、さらに

プログラムコード内の前記プログラムセグメント（220）を識別するステップ、
プログラムコード内の引き続くプログラムセグメント（221）を識別するステップ、
前記プログラムセグメント（220）から前記第 1 の計算カーネル（142）、前記第 2
の計算カーネル（152）、および関数メタ情報を含む実行時セグメント（32）を作成
するステップ、

引き続くプログラムセグメント（221）から、さらなる第 1 の計算カーネル、さらな
る第 2 の計算カーネル（152）、およびさらなる関数メタ情報を含む引き続く実行時セ
グメントを作成するステップ、

それぞれのプログラムセグメント（220、221）を中間表現、特にバイトコード表
現、または第 1 の処理ユニット（71）および第 2 の処理ユニット（72）上での実行を
円滑に進めるそれぞれの表現に翻訳するステップ、

決定された第 1 処理ユニット（71）の特性およびまたは第 2 処理ユニット（72）の
特性に従ってそれぞれの実行時セグメントを更新するステップ、そして

第 1 処理ユニット（71）および第 2 処理ユニット（72）での実行を容易にする中間

10

20

30

40

50

表現、バイトコード表現、およびそれぞれの表現のうちの少なくとも1つをそれぞれの計算カーネル（130－134、140－144、150－154）にコンパイルするステップ、

の少なくとも1つを含む方法。

【請求項13】

請求項1～12のいずれか1つに記載の方法であって、さらに前記プログラムセグメント（220）は前記異種計算システムのホストプロセッサ（73）上での実行を容易にする以下の表現、

異種計算システムのシングルコアCPU上での実行を容易にする表現、
 異種計算システムのマルチコアCPU上での実行を容易にする表現、
 異種計算システムのCPUベクトル拡張上での実行を容易にする表現、
 異種計算システムのGPU上での実行を容易にする表現、
 異種計算システムのSIMDアクセラレータ上での実行を容易にする表現、
 異種計算システムのDSP上での実行を容易にする表現、
 異種計算システムのマイクロプロセッサ上での実行を容易にする表現、
 異種計算システムのマイクロコントローラ上での実行を容易にする表現、
 異種計算システムの仮想デバイス上での実行を容易にする表現、
 異種計算システムのFPGA上での実行を容易にする表現、
 の少なくとも1つに翻訳されている方法。

【請求項14】

請求項1～13のいずれか1つに記載の方法であって、さらに次の複数のステップ、
 一般データ型またはその他の一般データタイプの複数の要素を格納する第1のデータ構造（A）およびまたは第2のデータ構造（B）を受け取るようにそれぞれ構成されたいくつかのプログラムセグメント（220－222）を含み、また数値演算の出力のサイズ、出力の構造、およびまたは出力を生成するための仕事量に関連するデータを含む関数メタ情報を含むシーケンス（210）をプログラムコード内で特定するステップ、

各実行時セグメント（30－34）がそれぞれの第1の計算カーネル、それぞれの第2の計算カーネル、および第1データ構造（A1）の実行時インスタンスおよびまたは第2データ構造の実行時インスタンスに対するそれぞれの数値演算を参照してそれぞれの関数メタ情報を含むように、各プログラムセグメント（220－222）からそれぞれの実行時セグメント（30－34）を作成するステップ、

各実行時セグメント（30－34）の第1および第2の計算カーネルを実行するためのそれぞれの第1および第2予想コストを計算するステップ、そして

それぞれの第1の予想コストとそれぞれの第2の予想コストを使用してそれぞれの計算カーネルが第1の処理ユニット（71）と第2の処理ユニット（72）のどちらで実行するかを決定するステップ、
 の少なくとも1つを含む方法。

【請求項15】

第1処理ユニット（71）および第2処理ユニット（72）を備えるコンピュータによって実行されるときに、前記請求項1～14のいずれかに記載の方法を上記コンピュータに実行させる命令を含むコンピュータ可読媒体。

【請求項16】

異種計算システムであって、

一般データ型の複数の要素を格納することができる第1のデータ構造（A）を受け取るように構成されたプログラムセグメントの実行時セグメント（30－34）を格納するメモリであって、前記プログラムセグメントは、数値演算の出力を生成する演算の仕事量と、前記出力のサイズに関するデータ及び前記出力の構造に関するデータの内の少なくとも1つと、を含む関数メタ情報を有する、メモリと、

少なくとも2つの異種の処理ユニット（71、72）と、

を含み、

そこで少なくとも2つの異なる処理ユニット（71、72）のそれぞれは、上記実行時セグメント（30-34）のそれぞれの計算カーネルを格納するメモリにアクセスしかつまたはその一部を形成し、各計算カーネルは数値演算を実行し、
そこで上記実行時セグメント（30-34）は、第1のデータ構造（A）の実行時インスタンス（A1）のデータメタ情報を決定するための実行可能コードを含み、そのデータメタ情報は、実行時インスタンス（A1）の実行時サイズ情報および実行時インスタンス（A1）の実行時場所情報を含み、

そこで実行時セグメント（30-34）は、関数メタ情報およびデータメタ情報を使用して、少なくとも2つの異なる処理ユニットのそれぞれについて、実行時インスタンス（A1）を用いて数値演算を実行するためのそれぞれの計算カーネル（130-134、140-144、150-154）を実行するそれぞれの予想コストを計算する実行可能コードを含み、そして

そこで実行時セグメント（30-34）は、それぞれの計算カーネルを実行するために少なくとも2つの異なる処理ユニットのうちの1つを選択するための実行可能コードを含み、少なくとも2つの処理ユニットのうちの選択された1つの計算された期待コストは決定された予想コストの最低値に対応する

システム。

【請求項17】

請求項16の異種計算システムであって、さらに

上記異種計算システムは、少なくとも2つの異なる処理ユニット（71、72）と結合されたベースプロセッサを備え

そこで上記ベースプロセッサは、少なくとも2つの異なる処理ユニット（71、72）のうちの少なくとも1つにそれぞれの計算カーネルをロードするように構成され、

そこで異種計算システムは異種コンピュータであり、

そこでベースプロセッサは異種コンピュータのホストコントローラのホストプロセッサであり、

ベースプロセッサは、少なくとも2つの異なる処理ユニットのうちの1つを形成し、

少なくとも2つの異なる処理ユニットのうちの少なくとも1つは、シングルコアCPU、マルチコアCPU、CPUベクトル拡張、GPU、SIMDアクセラレータ、DSP、マイクロプロセッサ、マイクロコントローラ、およびFPGAからなる群から選択され、
そしてまたは

そこで少なくとも2つの異なる処理ユニットのうちの少なくとも1つは、少なくとも1つのシングルコアCPU、少なくとも1つのマルチコアCPU、少なくとも1つのCPUベクトル拡張、少なくとも1つのGPU、少なくとも1つのSIMDアクセラレータ、少なくとも1つのDSP、少なくとも1つのマイクロプロセッサ、少なくとも1つのマイクロコントローラ、およびまたは少なくとも1つのFPGAを含む

システム。

【請求項18】

請求項16または17に記載の異種計算システムであって、

そこで前記データメタ情報は、実行時インスタンス（A1）の実行時同期情報、および実行時インスタンス（A1）の実行時タイプ情報の少なくとも1つを含み、

そしてまたは

そこで異種計算システムは、請求項1から10のいずれかに記載の方法を実行するように形成される

システム。

【発明の詳細な説明】

【技術分野】

【0001】

本発明の実施は、コンピュータ実装方法に関し、特に、異種計算システム上で数値アルゴリズムを実行するためのコンピュータ実装方法、異種計算システム、およびコンピュータ

10

20

30

40

50

可読媒体に関する。

【背景技術】

【0002】

特に数値計算の分野において、およびデータ分析のために、異種計算資源の効率的な利用に対する高い需要が存在する。これは、しばしば面倒で費用がかかるプログラムを並列化するタスクを含み、詳細なレベルのコンピュータアーキテクチャおよびそれに関連するプログラミング技術に関するかなりの知識を必要とする。

【0003】

出発点は伝統的な、C / C++、C#、FORTRAN、パスカルおよびジャバなどの一般的な汎用プログラミング言語（GPL）のセットであった。そのような言語は一般的に基本的なデータ要素としてのスカラーデータと任意の制御命令を扱って、低い粒度のデータを処理している。シングルプロセッサ実行のためのコンパイラ技術の改善に成功しているが、異種計算システム用の（GPL）プログラムのコンパイルは依然として挑戦的である。

【0004】

例えば、コンパイラは、どの処理ユニットを対象とするか、どのように処理ユニットにデータを処理するように指示するか、およびどのようにデータを処理ユニットに提供するかという、ある種の決定を下さなければならない。課題には、多様な（急速に変化する）コンピューティングハードウェア、任意のGPL言語要素および命令グラフの複雑さ、処理されるデータの動的性質、そしてすべての制約情報が最終的に利用可能になる時期が含まれる。さらに、コンパイラはしばしばプログラマからの重要な意思決定支援を必要とする。その結果、多くのソフトウェアプログラムの並列化の可能性、例えば、ほとんどのGPLベースのプログラムは、通常、それぞれ部分的にしかアクセスできず、それぞれの異種システムに部分的にのみ効率的である。さらに、導入されたオーバーヘッドのために、これまで使用されてきた自動並列化アプローチは、それぞれ、比較的大きなデータサイズおよび大きなコードセグメントに対してのみ価値があるかもしれない。

【0005】

したがって、既存のシステムは、「トップダウン」アプローチに従って、より速い計算のためにより大きなプログラムの部分を加速デバイスに分配しようと試み、それらの部分を最大化しようと試みる。課題として、潜在的な実行パスの複雑さ（パラメータ空間のサイズ）、データの変動、および実行中のシステム状態の複雑さ（多様性、予測不可能性）が含まれる。

【0006】

したがって、異種コンピューティングアーキテクチャ用のソフトウェアプログラムの並列化を改善する必要がある。

この出願の発明に関連する先行技術文献情報としては、以下のものがある（国際出願日以降国際段階で引用された文献及び他国に国内移行した際に引用された文献を含む）。

（先行技術文献）

（特許文献）

（特許文献1） 国際公開第2017/016590号

（特許文献2） 米国特許出願公開第2011/0173155号明細書

（特許文献3） 米国特許出願公開第2007/0283358号明細書

（非特許文献）

（非特許文献1） Philchaya Mangpo Phothilimthana, et al. "Portable Performance on Heterogeneous Architectures", In: Proceedings of the ASPLOS '13, Houston, Texas, USA, March 16-20, 2013, S. 1-13.

（非特許文献2） J. Ansel, et al.: PetaBricks: "A Language and Compiler for Algorithmic Choi

10

20

30

40

50

ce. In: Proceedings of the PLDI'09, Dublin, Ireland, June 15-20, 2009, S. 38-49.

【発明の概要】

【課題を解決するための手段】

【0007】

コンピュータ実装方法の一実施形態によれば、この方法は、第1計算カーネルを用いて異種コンピューティングシステムの第1処理ユニットを初期化し、第2計算カーネルを用いて異種コンピューティングシステムの第2処理ユニットを初期化することを含む。第1の計算カーネルと第2の計算カーネルの両方は、一般的データ型の複数の要素を格納する第1のデータ構造を受け取り、プログラムセグメントから導出された数値演算を実行するように構成されている。プログラムセグメントは、数値演算の出力のサイズ、出力の構造、およびまたは出力を生成するためのエフォートに関するデータを含む関数メタ情報を含む。第1のデータ構造の実行時インスタンスの関数メタ情報およびデータメタ情報は、第1の処理ユニット上で第1のカーネルに対し実行時インスタンスで数値演算を実行するための第1の期待コストを計算するために使用される。また同様に第2の処理ユニット上で第2のカーネルに対し実行時インスタンスで数値演算を実行するための第2の期待コストを計算するために使用される。データメタ情報は、実行時インスタンスのサイズ情報および実行時インスタンスの場所情報を含んでいる。本方法は、第1の予想コストが第2の予想コスト以下である場合に実行時インスタンスに対して数値演算を実行するために第1の処理ユニット上で第1の計算カーネルを実行することと、第1の予想コストが第2の予想コストよりも高い場合に実行時インスタンスに対して第2の処理ユニット上で第2の計算カーネルを実行することを含んでいる。データメタ情報は、さらに実行時インスタンスの実行時同期情報およびまたは実行時インスタンスの実行時タイプ情報を含み得る。

【0008】

本明細書の文脈において、「処理ユニット」という用語は、ハードウェア装置、特に典型的には特定の計算カーネルに格納されている命令に従って演算およびまたは計算を実行することができるプロセッサを記述するものである。

【0009】

処理装置は、シングルコアCPU（中央処理装置）、マルチコアCPU、ベクトル拡張を有するCPU（SSE / AVXなど）、GPU（グラフィックス処理装置）、少なくとも1つのGPUを統合したCPUであってもよい。数値計算をサポートするコプロセッサ、SIMDアクセラレータ（単一命令、多重データアクセラレータ）、マイクロコントローラ、DSP（デジタルシグナルプロセッサ）などのマイクロプロセッサ、およびFPGA（フィールドプログラマブルゲートアレイ）またはこれらの1つ以上の組み合わせであり、典型的なものはいわゆる仮想プロセッサを形成する。

【0010】

本明細書の文脈において、「出力を生成するためのエフォート」という用語は、出力を生成するための数値的なエフォート（純粋に計算コスト）を記述することを意図している。それぞれの関数メタ情報は、1つの数値的仕事量係数またはさらには数個の数値的仕事量係数を含むことができる。実行前に、関数メタ情報はハードウェア特性（処理ユニットの特性）に従って更新されてもよい。本明細書の文脈では、「予想コスト」という用語は、実行時に所与の処理ユニット上で実行時インスタンスを用いて数値演算を実行するための予想全体コストを表すことを意図している。予想コストは通常、実行時における与えられた処理ユニット上の計算コストを指し、（処理ユニットがまだ実行時インスタンスにアクセスしていないときの）転送コストを含み得る。予想コストは、関数メタ情報、すなわち数値エフォート係数、およびデータメタ情報、特に実行時インスタンスの実行時サイズ情報および実行時インスタンスの実行時位置情報、を使用して計算することができる。予想コストは通常、実行時における所与の処理ユニット上の計算コストを指し、（処理ユニットがまだ実行時インスタンスにアクセスしていないときの）転送コストを含み得る。期待コストは、関数メタ情報、例えば数値仕事量係数、データメタ情報、特に実行時インス

タンスの実行時サイズ情報および実行時インスタンスの実行時位置情報、を使用して計算することができる。本明細書の文脈において、用語「計算カーネル」は、実行時に一般データ型の複数の要素を格納することができるデータ構造の実行時インスタンスを用いて特定の数値演算を処理ユニットで実行することができるコンパイル済みソフトウェアコードを表すことを意図する。

【0011】

第1のデータ構造およびまたは第1のデータ構造の実行時インスタンスは、整数、浮動小数点数、複素数などの可変数個のそれぞれの一般データ型を格納することが可能であり得る。

【0012】

第1のデータ構造および／または第1のデータ構造の実行時インスタンスは、典型的にはn次元直線状データ構造である。

【0013】

第1のデータ構造および／または第1のデータ構造の実行時インスタンスは、(.NETフレームワークの意味での) 集合、ベクトル、配列、リスト、キュー、スタック、グラフ、ツリーやそのような構造へのポインタであり得る。簡単にするために、データ構造は多次元配列(ベクトル、行列またはより高次元の数学的データ)を表すと仮定され、数値演算は、特記しない限り詳細な説明では配列演算と仮定される。以下では、直線配列と行列という用語は同義語として使用される。直線配列は、科学的または技術的なアルゴリズムでよく使用される。例えば、それぞれ行および列に配置された表示画面またはカメラ画像センサの画素要素は、対応する2D／または3D直線配列に格納されてもよい。

【0014】

典型的には、第1のデータ構造の実行時インスタンスは、測定データ、画像、音声データ、シミュレーションデータ、制御データ、状態データなど、中間結果と共にプログラム全体にわたって処理されている様々な完成段階における技術情報を表すデータを格納する。

【0015】

コストは、数値演算を実行する前(例えば、数値演算を実行する直前)の実行時に決定され、関数メタデータ(関数メタ情報)とその第1のデータ構造の実行時インスタンスのデータメタ情報の両方を考慮に入れるので、数値演算は、この目的に最も適した特定の(実行時)構成にある処理ユニットによって実行することができる。他の方法と比較して、これはより低いオーバーヘッドで達成することができる。その結果、異種コンピューティングシステムは異なるサイズのデータ構造を他の手法と比較して、(平均して)より効率的に処理することができる。

【0016】

例えば、行列Aの全要素の二乗和 $\text{sum}(A^2)$ を計算する数値演算を実行するコストは、Aのサイズの関数として表すことができる。複数回実行される場合、個々の呼び出しは個々のAのサイズを関連付けることができる。通常、Aが十分に大きい場合、特にGPUでサポートされている要素タイプの場合、操作の $\text{sum}(A^2)$ はGPUで効率的に実行される。ただし、Aに含まれる要素がごくわずか、または1つのみの場合は、CPUが適している可能性がある。特に、GPUではなくCPUが既に操作を実行する時点で行列Aに直接アクセスしている場合、すぐに実行される。

【0017】

したがって、予想されるコストは、通常、数値演算、実行時インスタンスのサイズ、データ構造の実行時インスタンスの格納場所、および利用可能な処理ユニット(CPU、GPU、...)の(計算上の)特性に依存する。

【0018】

この理由で、伝統的なコンパイラは実行時に利用可能なハードウェアの広範な多様性(CPU、マルチコアCPU、CPUベクトル拡張、SIMDアクセラータ、GPUデバイス、FPGA、DSP、複数メモリモデル)一般目的の言語と命令グラフ、処理すべきデータの多様性(データの大きさ、同じコードセグメントの複数呼び出しの間での場所の変

10

20

30

40

50

化)の全てを考慮に入れることができない。すべての制約情報が最終的に利用可能になるときは重要な実行時サポートを必須にする。静的コンパイルと動的コンパイル(実行時)の両方とも、実際の数値演算に大きなオーバーヘッドを追加する。これは、結果として得られるプログラムを比較的小さい入力データ、さらには中サイズの入力データ、ひいてはさまざまなサイズおよび／または構造の入力データに対して非効率的にする(遅い)傾向がある。

【0019】

同様に、伝統的にMathworksのMATLAB、ILNumericsおよびNumPyなどのn次元配列構造に基づく数値アルゴリズムの定式化を可能にする技術計算で使用される特殊言語(ドメイン固有言語、DSL)のコンパイルは、十分に大きい入力データ用にのみ優れた実行性能を示す傾向がある。しかしながら、(十分に大きい)実行ユニットのプログラム部分をに手で割り当てるという負担は、開発時にプログラマに委任される。明らかに、このアプローチは(プログラムを導入せずに)処理ハードウェア変更をすることも、小さなプログラムセグメントおよび変化する(小さな)入力データサイズのために異種コンピューティングリソースを効率的に利用することもできない。

【0020】

それとは異なり、本明細書に記載の手法は、特定の適切な時期に、すなわちコンパイル時に、起動時に、および実行時に所望の(特に情報を収集し、決定を下す)アクションを実行することを可能にする。実行経路に関する決定は、特に実行時に、したがってデータ構造の実行時インスタンスおよびハードウェア(処理装置)すなわち現在のハードウェア負荷の両方に対して所望の情報がすべて利用可能になったときに、その時にのみ、行われる。これにより、オーバーヘッドを削減し、さまざまなデータサイズやハードウェア構成に自動的に適応することができる。

【0021】

さらに、異なる試行データが異なるサイズクラスに分類される所与のハードウェア上での所与の数値演算の実行時間(典型的には数時間)を決定するための時間のかかるシミュレーションは、ワークロードデータのサイズクラスに基づいて実行経路を実行時に決定する前のものであり、必須ではない。そのようなアプローチはいくつかの大きなデータといくつかの(複雑度の)低いアルゴリズムには効果的かもしれないが、(多次元)サイズ等級付けはさらに長いシミュレーション時間を避けるために正確でなく最善の実行パスを保証できない。さらに、このアプローチでは、ハードウェア負荷も実行時のワークロードデータの実際の格納場所も考慮に入れることができない。さらにまた、ユーザがプログラムコードを修正したり処理ハードウェアが変更されたりした場合には、このアプローチでは通常まったく新しいシミュレーションが必要とされる。

【0022】

このようなアプローチとは反対に、本明細書に記載のアプローチはまた、任意の複雑性のアルゴリズムまたはプログラム全体にさえもスケールアップすることができる。実行時におけるいくつかの後続のプログラムセグメントおよび実行時データに対する最適実行経路を予測するために、事前のシミュレーションまたは最適化パスは必要とされない。前述のような前もってのシミュレーションまたは最適化は一般に非常に時間および／またはエネルギーを消費するが、この方法で見いだされる解決策の質(実行経路構成)は限られている。また一方では、次のセグメントの数は、シミュレーション／最適化のために(パラメータスペースのかなり小さいサイズのために)小さく保たなければならない、それは粒度の荒いセグメンテーションをもたらすとして／または全体のプログラムサイズを制限する。一方、ワークロードのデータ値が変更されたために実行パスが変更された場合の対処には、きめの細かいセグメンテーションが必要になることがよくある。さらに、最適化によって見つけれられた解は、通常、極小値のみをマークする。異質コンピューティングシステム上のアルゴリズムに対する最適実行経路(大域的最適)を見つけるために、実際の作業負荷データを含めて、すべての依存パラメータが考慮されるべきである。そのため、大域的な最適化はしばしば実現不可能である。

【0023】

本明細書に記載の方法およびシステムは、ボトムアップアプローチに基づいている。より複雑なアルゴリズムの基本的な構成要素として基本数値演算から始めて、最良の実行オプションが実行時に直接評価される。ここでは、データと命令の両方の粒度が重要な役割を果たし、最高の実行パフォーマンスを得るために考慮される。

【0024】

特に、絡み合ったシステムの「プロセッサおよび命令」は、束縛されず、スカラーデータの代わりに配列を処理することができる機能（命令）によって置き換えられ、かつ抽象的である、すなわち具体的なプロセッサに割り当てられず、実行前にコスト関連情報（関数メタデータ）を提供することができる。単一のメインプロセッサでの焦点は、適切な実行装置の全体性に置き換わる。

【0025】

機能／命令の複雑さは、動的データサイズを処理する能力、ならびに関連するコストについて事前に知らせる能力によって増大する。作業負荷（配列機能／命令＋動的データ）の最適な処理ユニット／装置への割り当ては、実行時に、通常は実行直前に行われ、したがって現在のシステム全体状態を考慮に入れることができる。

【0026】

従って、小さな入力データ、中サイズの入力データ、および大きな入力データは、プログラムセグメントの単一のコンパイルされた表現によってオーバーヘッドをほとんど伴わずに、それぞれの最適な実行ユニットによって効率的に処理されることができる。

【0027】

例えば、ユーザ生成プログラムコードは、コンパイル時にプログラムコード内でそれぞれの（既知の）数値演算を含むプログラムセグメントを識別するために解析されてもよい。さらに、それぞれの数値演算の関数メタ情報を決定し、それぞれの第1の計算カーネル、それぞれの第2の計算カーネル、およびそれぞれの関数メタ情報を含むそれぞれの実行時セグメントをコンパイル時に形成することができる。

【0028】

プログラムコードは、プログラムシーケンスを形成するいくつかの典型的には後続のプログラムセグメントを含むことができる。

【0029】

本明細書で使用される「シーケンス」という用語は、ユーザプログラムまたはプログラムセグメントを含むユーザプログラムのセクションを表現することを意図している。

【0030】

本明細書で使用される「プログラムセグメント」という用語は、それぞれの一般データタイプの複数の要素を記憶することができる1つまたは複数のデータ構造の入力および／または出力引数を処理するための抽象レベルの命令を指すソフトウェアコードを表現することを意図する。「プログラムセグメント」は、コンパイラによって識別可能な、アレイ関数、ステートメント、演算子、および／またはそれらの組み合わせなどの単純な言語エンティティを含むことができる。たとえば、行列AとBを含む式 $\sin(A) + \text{linsolve}(B + 3 * \cos(A))$ は、セグメントになる。あるいは $\sin(A)$ と $\text{linsolve}(B + 3 * \cos(A))$ がそれぞれセグメントであり得る。

【0031】

本明細書で使用される用語「実行時セグメント」は、プログラムセグメントのコンパイルされた表現を記述することを意図している。コンパイルされた表現は、異種コンピューティングシステム上で実行可能であり、通常、異種コンピューティングシステムの各処理ユニットに対するそれぞれの計算カーネルと、データ構造の実行時インスタンスに対する予想コストを計算するためと、その計算カーネルを実行するための処理ユニットを選択するため、および／または予想コストを使用して2つ以上の処理ユニットにわたって作業負荷を分散するための制御コードを含む。さらに、処理ユニットの選択および／または作業負荷の分配のために、処理ユニットの現在の作業負荷および／またはデータの場所などの

10

20

30

40

50

さらなる情報を考慮に入れることができる。

【0032】

実行時セグメントは、コンパイラによって作成、マージ、および／または修正、さらには最適化され、実行時にインスタンス化および更新されてもよい。

【0033】

関数メタデータ（関数メタ情報）は、通常、実行時セグメントに格納される。多くの場合、関数メタ情報はプログラムセグメントと共に格納され、そこから引き出される。コンパイラは、セグメントがマージされ、修正され、および／または最適化されたとき、および実行時に具体的なデバイス機能が知られているときに、情報を更新および維持することがある。

【0034】

プログラムコード内の数値演算は、一般データタイプの複数の要素を格納するデータ構造のための計算命令の縮小セットの項目に対応し得る。これは、計算カーネルおよび／または関数メタ情報の効率的な生成および／または実行を容易にすることができる。

【0035】

しかしながら、プログラムコードはまた、コンパイラによって識別可能な数値演算なしの中間部分を含んでもよい。これらの部分は伝統的な方法で扱われるかもしれない。

【0036】

プログラムセグメントは、最初に中間表現、特にバイトコード表現、または処理ユニット上での実行を容易にするそれぞれの表現に変換することができ、例えば LLVM（低レベルバーチャルマシン）や Roslyn などのコンパイラインフラストラクチャを使用する。

【0037】

後に、すなわち特定の異種コンピューティングシステムの起動時に、中間表現はバイトコードおよび／または機械命令を含む直接実行可能な計算カーネルにコンパイルされてもよい。

【0038】

起動時に、特定の異種コンピューティングシステムの適切な処理ユニットを決定することができ、実行時セグメントは決定された処理ユニットに従って更新することができる。これは、計算カーネル、計算カーネルの中間表現、および／または関数メタ情報を更新することを含み得る。例えば、CPUカーネルテンプレートおよびGPUカーネルテンプレートは、実際のデバイスの特性に従って更新され得る。より具体的には、実際の処理ユニット上で特定の数値演算を実行するための関数メタ情報および／または計算カーネルの中間表現内のベクトルデータ型の長さおよび／または型における数値エフォート係数を、テンプレート内で更新することができる。

【0039】

さらに、処理ユニットはそれぞれの計算カーネルで初期化されてもよい。

【0040】

初期化の後、データ構造の実行時インスタンスの関数メタ情報およびデータメタ情報は、それぞれの処理ユニット上で計算カーネルを実行するそれぞれの予想コストを実行時に決定するために使用される。

【0041】

一実施形態では、遅延実行方式を実施することができる。これは、初期化の結果が最初の実行に必要とされるときまで潜在的に時間のかかる初期化タスクを遅らせることを含む。例えば、特定の処理ユニットに対する特定の実行時セグメントの計算カーネルは、対応する実行時セグメントが対応する処理ユニット上で対応する計算カーネルを実行することが決定されたときに初めて処理ユニットに割り当てられ初期化される。

【0042】

通常、予想コストを決定するために、計算コストと転送コスト（実行時インスタンスの別の処理ユニットへの転送に関連して発生する可能性があるコストを指す）の両方が考慮される。

10

20

30

40

50

【0043】

さらに、予想コストを決定するために多項式コスト関数を使用することができる。

【0044】

特に、双一次費用関数は、多くの場合、予想費用を決定するのに適していることが分かっている。多くの単純な数値演算では、すべての要素の関数結果を個別に計算するなど、格納されている要素の数のみにまたは実質的に依存する数値計算（計算コスト）が発生する。したがって、予想される計算コストを計算するために、要素の数が処理ユニットの並列計算能力を表すそれぞれの数値エフォート係数と単に掛け合わされてもよい。同様に、予想転送コストは、コピーされる要素の数に転送係数（または処理ユニットが記憶された要素に直接アクセスする場合はゼロ）を掛け、予想計算コストに加算することによって現在の状況下で適切な各処理ユニットに対してそれぞれの数値演算を実行するためのコストを計算できる。

10

【0045】

行列の反転、行列・ベクトル積の計算、連立方程式の解法、行列の固有値や固有ベクトルの決定（eig）、行列の高速フーリエ変換（fft）の計算など、より複雑な数値演算では、予想される計算コストはまた、数値演算の出力のサイズ、出力の構造もしくは形状、または値、構造、および／または入力の実行時インスタンスの形状にも依存し得る。例えば、入力データの特性（疎行列対密集行列、単一対2倍精度浮動小数点数、行列対称性、サイズなど）に応じて、異なる数値アルゴリズムを使用することができる。数値演算の出力は、プログラムセグメントのメタデータからコンパイラによって導出することができる。

20

【0046】

このアプローチは、以前に使用された実行コストによる決定のアプローチの制限を克服する。

【0047】

例えば、プラットフォーム／処理ユニット上で実行可能な中間表現またはバイナリ／オブジェクトコードで実行可能命令の数および種類を「コスト」の尺度として使用することは、プログラムセグメント内のプログラムフローを考慮に入れる必要があるので特に困難である。特に、セグメントに条件付き命令が含まれている場合（これは、ループ構造が関係している場合によくある）、正確な分析は少なくとも面倒であり、したがって費用がかかる。さらに、そのような条件式の値がプログラムセグメントによって処理される入力データのサイズに動的に依存している場合、このようなデータサイズがわかっている実行時にこの分析を実行する必要がある。問題は、ほとんどの試みが動的セグメントサイズを処理しようと試みるという事実によって増大し、これは、より良い効率のためにさまざまな数の命令をマージすることから得られる。それぞれのマージされたセグメントは、やはり実行時に実行されなければならない各ターゲットプラットフォーム用の特定のコンパイラで再びコンパイルする必要がある。

30

【0048】

さらに、プログラムセグメントを実行するのに必要な実際の時間が「コスト」の尺度として使用される場合、どこでコードを実行するかを決定するための大きなオーバーヘッドが必要とされる。さらに、信頼性のあるコストモデルを導出するために、異なるサイズおよび／または形状の入力データを用いた多数のランが必要とされる可能性がある。例えば、実際の計算のための信頼できるコストモデルを構築することを可能にするために、入力配列の次元および各次元における配列の長さの両方ならびに要素のデータ型を処理ユニット上で変化させる必要があるかもしれない。したがって、このアプローチは、小さい入力データサイズに対して動的に実行される場合、本質的に非効率的である。

40

【0049】

さらに、プログラムセグメントは、一般データタイプまたは別の一般データタイプの複数の要素を格納することができる第2のデータ構造を受け取るように構成されていることもある。

【0050】

50

例えば、行列-ベクトル積を計算するためのプログラムセグメントは、行列およびベクトルを受け取るように構成される。同様に、2つの行列を加算するためのプログラムセグメントは、2つの行列を受信するように構成される。

【0051】

コンピュータ可読媒体の一実施形態によれば、コンピュータ可読媒体は、第1の処理ユニットおよび第2の処理ユニットを備えるコンピュータによって実行されると、本明細書に記載の方法をコンピュータに実行させる命令を含む。

【0052】

コンピュータは、通常、CPUおよびGPUなどの2つ（すなわち第1および第2の処理装置）の異なる処理装置またはより多くの異なる処理装置を有する異種コンピュータである。典型的には、異なる処理ユニットは、典型的には異なる物理的性質に起因して、計算上の性質および／または能力に関して互いに異なる。しかしながら、2つの異なる処理ユニットはまた、同じ計算上の特性および／または能力を有することがあるが、プログラムセグメントが実行されるときには異なって利用される（異なる作業負荷を有する）。

【0053】

コンピュータ可読媒体は通常、非一時的記憶媒体である。

【0054】

異種コンピューティングシステムの一実施形態によれば、異種コンピューティングシステムは、少なくとも2つの異なる処理ユニットと、一般データタイプの複数の要素を格納することができる第1のデータ構造を受け取るように構成されるプログラムセグメントの実行時セグメントを格納するメモリとを含む。プログラムセグメントは、プログラムセグメントの数値演算の出力のサイズ、出力の構造、および／または出力を生成するための数値的仕事量に関連するデータを含む関数メタ情報を（コンパイラによって）含み、提供し、および／または可能にする。少なくとも2つの異なる処理装置のそれぞれは、少なくとも2つの処理装置のそれぞれについて実行時セグメントのそれぞれの計算カーネルを格納するメモリの少なくとも一部にアクセスし、かつ／またはそれを形成する。各計算カーネルは数値演算を実行する。実行時セグメントは、第1のデータ構造の実行時インスタンスのデータメタ情報を決定するための実行可能コードを含むかまたは参照する。データメタ情報は、実行時インスタンスの実行時サイズ情報、実行時インスタンスの実行時位置情報、実行時インスタンスの実行時同期情報、および実行時インスタンスの実行時タイプ情報のうちの少なくとも1つを含む。実行時セグメントはさらに、関数メタ情報およびデータメタ情報を使用して、少なくとも2つの処理ユニットのそれぞれについて、それぞれの計算カーネルを実行して実行時インスタンスの数値演算を実行するそれぞれの予想コストを計算するように構成された実行可能コードを含むか参照する。実行時セグメントはさらに、少なくとも2つの処理ユニットのうちの選択された1つの決定された予想コストが、それぞれの計算カーネルを実行するための少なくとも2つの異なる処理ユニットのうちの1つを最小値に対応するように選択するための実行可能コードを含む。

【0055】

一連の実行時セグメントはメモリに格納することができる。

【0056】

予想コストを計算するための実行可能コードと、少なくとも2つの異なる処理ユニットのうちの1つを選択するための実行可能コードとは、実行時セグメントの制御カーネルに含まれてもよい。

【0057】

典型的には、異種コンピューティングシステムは、処理ユニットを形成することができ、かつ少なくとも2つの処理ユニットのうちの第1の処理ユニットおよび第2の処理ユニットと結合されたベースプロセッサを含む。基本プロセッサは、通常、特に処理ユニットの特性を決定すること、数値演算を実行するのに適した異種コンピューティングシステムの処理ユニットを決定すること、決定された値に従って関数メタ情報を更新すること、それぞれの計算カーネルを処理ユニットにロードするための（実行時に）予想されるコスト

10

20

30

40

50

を（数値的に）決定するために、CPUのコアおよびGPUの陰影を付けるシェーダの数および／または計算特性などの処理ユニットの決定された特性に従って関数メタ浄法を更新すること、それらの処理ユニットにそれぞれ軽さカーネルをロードすること、（実行時の）予想コストを（数値的に）決定すること、決定された予想コストに基づいて、（望ましい）処理ユニットを選択すること、および／または選択された処理装置上でそれぞれのカーネルの実行を初期化すること、の制御機能を行うよう形成される。

【0058】

例えば、ベースプロセッサは制御カーネルをホストし実行することができる。以下では、ベースプロセッサは制御プロセッサとも呼ばれる。他の実施形態では、制御機能のうちの1つ、それ以上、または全てさえもが、処理ユニットのうちの1つによって実行され得る。制御機能は、処理ユニットとベースプロセッサとの間に分散させることもできる。例えば、制御カーネルは、異なるハードウェアユニット上で実行されるサブカーネルを有することができる。

10

【0059】

さらに、ベースプロセッサは、処理能力が異種コンピューティングシステムの（他の）処理装置と比較して低い場合であっても、処理装置のうちの1つであり、それぞれ処理装置のうちの1つとして機能し得る。例えば、ベースプロセッサは、（第1の）データ構造の実行時インスタンスに格納されている要素の数が比較的少ないときに数値演算を実行することができる。

【0060】

20

各処理ユニットは、シングルコアCPU（中央処理装置）、マルチコアCPU、ベクトル拡張を有するCPU（SSE / AVXなど）、GPU（グラフィックス処理装置）、少なくとも1つのGPUを統合したCPU、数値計算をサポートするコプロセッサ、SIMDアクセラレータ（単一命令多重データアクセラレータ）、マイクロプロセッサ、DSP（デジタル信号プロセッサ）などのマイクロコントローラ、およびFPGA（フィールドプログラマブルゲートアレイ）、典型的にはいわゆる仮想プロセッサを形成するこれらの装置のうちの1つ以上の組み合わせ、などによって構成することができる。

【0061】

異種コンピューティングシステムは、通常ベースプロセッサを形成するホストプロセッサを有するホストコントローラを含む異種コンピュータである。

30

【0062】

一例では、異種コンピュータは、CPUと、それぞれがそれぞれの処理ユニットを形成する1つまたは複数のGPUとをさらに含む。

【0063】

異種コンピューティングシステムは、ワークステーション、すなわち、技術的、医学的および／または科学的用途および／または数値的および／または配列集約的な計算用に特別に設計されたコンピュータであってもよい。例えば、ワークステーションは、測定データ、シミュレーションデータおよび／または画像、特にシミュレーション結果、科学データ、科学画像および／または医用画像を分析するために使用される。

【0064】

40

しかしながら、異種コンピューティングシステムは、それぞれが処理ユニットとして機能するいくつかの相互接続されたノードを有するグリッドまたはクラウドコンピューティングシステムとして、または組み込みコンピューティングシステムとしてさえも実装され得る。

【0065】

当業者は、以下の詳細な説明を読み、添付の図面を見れば、さらなる特徴および利点を認識するであろう。

【図面の簡単な説明】

【0066】

図中の構成要素は必ずしも縮尺通りではなく、代わりに本発明の原理を説明することに

50

重点が置かれている。さらに、図中、参照番号は対応する部分を示す。以下図面において、

【図 1】図 1 は一実施形態によるコンピュータ実装方法を示す図である。

【図 2】図 2 は一実施形態によるコンピュータ実装方法を示す図である。

【図 3】図 3 は実施形態によるコンピュータ実施方法を示す図である。

【図 4】図 4 は一実施形態によるコンピュータ実装方法および異種コンピューティングシステムの方法のステップを示す図である。

【図 5】図 5 は一実施形態によるコンピュータ実装方法のステップを示す図である。

【図 6】図 6 は一実施形態によるコンピュータ実装方法のステップを示す図である。

【図 7】図 7 は一実施形態によるコンピュータ実装方法のステップを示す図である。

【図 8】図 8 は一実施形態によるコンピュータ実装方法のステップを示す図である。

【図 9】図 9 は一実施形態によるコンピュータ実装方法のステップを示す図である。

【図 10】図 10 は一実施形態によるコンピュータ実装方法のステップを示す図である。

【発明を実施するための形態】

【0067】

以下の詳細な説明では、本明細書の一部を形成し、本発明を実施することができる特定の実施形態を例示として示す添付の図面を参照する。この点に関して、「上」、「下」、「前」、「背」、「先」、「後」などの方向の用語は、説明した図の向きに関して使用される。実施形態の構成要素は多数の異なる向きに配置することができるので、方向を表す用語は例示の目的で使用されており、決して限定的なものではない。本発明の範囲から逸脱することなく、他の実施形態を利用することができ、構造的または論理的な変更を加えることができることを理解されたい。したがって、以下の詳細な説明は限定的な意味で解釈されるべきではなく、本発明の範囲は添付の特許請求の範囲によって定義される。

【0068】

ここで様々な実施形態を詳細に参照するが、その 1 つまたは複数の例が図面に示されている。各実施例は説明のために提供され、本発明を限定することを意味しない。例えば、一実施形態の一部として図示または説明された特徴は、さらに他の実施形態を生み出すために他の実施形態上でまたは他の実施形態と共に使用され得る。本発明はそのような修正および変更を含むことを意図している。実施例は添付の特許請求の範囲を限定するものとして解釈されるべきではない特定の言語を用いて説明される。図面は縮尺通りではなく、例示目的のみのためのものである。明確にするために、特に明記しない限り、同じ要素または製造ステップは異なる図面中で同じ参照符号によって示されている。

【0069】

図 1 に関して、コンピュータ実施方法 1000 の方法ステップが説明される。

【0070】

起動フェーズ II のブロック 1300 において、異種コンピューティングシステムの処理ユニットは、一般データ型の複数の要素を格納できる実行可能な第 1 のデータ構造を受け取るように構成された（より高いレベルの）プログラムセグメントからコンパイルされた実行時セグメントのそれぞれの計算カーネルで初期化される。これは、計算カーネルをそれぞれの処理ユニットのメモリまたは異種計算システムの共有メモリにコピーすること、および／またはそれぞれの処理ユニットに計算カーネルを割り当てることを含むことができる。

【0071】

各計算カーネルは、プログラムセグメントから導出された数値演算を実行するように構成され得る。

【0072】

実行時セグメントは、通常、数値演算の出力のサイズ、出力の構造、およびまたは出力を生成するためのエフォートに関連するおよびまたはそのデータのサイズを表すデータを含む関数メタ情報を含むか、またはそれへのアクセスを有する。

【0073】

実行時フェーズⅠⅠⅠの次のブロック1500では、第1のデータ構造の実行時インスタンスの関数メタ情報およびデータメタ情報を使用して、典型的には処理ユニット上の計算カーネルを実行する上でそれぞれの予想コストを計算する。データメタ情報は、実行時インスタンスの実行時サイズ情報、実行時インスタンスの実行時（ストレージ）位置情報、および／または実行時インスタンスの実行時タイプ情報を含み得る。

【0074】

この目的のために、処理ユニットのうちの1つのメモリまたは異種コンピューティングシステムの共有メモリに格納されたデータ構造の実行時インスタンスは、実行時セグメントの制御コードによってアクセスされてもよい。

【0075】

実行時フェーズⅠⅠⅠの次のブロック1600では、予想コストを使用して、実行時インスタンスで数値演算を実行するためにそれぞれの計算カーネルがどの処理ユニット上で実行されるかを決定することができる。

【0076】

その後、選択された処理ユニットがまだ実行時インスタンスにアクセスしていない場合、実行時インスタンスは実行のために選択された処理ユニットにコピーされてもよい。

【0077】

その後、計算カーネルは、選択された処理ユニット上で実行されてもよい。

【0078】

図2に関して、コンピュータ実施方法2000の方法ステップが説明される。方法2000は、図1に関して上述した方法1000と同様である。

【0079】

図2に示すように、ブロック2100において、起動段階ⅠⅠの前にコンパイル段階Ⅰを使用し、プログラムコード内のプログラムセグメントを識別し、識別されたプログラムセグメントについての実行時セグメントをコンパイルすることができる。

【0080】

さらに、識別されたプログラムセグメントから導出された数値演算を実行するのに適した処理ユニットは、通常、方法2000の起動段階ⅠⅠのブロック2200において決定される。

【0081】

起動段階ⅠⅠの次のブロック2300において、決定された処理ユニットは実行時セグメントのそれぞれの計算カーネルで初期化される。

【0082】

実行時フェーズⅠⅠⅠの次のブロック2400では、第1のデータ構造の実行時インスタンスのデータメタ情報が決定される。これは、第1のデータ構造の次元を決定すること、第1のデータ構造の各次元に格納されている要素の数を決定すること、第1のデータ構造に格納されている要素の数を決定すること、要素のデータタイプを決定すること、実行時インスタンスのタイプ情報を決定すること、および／または実行時インスタンスの実行時場所情報の決定することを含む。

【0083】

実行時ロケーション情報が第1のデータ構造の実行時インスタンス内に格納されている場合、後者は容易に進められ得る。

【0084】

数値演算が入力として2つ以上のデータ構造を使用する実施形態では、ブロック2400において、すべての実行時インスタンスについてデータメタ情報を決定することができる。

【0085】

次のブロック2500では、実行時インスタンスの関数メタ情報およびデータメタ情報を使用して、処理ユニット上でカーネルを実行するそれぞれの予想コストを決定する。

【0086】

10

20

30

40

50

実行時フェーズⅠⅠⅠの次のブロック2600において、予想されるコストは、実行時インスタンスを用いて数値演算を実行するためにそれぞれのカーネルを実行するための処理ユニットを選択するために使用されてもよい。

【0087】

その後、ブロック2700において、（それぞれの）計算カーネルが、選択された処理ユニット上で実行されてもよい。

【0088】

図2に破線の矢印で示されるように、いくつかの実行時セグメントが実行されるべきである場合、ブロック2400から2700はループで実行されてもよい。

【0089】

任意選択で、アウトオブオーダー実行方式を実施することができる。

【0090】

図3について、コンピュータ実施方法3000の方法ステップを説明する。方法3000は、図2に関して上述した方法2000と同様である。

【0091】

方法3000は、プログラムコード内のプログラムセグメントを有するシーケンスを識別するためのブロック3110から始まる。この目的のために、パーサを使用してプログラムコード内の数値演算を検索することができる。

【0092】

後続のブロック3150において、いくつかの後続の実行時セグメントを作成することができる。各実行時セグメントは、同じでも異なってもよいそれぞれの一般データ型の複数の要素を格納する1つまたは複数のデータ構造を受け取るように構成することができる。さらに、各実行時セグメントは、数値演算の1つまたは複数の出力のサイズ、出力の構造または形状、および／または出力を生成するためのエフォートに関するデータを含むそれぞれの関数メタ情報を含むことができる。

【0093】

コンパイルフェーズⅠの後続のブロック3170において、各プログラムセグメントは中間表現に変換されてもよい。これは、コンパラインフラストラクチャを使用して実行できる。

【0094】

各実行時セグメントは、1つまたは複数のデータ構造の実行時インスタンスを用いて実行されるべきそれぞれの数値演算を参照するそれぞれの中間表現およびそれぞれの関数メタ情報を含むことができる。

【0095】

中間表現は、シーケンスまたは対応するプログラムセグメントと共に、またはそれらを参照して記憶されたメタデータから、またはメタデータを使用して作成することもできる。

【0096】

方法3000の起動段階ⅠⅠの次のブロック3200では、識別されたプログラムセグメントから導出された数値演算を実行するのに適した処理ユニットが通常決定される。

【0097】

起動段階ⅠⅠの後続のブロック3270において、中間表現は、それぞれの機械語表現（カーネル計算）にコンパイルされ得る。これは、利用可能な対応する処理ユニットに特化されてもされなくてもよく、利用可能な処理ユニットの特性に対する機械語および／または中間表現をよりよくカスタマイズすることを容易にするジャストインタイムコンパイラを使用して達成され得る。

【0098】

起動フェーズⅡの次のブロック3300において、決定された処理ユニットは、ブロック3270で形成されたそれぞれの計算カーネルで初期化される。

【0099】

実行時フェーズⅠⅠⅠの後続のブロック3400において、データ構造の実行時インス

10

20

30

40

50

タンスのデータメタ情報が決定される。

【0100】

次のブロック3500では、実行時インスタンスの関数メタ情報およびデータメタ情報を使用して、処理ユニット上でカーネルを実行するそれぞれの予想コストを決定することができる。

【0101】

実行時フェーズⅡⅡⅡの次のブロック3600では、ブロック3500で決定された予想コストを使用して、実行時インスタンスを用いて数値演算を実行するためにそれぞれのカーネルを実行するための処理ユニットを選択することができる。

【0102】

その後、ブロック3700で、計算カーネルを選択された処理装置上で実行することができる。

【0103】

図3の破線の矢印は、いくつかの実行時セグメントが実行されるべき場合、ブロック3400から3700がループで実行され得ることを示す。

【0104】

別の実施形態では、ブロック3600は、予想コストを使用して、計算カーネルを実行する作業負荷および計算カーネルの数値演算をそれぞれ実行することができる少なくとも2つの処理ユニットに分散するブロックによって置き換えられ、それらはブロック3700のための置換ブロック内の実行時インスタンスのそれぞれの部分に対するそれぞれの数値演算を実行する。

【0105】

図4は、異種コンピュータ70として実装された異種コンピューティングシステムの例示的な構成要素、およびコンピュータ実装方法4000の実行時フェーズⅡⅡⅡで一連の実行時セグメントを実行する典型的なプロセスを示す。方法4000の実行時段階ⅡⅡⅡの前に、図1から図3に関して上述したものと同様の始動段階(ⅡⅡ)がある。

【0106】

例示の実施形態では、異種コンピュータ70は、やはりホストコントローラ73のメインプロセッサまたはホストプロセッサを形成するCPU71とGPU72とを備える。

【0107】

複雑なユーザプログラムは、セグメントに分割されているシーケンスに分割される。セグメントは、シーケンスから派生した順序で連続して並ぶ。

【0108】

図4では、数字は、元のシーケンスの連続部分を表す引き続く実行時セグメント30～34に対してのみ与えられている。

【0109】

起動段階(プログラムの実行開始時)において、シーケンスを処理することができることが判明した各処理ユニット71、72、73は、シーケンスのすべての実行時セグメント30～34を処理するために初期化されている。以下では、処理ユニットは、装置(装置3、装置2、および装置1それぞれ)とも呼ばれる。

【0110】

したがって、シーケンスの実行時セグメント30～34に対応するそれぞれの計算カーネルがコンパイルされ、各デバイスに割り当てられている。カーネル140～144は、CPU71用にコンパイルされ割り当てられている。コンパイルされたカーネル140～144は、CPU上で元のシーケンスの実行可能表現111を形成する。

【0111】

同様に、同じ実行時セグメントの計算カーネル150から154がコンパイルされ、GPU72上に割り当てられ、GPU72上に元のシーケンスの実行可能表現112を形成する。

【0112】

10

20

30

40

50

これに加えて、セグメントのコードのスカラバージョンを表すスカラ計算カーネル 130～134 を各実行時セグメントに対して用意する。スカラ計算カーネル 130～134 は、プログラムが提供されたのと同じホスト言語を使用して、ほとんど CPU 上で動作する。スカラコードカーネル 130～134 は、CPU 上の元のシーケンスの表現 100 を形成するが、入力データ構造の通常はスカラ入力そしてまたは典型的には比較的小さいインスタンスのデータおよび／または順次処理に最適化される。

【0113】

実行時セグメント 30～34 のそれぞれは、対応するコンパイルされ割り当てられた計算カーネル 130～154 への参照を、決定されたデバイスのそれぞれについて少なくとも 1 つ格納することができ、これはいつでもその実行をトリガするために使用することができる。

10

【0114】

起動段階の終わりに、シーケンスのすべてのセグメントは、サポートされている各デバイスについてプログラムセグメントの実行可能バージョンへの 1 つの参照を格納することができる。

【0115】

例示的な実施形態では、各セグメントは、非常に低いオーバーヘッドで 3 つの異なる方法でそのコードを実行することができる。それらは、スカラデータのための 1 つのバージョン（それぞれデバイス 1 とホストコントローラ 73）、CPU 上で走るための配列バージョン（71 および デバイス 2）、および GPU 上で走るための 1 つのバージョン（72 および デバイス 3）である。

20

【0116】

各セグメント 30～31 は、実行時に実行パスを切り替えて準備された実行パスのうちの 1 つを選択するために使用される制御コード 50 を実装することができる。実行経路は、図 4 に太い破線の矢印として示されている。

【0117】

どの経路を実行に使用するべきかの決定は、典型的には、データサイズ、データ形状、およびデータ局所性、セグメントの個々の実行経路の計算コスト、デバイスの特性、デバイスの現在の利用状況、各デバイスに関連する追加のオーバーヘッドに関するヒューリスティックまたは測定したデータ、データサイズに関連する固定または動的しきい値に関するヒューリスティックなどの複数の要因に基づく。どの経路が使用されるべきかの決定はさらに、前のおよび／または連続するセグメント、およびこれらのセグメントについての最低実行時間に対応するデータ局所性構成に依存し得る。関連情報はまた、セグメントの過去の使用データによって、および／または投機的処理分析および／または評価によって得ることができる。

30

【0118】

上記で説明したように、最良の実行経路は、セグメントコードの実行を最も早く終了するもの、または実行中に最も低い電力を使用するものであることが判る。これについては以下で詳しく説明する。

【0119】

シーケンスの実行が時間 42 の方向にセグメントを通して進行するとき、シーケンスの実行経路は準備された装置のうちのいずれかの間で迅速に切り替わることができる。

40

【0120】

任意の入力アレイ引数 10 が実行のために各セグメントに提供されるべきである。

【0121】

各アレイオブジェクト（実行時インスタンス）A は、そのデータがコピーされた場所（デバイスメモリ）を追跡することができる。アレイ A が最初にデバイス 71、72 で使用されたとき、そのデータは通常デバイスメモリにコピーされ、アレイ A への参照 20～22 は通常アレイ A と共に記憶される。そのようにして、配列 A が入力引数として複数回使用される場合はデバイス間のデータ転送は最小化できる。

50

【0122】

デバイス上のセグメントの結果記憶としてアレイが使用されるとき、このデバイスは、アレイが現在記憶されている唯一のデバイスとしてアレイ内にマークされる。アレイがコピーされた可能性のある他のデバイス位置は、そのアレイのアレイ位置のリストから消去される。

【0123】

図5によって、図2および図3に関して上で説明された方法のうちのいずれかにおいて使用され得るコンパイラフェーズIのさらなるプロセスを説明する。

【0124】

第1のプロセスでは、ユーザプログラム200が分析される。実行時に実行コストを決定することを可能にするプログラム200のシーケンスが識別される。明確にするために、例示的なユーザプログラムはただ1つのシーケンス210を有する。

【0125】

シーケンスは、スカラデータ205を扱う単純な命令を含むか、またはそれからさえなり得る。この場合、コンパイラは、その命令に対して生成された実行可能コードのみに基づいてコスト情報を導き出すことができる。

【0126】

それに替ってまたは追加的に、シーケンス210は、より複雑な数値演算および／またはそれぞれ共通のデータタイプの複数の要素を格納することができる1つまたは複数のデータ構造10を潜在的に扱うより複雑な命令を有することができる。

【0127】

そのようなデータ構造は、プログラム200を書くために使用される言語のユーザに利用可能な任意の所定の配列データタイプであり得る。あるいは、データ構造はライブラリまたはスタンドアロンモジュールまたはプログラムにインポートされる外部モジュール（動的リンクライブラリ/静的リンクライブラリ）から得ることも可能である。このようなデータ構造は通常、要素数、データ構造内の個々の次元の数と長さなど、同じまたは類似のデータ型の複数の要素とデータ構造に関する付随情報、例えば要素の数、個別のデータ構造（配列と直線配列の「形状」として）における次元、タイプと地域情報（保管場所）をサポートするそのような言語の要素の型を格納できる。

【0128】

さらに、データ構造10は、一般データ構造について必要な情報も内包し、一般データ構造がこの情報を自然に提供しない場合にはそれを一般データ構造10として利用可能にすることを可能にする。

【0129】

ユーザプログラム200の少なくとも一部は、データ構造10に対して数値演算を実行する命令から構成される。そのような命令は、コンパイラに知られているか、またはコンパイラによって自動化された方法で認識されるようになる方法を提供する。

【0130】

そのような命令の一例は、Matlab (The MathWorks Inc.)、numpy (numpy.org)、ILNumerics (ILNumerics GmbH)、Julia (julialang.org)、Fortran、Scilab (scilab.org)、Octave、FreeMat (freemat.sf.net) そして Mathematica (Wolfram Research) などの技術的言語による関数のセットである。そのような言語は、それら自身のコンパイラと開発インフラストラクチャを提供するかもしれない。あるいは、その言語は、別の通常はより一般的な言語 (GPL) のドメイン固有言語拡張 (DSL) として実現され得て、GPLのコンパイラおよび開発ツールの全部または一部を利用してもなくてもよい。ここで、データ構造10は、任意のサイズおよび次元の直線配列として実現することができる。これらの関数の中には、1つ以上の要素の配列（多値入力パラメータ）を取り、map（要素に対する関数の実行）、sin（その要素の正弦の計算）、加算（2つの配列の要素の加算）のように、パラメータの要素の多くまたはすべてに対して操作を実行するものがある。他の関数は、'0'（ゼロ値要素の配列の作成）、'ones'（1値要素の配列

10

20

30

40

50

の作成)、rand (ランダムな値の要素の配列を作成する)、clone (別の配列のコピーを作成する) のように、0個以上の配列またはスカラー入力パラメータに基づいて1つ以上の新しい配列データを作成し得る。他の典型的な関数は、'reduce' (配列の次元に対して集約関数を実行する)、'sum' (特定の次元に沿って要素を合計する)、'max' (次元に沿って最大値を計算する) のように集約操作を実行する。他の関数は、例えば 'vertcat' や 'concat' (特定の次元に沿って配列を連結する) のように、データ構造からサブアレイを作成したり連結を行ったりする。さらに、より複雑な操作例えば、'matmult' (行列の乗算)、'linsolve' (線形方程式システムの解法)、'svd' (特異値分解)、'qr' (QR行列分解)、'fft' (高速フーリエ変換の実行)、'eig' (固有値/固有ベクトル分解)、'interpolation' (既知の離散値による新しい値の補間)、'fmin' (関数の最小値を求める) を使用することができる。

10

【0131】

関数は、例えば内の操作を制御するため、追加の補足パラメータを利用することができる。一例として、sum (A, 1) 関数を取り上げると、ここで1は、入力配列Aの次元のインデックスを決定して、それに含まれる要素を合計する補足のスカラーパラメータである。いくつかの関数は、入力配列と同じ型で同じサイズの単一の出力配列を生成する。他の関数は複数の出力を生成するかもしれない、いくつかの出力引数はサイズやタイプが異なるかもしれない、例えばブロードキャストバイナリ関数 'add'、'find' (ゼロ以外の要素を探す)、'max (A, 1, I)' (Aの次元 '1' に沿って最大値を持つ要素を見つけ、そのような要素のインデックスも与える)。さらに他の関数は、ゼロから、または入力配列の一部を抽出すること、および／またはその一部を修正することによって、新しい配列を生成することができる。後者の関数カテゴリの一例は、A [2 ;:] = Bのような修正 (左側) サブアレイ演算子式であり、ここで、Bの値はAの対応する要素に割り当てられ、インデックス2の行に沿って格納される。

20

【0132】

ユーザプログラム200内の識別可能な命令の別の例は、FORTRAN、C / C + +、Perl、C #、Visual Basic、F #、python、JavaScriptなどの言語のための実行時補足として同様に提供される一組の関数である。そのような (登録商標) 1つの例示的な機能の組は、.NET クラス System. Math の静的機能メンバによって記述される。これには、Abs、Acos、Asin、Atan、Atan2、Ceiling、Cos、Coshで始まり、Sin、Sin、Sqrt、Tan、Tan、Truncateで終わる名前の関数の範囲が含まれるが、これらに限定されない。そのような関数のスカラー性のために、記述された方法のコンパイラは、配列の要素上の反復を単一の配列関数呼び出しに変換するために、ユーザコード中の共通言語ループ構成パターン (for、while、goto- ループ) を検出することができ、記述された方法に従ってプログラム・セグメントに処理するのに適している。

30

【0133】

開示された方法の一実施形態は、Python、JuliaまたはMATLABのような動的に型付けされた言語 (例えばダックタイピング) で実施されてもよい。ただし、C #やJava (登録商標) などの静的型付け言語を使用した実装では、より優れた実行時パフォーマンスを達成することが期待されている。静的型付け言語で実施される実施形態および動的型付け言語で実施されるそのような実施形態の両方は、プログラムの両方のカテゴリ、すなわち静的型付けプログラムおよび動的型付けプログラムに適用可能である。プログラムが静的に型付けされ、開示された方法が動的に型付けされた言語で実施される場合、プログラムのデータおよび関数に関する情報は省略されてもよい。それに対応して、プログラムが動的型付け言語で提供され、開示された方法が静的型付け言語で実施される場合、欠けている型情報はコンパイラまたは実行時実行をサポートする実行時フレームワークによってプログラムから推論される。したがって、言語コンパイラおよび／または元の動的型付けプログラムの言語を補完する実行時実行フレームワークにおける (直接実行に使用される) 最終型の決定を容易にする前記と同じメカニズムを使用することができる。あるいは、コン

40

50

パイラはプログラムコードを検査して、プログラムセグメントのコードまたはコンパイルされたバイナリ表現内または補足資料内に見られるプログラムコード式（例えば定数）やマークアップ（属性、型宣言）から、コンパイラで利用可能なデフォルトタイプのリストから、そしてまたは連続したプログラムセグメントまたは他のプログラムコンテキストから、データ構造要素およびプログラムセグメントの型を推測する。

【0134】

一連の命令210は、ユーザによって作成されていてもよい。コンパイラは、シーケンス210の個々の命令を組み合わせ、既知の命令のプログラムセグメント220～222を形成することができる。このようなグループ化（結合）の主な目的は、実行パフォーマンスを向上させることである。この場合、プログラムセグメント220の数値的コストは、後述するようにプログラムセグメントにグループ化された個々の命令のコスト関数に基づいてコンパイラによって自動的に導出される。

10

【0135】

あるいは、プログラムセグメント220は単一のアレイ命令のみを含むことができる。

【0136】

図5の例示的实施形態では、シーケンス210は3つのプログラムセグメント220～222からなる。

【0137】

ユーザプログラム200の一部が既知の命令としてコンパイラによって識別可能ではない場合、現在のプログラムシーケンス210は終了されてもよく、新しいプログラムシーケンスが次の識別可能な命令で開始される。

20

【0138】

例示的な実施形態では、シーケンス210は、ユーザプログラム200内のプログラムの完全なコード行からの2つの入力アレイA、Bおよび1つの補足スカラー入力iを有する。

$$C = \text{sum}(\text{abs}(A - B[\text{full}, i]))$$

【0139】

全ての命令（sum、abs、マイナス、サブアレイ）はコンパイラに知られているので、単一のシーケンスがプログラム行全体から作成される。この場合、割り当て '=' が生成されたシーケンスに含まれる。

30

【0140】

他の実施形態では、割り当てをプログラムセグメントから除外することができ、または複数の割り当てをプログラムセグメントに含めることができる。これにより、プログラムセグメントは、任意の数の命令、命令行、および命令セクションにまたがることができる。

【0141】

さらに、プログラムセグメントは、制御構造および／またはループ構成を含んでも含まなくてもよい。どのような場合でも、コンパイラが制限付き言語構成要素の1つを識別すると、シーケンス境界が作成される。

【0142】

さらにまた、ユーザプログラム200内のマークはまた、シーケンスまたはプログラムセグメントの境界を示してもよい。そのようなマークは、ユーザによってまたは自動化プロセスによって作成される。例えば、プリプロセッサは、プログラムコード200を分析し、その後のコンパイルステップへのヒントを示すためにプログラムと共にマークを作成し格納することができる。

40

【0143】

あるいは、および／またはさらに、ユーザは、例えば言語属性の形でマークを編集、作成、使用および／または格納して、コンパイラに識別、セグメンテーション、および／または、ユーザ提供の機能またはプログラムに関する必要なメタデータの抽出をサポートまたは制御するための情報を提供することができる。ただし、コンパイラは一部またはすべてのマークを無視することもある。

50

【0144】

プログラム200をシーケンス210に分割し、シーケンス210をプログラムセグメント220～222に分割するプロセスは、識別されるべきシーケンスを表すグラフデータ構造の作成を含み得る。

【0145】

そのツリーは抽象構文木(AST)として作成することができ、その場合、それはデータ依存性および命令のプログラムフローおよびシーケンス内の対応するデータのプログラムフローを識別するために必要なすべての情報を保持する。グラフ内のこの情報は、機能のメタデータなどの付随する情報と共に、プログラムセグメント境界を決定するため、および／またはプログラムセグメントコスト、プログラムセグメントパラメータセット、および／またはプログラムセグメント出力サイズを計算するためにも使用され得る。

10

【0146】

実行中に高い効率を達成するために、コンパイラは通常、プログラムセグメントにマージされる命令の数を最大化しようと試みる。これはセグメントの複雑さおよび作業負荷を増大させ、そしてセグメント間の移行のためのオーバーヘッドおよび中間結果の記憶／転送コストを最小にすることによってしばしば処理ユニットのより良い利用をもたらす。

【0147】

例えば、コンパイラは、少なくとも1つの特定の限度に達し、新しいセグメントへのさらなる命令の追加が停止されるまで、プログラムコード内の隣接する命令を反復して組み合わせることによってプログラムセグメントのサイズを増加させようとする。そのような制限は、新しいセグメントに対する組み合わせ演算の複雑さ、新しいセグメントの入力／出力パラメータの数および／またはタイプ、結合された命令のメタ情報に基づいてセグメントの入出力データ構造の特性の機能として必要な関数メタ情報(機能サイズおよび機能仕事量として)を作成／導出するためのコンパイラロジックの能力、そしてまたは個々の命令から組み合わせた数値演算を表す計算カーネルの中間表現を作成する能力、に関するコンパイラ固有の規則(発見的)またはコンパイラ固有の制限によって確立され得る。

20

【0148】

組み合わせられた命令を用いてプログラムセグメント境界を定義する別の方法は、既知の命令のセットを、入力パラメータサイズに関していかなる出力パラメータについてもサイズの変更を引き起こさない命令(マップタイプ命令)と、入力パラメータのサイズに関連して少なくとも1つの出力パラメータのサイズを変更する可能性がある命令(縮小型命令、サブアレイ、および作成型命令)とに分類することである。一実施形態によれば、プログラムセグメントは、ユーザプログラムコード内のサイズ変更命令の直接の近隣から導出される任意の数の既知のマップタイプ命令を含む最大1つのサイズ変更命令を含む。

30

【0149】

当業者が特に有用であると思うプログラムセグメント境界配置の決定を支援する方法は、各処理ユニットの実行時入力／出力データ構造のインスタンスを用いてセグメントの計算カーネルを実行する仕事量を決定する複雑さ(コスト)を制限することである。一実施形態によれば、そのような仕事量は、セグメント化中のコンパイル時に新しい命令がプログラムセグメントに追加されるときに作成／採用されるセグメント関数メタ情報から導出される関数によって計算される(下記を参照)。マップタイプの計算命令がセグメントに追加されても、そのようなエフォート関数の複雑さは通常は増加しない。他の種類の命令は、より複雑な費用関数につながる可能性があり、実行時に評価のためにより高い労力を必要とする。セグメントに対する処理ユニットを決定するオーバーヘッドを小さく保つために、コンパイラは新しい命令の追加がプログラムセグメントのコストを評価するための複雑さまたは仕事量を増大させかねないときはいつでも、セグメント境界を置くことができる。

40

【0150】

サポートされている命令を見つけるために、プログラム上またはその一部に対して意味解析を実行することができる。そのような分析の目的は、命令および／または命令が処理

50

しているデータのタイプまたは他の情報を決定することである。

【0151】

あるいは、および／またはさらに、単純なテキストマッチングをプログラム200のトークンに対して実行することができる。

【0152】

プログラム200が十分なタイプ情報を直接提供しない場合には、意味解析を使用してデータタイプを推測することもできる。一例は、技術的プロトタイピング言語に共通な動的または弱型付け言語および／または構造の集まりを参照できる。

【0153】

コンパイル段階Iの間、識別された各プログラムセグメントに対する実行時セグメント240が通常作成される。実行時セグメント240は通常、実行時に処理ユニット(71、72、73)上でプログラムセグメント220の数値演算を実行するのに必要なすべての情報を格納している。

【0154】

一実施形態では、実行時セグメント240は、起動時IIにインスタンス化されることになるコンパイラ生成クラス定義249として実現される。通常、このクラスは、サポートされているデバイスカテゴリ(カーネル241、242、243)の数値演算の中間表現、セグメント計算カーネル241~243の配列引数の決定、キャッシュ、ロードを容易にする引数アダプタ245、セグメントの(組み合わせされた)命令に関する情報248、および実行時に処理コストの決定および処理ユニットの選択を実行するコードを含む装置選択コード247を格納している。デバイスセクタ247は、プログラムが実行されるときに実行時セグメント240、249のエントリポイントとして機能することができる。したがって、デバイスセクタ247は、プログラムセグメント220と同じ数および同じタイプの入力および出力引数をサポートする機能インターフェースを提供することができる。これにより、コンパイラは、プログラムセグメント220が成り立っている命令群の直接の代用としてデバイスセクタ247を使用することができる。

【0155】

この代用は、プログラムコード200の修正によって、またはユーザプログラム200から導出されたコンパイル済みアセンブリ(実行可能バイナリ)の修正によるコードインジェクションを介して実現され得る。

【0156】

実行時セグメントの実装は、別個のモジュール、アセンブリ、ライブラリ、リモートサービス、またはローカルサービスとして実現することができる。あるいは、実行時セグメントコードをユーザプログラム200に追加することができる。

【0157】

一実施形態では、起動フェーズIは、実行時セグメント249をプログラム内の静的コンテキスト変数としてインスタンス化することによって実施およびトリガすることができる。起動段階Iの間、利用可能な処理ユニット(73、71、72)に関する情報が収集され、プログラムセグメント220の中間表現241、242、243が各サポートされている処理ユニット(73、71、72)に対して調整され、計算カーネルはコンパイルされ、処理ユニット(73、71、72)にロードされる。

【0158】

代替的または追加的に、上記のリソースのいくつかに対する初期化ステップのいくつかは、対応するリソースの利用が実行中の最初に必要とされる時間まで遅延されてもよい。

【0159】

上述のように、各個々のセグメントを実行するコストは実行時に決定可能である。他の手法とは異なり、コンパイラは、コンパイル時または異種コンピューティングシステムのデバイス(処理ユニット)にセグメントを割り当てる前に、命令の実際のコストを決定しようとしな。代わりに、すべてのデバイスが同様に実行可能実行時セグメントを備えており、予想コストは実行時に、通常は実行直前に決定される。

10

20

30

40

50

【0160】

実行時セグメントの予想コストは、通常、特定の処理ユニット上の実行可能な表現の命令の固有コストおよびプログラムセグメントの命令から導出された数値演算によって扱われる具体的なデータを考慮に入れたコスト関数に基づいて評価される。

【0161】

固有セグメントコストは、対応するシーケンス210の命令のグラフを利用してプログラムセグメント内の各単一命令のコストから導き出すことができる。個別の命令のコストは実行時に効率的な評価を可能にする方法でセグメント全体の固有コストの表現に合併される。

【0162】

このような表現248は、実行時セグメント240と共に格納され、実行時に具体的な入力データ構造10についてプログラムセグメント220からコンパイルされた実行時セグメントを実行する実際のコストを決定するために使用できる。

【0163】

したがって、実行時セグメントの計算コストはそのような入力引数の関数であり、種々の方法で実装することができる。例えば、コンパイラはコスト関数を実行時セグメントに提供するためにいくつかの方法を利用することができ、そのうちのいくつかを以下に説明する。

【0164】

通常、コンパイラ250は、実行時に最高の実行性能または最低の電力消費をもたらすと予想されるコスト関数の実装を選択するであろう。そのような選択は多くの場合、実行時に取得するのに必要な最小限の情報に対応する。

【0165】

多くの命令について、コスト関数は、出力において単一の要素Cを生成するために実行される演算の数に関して決定できる。全体的なコストは、単一の要素Cのコストに係数Nを掛けた結果であり得る。ここで、Nは、命令の仕事量に応じて、生成される（べき）出力要素の数またはその一定のべき乗に対応する。

【0166】

$\sin(A)$ のような単純な演算では、Nは入力配列Aの要素数に等しいであろう。O(n^2)の仕事量で配列Aをソートする命令の場合、NはA内の要素数(n)の2乗に等しいだろう。他の命令はより高いまたはより低い仕事量を持ち、整数以外のべき乗を含む他のべき乗になる。

【0167】

単一要素Cの計算コストは、ルックアップテーブルまたはキャッシュから取得することができ、例えば命令を実行するために使用される具体的なハードウェア上での測定によって、または異種コンピューティングシステムの特定の処理ユニット(CPU、GPUなど)に対する命令のコンパイルから生じる実行時セグメントの計算カーネルの分析によって決定することができる。

【0168】

他の命令はより複雑なコスト関数を持つことがある。たとえば、コスト関数は、出力結果の要素数だけでなく、少なくとも1つの入力引数（つまり、個々のデータ構造の実行時インスタンス）の次元数およびまたは構造（すなわち個々の次元の長さ）、少なくとも1つの入力引数内の要素の値、他の入力パラメータおよびまたは出力パラメータの存在、構造、タイプ、値、およびまたは補足パラメータの存在および値に依存する可能性がある。

【0169】

コスト関数の情報は、コンパイラ250によって命令から自動的に導出することもできる。コストの自動導出は、関数を分析することを含むことができ、単純な命令に特に有用である。コンパイラ250は、機能のコストを自動的に導出するために、命令のコード、名前、可変数のパラメータ、およびまたは実行可能バイナリ表現のうちの少なくとも1つを使用することができる。

10

20

30

40

50

【0170】

あるいは、およびまたはさらに、実行時に計算コストを評価するのに適した機能に関連する補足的なメタデータがコンパイラによって使用され得る。そのようなメタデータは、関数のオーサリング中に作成されることがある。そのようなメタデータの指定されたセットおよびまたはフォーマットに同意することによって、プログラム200のプログラマは、それ自体の機能およびまたは動作を用いてサポートされる命令の集まりを拡張することができる。

【0171】

コンパイラ250はまた、定義済みの命令セットおよび関連するコスト関数を知ることがあり、およびまたはプログラムコード内の命令を既知の命令のコスト関数と一致させ得る。

10

【0172】

コンパイラ250は、対応する命令のコスト関数を単一のセグメントコスト関数に結合すること、あるいは個々の命令またはすべての結合された命令についてのコスト関数を、同様にある入力データ構造のインスタンスに対するセグメントの仕事量を表現するコスト関数で置換するように決定してもよい。

【0173】

あるいは、セグメントコスト関数は、セグメント内のすべての個々の命令のコスト関数を考慮することにより実装およびまたは評価されてもよい。これは、シーケンスグラフ210のノードをめぐり歩いて、命令を表す各ノードのコスト関数を評価することにより実現され得る。

20

【0174】

さらに、コンパイラ250は、結果として得られるコスト関数を単純化するために最適化を適用することができる。そのような最適化は、命令コスト関数のインライン化、セグメントに対するコスト値の品質／正確さの改善に（大きく）寄与しないようなパラメータの省略、セグメント内の命令の部分的または完全な集約、および他の方法を含み得て、実行時にセグメントコスト関数を評価することによる計算オーバーヘッドを削減する。例えば、コンパイラ250がプログラムセグメントについて、コストがその入力パラメータ10内の要素の数にのみ依存していることを識別した場合、他の情報、例えば入力パラメータ10の次元数およびまたは構造などがあり得るが、それは結果のコスト関数から省略される。

30

【0175】

一実施形態によれば、命令情報はメタデータ属性によって提供される。各命令は、命令に対するメタデータをコンパイラに問い合わせる方法を提供することができる。たとえば、適切なすべてのILNumerics関数は、コンパイラが関数を命令として識別するのをサポートする特定の属性で装飾されている。したがって、属性タイプはコンパイラに認識されており、属性タイプによって実装された共通インターフェースを介して関数の包括的なメタデータを提供するために使用できる。

【0176】

関数メタデータは、典型的には、配列であり得る出力データ構造の各生成インスタンスのサイズおよび構造を決定するためのデータ、命令（数値演算）がインスタンスを生成するために費やす「仕事量」を決定するためのデータ出力データ構造、および特定の処理ユニットおよびまたは一般的な処理ユニットタイプのためのカーネルコードを導出するためのオプションのデータを含む。この処理ユニットは、入力データ構造のインスタンスのセットに対して数値演算を実行するのに適している。

40

【0177】

上記のサポート関数メタデータは、通常、命令が生成できる各出力およびフレームワークがサポートする処理ユニットの各タイプに対して個別に提供される。

【0178】

たとえば、入力データln0のインスタンスの絶対または要素の大きさを計算するための

50

高レベルILNumerics関数abs (In0) は、クラスILMetadata_abs001へのリンクを含むILAcceleratorMetadata属性で装飾されている。両方とも、ILNumericsフレームワークのabs関数の作成者によって提供される。コンパイラーは、クラスをインスタンス化し、そのインターフェースを照会して、次の情報、入力データIn0のインスタンスに関してabs (In0) 命令によって生成される出力Out0データ構造のインスタンスのサイズ情報を収集できる。個々のサイズ情報は、以下を含むが、これらに限定されずに提供される、(1) Out0に格納されている要素の総数 (Numel (In0))、(2) Out0の構造を記述するために使用される次元数 (Numdim (In0)) および (3) Out0 (Size (In0, 0) to Size (In0, 2)) の各次元の長さ (要素の数)。これらは通常、セグメントサイズベクトルまたはリストを表す表Iに示されている。

10

Numdim(In0)
Numel(In0)
Size(In0, 0)
Size(In0, 1)
Size(In0, 2)

20

30

【0179】

そのような情報は、典型的には、入力データ構造In0のインスタンスを受け取り、引き続き要素として記憶されたサイズ情報を有するベクトル (サイズ記述子) を生成する関数に関して提供される。abs (In0) によって生成される出力は1つのみであるため、ILMetadata_abs001クラスの例では、単一のサイズの記述子のみが提供される。命令の種類によっては、サイズ記述子関数は、命令に定義された引数の数に対応する追加の引数を必要とする場合がある。たとえば、2つの入力配列の要素を追加するadd (In0, In1) 命令には、サイズ記述子関数で必要または不必要で、提供され、使用される2つの配列引数が必要である。

40

【0180】

abs (In0) 命令の関数メタ情報としてさらに提供されるのは、(1) In0の要素数、(2) In0の次元数、(3) In0の各次元の長さ、に各々関係した (一般化された) CPUプラットフォーム上の入力データIn0のインスタンスに関する命令の「仕事量」である。

50

【0181】

そのような仕事量情報は、浮動小数点または整数値の仕事量ベクトルを形成する正規化された値または定数として提供され得る。仕事量ベクトルの各要素は、サイズ記述子ベクトルの要素の情報に対応する、命令の正規化された仕事量を記述する仕事量情報ベクトル（すなわち、入力データ構造の要素ごとの仕事量）および入力データ構造のインスタンスに対応するサイズ記述子ベクトルの両方でベクトル積（すなわち、スカラー積の計算）を実行する場合、結果の値は、入力データ構造のインスタンスで命令を実行する仕事量に対応する。

【0182】

同様に、（一般化された）GPUおよびまたは任意の他の処理ユニット上の入力データ `In0` に関する `abs(In0)` 関数の「仕事量」が提供されてもよい。

10

【0183】

他の実施形態は、実行時に命令の仕事量情報を提供する他の方法を使用してもよい。このような方法は、上記の方法よりも簡単である。計算から情報を省略したり、一部の情報を適切な推定値に置き換えたりできる。他の実施形態では、実行時に命令の仕事量情報を提供する他の方法を使用してもよい。このような方法は、上記の方法よりも簡単である。計算から情報を省略したり、一部の情報を適切な推定値に置き換えたりできる。または、メソッドはより複雑で、関連するエンティティの同じプロパティや他のプロパティを考慮します。たとえば、仕事量情報は、関数ではなく定数として提供される場合がある。または、計算では、入力データ構造インスタンスの要素の値およびまたはタイプ、処理ユニットの状態情報、ホストコントローラおよびまたは他のシステムコンポーネント、入力/出力データ構造インスタンスによって提供される詳細情報、関数メタ情報およびまたは計算カーネルを考慮に入れることができる。

20

【0184】

関数メタ情報は、具体的なCPUプラットフォーム上の特殊なカーネル機能本体のテンプレートとして使用するのに適したテンプレートカーネルコード（計算カーネル）をさらに含むことができる。通常、テンプレートは、具体的な処理ユニットがわかっているときにプログラムの実行で完成する。テンプレートには、たとえば起動時に、たとえば処理ユニットのプロパティで、後で置換されるプレースホルダーが含まれる場合がある。そのようなプロパティには、プロセッサ、コア、キャッシュサイズ、キャッシュラインサイズ、メモリサイズと速度、ビットレート、機能およびまたは処理ユニットでサポートされる命令セット、電力消費、およびまたは処理ユニットの周波数が含まれる。

30

【0185】

同様に、具体的なGPUおよびまたはホストコントローラを含む他の処理ユニット上の特殊なカーネル機能本体のテンプレートとして使用するのに適したカーネルコード（計算カーネル）は、関数メタ情報の一部として提供されてもよい。このようなテンプレートカーネルコードには、起動時に、プロセッサ数、コア数、キャッシュサイズ、キャッシュラインサイズ、メモリサイズと速度、ビットレート、機能およびまたは処理ユニットによってサポートされる命令セット、電力消費およびまたは処理ユニットの周波数などの具体的な処理ユニットのプロパティで置換えられるプレースホルダーが装備されている場合がある。

40

【0186】

コンパイラ250は、収集された情報を使用して実行時セグメントを作成する。実行時セグメントには、ユーザープログラムが作成されたのと同じ言語の実行可能コードが含まれる場合がある。コンパイラ250は、セグメントが由来した元のプログラム部分の代わりに実行時セグメントの実行を引き起こす方法でプログラムコードを修正してもよい。

【0187】

あるいは、コンパイラは、実行システムと互換性のある実行可能命令を作成し、およびまたはユーザプログラム200から作成された実行可能リソースを変更して、実行時にユーザプログラム200の対応する部分の代わりに実行時セグメントを実行することができる。

50

【0188】

実行時セグメントは、識別された単一の命令に広がっていることがある。この場合、セグメント仕事量（以下で説明）は命令仕事量に対応し、セグメントサイズ（以下で説明）は命令サイズに対応する。他の場合では、セグメントは複数の命令にまたがり、セグメントのサイズと仕事量は、複数の命令仕事量と-sizesのセットから計算されます。セグメント仕事量と-sizeの作成の説明については、以下が参照される。

【0189】

実行時セグメントは、命令の元の意図のために複数の実行パスを実装することもある。1つのパスはCPU上で実行時セグメントを実行し、プログラムコードが作成された同じ実行時システムを潜在的に利用する。他のパスでは、OpenCL、CUDA、OpenGLなどの低レベルの加速インターフェイスを使用して、GPUなどの加速デバイスで実行時セグメントを実行できます。オプションで、スカラーまたは小さな入力構造に対して高速トラックが提供されます。コンピューティングカーネルが初めて実行されるときに、そのようなロードがのろのろした方法で実行される場合でも、すべてのパスは、それぞれのコンピューティングカーネルに対応する処理ユニットにプリロードすることで容易に初期化されます。

【0190】

一実施形態における実行時、どのパス（どのデバイス）が最速/最も効率的な結果の作成に使用されるかの決定は、多くの場合、動的な実行時データ、たとえばデータメタ情報、特にサイズ、構造、およびまたは入力データ構造の実行時インスタンスの値に依存する。他のすべての条件情報は、コンパイル時およびまたは起動時にコンパイラによって準備済みセグメントの仕事量に「焼き付け」られている。これは通常、コンパイル時または起動時に既知であり、実行時の動的な変更の問題ではない情報に特に当てはまる。

【0191】

通常、各実行時セグメントは、セグメントサイズとセグメント仕事量に基づいて、実行パスを動的に決定するための高速切り替えを実装する。

【0192】

セグメント仕事量は、実行時セグメントによる単一の結果要素の計算に関連する操作の数に対応する。実行時セグメントが複数の出力結果を生成できる場合、実行時セグメントに対して複数の仕事量が存在する可能性がある。

【0193】

セグメントの仕事量は通常、入力データ構造インスタンスのサイズの関数として実現およびまたは実装される。

【0194】

セグメントの仕事量は、実行時セグメントに実装され、セグメントを組み立てる命令から蓄積された命令のコストの測定値を運ぶことができる。命令が個々のプラットフォームに実装される方法は異なる場合があるため、通常、サポートされる各プラットフォームの命令を個別にアセンブルすることから、セグメント仕事量が計算される。

【0195】

一実施形態によれば、セグメント仕事量は、多次元配列のサイズ記述に対応する長さnのセグメント仕事量ベクトルとして実装され、そこでnは、システムによってサポートされる最大の次元数に対応する。

【0196】

各セグメントの仕事量ベクトルSEV0、SEV1は、以下の要素を含むか、それらから構成される。

- 要素#0：入力次元数による関数の仕事量、
- 要素#1：入力要素の数による仕事量、
- ...
- 要素#2... (n-3)：対応する次元の長さによる仕事量。

【0197】

セグメント仕事量ベクトルの要素は、数値定数またはスカラー数値関数であり得る。コ

10

20

30

40

50

ンパイラは、セグメント仕事量ベクトルのすべての要素を埋めて維持するように、およびまたは。セグメント中の関数のプロパティ（複雑さ）に応じて計算から一部の要素を除外することでパフォーマンスを改善または最適化することを決定するように構成され得る。

【0198】

しかし、セグメント仕事量ベクトルの少なくとも1つの要素はゼロではない。

【0199】

セグメントサイズは、 n 次元配列のサイズ記述に対応し、長さ $(n+2)$ のセグメントサイズベクトルとして実装できます。 n は、システムでサポートされる最大の次元数に対応します。

【0200】

通常、セグメントサイズベクトルとセグメント仕事量は同じ長さである。

【0201】

セグメントサイズベクトルは、以下の要素を含むか、それから構成される：

-要素#0：次元数 (n) 、

-要素#1：要素の数、

-...

-要素#2... $(n+1)$ ：次元1... n の長さ。

【0202】

セグメントサイズベクトルの要素は、数値定数またはスカラー数値関数である。通常、要素は入力データ構造のインスタンスのスカラー関数で構成される。繰り返しになるが、コンパイラは（通常は実行前に）パフォーマンスを向上させるために、セグメントサイズのベクトルを調整およびまたは最適化することもできる。

【0203】

図6によって、 200×300 要素の配列インスタンスAに対して関数 $3 + \sin(A)$ を備えた例示的なプログラムセグメントを実装する実行時セグメントのそれぞれの計算カーネルを実行する際の実行時III予想コストの決定について説明する。例示的な実施形態では、予想コストは、2つの処理ユニット、すなわちCPU（デバイス0）およびGPU（デバイス1）を備えた異種コンピューティングシステムについて決定される。

【0204】

実行時IIIおよび計算カーネルの1つを実行する前に、使用可能な各デバイスでの各計算カーネルの実行の予想コストを計算することにより、実行時セグメントの実行に必要な処理ユニットを選択する。

【0205】

最初に、セグメントサイズベクトルSSVおよびセグメント仕事量ベクトルSEV0、SEV1が決定され得る。例示的な実施形態では、セグメントサイズベクトルSSVは、テーブルIに関して上記で説明したように実装され、配列インスタンスAに従って決定される。

【0206】

関数に関連するセグメントサイズベクトルSSVの任意の要素について、関数は、要素を数として取得するために評価される。セグメント出力の結果のサイズはデバイス間で変わらないため、これは1回だけ実行される。これを実現する方法については、以下で参照できる。

【0207】

セグメント仕事量ベクトルSEV0、SEV1は、通常、各処理ユニット（CPU—インデックス0、GPU—インデックス1）ごとに決定される。関連する要素関数が評価される（詳細については、以下を参照）。

【0208】

その後、各デバイスの計算コストは、セグメントサイズベクトルとセグメント仕事量ベクトルとの間のスカラー積を計算することにより決定される。

【0209】

その後、セグメントサイズベクトルと配列オブジェクトAで維持される現在のデータ局

10

20

30

40

50

所性情報に基づいて、計算コストと各デバイスへの潜在的なデータ転送コストとを追加することにより、予想（合計）コストを決定できる。

【0210】

その後、最も低い予想コストに対応するデバイスが実行のために選択され得る。

【0211】

さらに、予想されるコストを考慮して、より多くの情報を考慮することができる。たとえば、追加のデバイスプロパティ（キャッシュサイズ、CPUベクトル拡張のSIMDレーン長など）、データ転送の開始に必要なオーバーヘッドに対応する要因（実際のデータ転送コストに次ぐ）、計算カーネルのプリロードの実行をトリガーするオーバーヘッド、プロセッサ周波数、処理ユニットの使用率などが考慮される。

【0212】

あるいは、実行のためのデバイス選択は、上記よりも少ない情報に基づいてもよい。

【0213】

転送コストテーブルTCTは、通常、数値演算の実行に適したすべてのデバイス（たとえば、OpenCLの事前定義バージョンに基づく）が識別されると、起動時（起動フェーズ）に作成される。起動オーバーヘッドを減らすために、テーブルTCTをキャッシュできる。

【0214】

一実施形態によれば、配列データは、（ホスト上だけでなく）任意のデバイス上に存在してもよい。実行のために現在のデバイスから別のデバイスにデータをコピーするために必要な仕事量を推定するために、現在のデバイスから他のデバイスへの正規化された転送コストに対応する転送係数に基づいて転送のコストを推定することができる。

【0215】

伝達係数は、起動時の測定によって、またはシステムで提供される特定のヒューリスティックに基づいて取得できる。

【0216】

転送係数は、実行時に実際の配列サイズと乗算される。

【0217】

さらに、伝達係数は、実際の測定データに基づいてシステムの実行中に改良され得る。これは、実行時コンピューティングシステムのリソース状態の変化を取り入れるのに役立つ。

【0218】

一実施形態によれば、ロケーションベクトルLVが使用される。データ構造（配列）の実行時インスタンスは、複数のストレージロケーションに共存できる。そのような場所へ参照する場所ベクトルは、通常、各配列操作によって維持される。

【0219】

ロケーションベクトルLVの長さは、サポートされるデバイスの数に対応し、通常、プログラムの実行中は一定の長さであるが、個々の実行およびまたはコンピューティングシステムの間で異なり得る。

【0220】

ロケーションベクトルLVの各要素は、通常、すべての実行時セグメントの1つの（固定）デバイスインデックス0、1に対応する。デバイスインデックスは、処理ユニットの決定の一部として、プログラムの起動時に割り当てられる。

【0221】

ロケーションベクトルLVの要素値は、それぞれのデバイス上のデータバッファへの参照を指してもよい。

【0222】

位置データLVの要素は、配列データが要素インデックスに対応するデバイスで（現在）利用可能である場合、ゼロではなく、そうでない場合、ゼロである。

【0223】

図6に示す例示的な実施形態では、入力配列Aは、CPUデバイス（インデックス0）

10

20

30

40

50

上にのみ存在し、例示的に示されるように、位置ベクトル $LV = (1, 0)$ によってCPUのインデックスがゼロのロケーションベクトルLV要素にのみゼロでない値(1)が保存されている。配列AはGPUメモリ上に存在しないため、転送コストベクトルTCV1で図6に示するように入力配列Aの要素ごとに0.2の転送コストの例を考慮する必要があり、一方対応する転送コストベクトルTCV0がゼロで埋ったことになったCPUに対しては転送コストを考慮する必要がない。

【0224】

各処理ユニットについて、予想コストは、デバイスインデックス i ($i = 0, 1$) につきスカラー積 $SSV * (SEVi + TCVi)$ として決定できる。

【0225】

例示的な実施形態では、CPU上で実行時セグメントを実行するための予想コストは、GPU上で実行時セグメントを実行するための予想コストよりもはるかに高い(16980のみと比較して90000)。したがって、GPUを実行用を選択できる。

【0226】

明確にするために、図6は、セグメント仕事量ベクトルSEV0で結果として(例示的な処理ユニットについて)要素ごとに実行される数値演算($3 + \sin(A)$)の実行時のコスト決定のみを示している。それぞれがゼロ以外の要素を1つだけ持つSEV1(つまり、2番目の要素は、格納されている要素の総数に応じたコストを参照している)。

【0227】

セグメント仕事量ベクトルおよびセグメントサイズベクトルを計算するさらなる実施形態が、操作 $3 + \sin(A)$ を実行するセグメントに付加される関数メタ情報の例示的な作成を説明する図7~9について以下で説明される。

【0228】

良好なパフォーマンスのために、複数の命令をマージして単一のマージされたセグメントを形成することが好ましい場合がある。マージを実行し、実行時に利用可能な全体のセグメントサイズとセグメントの仕事量を提供するために、いくつかのオプションが存在する。これらの方法は、サポートするセグメントの複雑さと、情報を計算するコストが異なる。

【0229】

コンパイラは、コンパイル時にセグメントの構築を開始すると、図7の右側に示されているセグメントのコードツリーまたは抽象構文ツリー(AST)を最初に識別する。

【0230】

例示的な実施形態では、ツリーは比較的単純である。より複雑なツリーもサポートされる。通常、ツリーの各ノードは、プログラムの命令またはデータ引数(定数、配列)に対応している。コンパイラは、ツリーのノードをたどって、関係するすべての命令のメタデータを照会することから始める。

【0231】

ノードの巡り歩く間に、コンパイラは汎用GPU、汎用CPU、CPUホスト、汎用DSP、スカラーデバイスなど、サポートされているすべての処理ユニットカテゴリに対して以下を構築する

- セグメント仕事量ベクトル、
- セグメントサイズベクトル、そして
- セグメントカーネルテンプレート。

この段階では、個々の処理ユニットのカテゴリ用に作成された仕事量ベクトルは、具体的なデバイスに固有の情報をまだ運んでいないことに注意が必要である。むしろ、個々の仕事量ベクトルは、さまざまな処理ユニットのカテゴリで命令を実装および実行する個々の方法を取り得る。たとえば、一部のデバイスカテゴリでは、一部の命令を効率的に実行するためにループ構造やスレッド管理オーバーヘッドを実装するカーネルコードを必要とするが、他のデバイスカテゴリでは必要としない。

【0232】

10

20

30

40

50

セグメントサイズベクトルは、個々の処理ユニットで変化しないため、一度だけ構築する必要がある。

【0233】

セグメントのすべての出力スロットには、独自のセグメントサイズベクトルが割り当てられる。ASTで処理される命令の複雑さに応じて、セグメントの個々の出力スロットによって生成される結果は、個々の計算コストに対応する。いくつかの実施形態では、セグメントの個々の出力について個々の仕事量ベクトルを維持することにより、そのような違いを考慮に入れる。ただし、わかりやすくするために、この説明ではセグメントが十分に単純であるため、セグメントの実行ごとにすべての出力が一度に作成されるため、各処理ユニット（カテゴリ）の単一の仕事量ベクトルは実際のすべての出力の計算コストを表すことができる。

10

【0234】

入力のサイズを変更しないセグメント内の命令は、単一のサイズ関数に集約することができる。入力のサイズを変更しないこのような命令を含むセグメントは、通常、出力スロットに割り当てられた対応する入力のサイズ記述子を反映するサイズ関数ベクトルを取得するだけである。

【0235】

さもなければ、各出力スロットの最終セグメントサイズは、セグメントシンタックスツリーに対応する「サイズ変更ノード」のツリーとして実現される。すべてのノードサイズ関数は、それに応じて入力のサイズを変更し、親ノードのサイズ関数の入力として機能する新しいサイズを「生成」する。

20

【0236】

上記のサイズツリーは、通常、後の評価のためにセグメントに格納される。サイズ情報は、実行時のより効率的なセグメントサイズ評価を促進するために、コンパイラによって調整およびまたは最適化され、およびまたは特殊な形式で保存される。

【0237】

たとえば、セグメント $3 + \sin(A)$ は、 $\text{Add}()$ と $\text{Sin}()$ の2つの命令を実装します。 $\text{Add}()$ は2つの入力配列の要素を追加する二項演算子である。2つの引数の1つがスカラー定数「3」であるため、 $\text{Add}()$ によって生成される出力のサイズは、一般的な次元のブロードキャストルールに従って、 $\text{Add}()$ への2番目の入力（配列）引数のサイズのみ

30

【0238】

同様に、 $\text{Sin}()$ 命令は、入力配列内のすべての要素のサインを計算する。 $\text{Sin}()$ によって生成される出力のサイズは、入力配列のサイズと等しくなる。したがって、セグメント内の両方の命令はサイズが変更されず、セグメントサイズは入力サイズに対応する。

【0239】

これは、図8に反映されており、出力スロット Out0 （のみ）に作成されたセグメント Segment_0001 のセグメントサイズベクトル SSV は、入力配列引数 In0 （のみ）のサイズ記述子に対応する。

【0240】

多くのセグメントは、（上記の例のように）サイズ変更以外の操作のみを実装するため、多くのサイズのツリーは単一ノードツリーである。さらに、各処理ユニット PU0 から PU2 を参照するセグメント仕事量ベクトル SEV0 から SEV1 のスパース性と非サイズ変更演算子（下記参照）により、多くの場合、入力配列のみの要素の総数に基づいて、多くの場合、実行時にスカラー値の評価のみが特定の処理ユニットのセグメントの最終仕事量の計算に必要である。

40

【0241】

ただし、入力引数のサイズを変更する複数の集約関数（ $\text{Sum}()$ など）を含むセグメントの場合、サイズ関数は、命令のツリーと対応するメタデータに基づいて、実行時に再帰的に評価する必要がある。コンパイラは、セグメントサイズの評価を簡単に保つために、

50

セグメントの関数のスパンをそれに応じて制限するように決定できる。

【0242】

セグメント仕事量ベクトル $SEV0$ から $SEV1$ は、同様の方法で作成されてもよい。すべての命令のメタデータは、コンパイラによって特定の処理ユニットカテゴリごとに個別に照会される。各命令の仕事量は、スカラーまたはスカラー関数の単純なベクトルとして提供される。通常、セグメント仕事量ベクトル $SEV0 \sim SEV1$ は、対応する出力の単一要素を計算するそれぞれの正規化された命令仕事量に対応する。個々の処理ユニット $PU0$ から $PU2$ は、たとえば、処理ユニットでサポートされている異なる命令セット、処理ユニットでの意図された数値演算の異なる実装、およびまたは必要な数値要素データ型または数値精度の異なるサポートに依存して、単一出力要素を計算する異なった仕事量に連携する。

10

【0243】

非サイズ変更配列命令の仕事量データベクトルは、通常、関与する非サイズ変更命令の仕事量ベクトルを追加することにより、単一の仕事量ベクトルに集約される。図8は、セグメント $3 + \sin(A)$ によって生成された出力の単一要素を計算する仕事量が示されている一実施形態を示している。両方の命令は、サイズを変更しない命令である。したがって、出力のすべての要素には、1つの加算の計算と（固有の）正弦関数の1つの評価が必要である。両方の演算の仕事量は、命令のメタデータに関して命令の作成者によって提供され、セグメント仕事量ベクトルを形成するために追加される。

【0244】

多くの場合、出力の単一要素を計算する仕事量は、計算される要素の総数に関連付けられる。この場合および実行時に、「単一要素の仕事量」および「出力要素の数」という要因を使用した単一の乗算演算により、特定の入力データ構造に対する全体的なセグメント実行の仕事量が推定される。多くの場合、セグメント仕事量ベクトルはまばらであり単一要素、つまりサイズ記述子内の要素数の情報のインデックスに対応する要素のみが非ゼロである。

20

【0245】

図8に示す例示的な実施形態では、単一要素を作成する仕事量は、配列仕事量要素 $SEV0 \sim SEV2$ のインデックス1（第2のインデックス値）に格納され、これは配列の要素数がそれぞれ配列サイズ記述子と SSV に保存される位置に対応する。ここで、格納されている値6は、 $Add()$ 命令の単一要素の仕事量1と $Sin()$ 命令の単一要素の仕事量5を加算した結果である。セグメントサイズベクトルの他の要素は0であり、これは、そのセグメント仕事量が実行時の入力データ構造内の要素の総数の単独で計算できることを示す。

30

【0246】

他のセグメントは、例えば集約命令 $Sum()$ などの他の命令を実装するかもしれない。そのような命令がセグメントに含まれる場合、対応する命令の仕事量は、入力データ構造の次元数または個々の次元の長さなど、他のサイズ情報に基づいて計算される。

【0247】

サイズ変更命令、すなわち、入力引数とは異なるサイズの出力配列を生成する命令の場合、命令仕事量情報を事前に集約することは容易ではないことが多い。代わりに、単一の出力要素を作成するための仕事量は、実行時にセグメントの個々の命令に提供される入力/出力データ構造インスタンスに対応するサイズ情報に依存する。したがって、これらのセグメントに対して仕事量評価ツリーを作成できる。仕事量評価ツリーのノードには、後で評価するために関係する命令の仕事量情報が格納されている。

40

【0248】

実行時に仕事量評価ツリーを使用して、特定の入力データ構造インスタンスのセットのセグメントのセグメント仕事量ベクトルを収集することができる。したがって、ツリーのノードに格納されている命令仕事量関数と命令サイズ関数は、データ構造インスタンスの具体的なサイズ（入力および中間引数のサイズ）を考慮して適切に評価される。

【0249】

このように、仕事量評価ツリーにより、実際にその演算をその処理ユニットで実行する

50

必要もなく、関連するメタデータに基づいて特定のデータ構造インスタンスを持つ特定の処理ユニットでセグメントの数値演算を計算するために必要な仕事量を予測できる。

【0250】

通常、命令作業ベクトルは、各デバイスカテゴリの命令作成者によって個別に維持される。これは、個々のデバイスが同じ演算を異なる方法で実装する可能性があること、または同じ演算が個々のデバイスカテゴリで異なる実行コストを引き起こすという事実を考慮に入れている。

【0251】

仕事量推定結果の精度を改善するために、メモリ転送コストおよび処理ユニットの周波数に関連するデータなど、さらなるデータが考慮される。

【0252】

コンパイラ段階後のセグメント仕事量ベクトルは、単一の要素を作成する仕事量を表すかもしれないが、セグメント仕事量ベクトルは、さらなるデータを考慮に入れることによってさらに洗練されるかもしれない。図9は、2つの処理ユニット（CPUとGPU）で使用可能なコアの数に関する情報とともに、セグメント仕事量ベクトルの採用を示している。

【0253】

起動段階（例えば起動時）IIにおいて、利用可能な処理ユニットのコアの数決定され、それに応じて所定のセグメント仕事量ベクトルSEV0～SEV2が更新される。例示的な実施形態では、CPUに対して検出された4つのコアにより、同時に4つの命令の実行が可能になる。したがって、所定の対応するセグメント仕事量ベクトルSEV1の要素（値0,6,0,0,0、図（8）を参照）は4で除算される。同様に、GPUには96個のコアがあり、GPUのセグメント仕事量ベクトルSEV2は96で除算される（または約0.01で乗算される）。それに応じて、CPUでセグメントを順次/スカラー方式で実行するための仕事量を表す所定のセグメント仕事量ベクトルSEV0（明確化のため図9には示されていない）は、ここで更新される場合とされない場合がある。

【0254】

実施形態によれば、セグメント仕事量情報を作成およびまたは更新するためにさらなるおよびまたは他のデータが考慮され、およびまたはデータ構造インスタンスのサイズ記述子内の要素の数以外の要素を使用できる。たとえば、カーネル実行トリガーオーバーヘッド、CPUベクトル拡張ユニットの長さ、デバイス間のメモリバス転送速度、コアグループ情報、およびまたは実際のまたは意図したデバイス使用率など、デバイス固有のプロパティをさらに考慮できる。

【0255】

また、さらなるサイズ情報を使用して、仕事量情報をセグメントの実行のためにインスタンス化された実際のデータ構造に関連付けることができる。たとえば、定数係数がサイズ記述子に含まれるか、計算された仕事量に追加される。または、上記のサイズ情報に加えて、次元の数、要素の数、次元の長さ、そのようなまたはさらなる数の整数または小数の累乗を使用して、データ構造インスタンスのサイズを記述できる。たとえば、データ構造のさらなる性質を記述するデータ構造の要素の値、対称性情報、データ範囲情報、ブール値または整数のフラグなど、データ構造インスタンスがセグメントの仕事量に及ぼす影響に関連する情報は、セグメント仕事量評価内で使用できる。

【0256】

コンパイル段階では、通常、デバイスカテゴリごとに1つのカーネルテンプレートが作成される。カーネルテンプレートは、セグメントの命令が意図する数値演算を実装する。

【0257】

そのようなテンプレートは、実際の実行ユニットが既知である場合、起動段階でカーネルコードを効率的に実装および適合させるために使用され得る。適合の例には、キャッシュサイズ、キャッシュライン長、SIMDベクトル拡張ユニットの長さ、サポートされる機能セット、デバイスでサポートされる精度、周波数、コア数などのデバイスプロパティの考慮が含まれる。そのような情報は処理ユニットでの実行を高速化するためカーネルそして

10

20

30

40

50

またはカーネルテンプレートの適合に使用され得る。

【0258】

この方法を数値アルゴリズムの広範囲の用途で有用にするために、方法のオーバーヘッドを可能な限り小さく保つことが望ましい。これにより、損益分岐点、さらに特定すれば補助データデバイスで実行することによるメソッドの利点がメソッド自体によって追加されるオーバーヘッドを超える配列データインスタンスのサイズが、シフトダウンされる。

【0259】

本明細書に記載された方法は、合理的な低い数ですでに有益である。このようにして、例えば、比較的小さな配列サイズでも並列の可能性を利用できる。

【0260】

他のコンピューティング分野にも当てはまるように、メモリ転送は現代の（フォンノイマン）コンピュータアーキテクチャの主要なボトルネックの1つである。したがって、デバイス間のデータバッファ交換は慎重に処理する必要があり、デバイス上のメモリ管理は特に効率的である必要があり、この入力データ構造は以下に説明する特定の設計上の利点で本発明をサポートする。

【0261】

この点で、配列（データ構造インスタンス）が現在の記憶場所を追跡すると有利である、例えば、配列は、その要素がコピーされたデバイスバッファの場所を保存できる。この方法では、配列に格納された情報が複数のデバイスメモリに同時に存在する場合がある。このようなデバイスはすべて、ほとんどオーバーヘッドなしで配列にアクセスできる。

【0262】

コンパイラは、その引数の意図に合わせてセグメント実装を最適化することにより、処理ユニットでのメモリ管理をサポートすることを決定する場合がある。これにより、「意図」とは、引数のライフタイムの少なくとも1つと交換可能性を指します。したがって、セグメント引数の意図の間の区別を言語に組み込むことができる（異なるInArray、OutArray、LocalArray、およびReturnArrayタイプによって）。不変性（読み取り専用入力配列）と揮発性（自己破壊的な戻り配列型）を利用すると、配列のコピーが節約され、最初から確定的なオブジェクトの破壊をサポートしていない言語でも。配列ストレージの早めの決定論的な廃棄が実現する。

【0263】

デバイスバッファーストレージは、入力配列引数にオンデマンドで割り当てられ、配列のストレージ自体と同じ有効期間を取得できる。デバイスにコピーされると、配列が破棄または変更されるまで、バッファはそこに残る。これにより、潜在的な後続のセグメント呼び出しでの配列（バッファ）の安価な再利用が可能になる。

【0264】

セグメントの出力引数は通常、配列の他のすべての保存場所を無効化（破棄/解放）し、新しいバッファが配列の唯一の保存場所になる。

【0265】

解放されたバッファは、後で再利用（プーリング）するためにデバイス上に残る場合がある。

【0266】

配列データストレージは、通常、配列サイズ記述子とは別に維持される。これにより、同じソース配列のサブ配列間で（大きな）データストレージを共有でき、個々のサイズ記述子（「ビュー」とも呼ばれる）のみが異なる。

【0267】

配列は、ストレージの怠惰なコピーのみを作成できる（「書き込み時の怠惰なコピー」）。

【0268】

さらに、実行時セグメントは、それをサポートするデバイス上で非同期的に実行される場合がある。通常、一度に1つのデバイス（多くの場合、ホストコントローラー）は、実

10

20

30

40

50

行時セグメントに最適な処理ユニットを決定し、およびまたは実行時セグメントを決定された処理ユニットに非同期でキューする。つまり、制御デバイスは、メソッドが次の実行時セグメントに進むまで、実行時セグメントの計算カーネルの実行が終了させるのを待たない。

【0269】

図10は、配列オブジェクトなどの一般データタイプの複数の要素を保存できるデータ構造の作成、アクセス、解放、および計算を含む複数の実行時セグメントおよび命令の非同期実行に関連する方法のステップを示している。

【0270】

一実施形態によれば、サポートされる各デバイスの配列オブジェクト、すなわちバッファハンドルおよび同期ハンドルに2つの追加アイテムが格納および維持される。

【0271】

バッファハンドルは、該当する場合、デバイス固有のメモリ上の配列データ（データ構造インスタンス）のバッファへの参照を格納している。バッファハンドルのタイプは、デバイスのタイプおよびまたはデバイスへのアクセスに使用されるインターフェイスによって異なる場合がある。

【0272】

同期ハンドルを使用して、特定のデバイスのバッファストレージへのアクセスを同期できる。

【0273】

同期ハンドルは、デバイス固有のインターフェイス（OpenCL、MPIなど）から取得するか、ホストアプリケーションコードによって作成および管理できる。

【0274】

図10に示される例示的な実施形態では、2つの処理ユニットが存在する、すなわち、管理ヒープとしてメモリを実装するCPUホストデバイス（デバイス0）および専用GPUメモリを有するGPU（デバイス2）である。

【0275】

すべてのデバイスはデバイスの共通インターフェイスを介して利用される可能性があるため、実際のタイプのデバイス（処理ユニット）は重要でない場合があることに注意されたい。たとえば、OpenCLでは、GPUメモリと同じ方法でCPUメモリを処理できる。同様に、個々の共有メモリデバイスは同じハードウェアメモリにアクセスできる。それでも、全体的な機能は同じであるが、ただし、図10の一部の呼び出しは不要であり、パフォーマンス上の理由で省略できる。

【0276】

同様に、デバイスに割り当てられた位置インデックスは選択の問題である。図10では、CPUにインデックス0が割り当てられ、GPUにインデックス2が割り当てられているが、サポートされているデバイスのセットの（デバイス）ロケーションインデックスは、このスキームに影響を与えずに異なってもよい。

【0277】

起動時（起動フェーズ）、サポートされているデバイスが特定され、通常、固定インデックスが各デバイスに割り当てられる。このインデックスは、プログラム実行全体で使用され、ロケーション配列内のデバイスを特定する。

【0278】

さらに、デバイス0とデバイス2のロケーションスロットのみが図に示されていることに注意されたい。システム上の他のデバイスに対応する他のスロットは、明確にするために省略されている。

【0279】

図10は、メモリ管理と同期に焦点を当て、入力配列Aの以下の配列命令を非同期的に計算するために使用されるアクションの概略図を示している。

[1] {

10

20

30

40

50

```

[2]   IArray <double> A = rand(1000,2000);
[3]   ...
[4]   B = Abs(Sin(A * A));
[5]   var c = A.GetArrayForRead();
[6]   ...
[7] }

```

【0280】

行[1]および[7]は、ライフタイムスコープを配列Aに関連付けるコードブロックを確定する。配列スコープは、配列ストレージの決定論的な破壊、解放、および再利用を可能にするため、以下の考慮事項に含まれる。これは、例えば、ほとんどの状況でガベージコレクター（GC）による非決定論的なメモリー管理より優れていると期待できる。なぜならそれはメモリをタイムリーに決定論的に解放および再利用でき、ガベージコレクションのオーバーヘッドと関連コストを節約するためである。また、明示的なスコープは、ホストがシングルスレッドモードで動作できるようにすることでメモリ管理を簡素化し、ロックまたはその他の同期スキームによって生じる同期オーバーヘッドを節約する。一部のGCの実装は、マルチスレッド化された非決定的なオブジェクトの破壊やファイナライズのため、ここで干渉する可能性がある。この説明では、使用される言語は、それぞれオブジェクトのスコープ情報とライフタイム情報を提供できると想定されている。

10

【0281】

配列Aが現在のブロックの範囲を離れるとすぐに、配列Aは廃棄の準備ができているとみなされる。

20

【0282】

[2]行では、配列Aは、ホストコントローラー上で例示的な1000行2000列の乱数で満たされたマトリックスとして作成される。コンパイラーは、この配列命令を（単一命令）セグメントとして識別し、実行時にGPU（デバイス2）で乱数を計算することが決定される。

【0283】

この時点で、GPUのメモリは大きなマトリックスAの要素にまだ割り当てられていない。代わりに、GPUデバイスは、専用デバイスメモリ上のマトリックスAにメモリを割り当てるように指示される。この命令は非同期に行われ、ホストは、デバイスが実際にバッファを作成するのを待たずに、後で非同期操作を識別するために使用される同期ハンドルをデバイスから受け取る。ホストコントローラーは後続の操作をすぐに続行するが、デバイスのバッファにアクセスする今後のすべての試行で同期ハンドルが使用される。このようなアクセス試行は、バッファのアクセスを実行する前に、同期ハンドルが「準備完了状態」に切り替わるのを待機する。同期ハンドルは、配列の保存場所の対応するデバイススロットに保存される。

30

【0284】

同期ハンドルは、単純な共有アトミック（ユニバーサルまたはメイン）カウンターである。あるいは、ホストフレームワークによって提供される、およびまたはオペレーティングシステムまたはシステムの他の関連する実行層によってサポートされる、より洗練された同期オブジェクトである。

40

【0285】

さらに、バッファへのポインタが提供され、GPUデバイスに対応する配列Aの同じデバイススロットに格納されてもよい。バッファハンドルは、デバイス上の実際のストレージにアクセスするために使用される。そのタイプは、デバイスへのアクセス/インターフェースに使用されるテクノロジーに依存する（OpenCL、MPI、OpenMPなど）。バッファハンドルは、同期ハンドルを提供した同じ呼び出しで提供される。または、デバイスからバッファポインタを照会するために、synchハンドルを使用した後の呼び出しで提供される。または、バッファの作成はすべて同期方式で実装できる。

【0286】

バッファが確立されるとすぐに（バッファの作成がまだ完了しているかどうかはま

50

だわからない場合でも）、選択された計算カーネルの実行がトリガーされます。そのようなトリガーは、処理ユニットにアクセスするために使用される低レベルのインターフェースによって提供されるように、コマンドキューなどで対応するコマンドをキューすることにより実行される。

【0287】

通常、同期ハンドルとバッファハンドルの両方がカーネルに提供されます。この場合も、呼び出しは非同期に実行され、それは呼び出しに関連付けられた（潜在的に時間のかかる）アクションの完了を待たずに戻る。計算カーネルは、最後の操作が終了せず、バッファにまだアクセスできない場合に備えて、同期ハンドルで待機する。計算カーネルでのこの待機および潜在的に高価で時間のかかる計算は、ホストプロセッサスレッドを遅延させることなく実行される。

10

【0288】

計算カーネルの非同期実行をトリガーする呼び出しは、ホストに2番目の同期ハンドルを提供する。2番目の同期ハンドルは、後でカーネル操作を識別するために使用される。両方の非同期操作がどのようにチェーンされているかに注意されたし。カーネルの実行はバッファの作成を待機する。したがって、返される2番目の同期ハンドルは、以前のすべてのコマンドの完了後にデバイスバッファの準備完了状態を識別するのに適している。したがって、プログラムのこの時点ではもはや不要な最初の同期ハンドルを置き換えることにより、2番目の同期ハンドルを保存するだけで十分である。

【0289】

20

他の実施形態では、同期ハンドルは連鎖可能でないことがある。各操作では、新しいハンドルを生成する必要がある。同期は、低レベルのデバイスインターフェースによるサポートなしで実行できる。これらの実施形態では、上記のコマンドに関連付けられた同期情報を格納および管理するために、キューまたはキューに類似したデータ構造を実装する必要がある。その後、各デバイスの同期ハンドルの単一のストレージスロットが、キューまたは別のコレクションへの参照になり得る。ただし、この場合、より高い管理オーバーヘッドが予想される。

【0290】

rand(1000,2000)（図10のseg_001）の計算がトリガーされた後、ホストは呼び出しによって返された同期ハンドルをデバイス2の配列格納場所スロットに格納し、すぐに後続の操作を続行する。ストレージロケーションスロットに同期ハンドルが存在することにより、配列のストレージを利用する操作がまだ進行中であることを配列の他のコンシューマに通知する。同期ハンドルの存在を考慮する必要がある演算には、出力バッファとしてのバッファへのアクセス（つまり、カーネルによるバッファの変更）とバッファの解放が含まれる。

30

【0291】

上記のコードリストの行[4]によれば、次の操作は同じ配列Aに関係します。ホスト命令カウンタが対応するセグメントに到達するとき（これがセグメントseg_042であると仮定しましょう）、完了の状態 seg_001決定されていません。

【0292】

40

したがって、ホストはデバイス2の配列ロケーションスロットに格納されたバッファハンドルと同期ハンドルを提供することにより、新しいセグメントの実行を以前の実行と同期する。カーネルはすぐに新しい同期ハンドルを返し、実行を開始する前に提供された同期ハンドル（存在する場合）を待つ。同期ハンドルは一般にオプションであることに注意する必要がある。スキーム全体または個々のデバイスの任意の組み合わせも、同期して機能するか、同期のための他のスキームを実装するかもしれない。わかりやすくするために、ここではseg_0042の出力が省略されていることに注意されたし。

【0293】

上記のコードリストの[5]行によれば、次の操作は、配列要素を1Dシステム配列オブジェクトとして表すローカルホスト配列を要求する。AのデータはこれまでGPUデバイスに

50

のみ存在するため、Aの2つの要素をホストデバイス0にコピーする必要がある。このコピーも、非同期で実行される。新しい同期ハンドルは、コピーコマンドからデバイス0に直ちに返されます。今回は、デバイス0の新しいバッファに対応している。上記の配列バッファの作成と同様に、バッファハンドルはすぐに返されるか、後で要求/保存される。あるいは、データのコピーが、明示的なバッファ作成コマンドによって先行される場合もある。

【0294】

バッファハンドルは、デバイス2のバッファからのみ読み取られることに注意されたし。デバイス2のバッファは変更されないため、このステップではデバイスロケーション2の新しい同期ハンドルは必要がない。

【0295】

以下の命令がホスト上で実行され、A.GetArrayForRead () から返されたシステム配列にアクセスすることを想定している。したがって、すべての要素がすでにデバイス0バッファにコピーされていることが保証される。

【0296】

この目的のために、デバイス保存場所0の同期ハンドルによってコピー操作の完了を待つことができるメモリバリアを導入することができる。そのようなメモリバリアは一般的な動機方法から去られているのと同じように動作し、また、任意のデバイスでの操作の完了を同期的に待機するために使用できる。たとえば、バッファを解放しようとするときに、同様の待機メカニズムを使用できる。

【0297】

最後に、配列Aは現在のコードブロックの範囲を離れる。Aはその後参照されなくなるため、Aに関連付けられたストレージはできるだけ早く破棄され、(制限された) デバイスメモリで消費されたメモリ領域が解放される。このような廃棄では、同期ハンドルの存在が考慮されます。このメソッドは、メモリを解放する前に、暗黙的なメモリバリアを使用する場合がある(図10のWaitSynchHandle () メソッドと同様)。または、デバイスインターフェイスがサポートしている場合は、非同期リリース操作を使用できる。

【0298】

このメソッドは、配列オブジェクトとデバイスバッファの間に1対1の関係を実装してもしなくてもよいことに注意されたし。最も効率的なメモリ管理のために、複数の配列が同じバッファストレージを利用/参照できるようにすることが望ましい場合がある。例えば「遅延コピーオンライト」スキームがそうである。したがって、配列を廃棄しても、そのバッファがすぐに解放されない場合がある。代わりに、参照カウンタを使用して、共有ストレージオブジェクトの参照カウンタをデクリメントできる。参照カウンタがこのストレージを参照している他の配列が存在しないことを示すとすぐに、バッファは解放される。

【0299】

非同期実行は、異種コンピューティングシステムのデバイスをビジー状態に保ち、既存のコンピューティングリソースをより効率的に利用し、したがってシーケンスの実行をより早く終了するか、より少ないエネルギーを消費するのに役立つ。

【0300】

デバイス上で実行するために実行時セグメントをキューに入れるとき、セグメントのコストの測定値が(とにかく) 計算される。この指標は、キューで「待機中」の命令の数の表現としても使用できる。現在キュー内で実行を「待機」しているすべてのセグメントの累積測定値は、デバイスキュー内の「先」の操作に関する指標となる。

【0301】

したがって、各アイテムがアイテム内の(数値) 演算の個々の数に対応するという事実も考慮することができる。

【0302】

「最適な」デバイスの決定は、このコスト「先」に基づいてもよい。デバイスのキュー

10

20

30

40

50

に多くのセグメントがある場合、データバッファコピーのコストを別のデバイスに投資し、代わりに他のデバイスでセグメントを実行またはキューの方が良い場合がある。

【0303】

さらに、実行時セグメントを任意の順序で実行するように、低レベルインターフェイス（OpenCLなど）を構成できる。低レベルのデバイスインターフェースまたは高レベルの管理層が、セグメントが元々シーケンスに並んでいた順序とは異なる順序がパフォーマンスまたはその他の観点から有利であると判断し、順序を交換しても負の副作用が生じない場合シーケンスの結果、実行の順序が変更される場合がある。

【0304】

実行順序を並べ替えると、複数のデバイスでいくつかの実行時セグメントを同時に実行できる場合がある。たとえば、セグメントのシーケンスを並べ替えると、互いの結果に依存するセグメントをグループ化できる。また、同じまたはほとんど同じ引数/データバッファを必要とするこのようなセグメントをグループ化できる。グループ化は、データバッファの現在のローカリティに基づいて行うこともできる。これにより、このようなセグメントのグループが複数のデバイスで効率的に同時に実行される可能性が高くなる。たとえば、再配置された実行時セグメントの最初のグループを最初のデバイスで非同期に実行し、2番目のグループを形成する後続の実行時セグメントを非同期実行のために2番目のデバイスのキューに入れることができる。グループ間のデータ依存性は低く抑えられているため、両方の実行時セグメントグループを少ない同期作業で実行できる。

【0305】

したがって、この方法は、デバイスの多くまたは全てさえも有利な程度までビジーに保つことができる。したがって、計算リソースの利用効率が向上する。

【0306】

一実施形態によれば、コンピュータ実装方法は、プログラムコード内のプログラムセグメント、通常はプログラムセグメントのシーケンスを識別することを含む。プログラムセグメントは、第1の一般データタイプの複数の要素および計算命令の出力に関連するデータを含む関数メタ情報を格納できる第1のデータ構造のための計算命令、典型的には数値演算を含む。プログラムセグメントのそれぞれのコンパイルされた表現（典型的には計算カーネル）を実行するのに適した計算システムの第1処理ユニットおよび第2処理ユニットが決定される。第1の処理ユニットは、第2の処理ユニットとは異なる。第1処理ユニットは、プログラムセグメントの第1コンパイル済み表現で初期化され、第2処理ユニットは、プログラムセグメントの第2コンパイル済み表現で初期化される。第1データ構造の実行時インスタンスの関数メタ情報とデータメタ情報を使用して、第1処理ユニットで第1コンパイル済み表現を実行する第1予想コストを決定し、2番目の処理ユニットで第2コンパイル済み表現を実行する第2予想コストを決定するために使用される。第1の予想コストおよび第2の予想コストは、それぞれのコンパイルされた表現を実行するための第1の処理ユニットまたは第2の処理ユニットのいずれかを選択するために使用される。通常、1番目と2番目のコストは数値的に決定される（数値的に計算される）。

【0307】

あるいは、第1の予想コストおよび第2の予想コストを使用して、それぞれのコンパイルされた表現を実行するための第1の処理ユニットおよび第2の処理ユニットの一方のみを選択するか、第1処理ユニットおよび第2処理ユニットでコンパイルされた表現それぞれを実行するワークロードのそれぞれのシェアを決定することができる。

【0308】

計算命令が実行時インスタンスの一部で実行するのに適している場合、ワークロードの共有が引き起こされる。これは特に、要素ごとに、またはデータの分離された領域で実行できる命令を計算する場合である。例には、行列の追加、 $\sin(A)$ などの行列Aの関数の計算、データの平滑化などの数値演算が含まれる。

【0309】

ワークロードの共有は通常、実行時インスタンスの第1部分を第1処理ユニットに割り

10

20

30

40

50

当て、実行時インスタンスの第2部分を第2処理ユニットに割り当てることにより達成される。ワークロードは、予想コストに応じて、たとえば予想コストに反比例して、およびまたは予想実行時間で共有できる。予想される実行時間は、予想されるコストと、処理ユニットの実際の可用性またはワークロードから導き出すことができる。

【0310】

通常、ワークロードは、実行時インスタンスの実行時サイズが十分に大きい場合、およびまたはシステムで利用可能な少なくとも1つの処理ユニットの使用率が他の処理ユニットに比べて低いことが判明または予想される場合に共有される。後者の場合、実行時セグメント、実行時のコンパイラ、またはスケジューリングモジュールは、使用率の低い処理ユニットを、使用率の高い1つ以上のデバイスと共有して、使用中のデバイスから十分に活用されていないデバイスワークロードの少なくとも一部を分散することを決定する場合がある。

10

【0311】

共有デバイスは、いわゆる「仮想デバイス」を形成する場合があります、非同期実行および順不同実行が処理ユニットで利用できない場合に特に役立つ。これは、処理ユニットのアクセスを処理する基礎となる低レベルインターフェイスがそのような機能をサポートしていない、およびまたは命令が依存関係を示しすぎて命令/セグメントの並べ替えおよびまたは複数のデバイスでのセグメントの分散が可能でないか使用することができない場合である。

【0312】

さらに、データ構造のインスタンスが複数のデバイス（分散メモリ）のメモリにわたる場合、仮想デバイスにより、そのような分散インスタンスを有するセグメントの実行に関与するデバイスを本発明に従って使用することができる。これにより、仮想デバイスは処理ユニットと見なされ、上記のように処理される。

20

【0313】

仮想デバイスが実行シーケンスのパフォーマンスに有利な別のシナリオは、共有メモリデバイスで、たとえば、GPUデバイスとメモリを共有するCPUです。ここで、CPUコアのすべてまたは一部とGPUコアのすべてまたは一部を含む仮想デバイスは、基礎となる個別のデバイスよりも高い計算能力を示す場合がある。（共有メモリによる）転送コストが関係しないため、基盤となるデバイスのいずれかを仮想デバイスに置き換えると、実行パフォーマンスが向上する。

30

【0314】

一実施形態によれば、コンピュータ可読媒体は、コンピュータによって実行されると、そのコンピュータに、上述の方法のいずれかおよびまたは少なくとも部分的にはAPI（アプリケーションプログラミングインターフェイス）として実装される以下のプロセスの少なくとも1つを、実行させる命令を記憶する。

ソフトウェアコード（通常、GPLまたはMATLABやNumPyなどの科学計算言語で記述されたユーザーコード）を取得し、そのソフトウェアコードで、一般データの複数の要素を格納できる最初のデータ構造の数値演算を含むプログラムセグメントを識別すること、

数値演算の出力、特に出力のサイズ、出力の構造、およびまたは出力を生成するための仕事量を参照する関数メタ情報を決定すること、

40

数値演算を実行するのに適したコンピュータまたは別の異種コンピューティングシステムの処理ユニットを決定すること、そして

プログラムセグメントを中間表現、特にバイトコード表現、またはコンピュータまたは他の異種コンピューティングシステムで実行可能な表現、特にプログラムセグメントの実行時セグメントに変換すること。

【0315】

通常、実行時セグメントには、関数メタ情報、決定された各処理ユニットの計算カーネル、および関数メタ情報を使用して最初のデータ構造の実行時インスタンスで計算カーネルを実行するための予想コストを決定する実行可能コード、そして、最初のデータ構造の

50

実行時インスタンスのデータメタ情報が含まれる。そのデータメタ情報は、実行時インスタンスの実行時サイズ情報、実行時インスタンスの実行時ロケーション情報、および実行時インスタンスの実行時タイプ情報を含む。

【0316】

関数メタ情報を決定することは、それらが既にソフトウェアコードに保存されている場合、関数メタ情報を読み取ること、加算、サイン、FFT（高速フーリエ変換）などの（すでに）既知の数値演算の関数メタ情報を保存するルックアップテーブルで検索すること、およびまたはテスト実行に基づいて関数のメタ情報を導出することを含み得る。

【0317】

コンピュータ実装方法の一実施形態によれば、この方法は、第1の計算カーネルを用いて異種計算システムの第1の処理ユニットを初期化し、第2の計算カーネルを用いて異種計算システムの第2の処理ユニットを初期化することを含む。第1計算カーネルと第2計算カーネルの両方は、一般データ型の複数の要素を格納する第1データ構造を受け取るように構成されたプログラムセグメントから派生した数値演算を実行するように構成される。プログラムセグメントは、数値演算の出力のサイズ、出力の構造、およびまたは出力を生成する仕事量に関連するデータを含む関数メタ情報を含む。第1データ構造の実行時インスタンスの関数メタ情報とデータメタ情報は、第1処理ユニットで実行時インスタンスで数値演算を実行するために第1カーネルを実行する第1の予想コストを決定し、そして第2処理ユニットで実行時インスタンスで数値演算を実行するために第2カーネルを実行する第2予想コストを決定するために、使用される。データメタ情報は、実行時インスタンスの実行時サイズ情報、実行時インスタンスの実行時ロケーション情報、実行時インスタンスの実行時同期情報、および実行時インスタンスの実行時タイプ情報のうちの少なくとも1つを含む。この方法は、第1の予想コストが第2の予想コスト以下である場合、実行時インスタンスで数値演算を実行するために、第1の処理ユニットで第1の計算カーネルを実行すること、および第1の予想コストが第2の予想コストよりも高い場合、実行時インスタンスで数値演算を実行するために第2の計算カーネルを第2処理ユニットで実行すること、の1つをさらに含む。

【0318】

コンピュータ実装方法の一実施形態によれば、方法は、第1処理ユニットおよび第2処理ユニットを含む異種コンピューティングシステムを、一般的なデータ型の複数の要素を格納する第1データ構造で数値演算を実行するための実行時セグメントで初期化することを含む。実行時セグメントは、第1処理ユニットで数値演算を実行するように構成された第1計算カーネル、第2処理ユニットで数値演算を実行するように構成された第2計算カーネル、および第1処理ユニット上の一般データ型の1つの要素の数値演算を実行するための第1数値仕事量と、第2処理ユニット上の一般データ型の1つの要素の数値演算を実行するための第2数値仕事量を含む関数メタ情報を含む。異種コンピューティングシステムで、最初のデータ構造の実行時インスタンスが作成される。実行時インスタンスのデータメタ情報が決定される。このデータメタ情報は、実行時インスタンスの実行時サイズ情報と実行時インスタンスの実行時ロケーション情報を含む。データメタ情報と関数メタ情報は、実行時インスタンスで数値演算を実行するために最初の処理ユニットで最初の計算カーネルを実行する第1の予想コストを計算するために、そして実行時インスタンスで数値演算を実行するために2番目の処理ユニット上で2番目の計算カーネルを実行する第2の予想コストを計算するために使用される。最初の予想コストが2番目の予想コスト以下である場合、実行時インスタンスで数値演算を実行するため第1の計算カーネルが第1の処理ユニットで実行され、そうでない場合、実行時インスタンスの数値演算を実行試してに2番目の計算カーネルが2番目の処理ユニットで実行される。

【0319】

関数メタ情報は、さらに数値演算の出力のサイズに関係するデータ、およびまたは出力の構造に関係するデータを含む。

【0320】

10

20

30

40

50

第1および第2の予想コストを計算する前に、関数メタ情報、特に第1および第2の数値的仕事量は、通常、第1の処理ユニットの決定された特性および第2の処理ユニットの決定された特性に従って更新される。

【0321】

さらに、第1および第2の予想コストを計算する前に、第1の処理カーネルの決定された特性および第2の処理ユニットの決定された特性に従って、第1の計算カーネルおよびまたは第2の計算カーネルを更新する。

【0322】

実行時セグメントは、通常、第1および第2のコストを計算するための制御コード、およびまたは第1および第2のコストに基づいてどちらの処理ユニットでそれぞれの計算カーネルを実行するかを動的に決定するための制御コードを含む。

【0323】

実行時セグメントは、数値演算を定義するプログラムセグメントから導出されてもよい。

【0324】

本発明の様々な例示的实施形態を開示したが、本発明の精神および範囲から逸脱することなく本発明のいくつかの利点を達成する様々な変更および修正を行うことができることは当業者には明らかであろう。同じ機能を実行する他のコンポーネントが適切に置換され得ることは、当業者には明らかであろう。特定の図を参照して説明されている機能は、明示的に言及されていない場合でも、他の図の機能と組み合わせることができることを述べておく。本発明の概念に対するそのような修正は、添付の特許請求の範囲によってカバーされることを意図している。

【0325】

本明細書で使用する用語「有する」、「含有する」、「含む」、「構成される」などは、述べられた要素または特徴の存在を示すが、追加の要素または特徴を排除しない自由端の用語である。冠詞「a」、「an」、および「the」は、文脈からそうでないことが明確に示されていない限り、単数形だけでなく複数形も含むことを意図している。

【0326】

上記の範囲の変形および用途を念頭に置いて、本発明は前述の説明によって制限されず、添付の図面によっても制限されないことを理解されたい。代わりに、本発明は、以下の請求項およびそれらの法的同等物によってのみ制限される。

10

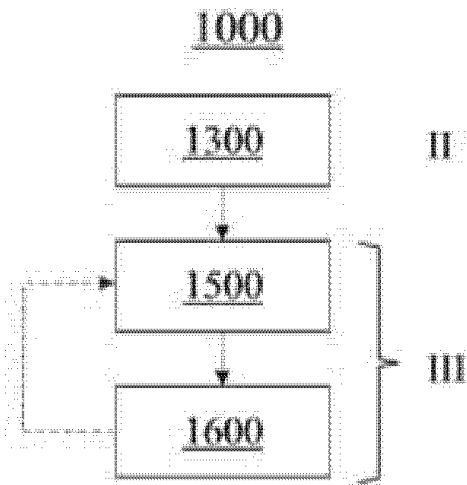
20

30

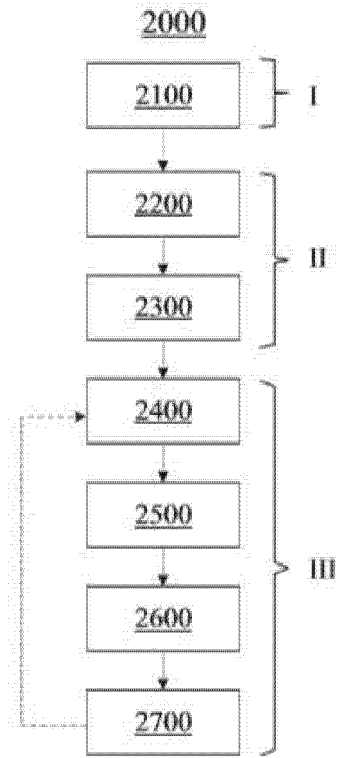
40

50

【図面】
【図 1】



【図 2】



10

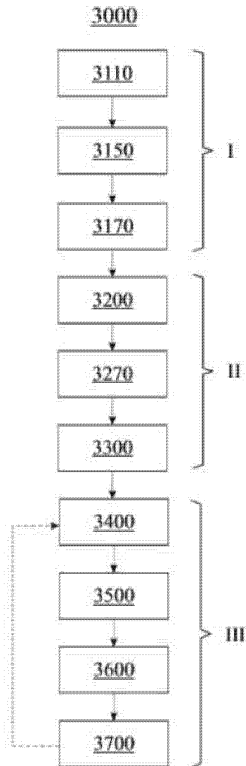
20

30

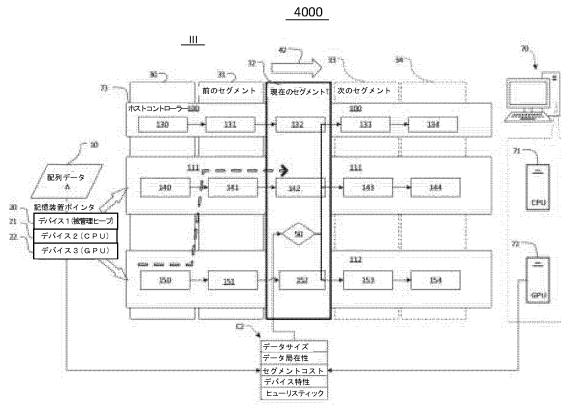
40

50

【図 3】



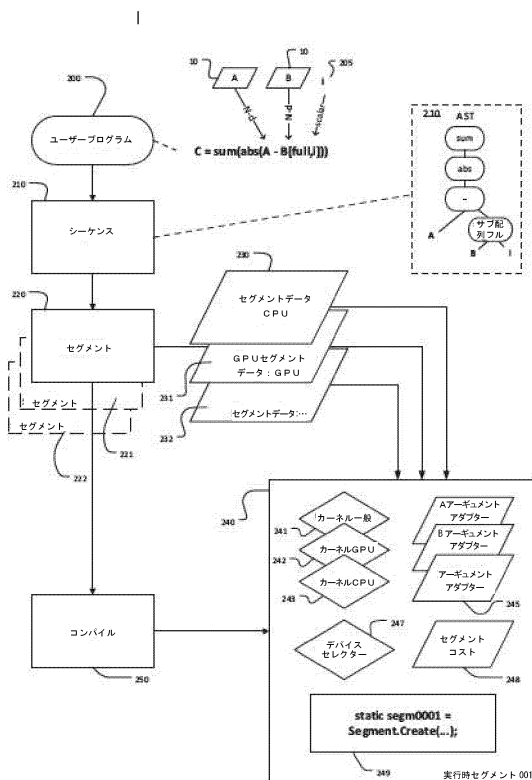
【図 4】



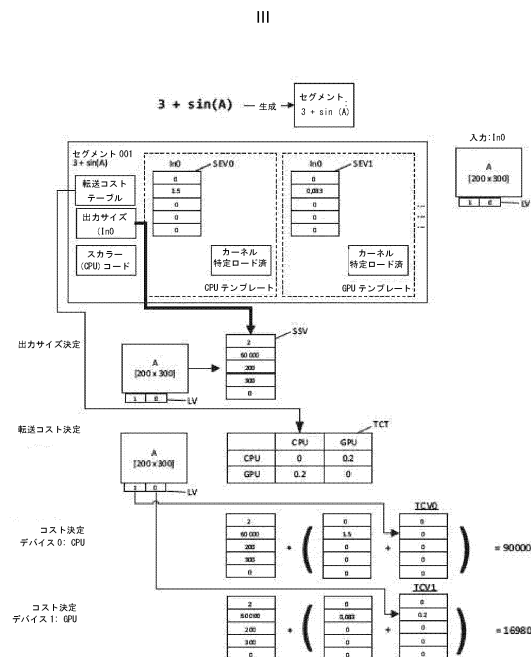
10

20

【圖 5】



【図 6】

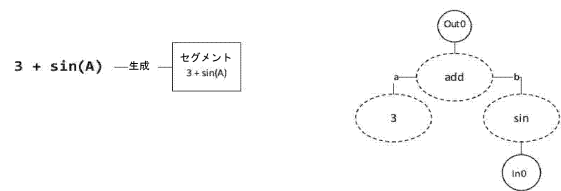


30

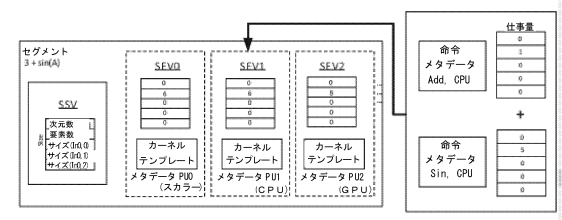
40

50

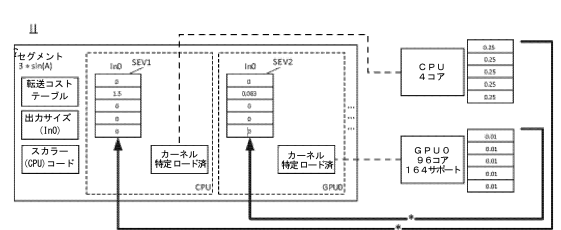
【図 7】



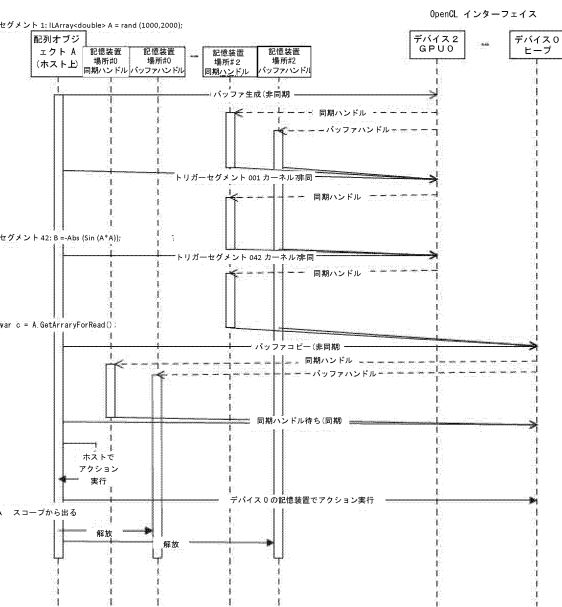
【図 8】



【図 9】



【図 10】



10

20

30

40

50

フロントページの続き

- (56)参考文献 特開 2014-102683 (JP, A)
 特表 2012-525753 (JP, A)
 特表 2015-509622 (JP, A)
 特開 2007-328415 (JP, A)
 米国特許出願公開第 2011/0173155 (US, A1)
- (58)調査した分野 (Int.Cl., DB名)
 G06F 9/455-9/54
 G06F 17/16