

(12) 特許協力条約に基づいて公開された国際出願

(19) 世界知的所有権機関
国際事務局

(43) 国際公開日
2023年8月3日(03.08.2023)



(10) 国際公開番号

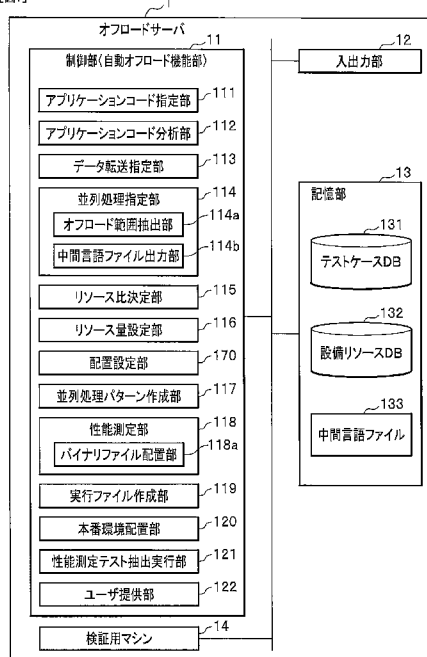
WO 2023/144926 A1

- (51) 国際特許分類:
G06F 11/34 (2006.01) G06F 8/60 (2018.01)
G06F 8/41 (2018.01) G06F 9/50 (2006.01)
- (21) 国際出願番号: PCT/JP2022/002880
- (22) 国際出願日: 2022年1月26日(26.01.2022)
- (25) 国際出願の言語: 日本語
- (26) 国際公開の言語: 日本語
- (71) 出願人: 日本電信電話株式会社 (NIPPON TELEGRAPH AND TELEPHONE CORPORATION) [JP/JP]; 〒1008116 東京都千代田区大手町一丁目5番1号 Tokyo (JP).
- (72) 発明者: 山登 庸次(YAMATO, Yoji); 〒1808585 東京都武蔵野市緑町3丁目9-11 N T 知的財産センタ内 Tokyo (JP).
- (74) 代理人: 特許業務法人磯野国際特許商標事務所 (ISONO INTERNATIONAL PATENT OFFICE, P.C.); 〒1050001 東京都港区虎ノ門一丁目1番18号 ヒューリック虎ノ門ビル Tokyo (JP).
- (81) 指定国(表示のない限り、全ての種類の国内保護が可能): AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, IT, JO, JP, KE, KG, KH,

(54) Title: OFFLOAD SERVER, OFFLOAD CONTROL METHOD, AND OFFLOAD PROGRAM

(54) 発明の名称: オフロードサーバ、オフロード制御方法およびオフロードプログラム

〔図1〕



- 1 Offload server
11 Control unit (automatic offload function unit)
12 Input/output unit
13 Storage unit
14 Machine for verification
111 Application code designation unit
112 Application code analysis unit
113 Data transfer designation unit
114 Parallel processing designation unit
114a Offload range extraction unit
114b Intermediate language file output unit
115 Resource ratio determination unit
116 Resource amount setting unit
117 Parallel processing pattern creation unit
118 Performance measurement unit
118a Binary file positioning unit
119 Executable file creation unit
120 Production environment positioning unit
121 Performance measurement test extraction execution unit
122 User provision unit
131 Test case database
132 Facility resource database
133 Intermediate language file
170 Positioning setting unit

(57) Abstract: This offload server (1) comprises: a performance measurement unit (118) for compiling an application program of a parallel processing pattern, positioning the application program in a device for accelerator verification, and executing a process for performance measurement when offloading to an accelerator has occurred; and a positioning setting unit (170) that, when positioning a converted application program in one of a cloud server, a carrier edge server, and a user edge server on a network in accordance with a condition pertaining to a cost or a response time as designated by

KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY,
MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ,
NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT,
QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL,
ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG,
US, UZ, VC, VN, WS, ZA, ZM, ZW.

- (84) 指定国(表示のない限り、全ての種類の広域保護が可能): ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), ユーラシア (AM, AZ, BY, KG, KZ, RU, TJ, TM), ヨーロッパ (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

添付公開書類：

一 国際調査報告 (条約第21条(3))

a user, calculates and sets the location for positioning of the application program on the basis of a linear planning formula in which device and link costs, a calculation resource upper limit, and a bandwidth upper limit are used as constraint conditions and the cost or response time of the calculation resource is used as an objective function.

(57) 要約：オフロードサーバ（１）は、並列処理パターン of アプリケーションプログラムをコンパイルして、アクセラレータ検証用装置に配置し、アクセラレータにオフロードした際の性能測定用処理を実行する性能測定部（１１８）と、変換したアプリケーションプログラムを、ユーザの指定するコストまたは応答時間の条件に応じて、ネットワーク上の、クラウドサーバ、キャリアエッジサーバ、ユーザエッジサーバのいずれかに配置する際、デバイスおよびリンクのコスト、計算リソース上限、帯域上限を制約条件とし、かつ計算リソースのコストまたは応答時間を目的関数とした線形計画式に基づいて、アプリケーションプログラムの配置場所を計算して設定する配置設定部（１７０）と、を備える。

明 細 書

発明の名称：

オフロードサーバ、オフロード制御方法およびオフロードプログラム

技術分野

[0001] 本発明は、機能処理をGPU (Graphics Processing Unit) やFPGA (Field Programmable Gate Array) 等のアクセラレータに自動オフロードし、変換したアプリケーションプログラム（以下適宜、アプリケーションという）を適切な場所に配置するオフロードサーバ、オフロード制御方法およびオフロードプログラムに関する。

背景技術

[0002] CPU (Central Processing Unit) 以外のヘテロな計算リソースを用いることが増えている。例えば、GPU (アクセラレータ) を強化したサーバで画像処理を行ったり、FPGA (アクセラレータ) で信号処理をアクセラレートすることが始まっている。FPGAは、製造後に設計者等が構成を設定できるプログラム可能なゲートアレイであり、PLD (Programmable Logic Device) の一種である。Amazon Web Services (AWS) (登録商標) では、GPUインスタンス、FPGAインスタンスが提供されており、オンデマンドにそれらリソースを使うこともできる。Microsoft (登録商標) は、FPGAを用いて検索を効率化している。

[0003] サービス連携技術等を用いて、多彩なアプリケーションの創出が期待されるが、更に進歩したハードウェアを生かすことで、動作アプリケーションの高性能化が期待できる。しかし、そのためには、動作させるハードウェアに合わせたプログラミングや設定が必要である。例えば、CUDA (Compute Unified Device Architecture)、OpenCL (Open Computing Language) といった多くの技術知識が求められ、ハードルは高い。OpenCLは、あらゆる計算資源 (CPUやGPUに限らない) を特定のハードに縛られず統一的に扱えるオープンなAPI (Application Programming Interface) である。

- [0004] GPUやFPGAをユーザのアプリケーションで容易に利用できるようにするため下記が求められる。すなわち、動作させる画像処理、暗号処理等の汎用アプリケーションを環境にデプロイする際に、プラットフォームがアプリケーションロジックを分析し、GPU、FPGAに自動で処理をオフロードすることが望まれる。
- [0005] GPUの計算能力を画像処理以外にも使うGP GPU (General Purpose GPU) のための開発環境CUDAが発展している。CUDAは、GP GPU向けの開発環境である。また、GPU、FPGA、メニーコアCPU等のヘテロハードウェアを統一的に扱うための標準規格としてOpenCLも登場している。
- [0006] CUDAやOpenCLでは、C言語の拡張によるプログラミングを行う。ただし、GPU等のデバイスとCPUの間のメモリコピー、解放等を記述する必要があり、記述の難度は高い。実際に、CUDAやOpenCLを使いこなせる技術者は数多くはいない。
- [0007] 簡易にGP GPUを行うため、ディレクティブベースで、ループ文等の並列処理すべき個所を指定し、ディレクティブに従いコンパイラがデバイス向けコードに変換する技術がある。技術仕様としてOpenACC (Open Accelerator) 等、コンパイラとしてPGIコンパイラ (登録商標) 等がある。例えば、OpenACCを使った例では、ユーザはC/C++/Fortran言語で書かれたコードに、OpenACCディレクティブで並列処理させる等を指定する。PGIコンパイラは、コードの並列可能性をチェックして、GPU用、CPU用実行バイナリを生成し、実行モジュール化する。IBM JDK (登録商標) は、Java (登録商標) のlambda形式に従った並列処理指定を、GPUにオフロードする機能をサポートしている。これらの技術を用いることで、GPUメモリへのデータ割り当て等を、プログラマは意識する必要がない。
- このように、OpenCL、CUDA、OpenACC等の技術により、GPUやFPGAへのオフロード処理が可能になっている。
- [0008] しかし、オフロード処理自体は行えるようになっても、適切なオフロード

には課題が多い。例えば、Intelコンパイラ（登録商標）のように自動並列化機能を持つコンパイラがある。自動並列化する際は、プログラム上のfor文（繰り返し文）等の並列処理部を抽出する。ところが、GPUを用いて並列に動作させる場合は、CPU-GPUメモリ間のデータやり取りによるオーバーヘッドのため、性能が出ないことも多い。GPUを用いて高速化する際は、スキル保持者が、OpenCLやCUDAでのチューニングや、PGIコンパイラ等で適切な並列処理部を探索することが必要になっている。

このため、スキルが無いユーザがGPUを使ってアプリケーションを高性能化することは難しいし、自動並列化技術を使う場合も、for文を並列するかわからないかの試行錯誤チューニング等により、利用開始までに多くの時間がかかっている。

[0009] 配置に関して、ネットワークリソースの最適利用として、ネットワーク上にあるサーバ群に対してVN（Virtual Network）の埋め込み位置を最適化する研究がある（非特許文献1参照）。非特許文献1では、通信トラヒックを考慮したVNの最適配置を決定する。しかし、単一リソースの仮想ネットワークが対象で、キャリアの設備コストや全体的応答時間の削減が目的で、個々に異なるアプリケーションの処理時間や、個々のユーザのコストや応答時間要求等の条件は考慮されていない。

[0010] 並列処理箇所の試行錯誤を自動化する取り組みとして、非特許文献2が挙げられる。

非特許文献2は、一度記述したコードで、配置先の環境に存在するGPUやFPGA、メニーコアCPU等を利用できるように、変換、リソース設定等を自動で行い、アプリケーションを高性能で動作させることを目的とした、環境適応ソフトウェアを提案している。併せて、非特許文献2は、環境適応ソフトウェアの要素として、アプリケーションコードのループ文を、GPUに自動オフロードする方式を提案し性能向上を評価している。

[0011] 非特許文献3は、環境適応ソフトウェアの要素として、アプリケーションコードのループ文を、FPGAに自動オフロードする方式を提案し性能向上

を評価している。

非特許文献4は、環境適応ソフトウェアの要素として、GPU等向けに自動変換した後、アプリケーションを実行するリソース量（仮想マシンコアの数など）を、適正化する手法を評価している。

先行技術文献

非特許文献

[0012] 非特許文献1：河島滉太，大歳達也，大下裕一，村田正幸，”分散型モデル予測制御にもとづくスケーラビリティを有する仮想ネットワーク埋め込み手法，” 電子情報通信学会技術報告，vol.115，pp.19-24，2015.

非特許文献2：Y. Yamato, “Study of parallel processing area extraction and data transfer number reduction for automatic GPU offloading of IoT applications,” Journal of Intelligent Information Systems, Springer, DOI: 10.1007/s10844-019-00575-8, Aug. 2019. (Electronic Publishing) Vol.54, No.3, pp.567-584, May 2020. (Print Publishing)

非特許文献3：Y. Yamato, “Automatic Offloading Method of Loop Statements of Software to FPGA,” International Journal of Parallel, Emergent and Distributed Systems, Taylor & Francis, DOI: 10.1080/17445760.2021.1916020, Apr. 2021.

非特許文献4：Y. Yamato, “Arranging resource amount after automatic GPU offloading,” The Ninth International Symposium on Computing and Networking (CANDAR 2021), Nov. 2021.

発明の概要

発明が解決しようとする課題

[0013] 非特許文献1～4では、自動オフロード時の処理時間の短縮を中心に評価している。

GPU、FPGA等のヘテロジニアスなデバイスに処理をオフロードする際に、変換したアプリケーションをユーザ要望（価格、応答時間）を満たし

て動作させることについては提案されていないという課題がある。

[0014] このような点に鑑みて本発明がなされたのであり、GPUやFPGA等のオフロードデバイスに配置できるよう自動変換した際に、変換したアプリケーションをユーザのコストまたは応答時間の要求を満たして最適に配置することを課題とする。

課題を解決するための手段

[0015] 前記した課題を解決するため、アプリケーションプログラムの特定処理をアクセラレータにオフロードするオフロードサーバであって、前記アプリケーションプログラムのソースコードを分析するアプリケーションコード分析部と、前記アプリケーションプログラムのループ文の中で用いられる変数の参照関係を分析し、ループ外でデータ転送してよいデータについては、ループ外でのデータ転送を明示的に指定する明示的指定行を用いたデータ転送指定を行うデータ転送指定部と、前記アプリケーションプログラムのループ文を特定し、特定した各前記ループ文に対して、前記アクセラレータにおける並列処理指定文を指定してコンパイルする並列処理指定部と、コンパイルエラーが出るループ文に対して、オフロード対象外とするとともに、コンパイルエラーが出ないループ文に対して、並列処理するかしないかの指定を行う並列処理パターンを作成する並列処理パターン作成部と、前記並列処理パターンの前記アプリケーションプログラムをコンパイルして、アクセラレータ検証用装置に配置し、前記アクセラレータにオフロードした際の性能測定用処理を実行する性能測定部と、変換した前記アプリケーションプログラムを、ユーザの指定するコストまたは応答時間の条件に応じて、ネットワーク上の、クラウドサーバ、キャリアエッジサーバ、ユーザエッジサーバのいずれかに配置する際、デバイスおよびリンクのコスト、計算リソース上限、帯域上限を制約条件とし、かつ計算リソースのコストまたは応答時間を目的関数とした線形計画式に基づいて、アプリケーションプログラムの配置場所を計算して設定する配置設定部と、を備えることを特徴とするオフロードサーバとした。

発明の効果

[0016] 本発明によれば、GPUやFPGA等のオフロードデバイスに配置できるよう自動変換した際に、変換したアプリケーションをユーザのコストまたは応答時間の要求を満たして最適に配置することができる。

図面の簡単な説明

[0017] [図1]本発明の第1実施形態に係るオフロードサーバの構成例を示す機能ブロック図である。

[図2]第1実施形態に係るオフロードサーバを用いた自動オフロード処理を示す図である。

[図3]第1実施形態に係るオフロードサーバのSimple GAによる制御部（自動オフロード機能部）の探索イメージを示す図である。

[図4]比較例の通常CPUプログラムの例を示す図である。

[図5]比較例の単純CPUプログラムを利用してCPUからGPUへデータ転送する場合のループ文の例を示す図である。

[図6]第1実施形態に係るオフロードサーバのネスト一体化をした場合のCPUからGPUへデータ転送する場合のループ文の例を示す図である。

[図7]第1実施形態に係るオフロードサーバの転送一体化をした場合のCPUからGPUへデータ転送する場合のループ文の例を示す図である。

[図8]第1実施形態に係るオフロードサーバの転送一体化をし、かつ一時領域を利用した場合のCPUからGPUへデータ転送する場合のループ文の例を示す図である。

[図9A]第1実施形態に係るオフロードサーバの実装の動作概要を説明するフローチャートである。

[図9B]第1実施形態に係るオフロードサーバの実装の動作概要を説明するフローチャートである。

[図10]第1実施形態に係るオフロードサーバのGPUオフロード試行の後に追加されるリソース比とリソース量の設定および新規アプリケーションの配置を説明するフローチャートである。

[図11]第1実施形態に係るオフロードサーバの計算ノードのトポロジーの一例を示す図である。

[図12]第1実施形態に係るオフロードサーバの平均応答時間のアプリケーション配置数変化を示すグラフである。

[図13]本発明の第2実施形態に係るオフロードサーバの構成例を示す機能ブロック図である。

[図14]第2実施形態に係るオフロードサーバの実装の動作概要を説明するフローチャートである。

[図15]上記第2実施形態に係るオフロードサーバの性能測定部の性能測定処理を示すフローチャートである。

[図16]第2実施形態に係るオフロードサーバのP L D処理パターン作成部の探索イメージを示す図である。

[図17]第2実施形態に係るオフロードサーバのCコードからOpenCL最終解の探索までの流れを説明する図である。

[図18]本発明の各実施形態に係るオフロードサーバの機能を実現するコンピュータの一例を示すハードウェア構成図である。

発明を実施するための形態

[0018] 以下、図面を参照して本発明を実施するための形態（以下、「本実施形態」という）におけるオフロードサーバについて説明する。

（本発明の自動オフロードの基本的な考え方）

本発明者は、環境適応ソフトウェアのコンセプトを具体化するために、これまでに、プログラムのループ文のGPU自動オフロード、FPGA自動オフロード、変換アプリケーションの実行リソース適正化の方式を提案してきた（非特許文献2、3、4参照）。これら非特許文献2、3、4の要素技術の検討も踏まえて、本発明の基本的な考え方を述べる。

[0019] <CPUとオフロードデバイスのリソース比の適切化>

まず、デバイスにオフロードするプログラム変換ができた後の、CPUとオフロードデバイスのリソース比の適切化について説明する。

非特許文献2等の手法により、GPUやFPGA等のオフロードデバイスに通常のプログラムを自動オフロードすることができる。

現在、マルチコアCPU、メニーコアCPUは、仮想マシンやコンテナによる仮想化により、全コアの何割を割り当てる等が柔軟にできるようになっている。GPUについても、近年CPU同様の仮想化が行われ、GPUの全コアの何割を割り当てる等の運用が可能になりつつある。FPGAに関しては、リソース使用量は、Look Up TableやFlip Flopの設定数で表されることが多く、利用されていないゲートについては別用途に使うことができる。

[0020] このように、CPU、GPU、FPGAとも全リソースの一部を使う運用が可能であり、CPUとオフロードデバイスのリソースを用途に応じて適切化することはコストパフォーマンスを高める上で重要である。

[0021] また、非特許文献2等の手法を用いて、アプリケーションをCPUとGPU処理のコードに変換することはできる。しかし、コード自体は、適切であっても、CPUとGPUとのリソース量が適切なバランスでない場合には、性能が出ない。例えば、ある処理を行う際に、CPUの処理時間が1000秒、GPUの処理時間が1秒では、オフロードできる処理をGPUである程度高速化しても、全体的にはCPUがボトルネックとなっている。

[0022] さらに、非特許文献5の「K. Shirahata, H. Sato and S. Matsuoka, "Hybrid Map Task Scheduling for GPU-Based Heterogeneous Clusters," IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), pp.733-740, Dec. 2010.」では、CPUとGPUを使ってMapReduce（登録商標）フレームワークでタスク処理している際に、CPUとGPUの実行時間が同じになるようMapタスクを配分することで、全体の高性能化を図っている。

[0023] 本発明者は、CPUとオフロードデバイスのリソース比を下記のように決めることを想到した。すなわち、何れかのデバイスでの処理がボトルネックとなることを避けるため、上記非特許文献等も参考に、テストケースの処理時間から、CPUとオフロードデバイスの処理時間が同等オーダになるよう

に、CPUとオフロードデバイスのリソース比（以下、「リソース比」という）を決定する。

[0024] また、本発明者は、非特許文献2の手法のように、自動オフロードの際、検証環境での性能測定結果に基づいて徐々に高速化していく手法を採る。理由としては、性能に関しては、コード構造だけでなく、実際に処理するハードウェアのスペック、データサイズ、ループ回数等の実際に処理する内容によって大きく変わるためである。また、性能は、静的に予測することが困難であり、動的な測定が必要だからである。そのため、コード変換の際に、既に検証環境での性能測定結果があるので、その結果を用いてリソース比を定める。

[0025] 性能測定の際には、テストケースを指定して測定を行う。例えば、検証環境でのテストケースの処理時間が、CPU処理：10秒、GPU処理：5秒の場合では、CPU側のリソースは2倍で同等の処理時間程度と考えられる。このため、リソース比は2：1となる。なお、特にある処理をオフロードで高速化したいといったユーザ要望については、その処理を含むテストケースを準備して、そのテストケースに対して非特許文献2等の手法で高速化することでユーザ要望が反映される。

[0026] <CPUとオフロードデバイスのリソース量の決定と自動検証>

次に、CPUとオフロードデバイスのリソース量（以下、「リソース量」という）の決定と自動検証について説明する。

上記<CPUとオフロードデバイスのリソース比の適切化>により、リソース比が定まった場合、次に商用環境へのアプリケーションの配置を行う。

商用環境への配置の際は、ユーザが指定したコスト要求を満たすように、リソース比は可能な限りキープ（維持）したまま、リソース量を決定する。例えば、CPUに関して、1VMは1000円／月、GPUは4000円／月、リソース比は2：1が適切であるとする。そして、ユーザの予算は、月10000円以内であると想定する。この場合には、リソース比を2：1としても、ユーザの予算内である月10000円以内に収まるので、適切なり

ソース比2 : 1をキープしたリソース量、すなわちCPUは「2」、GPUは「1」を確保して商用環境に配置することになる。また、ユーザの予算が、月5000円以内であった場合には、適切なリソース比2 : 1はキープできない。この場合、リソース量として、CPUは「1」、GPUは「1」を確保して配置する。

[0027] 商用環境にリソースを確保してプログラムを配置した後は、ユーザが利用する前に動作することを確認するため、自動検証が行われる。自動検証では、性能検証テストケースやリグレーションテストケースが実行される。性能検証テストケースは、ユーザが指定した想定テストケースをJenkins（登録商標）等の試験自動実行ツールを用いて行い、処理時間やスループット等を測定する。リグレーションテストケースは、システムにインストールされるミドルウェアやOS等のソフトウェアの情報を取得して、それらに対応するリグレーションテストをJenkins等を用いて実行する。これらの自動検証を、少ないテストケースの準備で行うための検討は非特許文献6（Y. Yamato, “Automatic verification technology of software patches for user virtual environments on IaaS cloud,” Journal of Cloud Computing, Springer, 2015, 4:4, DOI: 10.1186/s13677-015-0028-6, Feb. 2015.）等でなされており、この非特許文献6の技術を用いる。

[0028] 性能検証テストケースでは、オフロードした場合でも計算結果が不正でないかをチェックする。また、性能検証テストケースでは、オフロードしない場合との計算結果差分もチェックする。例えば、GPUを処理するPGIコンパイラは、PCAST（登録商標）という機能のPGI_compare（登録商標）やacc_compare（登録商標）というAPI（Application Programming Interface）で、GPUを使う場合使わない場合の計算結果差分を確認できる。

なお、GPUとCPUでは丸め誤差が異なる等、並列処理等を正しくオフロードしても完全に計算結果が一致しない場合もある。そのため、例えばIEEE 754仕様による確認等を行い、許容できる差分かをユーザに提示し、ユーザに確認をしてもらう。

[0029] 自動検証の結果として、性能検証テストケースの処理時間やそのスループット、計算結果差分およびリグレーションテストの実行結果の情報が、ユーザに提示される。ユーザには、さらに確保したリソース（VMの数やスペック等）とその価格が提示されており、ユーザはそれら情報を参照して運用開始を判断する。

[0030] <リソース、リソース比、テストケース処理時間>

本実施形態におけるリソース、リソース比、テストケース処理時間について述べる。

・リソースについて

CPU、GPU、FPGA等は仮想資源のインスタンスとして提供されるようになってきている。

リソースとして、CPUのコア数、クロック、メモリ量、ディスクサイズ、GPUのコア数、クロック、メモリ量、FPGAのゲート規模（Intel（登録商標）の場合はLE（登録商標）、Xilinx（登録商標）の場合LC（登録商標）が単位となる）がある。クラウド等の事業者は、それらをパッケージ化して、small sizeの仮想マシンやGPUインスタンスといった形で提供している。仮想化する場合は、利用するインスタンスの数が利用するリソース量といえる。

[0031] ・リソース比について

CPU、GPU、FPGAのインスタンス数の比がリソース比となる。インスタンス数が1つ、2つ、3つであれば、リソース比は1：2：3である。

[0032] ・テストケース処理時間について

本実施形態は、ユーザが指定するテストケースを高速化するオフロードパターンを探索して発見する。テストケースは、DB（データベース）であればTPC-C（登録商標）のようなトランザクション処理数であり、FFTであればサンプルデータでのフーリエ変換処理の実行である。処理時間は、そのサンプル処理を実行した際の実行時間である。例えば、処理Aの処理時間は、

オフロード前は10秒であったものが、オフロード後は2秒になるといった形で、CPUで実行した場合と、オフロードデバイスで実行した場合との実行時間がそれぞれ取得される。

[0033] 〈ループ文の発見〉

コンパイラが、このループ文はGPUの並列処理に適しているという適合性を見つけることは難しいのが現状である。GPUにオフロードすることでどの程度の性能、電力消費量になるかは、実測してみないと予測は難しい。そのため、このループ文をGPUにオフロードするという指示を手動で行い、測定の試行錯誤が行われている。

本発明は、GPUにオフロードする適切なループ文の発見を、進化計算手法である遺伝的アルゴリズム（GA：Genetic Algorithm）を用いて自動的に行う。すなわち、並列可能ループ文群に対して、GPU実行の際を1、CPU実行の際を0に値を置いて遺伝子化し、検証環境で反復測定し適切なパターンを探索する。

[0034] （第1の実施形態）

次に、本発明を実施するための形態（以下、「本実施形態」と称する。）における、オフロードサーバ1等について説明する。

[0035] [ループ文のGPU自動オフロード]

図1は、本発明の第1の実施形態に係るオフロードサーバ1の構成例を示す機能ブロック図である。

オフロードサーバ1は、アプリケーションの特定処理をアクセラレータに自動的にオフロードする装置である。

図1に示すように、オフロードサーバ1は、制御部11と、入出力部12と、記憶部13と、検証用マシン14（Verification machine）（アクセラレータ検証用装置）と、を含んで構成される。

[0036] 入出力部12は、各デバイス等との間で情報の送受信を行うための通信インターフェイスと、タッチパネルやキーボード等の入力装置や、モニタ等の出力装置との間で情報の送受信を行うための入出力インターフェイスとから

構成される。

- [0037] 記憶部 13 は、ハードディスクやフラッシュメモリ、RAM (Random Access Memory) 等により構成され、制御部 11 の各機能を実行させるためのプログラム (オフロードプログラム) や、制御部 11 の処理に必要な情報 (例えば、中間言語ファイル (Intermediate file) 133) が一時的に記憶される。
- [0038] 記憶部 13 は、テストケース DB (Test case database) 131、設備リソース DB 132、中間言語ファイル (Intermediate file) 133 を備える。
- [0039] テストケース DB 131 は、検証対象ソフトに対応した試験項目のデータを格納する。試験項目のデータは、例えば MySQL 等のデータベースシステムの場合、TPC-C 等のトランザクション試験のデータである。
- [0040] 設備リソース DB 132 は、事業者が保持するサーバ等のリソースと価格等の事前に準備された情報と、それらがどの程度使われているかの情報を保持する。例えば GPU インスタンスを 3 収容できるサーバが 10 台あり、1 GPU インスタンスは月 5000 円であり、10 台のうち、A、B の 2 台はフルに使われており、C の 1 台はインスタンスが 1 つだけ使われている等の情報である。この情報は、ユーザが運用条件 (コスト、性能等の条件) を指定した際に、確保するリソース量を決定するために利用される。ユーザ運用条件は、ユーザがオフロード依頼時に指定したコスト条件 (例えば、月 10000 円以内の予算等) と性能条件 (例えば、TPC-C 等のトランザクションスループットが何以上やサンプルフーリエ変換処理が 1 スレッドで何秒以内等) である。
- [0041] 中間言語ファイル 133 は、高水準言語と機械語の中間に介在するプログラミング言語の形で制御部 11 の処理に必要な情報を一時的に記憶する。
- [0042] 検証用マシン 14 は、環境適応ソフトウェアの検証用環境として、CPU、GPU、FPGA を備える。
- [0043] 制御部 11 は、オフロードサーバ 1 全体の制御を司る自動オフロード機能部 (Automatic Offloading function) である。制御部 11 は、例えば、記憶部 13 に格納されたアプリケーションプログラム (オフロードプログラム)

を不図示のCPU (Central Processing Unit) が、RAMに展開し実行することにより実現される。

[0044] 制御部11は、アプリケーションコード指定部 (Specify application code) 111と、アプリケーションコード分析部 (Analyze application code) 112と、データ転送指定部113と、並列処理指定部114と、リソース比決定部115と、リソース量設定部116と、配置設定部170と、並列処理パターン作成部117と、性能測定部118と、実行ファイル作成部119と、本番環境配置部 (Deploy final binary files to production environment) 120と、性能測定テスト抽出実行部 (Extract performance test cases and run automatically) 121と、ユーザ提供部 (Provide price and performance to a user to judge) 122と、を備える。

[0045] <アプリケーションコード指定部111>

アプリケーションコード指定部111は、入力されたアプリケーションコードの指定を行う。具体的には、アプリケーションコード指定部111は、受信したファイルに記載されたアプリケーションコードを、アプリケーションコード分析部112に渡す。

[0046] <アプリケーションコード分析部112>

アプリケーションコード分析部112は、処理機能のソースコードを分析し、ループ文やFFTライブラリ呼び出し等の構造を把握する。

[0047] <データ転送指定部113>

データ転送指定部113は、アプリケーションプログラムのループ文の中で用いられる変数の参照関係を分析し、ループ外でデータ転送してよいデータについては、ループ外でのデータ転送を明示的に指定する明示的指定行 (後記する#pragma acc kernels、#pragma acc data copyin(a, b)、#pragma acc data copyout(a, b)、#prama acc parallel loop、#prama acc parallel loop vectorなど) を用いたデータ転送指定を行う。

[0048] <並列処理指定部114>

並列処理指定部114は、アプリケーションプログラムのループ文 (繰り

返し文)を特定し、各ループ文に対して、アクセラレータにおける並列処理指定文を指定してコンパイルする。

並列処理指定部114は、オフロード範囲抽出部(Extract offload able area)114aと、中間言語ファイル出力部(Output intermediate file)114bと、を備える。

[0049] オフロード範囲抽出部114aは、ループ文やFFT等、GPU・FPGAにオフロード可能な処理を特定し、オフロード処理に応じた中間言語を抽出する。

中間言語ファイル出力部114bは、抽出した中間言語ファイル133を出力する。中間言語抽出は、一度で終わりではなく、適切なオフロード領域探索のため、実行を試行して最適化するため反復される。

[0050] <リソース比決定部115>

リソース比決定部115は、性能測定結果をもとに、CPUとオフロードデバイスの処理時間(テストケースCPU処理時間とオフロードデバイス処理時間)を、リソース比として決定する(後記)。具体的には、リソース比決定部115は、CPUとオフロードデバイスの処理時間が同等オーダになるように、リソース比を決定する。また、リソース比決定部115は、CPUとオフロードデバイスの処理時間の差分が所定閾値以上の場合、リソース比を所定の上限値に設定する。

[0051] <リソース量設定部116>

リソース量設定部116は、決定したリソース比をもとに、所定のコスト条件を満たすように、CPUおよびオフロードデバイスのリソース量を設定する(後記)。具体的には、リソース量設定部116は、決定したリソース比を維持して、所定のコスト条件を満たす最大のリソース量を設定する。また、リソース量設定部116は、決定したリソース比を維持した最小リソース量の設定で所定のコスト条件を満たさない場合は、リソース比を崩してCPUとオフロードデバイスのリソース量をコスト条件を満たすより小さい値(例えば、最小)で設定する。

[0052] <配置設定部 170>

配置設定部 170 は、変換したアプリケーションを、ユーザの指定するコストまたは応答時間の条件に応じて、ネットワーク上の、クラウドサーバ、キャリアエッジサーバ、ユーザエッジサーバのいずれかに配置する際、デバイスおよびリンクのコスト、計算リソース上限、帯域上限を制約条件とし、かつ計算リソースのコストまたは応答時間を目的関数とした線形計画式に基づいて、アプリケーションの配置場所を計算して設定する。具体的には、配置設定部 170 は、設備リソース DB 132 のサーバ、リンクのスペック情報、既存アプリケーションの配置情報に基づいて、線形計画手法で、新規アプリケーションの配置先（APL の配置場所）を計算して設定する。線形計画手法では、例えば、後記 [数 1] [数 2] に示す線形計画式の目的関数および制約条件を用いる。後記 [数 1] [数 2] に示す線形計画式は、設備リソース DB 132 に保存されており、配置設定部 170 が、設備リソース DB 132 から読み出し、配置設定部 170 が処理するメモリ上で展開される。

[0053] <並列処理パターン作成部 117>

並列処理パターン作成部 117 は、コンパイルエラーが出るループ文（繰り返し文）に対して、オフロード対象外とするとともに、コンパイルエラーが出ない繰り返し文に対して、並列処理するかしないかの指定を行う並列処理パターンを作成する。

[0054] <性能測定部 118>

性能測定部 118 は、並列処理パターンのアプリケーションプログラムをコンパイルして、検証用マシン 14 に配置し、アクセラレータにオフロードした際の性能測定用処理を実行する。

性能測定部 118 は、バイナリファイル配置部 (Deploy binary files) 118a を備える。バイナリファイル配置部 118a は、GPU や FPGA を備えた検証用マシン 14 に、中間言語から導かれる実行ファイルをデプロイ（配置）する。

[0055] 性能測定部 118 は、配置したバイナリファイルを実行し、オフロードした際の性能を測定するとともに、性能測定結果を、オフロード範囲抽出部 114 a に戻す。この場合、オフロード範囲抽出部 114 a は、別の並列処理パターン抽出を行い、中間言語ファイル出力部 114 b は、抽出された中間言語をもとに、性能測定を試行する（後記図 2 の符号 a 参照）。

[0056] <実行ファイル作成部 119>

実行ファイル作成部 119 は、所定回数繰り返された、性能測定結果をもとに、複数の並列処理パターンから高処理性能の並列処理パターンを複数選択し、高処理性能の並列処理パターンを交叉、突然変異処理により別の複数の並列処理パターンを作成する。そして、実行ファイル作成部 119 は、新たに性能測定までを行い、指定回数の性能測定後に、性能測定結果をもとに、複数の並列処理パターンから最高処理性能の並列処理パターンを選択し、最高処理性能の並列処理パターンをコンパイルして実行ファイルを作成する。

[0057] <本番環境配置部 120>

本番環境配置部 120 は、作成した実行ファイルを、ユーザ向けの本番環境に配置する（「最終バイナリファイルの本番環境への配置」）。本番環境配置部 120 は、最終的なオフロード領域を指定したパターンを決定し、ユーザ向けの本番環境にデプロイする。

[0058] <性能測定テスト抽出実行部 121>

性能測定テスト抽出実行部 121 は、実行ファイル配置後、テストケース DB 131 から性能試験項目を抽出し、性能試験を実行する（「最終バイナリファイルの本番環境への配置」）。

性能測定テスト抽出実行部 121 は、実行ファイル配置後、ユーザに性能を示すため、性能試験項目をテストケース DB 131 から抽出し、抽出した性能試験を自動実行する。

[0059] <ユーザ提供部 122>

ユーザ提供部 122 は、性能試験結果を踏まえた、価格・性能等の情報を

ユーザに提示する（「価格・性能等の情報のユーザへの提供」）。テストケースDB131には、性能試験項目が格納されている。ユーザ提供部122は、テストケースDB131に格納された試験項目に対応した性能試験の実施結果に基づいて、価格、性能等のデータを、上記性能試験結果と共にユーザに提示する。ユーザは、提示された価格・性能等の情報をもとに、サービスの課金利用開始を判断する。ここで、本番環境への一括デプロイには、非特許文献7（Y. Yamato, M. Muroi, K. Tanaka and M. Uchimura, “Development of Template Management Technology for Easy Deployment of Virtual Resources on OpenStack,” Journal of Cloud Computing, Springer, 2014, 3:7, DOI: 10.1206/s13677-014-0007-3, 12 pages, June 2014.）の技術を用い、また、性能自動試験には、前述の非特許文献6の技術を用いればよい。

[0060] [遺伝的アルゴリズムの適用]

オフロードサーバ1は、オフロードの最適化にGA（Genetic Algorithms）を用いることができる。GAを用いた場合のオフロードサーバ1の構成は下記の通りである。

すなわち、並列処理指定部114は、遺伝的アルゴリズムに基づき、コンパイルエラーが出ないループ文（繰り返し文）の数を遺伝子長とする。並列処理パターン作成部117は、アクセラレータ処理をする場合を1または0のいずれか一方、しない場合を他方の0または1として、アクセラレータ処理可否を遺伝子パターンにマッピングする。

[0061] 並列処理パターン作成部117は、遺伝子の各値を1か0にランダムに作成した指定個体数の遺伝子パターンを準備する。性能測定部118は、各個体に応じて、アクセラレータにおける並列処理指定文を指定したアプリケーションコードをコンパイルして、検証用マシン14に配置する。性能測定部118は、検証用マシン14において性能測定用処理を実行する。

[0062] ここで、性能測定部118は、途中世代で、以前と同じ並列処理パターンの遺伝子が生じた場合は、当該並列処理パターンに該当するアプリケーションコードのコンパイル、および、性能測定はせずに、性能測定値としては同

じ値を使う。

また、性能測定部 118 は、コンパイルエラーが生じるアプリケーションコード、および、性能測定が所定時間で終了しないアプリケーションコードについては、タイムアウトの扱いとして、性能測定値を所定の時間（長時間）に設定する。

[0063] 実行ファイル作成部 119 は、全個体に対して、性能測定を行い、処理時間の短い個体ほど適合度が高くなるように評価する。実行ファイル作成部 119 は、全個体から、適合度が所定値（例えば、全個数の上位 $n\%$ 、または全個数の上位 m 個 n, m は自然数）より高いものを性能の高い個体として選択し、選択された個体に対して、交叉、突然変異の処理を行い、次世代の個体を作成する。実行ファイル作成部 119 は、指定世代数の処理終了後、最高性能の並列処理パターンを解として選択する。

[0064] 以下、上述のように構成されたオフロードサーバ 1 の自動オフロード動作について説明する。

[自動オフロード動作]

図 2 は、オフロードサーバ 1 を用いた自動オフロード処理を示す図である。

図 2 に示すように、オフロードサーバ 1 は、環境適応ソフトウェアの要素技術に適用される。オフロードサーバ 1 は、制御部（自動オフロード機能部）11 と、テストケース DB 131 と、設備リソース DB 132 と、中間言語ファイル 133 と、検証用マシン 14 と、を有している。

オフロードサーバ 1 は、ユーザが利用するアプリケーションコード（Application code）125 を取得する。

[0065] ユーザは、例えば、各種デバイス（Device 151、CPU-GPU を有する装置 152、CPU-FPGA を有する装置 153、CPU を有する装置 154）の利用を契約した人である。

オフロードサーバ 1 は、機能処理を CPU-GPU を有する装置 152、CPU-FPGA を有する装置 153 のアクセラレータに自動オフロードする。

[0066] 以下、図2のステップ番号を参照して各部の動作を説明する。

＜ステップS11 : Specify application code＞

ステップS11において、アプリケーションコード指定部111（図1参照）は、受信したファイルに記載されたアプリケーションコードを、アプリケーションコード分析部112に渡す。

[0067] ＜ステップS12 : Analyze application code＞

ステップS12において、アプリケーションコード分析部112（図1参照）は、処理機能のソースコードを分析し、ループ文やFFTライブラリ呼び出し等の構造を把握する。

[0068] ＜ステップS13 : Extract offloadable area＞

ステップS13において、並列処理指定部114（図1参照）は、アプリケーションのループ文（繰り返し文）を特定し、各繰り返し文に対して、アクセラレータにおける並列処理指定文を指定してコンパイルする。具体的には、オフロード範囲抽出部114a（図1参照）は、ループ文やFFT等、GPU・FPGAにオフロード可能な処理を特定し、オフロード処理に応じた中間言語を抽出する。

[0069] ＜ステップS14 : Output intermediate file＞

ステップS14において、中間言語ファイル出力部114b（図1参照）は、中間言語ファイル133を出力する。中間言語抽出は、一度で終わりではなく、適切なオフロード領域探索のため、実行を試行して最適化するため反復される。

[0070] ＜ステップS15 : Compile error＞

ステップS15において、並列処理パターン作成部117（図1参照）は、コンパイルエラーが出るループ文に対して、オフロード対象外とするとともに、コンパイルエラーが出ない繰り返し文に対して、並列処理するかしないかの指定を行う並列処理パターンを作成する。

[0071] ＜ステップS21 : Deploy binary files＞

ステップS21において、バイナリファイル配置部118a（図1参照）

は、GPU・FPGAを備えた検証用マシン14に、中間言語から導かれる実行ファイルをデプロイする。

[0072] <ステップS22 : Measure performances>

ステップS22において、性能測定部118（図1参照）は、配置したファイルを実行し、オフロードした際の性能を測定する。

オフロードする領域をより適切にするため、この性能測定結果は、オフロード範囲抽出部114aに戻され、オフロード範囲抽出部114aが、別パターンの抽出を行う。そして、中間言語ファイル出力部114bは、抽出された中間言語をもとに、性能測定を試行する（図2の符号a参照）。

[0073] 図2の符号aに示すように、制御部11は、上記ステップS12乃至ステップS22を繰り返し実行する。制御部11の自動オフロード機能をまとめると、下記である。すなわち、並列処理指定部114は、アプリケーションプログラムのループ文（繰り返し文）を特定し、各繰り返し文に対して、GPUでの並列処理指定文を指定して、コンパイルする。そして、並列処理パターン作成部117は、コンパイルエラーが出るループ文を、オフロード対象外とし、コンパイルエラーが出ないループ文に対して、並列処理するかしないかの指定を行う並列処理パターンを作成する。そして、バイナリファイル配置部118aは、該当並列処理パターンのアプリケーションプログラムをコンパイルして、検証用マシン14に配置し、性能測定部118が、検証用マシン14で性能測定用処理を実行する。実行ファイル作成部119は、所定回数繰り返された、性能測定結果をもとに、複数の並列処理パターンから最高処理性能のパターンを選択し、選択パターンをコンパイルして実行ファイルを作成する。

[0074] <ステップS23 : ユーザ運用条件によるリソース量設定>

ステップS23において、制御部11は、ユーザ運用条件によるリソース量設定を行う。すなわち、制御部11のリソース比決定部115は、CPUとオフロードデバイスのリソース比を決定する。そして、リソース量設定部116は、決定したリソース比をもとに、設備リソースDB132の情報を

参照し、ユーザ運用条件を満たすように、CPUおよびオフロードデバイスのリソース量を設定する（図10により後記する）。

[0075] <ステップS24 : Deploy final binary files to production environment >

ステップS24において、本番環境配置部120は、最終的なオフロード領域を指定したパターンを決定し、ユーザ向けの本番環境にデプロイする。

[0076] <ステップS25 : Extract performance test cases and run automatically>

ステップS25において、性能測定テスト抽出実行部121は、実行ファイル配置後、ユーザに性能を示すため、性能試験項目をテストケースDB131から抽出し、抽出した性能試験を自動実行する。

[0077] <ステップS26 : Provide price and performance to a user to judge>

ステップS26において、ユーザ提供部122は、性能試験結果を踏まえた、価格・性能等の情報をユーザに提示する。ユーザは、提示された価格・性能等の情報をもとに、サービスの課金利用開始を判断する。

[0078] 上記ステップS11～ステップS26は、例えばユーザのサービス利用のバックグラウンドで行われ、例えば、仮利用の初日の間に行う等を想定している。

[0079] 上記したように、オフロードサーバ1の制御部（自動オフロード機能部）11は、環境適応ソフトウェアの要素技術に適用した場合、機能処理のオフロードのため、ユーザが利用するアプリケーションプログラムのソースコードから、オフロードする領域を抽出して中間言語を出力する（ステップS11～ステップS15）。制御部11は、中間言語から導かれる実行ファイルを、検証用マシン14に配置実行し、オフロード効果を検証する（ステップS21～ステップS22）。検証を繰り返し、適切なオフロード領域を定めたのち、制御部11は、実際にユーザに提供する本番環境に、実行ファイルをデプロイし、サービスとして提供する（ステップS23～ステップS26）。

[0080] [G Aを用いたGPU自動オフロード]

GPU自動オフロードは、GPUに対して、図2のステップS12～ステップS22を繰り返し、最終的にステップS23でデプロイするオフロードコードを得るための処理である。

[0081] GPUは、一般的にレイテンシーは保証しないが、並列処理によりスループットを高めることに向いたデバイスである。暗号化処理や、カメラ映像分析のための画像処理、大量センサデータ分析のための機械学習処理等が代表的であり、それらは、繰り返し処理が多い。そこで、アプリケーションの繰り返し文をGPUに自動でオフロードすることでの高速化を狙う。

[0082] しかし、従来技術で記載の通り、高速化には適切な並列処理が必要である。特に、GPUを使う場合は、CPUとGPU間のメモリ転送のため、データサイズやループ回数が多いと性能が出ないことが多い。また、メモリデータ転送のタイミング等により、並列高速化できる個々のループ文（繰り返し文）の組み合わせが、最速とならない場合等がある。例えば、10個のfor文（繰り返し文）で、1番、5番、10番の3つがCPUに比べて高速化できる場合に、1番、5番、10番の3つの組み合わせが最速になるとは限らない等である。

[0083] 適切な並列領域指定のため、PGIコンパイラを用いて、for文の並列可否を試行錯誤して最適化する試みがある。しかし、試行錯誤には多くの稼働がかかり、サービスとして提供する際に、ユーザの利用開始が遅くなり、コストも上がってしまう問題がある。

[0084] そこで、本実施形態では、並列化を想定していない汎用プログラムから、自動で適切なオフロード領域を抽出する。このため、最初に並列可能for文のチェックを行い、次に並列可能for文群に対してG Aを用いて検証環境で性能検証試行を反復し適切な領域を探索すること、を実現する。並列可能for文に絞った上で、遺伝子の部分の形で、高速化可能な並列処理パターンを保持し組み換えていくことで、取り得る膨大な並列処理パターンから、効率的に高速化可能なパターンを探索できる。

[0085] [Simple GAによる制御部（自動オフロード機能部） 11の探索イメージ]

図3は、Simple GAによる制御部（自動オフロード機能部） 11の探索イメージを示す図である。図3は、処理の探索イメージと、for文の遺伝子配列マッピングを示す。

GAは、生物の進化過程を模倣した組合せ最適化手法の一つである。GAのフローチャートは、初期化→評価→選択→交叉→突然変異→終了判定となっている。

本実施形態では、GAの中で、処理を単純にしたSimple GAを用いる。Simple GAは、遺伝子は1、0のみとし、ルーレット選択、一点交叉、突然変異は1箇所の遺伝子の値を逆にする等、単純化されたGAである。

[0086] <初期化>

初期化では、アプリケーションコードの全for文の並列可否をチェック後、並列可能for文を遺伝子配列にマッピングする。GPU処理する場合は1、GPU処理しない場合は0とする。遺伝子は、指定の個体数Mを準備し、1つのfor文にランダムに1、0の割り当てを行う。

具体的には、制御部（自動オフロード機能部） 11（図1参照）は、ユーザが利用するアプリケーションコード（Application code） 130（図2参照）を取得し、図3に示すように、アプリケーションコード130のコードパターン（Code patterns） 141からfor文の並列可否をチェックする。図3に示すように、コードパターン141から5つのfor文が見つかった場合（図3の符号b参照）、各for文に対して1桁、ここでは5つのfor文に対し5桁の1または0をランダムに割り当てる。例えば、CPUで処理する場合0、GPUに出す場合1とする。ただし、この段階では1または0をランダムに割り当てる。

遺伝子長に該当するコードが5桁であり、5桁の遺伝子長のコードは $2^5 = 32$ パターン、例えば10001、10010、…となる。なお、図3では、コードパターン141中の丸印（○印）をコードのイメージとして示している。

[0087] <評価>

評価では、デプロイ（配置）とパフォーマンスの測定（Deploy & performance measurement）を行う（図3の符号c参照）。すなわち、性能測定部118（図1参照）は、遺伝子に該当するコードをコンパイルして検証用マシン14にデプロイして実行する。性能測定部118は、ベンチマーク性能測定を行う。性能が良いパターン（並列処理パターン）の遺伝子の適合度を高くする。

[0088] <選択>

選択では、適合度に基づいて、高性能コードパターンを選択（Select high performance code patterns）する（図3の符号d参照）。性能測定部118（図1参照）は、適合度に基づいて、高適合度の遺伝子を、指定の個体数で選択する。本実施形態では、適合度に応じたルーレット選択および最高適合度遺伝子のエリート選択を行う。

図3では、選択されたコードパターン（Select code patterns）142の中の丸印（○印）が、3つに減ったことを探索イメージとして示している。

[0089] <交叉>

交叉では、一定の交叉率 P_c で、選択された個体間で一部の遺伝子のある一点で交換し、子の個体を作成する。

ルーレット選択された、あるパターン（並列処理パターン）と他のパターンとの遺伝子を交叉させる。一点交叉の位置は任意であり、例えば上記5桁のコードのうち3桁目で交叉させる。

[0090] <突然変異>

突然変異では、一定の突然変異率 P_m で、個体の遺伝子の各値を0から1または1から0に変更する。

また、局所解を避けるため、突然変異を導入する。なお、演算量を削減するために突然変異を行わない態様でもよい。

[0091] <終了判定>

図3に示すように、クロスオーバーと突然変異後の次世代コードパターン

の生成 (Generate next generation code patterns after crossover & mutation) を行う (図3の符号e参照)。

終了判定では、指定の世代数T回、繰り返しを行った後に処理を終了し、最高適合度の遺伝子を解とする。

例えば、性能測定して、速い3つ10010、01001、00101を選ぶ。この3つをGAにより、次の世代は、組み換えをして、例えば1番目と2番目を交叉させて新しいパターン(並列処理パターン)11011を作っていく。このとき、組み換えをしたパターンに、勝手に0を1にするなどの突然変異を入れる。上記を繰り返して、一番早いパターンを見付ける。指定世代(例えば、20世代)などを決めて、最終世代で残ったパターンを、最後の解とする。

[0092] <デプロイ(配置)>

最高適合度の遺伝子に該当する、最高処理性能の並列処理パターンで、本番環境に改めてデプロイして、ユーザに提供する。

[0093] <補足説明>

GPUにオフロードできないfor文(ループ文;繰り返し文)が相当数存在する場合について説明する。例えば、for文が200個あっても、GPUにオフロードできるものは30個くらいである。ここでは、エラーになるものを除外し、この30個について、GAを行う。

[0094] OpenACCには、ディレクティブ #pragma acc kernelsで指定して、GPU向けバイトコードを抽出し、実行によりGPUオフロードを可能とするコンパイラがある。この#pragmaに、for文のコマンドを書くことにより、そのfor文がGPUで動くか否かを判定することができる。

[0095] 例えばC/C++を使った場合、C/C++のコードを分析し、for文を見付ける。for文を見付けると、OpenACCで並列処理の文法である#pragma acc kernels、#prama acc parallel loopや#prama acc parallel loop vectorを使ってfor文に対して書き込む。詳細には、#pragma acc kernels、#prama acc parallel loopや#prama acc parallel loop vectorに、一つ一つfor文を入れてコン

パイルして、エラーであれば、そのfor文はそもそも、GPU処理できないので、除外する。

[0096] このようにして、残るfor文を見付ける。そして、エラーが出ないものを、長さ（遺伝子長）とする。エラーのないfor文が5つであれば、遺伝子長は5であり、エラーのないfor文が10であれば、遺伝子長は10である。なお、並列処理できないものは、前の処理を次の処理に使うようなデータに依存がある場合である。

以上が準備段階である。次にGA処理を行う。

[0097] for文の数に対応する遺伝子長を有するコードパターンが得られている。始めはランダムに並列処理パターン10010、01001、00101、…を割り当てる。GA処理を行い、コンパイルする。その時に、オフロードできるfor文であるにもかかわらず、エラーがでることがある。それは、for文が階層になっている（どちらか指定すればGPU処理できる）場合である。この場合は、エラーとなったfor文は、残してもよい。具体的には、処理時間が多くなった形にして、タイムアウトさせる方法がある。

[0098] 検証用マシン14でデプロイして、ベンチマーク、例えば画像処理であればその画像処理でベンチマークする、その処理時間が短い程、適応度が高いと評価する。例えば、処理時間の $-1/2$ 乗で、処理時間1秒かかるものは1、100秒かかるものは0.1、0.01秒かかるものは10とする。

適応度が高いものを選択して、例えば10個のなかから、3～5個を選択して、それを組み替えて新しいコードパターンを作る。このとき、作成途中で、前と同じものができる場合がある。その場合、同じベンチマークを行う必要はないので、前と同じデータを使う。本実施形態では、コードパターンと、その処理時間は記憶部13に保存しておく。

以上で、Simple GAによる制御部（自動オフロード機能部）11の探索イメージについて説明した。次に、データ転送の一括処理手法について述べる。

[0099] [データ転送の一括処理手法]

<基本的な考え方>

CPU-GPU転送の削減のため、ネストループの変数をできるだけ上位で転送することに加え、本発明は、多数の変数転送タイミングを一括化し、さらにコンパイラが自動転送してしまう転送を削減する。

転送の削減にあたり、ネスト単位だけでなく、GPUに転送するタイミングがまとめられる変数については一括化して転送する。例えば、GPUの処理結果をCPUで加工してGPUで再度処理させるなどの変数でなければ、複数のループ文で使われるCPUで定義された変数を、GPU処理が始まる前に一括してGPUに送り、全GPU処理が終わってからCPUに戻すなどの対応も可能である。

[0100] コード分析時にループおよび変数の参照関係を把握するため、その結果から複数ファイルで定義された変数について、GPU処理とCPU処理が入れ子にならず、CPU処理とGPU処理が分けられる変数については、一括化して転送する指定をOpenACCのdata copy文を用いて指定する。

GPU処理の始まる前に一括化して転送され、ループ文処理のタイミングで転送が不要な変数はdata presentを用いて転送不要であることを明示する。

CPU-GPUのデータ転送時は、一時領域を作成し(#pragma acc declare create)、データは一時領域に格納後、一時領域を同期(#pragma acc update)することで転送を指示する。

[0101] <比較例>

まず、比較例について述べる。

比較例は、通常CPUプログラム(図4参照)、単純GPU利用(図5参照)、ネスト一括化(非特許文献2)(図6参照)である。なお、以下の記載および図中のループ文の文頭の<1>~<4>等は、説明の便宜上で付したものである(他図およびその説明においても同様)。

図4に示す通常CPUプログラムのループ文は、CPUプログラム側で記述され、

<1> ループ [for(i=0; i<10; i++)] {

}

の中に、

<2> ループ [for(j=0; j<20; j++)] {

がある。図4の符号fは、上記<2>ループにおける、変数a, bの設定である。

また、

<3> ループ [for(k=0; k<30; k++)] {

}

と、

<4> ループ [for(l=0; l<40; l++)] {

}

と、が続く。図4の符号gは、上記<3>ループにおける変数c, dの設定であり、図4の符号hは、上記<4>ループにおける変数e, fの設定である。

図4に示す通常CPUプログラムは、CPUで実行される（GPU利用しない）。

[0102] 図5は、図4に示す通常CPUプログラムを、単純GPU利用して、CPUからGPUへのデータ転送する場合のループ文を示す図である。データ転送の種類は、CPUからGPUへのデータ転送、および、GPUからCPUへのデータ転送がある。以下、CPUからGPUへのデータ転送を例にとる。

図5に示す単純GPU利用のループ文は、CPUプログラム側で記述され、

、

<1> ループ [for(i=0; i<10; i++)] {

}

の中に、

<2> ループ [for(j=0; j<20; j++)] {

がある。

さらに、図5の符号iに示すように、<1> ループ [for(i=0; i<10; i++)]

} {

}の上部に、P G Iコンパイラによるfor文等の並列処理可能処理部を、Open A C Cのディレクティブ #pragma acc kernels（並列処理指定文）で指定している。

図5の符号 i を含む破線枠囲みに示すように、#pragma acc kernelsによって、C P UからG P Uへデータ転送される。ここでは、このタイミングで a , b が転送されるため10回転送される。

[0103] また、図5の符号 j に示すように、<3> ループ [for(k=0; k<30; k++)] { }の上部に、P G Iコンパイラによるfor文等の並列処理可能処理部を、Open A C Cのディレクティブ #pragma acc kernelsで指定している。

図5の符号 j を含む破線枠囲みに示すように、#pragma acc kernelsによって、このタイミングで c , d が転送される。

[0104] ここで、<4> ループ [for(l=0; l<40; l++)] { }の上部には、#pragma acc kernelsを指定しない。このループは、G P U処理しても効率が悪いのでG P U処理しない。

[0105] 図6は、ネスト一括化（非特許文献2）による、C P UからG P UおよびG P UからC P Uへのデータ転送する場合のループ文を示す図である。

図6に示すループ文では、図6の符号 k に示す位置に、C P UからG P Uへのデータ転送指示行、ここでは変数 a , b の copyin 節の #pragma acc data copyin(a, b)を挿入する。なお、本明細書では表記の関係でcopyin(a, b)について、括弧()を付している。後記copyout(a, b)、datacopyin(a, b, c, d)についても同様の表記方法を採用。

上記 #pragma acc data copyin(a, b)は、変数 a の設定、定義を含まない最上位のループ（ここでは、<1> ループ [for(i=0; i<10; i++)] { }の上部）に指定される。

図6の符号 k を含む一点鎖線枠囲みに示すタイミングで a , b が転送されるため1回転送が発生する。

[0106] また、図6に示すループ文では、図6の符号 l に示す位置に、G P Uから

CPUへのデータ転送指示行、ここでは変数 *a*, *b* の `copyout` 節の `#pragma acc data copyout(a, b)` を挿入する。

上記 `#pragma acc data copyout(a, b)` は、〈1〉ループ `[for(i=0; i<10; i++)]` {
}の下部に指定される。

[0107] このように、CPUからGPUへのデータ転送において、変数 *a* の `copyin` 節の `#pragma acc data copyin(a, b)` を、上述した位置に挿入することによりデータ転送を明示的に指示する。これにより、できるだけ上位のループでデータ転送を一括して行うことができ、図5に示す単純GPU利用のループ文のようにループ毎に毎回データを転送する非効率な転送を避けることができる。

[0108] 〈実施形態〉

次に、本実施形態について述べる。

《転送不要な変数を `data present` を用いて明示》

本実施形態では、複数ファイルで定義された変数について、GPU処理とCPU処理が入れ子にならず、CPU処理とGPU処理が分けられる変数については、一括化して転送する指定をOpenACCの `data copy` 文を用いて指定する。併せて、一括化して転送され、そのタイミングで転送が不要な変数は `data present` を用いて明示する。

[0109] 図7は、本実施形態のCPU-GPUのデータ転送時の転送一括化によるループ文を示す図である。図7は、比較例の図6のネスト一括化に対応する。

図7に示すループ文では、図7の符号 *m* に示す位置に、CPUからGPUへのデータ転送指示行、ここでは変数 *a*, *b*, *c*, *d* の `copyin` 節の `#pragma acc data copyin(a, b, c, d)` を挿入する。

上記 `#pragma acc data copyin(a, b, c, d)` は、変数 *a* の設定、定義を含まない最上位のループ（ここでは、〈1〉ループ `[for(i=0; i<10; i++)]` {
}の上部）に指定される。

[0110] このように、複数ファイルで定義された変数について、GPU処理とCPU

U処理が入れ子にならず、CPU処理とGPU処理が分けられる変数については、一括化して転送する指定をOpenACCのdata copy文#pragma acc data copyin(a, b, c, d)を用いて指定する。

図7の符号mを含む一点鎖線枠囲みに示すタイミングでa, b, c, dが転送されるため1回転送が発生する。

[0111] そして、上記#pragma acc data copyin(a, b, c, d)を用いて一括化して転送され、そのタイミングで転送が不要な変数は、図7の符号nを含む二点鎖線枠囲みに示すタイミングで既にGPUに変数があることを明示するdata present文#pragma acc data present (a, b)を用いて指定する。

[0112] 上記#pragma acc data copyin(a, b, c, d)を用いて一括化して転送され、そのタイミングで転送が不要な変数は、図7の符号oを含む二点鎖線枠囲みに示すタイミングで既にGPUに変数があることを明示するdata present文#pragma acc data present(c, d)を用いて指定する。

<1>、<3>のループがGPU処理されGPU処理が終了したタイミングで、GPUからCPUへのデータ転送指示行、ここでは変数a, b, c, dのcopyout節の#pragma acc datacopyout(a, b, c, d)を、図7の<3>ループが終了した位置pに挿入する。

[0113] 一括化して転送する指定により一括化して転送できる変数は一括転送し、既に転送され転送が不要な変数はdata presentを用いて明示することで、転送を削減して、オフロード手段のさらなる効率化を図ることができる。しかし、OpenACCで転送を指示してもコンパイラによっては、コンパイラが自動判断して転送してしまう場合がある。コンパイラによる自動転送とは、OpenACCの指示と異なり、本来はCPU-GPU間の転送が不要であるにもかかわらずコンパイラ依存で自動転送されてしまう事象のことである。

[0114] 《データの一時領域格納》

図8は、本実施形態のCPU-GPUのデータ転送時の転送一括化によるループ文を示す図である。図8は、図7のネスト一括化および転送不要な変数明示に対応する。

図8に示すループ文では、図8の符号qに示す位置に、CPU-GPUのデータ転送時、一時領域を作成するOpenACCのdeclare create文#pragma acc declare createを指定する。これにより、CPU-GPUのデータ転送時は、一時領域を作成し（#pragma acc declare create）、データは一時領域に格納される。

[0115] また、図8の符号rに示す位置に、一時領域を同期するためのOpenACCのdeclare create文#pragma acc updateを指定することで転送を指示する。

[0116] このように、一時領域を作成し、一時領域でパラメータを初期化して、CPU-GPU転送に用いることで、不要なCPU-GPU転送を遮断する。OpenACCの指示では意図しないが性能を劣化する転送を削減することができる。

[0117] [GPUオフロード処理]

上述したデータ転送の一括処理手法により、オフロードに適切なループ文を抽出し、非効率なデータ転送を避けることができる。

ただし、上記データ転送の一括処理手法を用いても、GPUオフロードに向いていないプログラムも存在する。効果的なGPUオフロードには、オフロードする処理のループ回数が多いことが必要である。

[0118] そこで、本実施形態では、本格的なオフロード処理探索の前段階として、プロファイリングツールを用いて、ループ回数を調査する。プロファイリングツールを用いると、各行の実行回数を調査できるため、例えば、5000万回以上のループを持つプログラムをオフロード処理探索の対象とする等、事前に振り分けることができる。以下、具体的に説明する（図2で述べた内容と一部重複する）。

[0119] 本実施形態では、まず、アプリケーションコード分析部112（図1）がアプリケーションを分析し、for, do, while等のループ文を把握する。次に、サンプル処理を実行し、プロファイリングツールを用いて、各ループ文のループ回数を調査し、一定の値以上のループがあるか否かで、探索を本格的に行うか否かの判定を行う。

[0120] 探索を本格的に行うと決まった場合は、GAの処理に入る（図2参照）。

初期化ステップでは、アプリケーションコードの全ループ文の並列可否をチェックした後、並列可能ループ文をGPU処理する場合は1、しない場合は0として遺伝子配列にマッピングする。遺伝子は、指定の個体数が準備されるが、遺伝子の各値にはランダムに1, 0の割り当てをする。

[0121] ここで、遺伝子に該当するコードでは、GPU処理すると指定されたループ文内の変数データ参照関係から、データ転送の明示的指示（`#pragma acc data copyin/copyout/copy`）を追加する。

[0122] 評価ステップでは、遺伝子に該当するコードをコンパイルして検証用マシンにデプロイして実行し、ベンチマーク性能測定を行う。そして、性能が良いパターンの遺伝子の適合度を高くする。遺伝子に該当するコードは、上述のように、並列処理指示行（例えば、図4の符号f参照）とデータ転送指示行（例えば、図4の符号h参照、図5の符号i参照、図6の符号k参照）が挿入されている。

[0123] 選択ステップでは、適合度に基づいて、高適合度の遺伝子を、指定の個体数分選択する。本実施形態では、適合度に応じたルーレット選択および最高適合度遺伝子のエリート選択を行う。交叉ステップでは、一定の交叉率 P_c で、選択された個体間で一部の遺伝子をある一点で交換し、子の個体を作成する。突然変異ステップでは、一定の突然変異率 P_m で、個体の遺伝子の各値を0から1または1から0に変更する。

[0124] 突然変異ステップまで終わり、次の世代の遺伝子が指定個体数作成されると、初期化ステップと同様に、データ転送の明示的指示を追加し、評価、選択、交叉、突然変異ステップを繰り返す。

[0125] 最後に、終了判定ステップでは、指定の世代数、繰り返しを行った後に処理を終了し、最高適合度の遺伝子を解とする。最高適合度の遺伝子に該当する、最高性能のコードパターンで、本番環境に改めてデプロイして、ユーザーに提供する。

[0126] 以下、オフロードサーバ1の実装を説明する。本実装は、本実施形態の有効性を確認するためのものである。

[実装]

C/C++アプリケーションを汎用のPGIコンパイラを用いて自動オフロードする実装を説明する。

本実装では、GPU自動オフロードの有効性確認が目的であるため、対象アプリケーションはC/C++言語のアプリケーションとし、GPU処理自体は、従来のPGIコンパイラを説明に用いる。

[0127] C/C++言語は、OSS (Open Source Software) およびproprietaryソフトウェアの開発で、上位の人気を誇り、数多くのアプリケーションがC/C++言語で開発されている。一般ユーザが用いるアプリケーションのオフロードを確認するため、暗号処理や画像処理等のOSSの汎用アプリケーションを利用する。

[0128] GPU処理は、PGIコンパイラにより行う。PGIコンパイラは、OpenACCを解釈するC/C++/Fortran向けコンパイラである。本実施形態では、for文等の並列可能処理部を、OpenACCのディレクティブ `#pragma acc kernels` (並列処理指定文) で指定する。これにより、GPU向けバイトコードを抽出し、その実行によりGPUオフロードを可能としている。さらに、for文内のデータ同士に依存性があり並列処理できない処理やネストのfor文の異なる複数の階層を指定されている場合等の際に、エラーを出す。併せて、`#pragma acc data copyin/copyout/copy` 等のディレクティブにより、明示的なデータ転送の指示が可能とする。

[0129] 上記 `#pragma acc kernels` (並列処理指定文) での指定に合わせて、OpenACCの`copyin` 節の`#pragma acc data copyout(a[...])`の、上述した位置への挿入により、明示的なデータ転送の指示を行う。

[0130] <実装の動作概要>

実装の動作概要を説明する。

実装は、以下の処理を行う。

下記図9A-Bのフローの処理を開始する前に、高速化するC/C++アプリケーションとそれを性能測定するベンチマークツールを準備する。

- [0131] 実装では、C/C++アプリケーションの利用依頼があると、まず、C/C++アプリケーションのコードを解析して、for文を発見するとともに、for文内で使われる変数データ等の、プログラム構造を把握する。構文解析には、LLVM/Clangの構文解析ライブラリ等を使用する。
- [0132] 実装では、最初に、そのアプリケーションがGPUオフロード効果があるかの見込みを得るため、ベンチマークを実行し、上記構文解析で把握したfor文のループ回数を把握する。ループ回数把握には、GNUカバレッジのgcov等を用いる。プロファイリングツールとしては、「GNUプロファイラ(gprof)」、「GNUカバレッジ(gcov)」が知られている。双方とも各行の実行回数を調査できるため、どちらを用いてもよい。実行回数は、例えば、1000万回以上のループ回数を持つアプリケーションのみ対象とするようにできるが、この値は変更可能である。
- [0133] CPU向け汎用アプリケーションは、並列化を想定して実装されているわけではない。そのため、まず、GPU処理自体が不可なfor文は排除する必要がある。そこで、各for文一つずつに対して、GPU処理の#pragma acc kernelsや#pragma acc parallel loopや#pragma acc parallel loop vectorディレクティブ挿入を試行し、コンパイル時にエラーが出るかの判定を行う。コンパイルエラーに関しては、幾つかの種類がある。for文の中で外部ルーチンが呼ばれている場合、ネストfor文で異なる階層が重複指定されている場合、break等でfor文を途中で抜ける処理がある場合、for文のデータにデータ依存性がある場合等がある。アプリケーションによって、コンパイル時エラーの種類は多彩であり、これ以外の場合もあるが、コンパイルエラーは処理対象外とし、#pragmaディレクティブは挿入しない。
- [0134] コンパイルエラーは自動対処が難しく、また対処しても効果が出ないことも多い。外部ルーチンコールの場合は、#pragma acc routineにより回避できる場合があるが、多くの外部コールはライブラリであり、それを含めてGPU処理してもそのコールがネックとなり性能が出ない。for文一つずつを試行するため、ネストのエラーに関しては、コンパイルエラーは生じない。また

、break等により途中で抜ける場合は、並列処理にはループ回数を固定化する必要があり、プログラム改造が必要となる。データ依存が有る場合はそもそも並列処理自体ができない。

[0135] ここで、並列処理してもエラーが出ないループ文の数がaの場合、aが遺伝子長となる。遺伝子の1は並列処理ディレクティブ有、0は無に対応させ、長さaの遺伝子に、アプリケーションコードをマッピングする。

[0136] 次に、初期値として、指定個体数の遺伝子配列を準備する。遺伝子の各値は、図3で説明したように、0と1をランダムに割当てて作成する。準備された遺伝子配列に応じて、遺伝子の値が1の場合はGPU処理を指定するディレクティブ `¥#pragma acc kernels`, `¥#pragma acc parallel loop`, `¥#pragma acc parallel loop vector`をC/C++コードに挿入する。single loop等はparallelにしない理由としては、同じ処理であればkernelsの方が、PGIコンパイラとしては性能が良いためである。この段階で、ある遺伝子に該当するコードの中で、GPUで処理させる部分が決まる。

[0137] 並列処理およびデータ転送のディレクティブを挿入されたC/C++コードを、GPUを備えたマシン上のPGIコンパイラでコンパイルを行う。コンパイルした実行ファイルをデプロイし、ベンチマークツールで性能と電力使用量を測定する。

[0138] 全個体数に対して、ベンチマーク性能測定後、ベンチマーク処理時間と電力使用量に応じて、各遺伝子配列の適合度を設定する。設定された適合度に応じて、残す個体の選択を行う。選択された個体に対して、交叉処理、突然変異処理、そのままコピー処理のGA処理を行い、次世代の個体群を作成する。

[0139] 次世代の個体に対して、ディレクティブ挿入、コンパイル、性能測定、適合度設定、選択、交叉、突然変異処理を行う。ここで、GA処理の中で、以前と同じパターンの遺伝子が生じた場合は、その個体についてはコンパイル、性能測定をせず、以前と同じ測定値を用いる。

[0140] 指定世代数のGA処理終了後、最高性能の遺伝子配列に該当する、ディレ

クティブ付きC/C++コードを解とする。

[0141] この中で、個体数、世代数、交叉率、突然変異率、適合度設定、選択方法は、GAのパラメータであり、別途指定する。提案技術は、上記処理を自動化することで、従来、専門技術者の時間とスキルが必要だった、GPUオフロードの自動化を可能にする。

[0142] 図9A-Bは、上述した実装の動作概要を説明するフローチャートであり、図9Aと図9Bは、結合子で繋がれる。

C/C++向けOpenACCコンパイラを用いて以下の処理を行う。

[0143] <コード解析>

ステップS101で、アプリケーションコード分析部112（図1参照）は、C/C++アプリのコード解析を行う。

[0144] <ループ文特定>

ステップS102で、並列処理指定部114（図1参照）は、C/C++アプリのループ文、参照関係を特定する。

[0145] <ループ文の並列処理可能性>

ステップS103で、並列処理指定部114は、各ループ文のGPU処理可能性をチェックする（`#pragma acc kernels`）。

[0146] <ループ文の繰り返し>

制御部（自動オフロード機能部）11は、ステップS104のループ始端とステップS117のループ終端間で、ステップS105-S116の処理についてループ文の数だけ繰り返す。

[0147] <ループの数の繰り返し（その1）>

制御部（自動オフロード機能部）11は、ステップS105のループ始端とステップS108のループ終端間で、ステップS106-S107の処理についてループ文の数だけ繰り返す。

ステップS106で、並列処理指定部114は、各ループ文に対して、OpenACCでGPU処理（`#pragma acc kernels`）を指定してコンパイルする。

ステップS107で、並列処理指定部114は、エラー時は、次の指示句

でGPU処理可能性をチェックする(#pragma acc parallel loop)。

[0148] <ループの数の繰り返し(その2)>

制御部(自動オフロード機能部)11は、ステップS109のループ始端とステップS112のループ終端間で、ステップS110-S111の処理についてループ文の数だけ繰り返す。

ステップS110で、並列処理指定部114は、各ループ文に対して、OpenACCでGPU処理(#pragma acc parallel loop)を指定してコンパイルする。

ステップS111で、並列処理指定部114は、エラー時は、次の指示句でGPU処理可能性をチェックする(#pragma acc parallel loop vector)。

[0149] <ループの数の繰り返し(その3)>

制御部(自動オフロード機能部)11は、ステップS113のループ始端とステップS116のループ終端間で、ステップS114-S115の処理についてループ文の数だけ繰り返す。

ステップS114で、並列処理指定部114は、各ループ文に対して、OpenACCでGPU処理(#pragma acc parallel loop vector)を指定してコンパイルする。

ステップS115で、並列処理指定部114は、エラー時は、当該ループ文からはGPU処理指示句を除去する。

[0150] <for文の数カウント>

ステップS118で、並列処理指定部114は、コンパイルエラーが出ないループ文(ここではfor文)の数をカウントし、遺伝子長とする。

[0151] <指定個体数パターン準備>

次に、初期値として、並列処理指定部114は、指定個体数の遺伝子配列を準備する。ここでは、0と1をランダムに割当てて作成する。

ステップS119で、並列処理指定部114は、C/C++アプリコードを、遺伝子にマッピングし、指定個体数パターン準備を行う。

準備された遺伝子配列に応じて、遺伝子の値が1の場合は並列処理を指定するディレクティブをC/C++コードに挿入する（例えば図3の#pragmaディレクティブ参照）。

[0152] 制御部（自動オフロード機能部）11は、図9BのステップS120のループ始端とステップS131のループ終端間で、ステップS121－S130の処理について指定世代数繰り返す。

また、上記指定世代数繰り返しにおいて、さらにステップS121のループ始端とステップS126のループ終端間で、ステップS122－S125の処理について指定個体数繰り返す。すなわち、指定世代数繰り返しの中で、指定個体数の繰り返しが入れ子状態で処理される。

[0153] <データ転送指定>

ステップS122で、データ転送指定部113は、変数参照関係をもとに、明示的指示行（#pragma acc data copy/copyin/copyout/presentおよび#pragma acc declarecreate, #pragma acc update）を用いたデータ転送指定を行う。

[0154] <コンパイル>

ステップS123で、並列処理パターン作成部117（図1参照）は、遺伝子パターンに応じてディレクティブ指定したC/C++コードをPGIコンパイラでコンパイルする。すなわち、並列処理パターン作成部117は、作成したC/C++コードを、GPUを備えた検証用マシン14上のPGIコンパイラでコンパイルを行う。

ここで、ネストfor文を複数並列指定する場合等でコンパイルエラーとなることがある。この場合は、性能測定時の処理時間がタイムアウトした場合と同様に扱う。

[0155] ステップS124で、性能測定部118（図1参照）は、CPU-GPU搭載の検証用マシン14に、実行ファイルをデプロイする。

ステップS125で、性能測定部118は、配置したバイナリファイルを実行し、オフロードした際のベンチマーク性能を測定する。

[0156] ここで、途中世代で、以前と同じパターンの遺伝子については測定せず、同じ値を使う。つまり、GA処理の中で、以前と同じパターンの遺伝子が生じた場合は、その個体についてはコンパイルや性能測定をせず、以前と同じ測定値を用いる。

[0157] ステップS 1 2 7で、性能測定部 1 1 8（図 1 参照）は、処理時間を測定する。

[0158] ステップS 1 2 8で、性能測定部 1 1 8は、測定した処理時間をもとに評価値を設定する。

[0159] ステップS 1 2 9で、実行ファイル作成部 1 1 9（図 1 参照）は、処理時間の短い個体ほど適合度が高くなるように評価し、性能の高い個体を選択する。実行ファイル作成部 1 1 9は、測定された複数パターンの中で、短時間かつ低電力使用量のパターンを解として選択する。

[0160] ステップS 1 3 0で、実行ファイル作成部 1 1 9は、選択された個体に対して、交叉、突然変異の処理を行い、次世代の個体を作成する。実行ファイル作成部 1 1 9は、次世代の個体に対して、コンパイル、性能測定、適合度設定、選択、交叉、突然変異処理を行う。

すなわち、全個体に対して、ベンチマーク性能測定後、ベンチマーク処理時間に応じて、各遺伝子配列の適合度を設定する。設定された適合度に応じて、残す個体の選択を行う。実行ファイル作成部 1 1 9は、選択された個体に対して、交叉処理、突然変異処理、そのままコピー処理のGA処理を行い、次世代の個体群を作成する。

[0161] ステップS 1 3 2で、実行ファイル作成部 1 1 9は、指定世代数のGA処理終了後、最高性能の遺伝子配列に該当するC/C++コード（最高性能の並列処理パターン）を解とする。

[0162] <GAのパラメータ>

上記、個体数、世代数、交叉率、突然変異率、適合度設定、選択方法は、GAのパラメータである。GAのパラメータは、例えば、以下のように設定してもよい。

実行するSimple GAの、パラメータ、条件は例えば以下のようにできる。

遺伝子長：並列可能ループ文数

個体数M：遺伝子長以下

世代数T：遺伝子長以下

適合度：(処理時間)^(-1/2)

- [0163] この設定により、ベンチマーク処理時間が短い程、高適合度になる。また、適合度を、処理時間の(-1/2)乗を含む形とすることで、処理時間が短い特定の個体の適合度が高くなり過ぎて、探索範囲が狭くなるのを防ぐことができる。また、性能測定が一定時間で終わらない場合は、タイムアウトさせ、処理時間1000秒等の時間（長時間）であるとして、適合度を計算する。このタイムアウト時間は、性能測定特性に応じて変更させればよい。

選択：ルーレット選択

ただし、世代での最高適合度遺伝子は交叉も突然変異もせず次世代に保存するエリート保存も合わせて行う。

交叉率P_c：0.9

突然変異率P_m：0.05

- [0164] <コストパフォーマンス>

自動オフロード機能のコストパフォーマンスについて述べる。

NVIDIA Tesla等の、GPUボードのハードウェアの価格だけを見ると、GPUを搭載したマシンの価格は、通常のCPUのみのマシンの約2倍となる。しかし、一般にデータセンタ等のコストでは、ハードウェアやシステム開発のコストが1/3以下であり、電気代や保守・運用体制等の運用費が1/3超であり、サービスオーダ等のその他費用が1/3程度である。本実施形態では、暗号処理や画像処理等動作させるアプリケーションで時間がかかる処理を2倍以上高性能化できる。このため、サーバハードウェア価格自体は2倍となっても、コスト効果が十分に期待できる。

- [0165] 本実施形態では、gcov, gprof等を用いて、ループが多く実行時間がかかっているアプリケーションを事前に特定して、オフロード試行をする。これに

より、効率的に高速化できるアプリケーションを見つけることができる。

[0166] <本番サービス利用開始までの時間>

本番サービス利用開始までの時間について述べる。

コンパイルから性能測定1回は3分程度とすると、20の個体数、20の世代数のGAで最大20時間程度解探索にかかるが、以前と同じ遺伝子パターンのコンパイル、測定は省略されるため、8時間以下で終了する。多くのクラウドやホスティング、ネットワークサービスではサービス利用開始に半日程度かかるのが実情である。本実施形態では、例えば半日以内の自動オフロードが可能である。このため、半日以内の自動オフロードであれば、最初は試し利用ができるとすれば、ユーザ満足度を十分に高めることが期待できる。

[0167] より短時間でオフロード部分を探索するためには、複数の検証用マシンにより個体数分並列で性能測定することが考えられる。アプリケーションに応じて、タイムアウト時間を調整することも短時間化に繋がる。例えば、オフロード処理がCPUでの実行時間の2倍かかる場合はタイムアウトとする等である。また、個体数、世代数が多い方が、高性能な解を発見できる可能性が高まる。しかし、各パラメータを最大にする場合、個体数×世代数だけコンパイル、および性能ベンチマークを行う必要がある。このため、本番サービス利用開始までの時間がかかる。本実施形態では、GAとしては少ない個体数、世代数で行っているが、交叉率 P_c を0.9と高い値にして広範囲を探索することで、ある程度の性能の解を早く発見するようにしている。

[0168] [指示句の拡大]

本実施形態では、適用できるアプリケーション増加のため、指示句の拡大を行う。具体的には、GPU処理を指定する指示句として、`kernels`指示句に加えて、`parallel loop`指示句、`parallel loop vector`指示句にも拡大する。

OpenACC標準では、`kernels`は、`single loop`や`tightly nested loop`に使われる。また、`parallel loop`は、`non-tightly nested loop`も含めたループに使われる。`parallel loop vector`は、`parallelize`はできないが`vectorize`は

できるループに使われる。ここで、tightly nested loopとは、ネストループにて、例えば、iとjをインクリメントする二つのループが入れ子になっている時、下位のループでiとjを使った処理がされ、上位ではされないような単純なループである。また、PGIコンパイラ等の実装においては、kernelsは、並列化の判断はコンパイラが行い、parallelは並列化の判断はプログラマが行うという違いがある。

[0169] そこで、本実施形態では、single、tightly nested loopにはkernelsを使い、non-tightly nested loopにはparallel loopを使う。また、parallelizeできないがvectorizeできるループにはparallel loop vectorを使う。

ここで、parallel指示句にすることで、結果がkernelsの場合より信頼度が下がる懸念がある。しかし、最終的なオフロードプログラムに対して、サンプルテストを行い、CPUとの結果差分をチェックしその結果をユーザに見せて、ユーザに確認してもらうことを想定している。そもそも、CPUとGPUではハードが異なるため、有効数字桁数や丸め誤差の違い等があり、kernelsだけでもCPUとの結果差分のチェックは必要である。

[0170] [リソース比とリソース量の設定、および新規アプリケーションの配置フローチャート]

図10は、GPUオフロード試行の後に追加されるリソース比とリソース量の設定および新規アプリケーションの配置を説明するフローチャートである。図10に示すフローチャートは、図9A-Bに示すGPUオフロード試行後に実行される。

[0171] ステップS51でリソース比決定部115は、ユーザ運用条件、テストケースCPU処理時間、オフロードデバイス処理時間を取得する。ユーザ運用条件は、ユーザがオフロードしたいコードを指定する際に合わせてユーザに指定してもらう。ユーザ運用条件は、リソース量設定部116が、設備リソースDB132の情報を参照してリソース量を決定する際に利用する。

[0172] <リソース比決定部115の処理>

ステップS52でリソース比決定部115は、性能測定結果をもとに、C

PUとオフロードデバイスの処理時間（テストケースCPU処理時間とオフロードデバイス処理時間）の比を、リソース比として決定する。

[0173] 本自動オフロードによって、コード変換の際には、既に検証環境での性能測定結果が得られている。この性能測定結果を用いて、リソース比決定部115は、CPUとオフロードデバイスのリソース比を決定する。具体的には、検証環境でのCPUとオフロードデバイスの処理時間の比に対して、適正なリソース比を決定する。例えば、検証環境でのテストケース処理時間が、CPU処理：10秒、GPU処理：5秒の場合は、リソース比は、CPU：GPU=2：1となる。

[0174] リソース比決定部115は、CPUとオフロードデバイスの処理時間が同等オーダになるように、リソース比を決定する。CPUとオフロードデバイスの処理時間が同等オーダになるように、リソース比を決定することで、CPUとオフロードデバイスの処理時間を揃え、CPUとアクセラレータがGPU、FPGA、メニーコアCPU等の混在環境であってもリソース量を適切に設定することができる。

[0175] リソース比決定部115は、CPUとオフロードデバイスの処理時間の差分が所定閾値以上の場合、リソース比を所定の上限値に設定する。すなわち、検証環境でのCPUとオフロードデバイスの処理時間が、例えば10倍以上差分がある場合にリソース比を10倍以上にしてしまうと、コストパフォーマンス悪化につながる。この場合は、例えば、5：1等のリソース比を上限にする（上限値は、処理時間の5：1のリソース比）。リソース比に上限を設けることで、VM数の大幅増加を防ぐことができる。

[0176] <リソース量設定部116の処理>

ステップS53でリソース量設定部116は、ユーザ運用条件と適切リソース比をもとに、リソース量を設定する。すなわち、リソース量設定部116は、ユーザが指定したコスト条件を満たすように、リソース比はできるだけキープして、リソース量を決定する。

[0177] リソース量設定部116は、適切リソース比を維持して、ユーザ運用条件

を満たす最大のリソース量を設定する。具体例を挙げると、CPU 1 VMは1000円/月、GPUは4000円/月、リソース比は2 : 1が適切であると、ユーザは月10000円以内の予算であるとする。この場合には、CPUは2、GPUは1を確保して商用環境に配置する。

[0178] リソース量設定部116は、リソース比を維持した最小リソース量でもユーザ運用条件を満たさない場合は、リソース比を崩してCPUとオフロードデバイスのリソース量をコスト条件を満たすよう最小で設定する。具体例を挙げると、CPU 1 VMは1000円/月、GPUは4000円/月、リソース比は2 : 1が適切であると、ユーザは月5000円以内の予算であるとする。この場合には、ユーザ予算が足りないため、リソース比はキープできないが、CPUとオフロードデバイスのリソース量をより小さく設定、すなわちCPUは1、GPUは1を確保して配置する。

[0179] 上記ステップS53の処理を終え、商用環境にリソースを確保して配置した後は、ユーザが利用する前に、性能およびコストを確認するため、図2で述べた自動検証を実行する。これにより、商用環境でリソースを確保して、自動検証後、性能とコストをユーザに提示することができる。

[0180] <リソース比とリソース量の設定のまとめ>

リソース比を適切化するため、オフロードパターンの解を決める際の性能測定結果を用いる。実装は、テストケースの処理時間から、CPUとGPUの処理時間が同等オーダになるようリソース比を定める。例えば、テストケースの処理時間が、CPU処理 : 10秒、GPU処理 : 5秒の場合では、CPU側のリソースは2倍で同等の処理時間程度と考えられるため、リソース比は2 : 1となる。なお、仮想マシン等の数は整数となるため、リソース比は処理時間から計算する際に、整数比となるように四捨五入する。

[0181] リソース比が決定されると、次に、商用環境へのアプリケーション配置を行う際のリソース量の設定を行う。実装は、リソース量決定には、ユーザがオフロード依頼時に指定したコスト要求を満たすように、リソース比はできるだけキープして、VM等の数を定める。具体的には、コスト範囲内で、リ

ソース比をキープする中では、VM等の数は最大値を選択する。

[0182] 例えば、CPUに関して1VMは1000円/月、GPUは4000円/月、リソース比は2:1が適切であり、ユーザは月10000円以内の予算であった場合には、CPUは2、GPUは1を確保する。また、コスト範囲内で、リソース比をキープできない場合は、CPU1単位、GPU1単位から始めてできるだけ適切なリソース比に近くなるよう、リソース量を設定する。例えば、月5000円以内の予算であった場合には、リソース比はキープできないが、CPUは1、GPUは1を確保する。

リソース量を設定すると、実装では、例えばXen Serverの仮想化機能を用いて、CPUやGPUのリソースを割り当てる。

[0183] ステップS54で配置設定部170は、設備リソースDB132のサーバ、リンクのスペック情報、既存アプリケーションの配置情報に基づいて、線形計画手法を用いて、新規アプリケーションの配置先（APLの配置場所）を計算して設定する。

[0184] [変換したアプリケーションの最適配置]

本実施形態のオフロードサーバ1は、CPU向けプログラムを、GPU等のデバイスにオフロードした際に、アプリケーションをユーザのコスト等要求を満たして、応答時間等を短く動作するように、配置先を適正化する。

[0185] <アプリケーション配置場所の適切化>

本実施形態では、アプリケーションはクラウドだけでなく、ネットワークエッジやユーザエッジに配置できることを前提とする。ただし、ネットワークエッジやユーザエッジは、クラウドに比べサーバの集約度が低く分散している。このため、計算リソースのコストは、クラウドに比べ割高となる。すなわち、一般にCPUやGPU等のハードウェアの価格は配置場所によらず一定であるものの、クラウドを運用するデータセンタでは集約されたサーバをまとめて監視や空調制御等できるため、運用費が割安となる。

例えば、計算ノードリンクの簡単なトポロジーとしては、図11が挙げられる。

[0186] 図11は、計算ノードのトポロジーの一例を示す図である。図11は、IoTシステムのように、ユーザ環境でデータを収集するIoTデバイス等から、ユーザエッジにデータが送られ、ネットワークエッジを介してクラウドにデータが送られ、分析結果を会社の幹部が見る等で使われるトポロジーである。

[0187] 図11に示すように、アプリケーションを配置するトポロジーは、3層で構成され、クラウドレイヤー（例えば、データセンタ）の拠点数は「2」（n13, n14）、キャリアエッジレイヤー（例えば、局舎）は「3」、ユーザエッジレイヤー（例えば、ユーザ環境）は「4」（n6－n9）、インプットノードは「5」（n1－n5）とする。

IOT等のアプリケーションを想定してインプットノードからIoTデータ（IoTデバイスの一つである花粉センサや体温センサ等）がユーザエッジに収集され、アプリケーションの特性（応答時間の要求条件等）に応じて、ユーザエッジ、キャリアエッジで分析処理がされたり、クラウドまでデータをあげてから分析処理されたりされる。アウトプットノードは「1」（n15）であり、分析結果を会社の幹部が見る。例えば、インプットノードがIoTデータ（花粉センサ）の場合は、アウトプットノードの統計・分析結果は気象庁の責任者が確認する。

図11に示す配置トポロジー3層は、一例であり、例えば5層であってもよい。また、ユーザエッジ、キャリアエッジの数は、実際には数十～数百の場合もある。

[0188] 計算ノードは、CPU、GPU、FPGAの3種に分けられる。GPUやFPGAを備えるノードには、CPUも搭載されているが、仮想化技術（例えば、NVIDIA vGPU）により、GPUインスタンス、FPGAインスタンスとして、CPUリソースも含む形で分割して提供される。

[0189] アプリケーションは、クラウド、キャリアエッジ、ユーザエッジに配置され、ユーザ環境に近い側程、応答時間を低減することが可能になる代わりに、計算リソースのコストが高くなる。本実施形態では、GPUやFPGA向

けに変換したアプリケーションを配置することになるが、配置する際に、ユーザは2種類のリクエストを発出できる。

一つ目は、コスト要求であり、アプリケーションを動作させるために許容できる計算リソースのコストを指定する形で、例えば月5000円以内で動作させる等である。二つ目は、応答時間要求であり、アプリケーションを動作させる際の許容応答時間を指定する形で、例えば10秒以内に応答を返す等である。従来から行われている設備設計では、例えば仮想ネットワークを収容するサーバを配置する場所を、トラフィック増加量等の長期的傾向を見て、計画的に設計している。

[0190] 本実施形態では、下記(1)、(2)の特徴がある。(1)配置されるアプリケーションは静的に定まっているのではなく、GPUやFPGA向けに自動変換され、GA等を通じて利用形態に適したパターンが実測を通じて抽出される。このため、アプリケーションのコードや性能は動的に変わり得る。

(2)キャリアの設備コストや全体的応答時間だけを低減すればよいのではなく、計算リソースのコストや応答時間に対する個々のユーザ要求を満たす必要がある。また、アプリケーションの配置ポリシーも動的に変わり得る。

[0191] 上記(1)、(2)の特徴も踏まえ、本実施形態のアプリケーション配置は、ユーザからの配置依頼があった場合、変換を行い、変換したアプリケーションをその時点で適切なサーバに順次配置していく形とする。アプリケーションを変換しても、コストパフォーマンスが向上しない場合は、変換前のアプリケーション配置とする。例えば、GPUインスタンスはCPUインスタンスの2倍のコストがかかる際に、変換しても2倍以上性能が改善されないならば、変換前を配置した方がよい。また、既に上限まで計算リソースや帯域が使われてしまっている場合はそのサーバには配置はできないことがある。

[0192] <アプリケーション適切配置のための線形計画式>

本実施形態では、アプリケーションの適切な配置場所を計算するための、線形計画手法の定式化を行う。線形計画手法は、具体的には、[式1]（以下の式（1）～式（4））、[式2]（以下の式（3）～式（6））に示す線形計画式のパラメータを用いる。

[0193] ここで、デバイスやリンクのコストや計算リソース上限、帯域上限等は、事業者が準備するサーバやネットワークに依存する。このため、それらのパラメータ値は事業者が事前に設定する。オフロードした際にアプリケーションが使用する計算リソース量、帯域、データ容量、処理時間は、自動変換する前の検証環境での試験での最終的に選択されたオフロードパターンでの計測値により決まり、環境適応機能により自動設定される。

ユーザ要求が計算リソースのコスト要求であるかまたは応答時間要求であるかで、線形計画式のパラメータにおける、目的関数と制約条件が変わる。

[0194] ・コスト要求による、線形計画式のパラメータ

コスト要求により、一月幾ら以内での配置が必要な要求の場合は、下記[式1]に示す線形計画式のパラメータを用いる。

[0195]

[数1]

$$R_k = \sum_{i \in Device} (A_{i,k}^d \cdot B_{i,k}^p) + \sum_{j \in Link} (A_{i,k}^l \cdot \frac{C_k}{B_k^l}) \quad \dots(1)$$

$$\sum_{i \in Device} a_i (\frac{A_{i,k}^d \cdot B_k^d}{C_i^d}) + \sum_{j \in Link} b_j (\frac{A_{j,k}^l \cdot B_k^l}{C_j^l}) \leq P_k \quad \dots(2)$$

$$\sum_{k \in App} (A_{i,k}^d \cdot B_k^d) \leq C_i^d \quad \dots(3)$$

$$\sum_{k \in App} (A_{j,k}^l \cdot B_k^l) \leq C_j^l \quad \dots(4)$$

a_i : デバイス使用コスト

b_j : リンク使用コスト

C_i^d : i 番目のデバイスの計算リソース上限

C_j^l : j 番目のリンクの帯域上限

C_k : k 番目のアプリで使うデータ容量

$A_{i,k}^d$: i 番目のデバイスで k 番目のアプリ使用有無

$A_{i,k}^l$: j 番目のリンクで k 番目のアプリ使用有無

B_k^d : k 番目のアプリが使用する計算リソース量

B_k^l : k 番目のアプリが使用する帯域

$B_{i,k}^p$: i 番目のデバイスでの k 番目アプリの処理時間

[0196] 式（１）の応答時間の最小化が目的関数である。式（２）の計算リソースのコストがいくら以内であるかは、制約条件の一つである。さらに、式（３）（４）のサーバのリソース上限を超えていないかの制約条件も加わる。

[0197] ・ 応答時間要求による、線形計画式のパラメータ

応答時間要求により、アプリケーションの応答時間が何秒以内での配置が必要な要求の場合は、下記〔式２〕に示す線形計画式のパラメータを用いる

。

[0198]

[数2]

$$P_k = \sum_{i \in Device} a_i \left(\frac{A_{i,k}^d \cdot B_k^d}{C_i^d} \right) + \sum_{j \in Link} b_j \left(\frac{A_{j,k}^l \cdot B_k^l}{C_j^l} \right) \quad \dots (5)$$

$$\sum_{i \in Device} (A_{i,k}^d \cdot B_{i,k}^p) + \sum_{j \in Link} (A_{j,k}^l \cdot \frac{C_k}{B_k^l}) \leq R_k \quad \dots (6)$$

$$\sum_{k \in App} (A_{i,k}^d \cdot B_k^d) \leq C_i^d \quad \dots (3)$$

$$\sum_{k \in App} (A_{j,k}^l \cdot B_k^l) \leq C_j^l \quad \dots (4)$$

a_i : デバイス使用コスト

b_j : リンク使用コスト

C_i^d : i 番目のデバイスの計算リソース上限

C_j^l : j 番目のリンクの帯域上限

C_k : k 番目のアプリで使うデータ容量

$A_{i,k}^d$: i 番目のデバイスで k 番目のアプリ使用有無

$A_{j,k}^l$: j 番目のリンクで k 番目のアプリ使用有無

B_k^d : k 番目のアプリが使用する計算リソース量

B_k^l : k 番目のアプリが使用する帯域

$B_{i,k}^p$: i 番目のデバイスでの k 番目アプリの処理時間

[0199] 式(2)に対応する式(5)の計算リソースのコストの最小化が目的関数である。式(1)に対応する式(6)の応答時間が何秒以内であるかは、制約条件の一つである。さらに、式(3)(4)の制約条件も加わる。

[0200] ・線形計画式のパラメータの説明

式(1)および式(6)は、アプリケーション k の応答時間を計算するための式であり、式(1)の場合は R_k が目的関数、式(6)の場合は R_k がユーザが指定した上限を設定する制約条件である。

[0201] 式(2)および式(5)は、アプリケーション k を動作させるコスト(価格) P_k を計算するための式であり、式(2)の場合は P_k がユーザが指定した上限を設定する制約条件、式(5)の場合は P_k が目的関数である。

[0202] 式(3)および式(4)は、計算リソースおよび通信帯域の上限を設定する制約条件であり、他者が配置したアプリケーション含めて計算され、新規ユーザのアプリケーション配置によるリソース上限の超過を防ぐ。

[0203] 式(1)乃至式(4)および、式(3)乃至式(6)の線形計画式を、ネットワークポロジや変換アプリケーションタイプ(CPUに対するコスト増と性能増等)、ユーザ要求、既配置アプリケーションの異なる条件に対して、GLPK(Gnu Linear Programming Kit)やCPLX(IBM Decision Optimization)等の線形計画ソルバで解を導出することで、適切なアプリケーション配置を計算できる。適切配置計算後に実際の配置を、複数のユーザに対して、順次行っていくことで、複数のアプリケーションが各ユーザの要求に基づいて配置される。

[0204] 以上のように、線形計画式に基づいて、新規にアプリケーションの配置依頼があった場合に計算し、順に配置することで、ユーザ要望を満たした配置が可能である。

ここで、アプリケーションプログラムの配置は、順次行われるため早い者勝ちと言えるが、アプリケーション100個毎等、定期的に、既に配置済みのアプリケーションプログラム群の適正配置を再計算する。そして、ユーザの指定するコスト、応答時間に応じて、目的関数が極小化される配置を計算

し、計算で定まった位置に、アプリケーションを再配置してもよい。

[0205] [評価]

線形計画手法の一態様である線形計画式に基づき、無償ソルバのGLPK（登録商標）を用いて、複数のアプリケーションが適切に配置されていくことを、いくつかの条件を変更して確認した。

[0206] <評価条件>

・対象アプリケーション

配置対象のアプリケーションは、多くのユーザが利用すると想定されるフーリエ変換による画像処理をする。フーリエ変換処理（FFT）は、振動周波数の分析等、IoTでのモニタリングの様々な場面で利用されている。

NAS.FT (<https://www.nas.nasa.gov/publications/npb.html>)（登録商標）は、FFT処理のオープンソースアプリケーションの一つである。備え付けのサンプルテストの 2048×2048 サイズの計算を行う。IoTで、デバイスからデータをネットワーク転送するアプリケーションについて想定した際に、ネットワークコストを下げるため、デバイス側でFFT 処理等の一次分析をして送ることが想定される。

[0207] MRI-Q (<http://impact.crhc.illinois.edu/parboil/>)（登録商標）は、非デカルト空間の3次元MRI再構成アルゴリズムで 사용되는キャリブレーション用のスキャナー構成を表す行列Qを計算する。IoT環境では、カメラビデオからの自動監視のために画像処理が必要になることが多く、画像処理の自動オフロードへのニーズはある。MRI-QはC言語アプリケーションで、パフォーマンス測定中に3次元MRI画像処理を実行し、Large の $64 \times 64 \times 64$ サイズのサンプルデータを使用して処理時間を測定する。CPU処理はC言語で、FPGA処理はOpenCL（登録商標）に基づき処理される。

本実施形態のGPU、FPGA自動オフロード技術により、NAS.FTはGPUで高速化でき、MRI-QはFPGAで高速化でき、それぞれ、CPUに比べて5倍、7倍の高速化ができる。

[0208] ・評価手法

アプリケーションを配置するトポロジは、図 11 に示すように 3 層で構成され、クラウドレイヤーの拠点数は「5」、キャリアエッジレイヤーは「20」、ユーザエッジレイヤーは「60」、インプットノードは「300」とする。IoT 等のアプリケーションを想定してインプットノードから IoT データ等がユーザエッジに収集され、アプリケーションの特性（応答時間の要求条件等）に応じて、ユーザエッジ、キャリアエッジで分析処理がされたり、クラウドまでデータをあげてから分析処理されたりされる。

[0209] [式 1] [式 2] に示す線形計画式のパラメータを元に、ユーザ要求条件に基づいて、例えば 1000 個のアプリケーションを配置する。アプリケーションは、IoT アプリケーションで、インプットノードから生じるデータを分析する想定である。インプットノード（「300」あるとする）から配置依頼をランダムに生じさせる。

例えば、配置依頼数として、NAS.FT : MRI-Q = 3 : 1 の割合で 1000 回アプリを配置依頼する。また、ユーザ要求として、配置依頼する際にアプリ毎に価格条件か応答時間条件が選ばれる。NAS.FT の場合、価格に関しては月 7000 円上限か 8500 円上限か 10000 円上限、応答時間に関しては 6 秒上限か 7 秒条件か 10 秒上限が選択される。MRI-Q の場合、価格に関しては月 12500 円上限か 20000 円上限、応答時間に関しては、4 秒上限か 8 秒上限が選択される。

[0210] ユーザ要求のバリエーションとして、3 パターンがある。

パターン 1 : NAS.FT では 6 種のリクエストを 1 / 6 ずつ、MRI-Q では 4 種のリクエストを 1 / 4 ずつ選択する。

パターン 2 : リクエストは最低価格が上限の条件を選択（最初は 7000 円、12500 円）し、空きがない場合は次に安い価格条件とする。

パターン 3 : リクエストは最低応答時間が上限の条件を選択（最初は 6 秒、4 秒）し、空きがない場合は次に速い応答時間条件とする。

[0211] ・配置のシミュレーション

配置は、評価ツールとしてソルバ G L P K 5.0（登録商標）を用いてシミ

ュレーション実験により行う。規模のあるネットワーク配置の模擬のため、評価ツールを用いたシミュレーションになる。実利用の際は、アプリケーションのオフロード依頼が来た場合、検証環境を用いた繰返し性能試験でオフロードパターンを作成し、検証環境での性能試験結果に基づいて適切なリソース量を決める（図10参照）。そして、ユーザ要望に応じてGLPK等を用いて適切な配置を定め、実際にデプロイした際の正常確認試験や性能試験を自動で行い、その結果と価格をユーザに提示して、ユーザ判断後利用を開始する。

[0212] 図12は、平均応答時間のアプリケーション配置数変化を示すグラフである。図12は、平均応答時間とアプリケーション配置数を、上記3パターンに対して取る。

パターン2ではクラウドから順に、パターン3ではエッジから順に埋まっていくことが確認できた。パターン1では、多様な依頼が来た場合に、ユーザ要求条件を満たして配置される。

図12に示すように、パターン2では、400配置位までは全てクラウドに配置され平均応答時間は最遅のままであるが、クラウドが埋まると段々下がっていくことが分かる。

パターン3では、NAS、FTはユーザエッジから、またMRI-Qはキャリアエッジから配置される。このため、平均応答時間については最短となる。しかし、数が増えるとクラウドにも配置されるため平均応答時間は遅くなる。パターン2では、平均応答時間は、パターン1やパターン3の中間であり、ユーザ要求に応じて配置される。このため、パターン2では、最初はクラウドに全て入るパターン1に比べて平均応答時間は適切に低減されている。

[0213] このように、ソフトウェアを配置先環境に合わせて自動適応させ、GPU等に自動オフロードした際に、ユーザのコスト要求、応答時間要求に応える。すなわち、GPU等のデバイスで処理できるよう、プログラムを変換し、アサインするリソース量が定まった後に変換したアプリケーションの最適配置を行う。

[0214] まとめると、まず、プログラム変換する際に検証環境で行った性能試験のデータから、アプリケーションの利用データ容量、計算リソース量、帯域、処理時間を設定する。変換アプリケーション毎に設定される値と、事前に設定されるサーバやリンクのコスト等の値から、線形計画式に基づき、アプリケーションの適切な配置が計算される。アプリケーション配置の際は、ユーザが指定する価格や応答時間のリクエストに基づき、一方が制約条件にもう一方が目的関数となる。線形計画ソルバにより適切な配置が計算され、提案方式は計算された場所にリソースを配置した際の、価格等をユーザに提示し、ユーザ承諾後に利用が開始される。

[0215] GPU、FPGAに自動オフロードしたアプリケーションに対して、ユーザのリクエストする価格条件や応答時間条件、アプリケーションの配置数等を変更して、適正配置を計算する。これにより、ユーザ要望に従った配置が可能になる。

[0216] (第2実施形態)

次に、本発明の第2実施形態における、オフロードサーバ1A等について説明する。

第2実施形態は、ループ文のFPGA自動オフロードに適用した例である。

本実施形態は、PLD (Programmable Logic Device) として、FPGA (Field Programmable Gate Array) に適用した例について説明する。本発明は、プログラマブルロジックデバイス全般に適用可能である。

[0217] (原理説明)

FPGAで、どのループをオフロードすれば高速になるかの予測は難しいため、GPU同様検証環境で自動測定することを提案している。しかし、FPGAは、OpenCLをコンパイルして実機で動作させるまで数時間以上かかるため、GPU自動オフロードでのGAを用いて何回も反復して測定することは、処理時間が膨大となり行うことはできない。そこで、FPGAにオフロードする候補のループ文を絞ってから、測定を行う形をとる。具体的には、

発見されたループ文に対して、ROSE（登録商標）等の算術強度分析ツールを用いて算術強度が高いループ文を抽出する。更に、gcov（登録商標）等のプロファイリングツールを用いてループ回数が多いループ文も抽出する。

[0218] 算術強度やループ回数が多いループ文を候補として、OpenCL 化を行う。OpenCL 化時には、CPU 処理プログラムを、カーネル（FPGA）とホスト（CPU）に、OpenCL の文法に従って分割する。候補ループ文に対して、作成したOpenCL をプレコンパイルして、リソース効率が高いループ文を見つける。これは、コンパイルの途中で、作成するリソースは分かるため、利用するリソース量が十分少ないループ文に更に絞り込む。

候補ループ文が幾つか残るため、それらを用いて性能や電力使用量を実測する。選択された単ループ文に対してコンパイルして測定し、更に高速化できた単ループ文に対してはその組み合わせパターンも作り 2 回目の測定をする。測定された複数パターンの中で、短時間かつ低電力使用量のパターンを解として選択する。

[0219] ループ文のFPGA オフロードについては、算術強度等を用いて絞り込んでから、測定を行い、低電力パターンの評価値を高めることで、自動での高速化、低電力化を行う。

[0220] 図 13 は、本発明の第 2 実施形態に係るオフロードサーバ 1 A の構成例を示す機能ブロック図である。本実施形態の説明に当たり、図 1 と同一構成部分には同一符号を付して重複箇所の説明を省略する。

オフロードサーバ 1 A は、アプリケーションの特定処理をアクセラレータに自動的にオフロードする装置である。

また、オフロードサーバ 1 A は、エミュレータに接続可能である。

図 13 に示すように、オフロードサーバ 1 A は、制御部 21 と、入出力部 12 と、記憶部 13 と、検証用マシン 14（Verification machine）（アクセラレータ検証用装置）と、を含んで構成される。

[0221] 制御部 21 は、オフロードサーバ 1 A 全体の制御を司る自動オフロード機能部（Automatic Offloading function）である。制御部 21 は、例えば、記

憶部 13 に格納されたプログラム（オフロードプログラム）を不図示の CPU が、RAM に展開し実行することにより実現される。

[0222] 制御部 21 は、アプリケーションコード指定部 (Specify application code) 111 と、アプリケーションコード分析部 (Analyze application code) 112 と、PLD 処理指定部 213 と、算術強度算出部 214 と、配置設定部 170 と、PLD 処理パターン作成部 215 と、性能測定部 118 と、実行ファイル作成部 119 と、本番環境配置部 (Deploy final binary files to production environment) 120 と、性能測定テスト抽出実行部 (Extract performance test cases and run automatically) 121 と、ユーザ提供部 (Provide price and performance to a user to judge) 122 と、を備える。

[0223] <PLD 処理指定部 213>

PLD 処理指定部 213 は、アプリケーションのループ文（繰り返し文）を特定し、特定した各ループ文に対して、PLD におけるパイプライン処理、並列処理を OpenCL で指定した複数のオフロード処理パターンを作成してコンパイルする。

PLD 処理指定部 213 は、オフロード範囲抽出部 (Extract offload able area) 213a と、中間言語ファイル出力部 (Output intermediate file) 213b と、を備える。

[0224] オフロード範囲抽出部 213a は、ループ文や FFT 等、FPGA にオフロード可能な処理を特定し、オフロード処理に応じた中間言語を抽出する。

[0225] 中間言語ファイル出力部 213b は、抽出した中間言語ファイル 133 を出力する。中間言語抽出は、一度で終わりではなく、適切なオフロード領域探索のため、実行を試行して最適化するため反復される。

[0226] <算術強度算出部 214>

算術強度算出部 214 は、例えば ROSE フレームワーク（登録商標）等の算術強度 (Arithmetic Intensity) 分析ツールを用いて、アプリケーションのループ文の算術強度を算出する。算術強度は、プログラムの稼働中に実行し

た浮動小数点演算 (floating point number, FN) の数を、主メモリへのアクセスしたbyte数で割った値 (FN演算／メモリアクセス) である。

算術強度は、計算回数が多いと増加し、アクセス数が多いと減少する指標であり、算術強度が高い処理はプロセッサにとって重い処理となる。そこで、算術強度分析ツールで、ループ文の算術強度を分析する。PLD処理パターン作成部215は、算術強度が高いループ文をオフロード候補に絞る。

[0227] 算術強度の計算例について述べる。

1回のループの中での浮動小数点計算処理が10回 (10FLOP) 行われ、ループの中で使われるデータが2byteであるとする。ループ毎に同じサイズのデータが使われる際は、 $10 / 2 = 5$ [FLOP/byte]が算術強度となる。

なお、算術強度では、ループ回数が考慮されないため、本実施形態では、算術強度に加えて、ループ回数も考慮して絞り込む。

[0228] <PLD処理パターン作成部215>

PLD処理パターン作成部215は、算術強度算出部214が算出した算術強度をもとに、算術強度が所定の閾値より高い (以下、適宜、高算術強度という) ループ文をオフロード候補として絞り込み、PLD処理パターンを作成する。

また、PLD処理パターン作成部215は、基本動作として、コンパイルエラーが出るループ文 (繰り返し文) に対して、オフロード対象外とするとともに、コンパイルエラーが出ない繰り返し文に対して、PLD処理するかしないかの指定を行うPLD処理パターンを作成する。

[0229] ・ループ回数測定機能

PLD処理パターン作成部215は、ループ回数測定機能として、プロファイリングツールを用いてアプリケーションのループ文のループ回数を測定し、ループ文のうち、高算術強度で、ループ回数が所定の回数より多い (以下、適宜、高ループ回数という) ループ文を絞り込む。ループ回数把握には、GNUカバレッジのgcov等を用いる。プロファイリングツールとしては、「GNUプロファイラ(gprof)」、「GNUカバレッジ(gcov)」が知られている。双方と

も各ループの実行回数を調査できるため、どちらを用いてもよい。

[0230] また、算術強度分析では、ループ回数は特に見えないため、ループ回数が多く負荷が高いループを検出するため、プロファイリングツールを用いて、ループ回数を測定する。ここで、算術強度の高さは、FPGAへのオフロードに向けた処理かどうかを表わし、ループ回数×算術強度は、FPGAへのオフロードに関連する負荷が高いかどうかを表わす。

[0231] ・OpenCL（中間言語）作成機能

PLD処理パターン作成部215は、OpenCL作成機能として、絞り込まれた各ループ文をFPGAにオフロードするためのOpenCLを作成（OpenCL化）する。すなわち、PLD処理パターン作成部215は、絞り込んだループ文をオフロードするOpenCLをコンパイルする。また、PLD処理パターン作成部215は、性能測定された中でCPUに比べ高性能化されたループ文をリスト化し、リストのループ文を組み合わせでオフロードするOpenCLを作成する。

[0232] OpenCL化について述べる。

PLD処理パターン作成部215は、ループ文をOpenCL等の高位言語化する。まず、CPU処理のプログラムを、カーネル（FPGA）とホスト（CPU）に、OpenCL等の高位言語の文法に従って分割する。例えば、10個のfor文の内一つのfor文をFPGAで処理する場合は、その一つをカーネルプログラムとして切り出し、OpenCLの文法に従って記述する。OpenCLの文法例については、後記する。

[0233] さらに、分割する際、より高速化するための技法を盛り込むこともできる。一般に、FPGAを用いて高速化するためには、ローカルメモリキャッシュ、ストリーム処理、複数インスタンス化、ループ文の展開処理、ネストループ文の統合、メモリインターリーブ等がある。これらは、ループ文によっては、絶対効果があるわけではないが、高速化するための手法として、よく利用されている。

[0234] OpenCLのC言語の文法に沿って作成したカーネルは、OpenCLのC言語のラ

ンタイムAPIを利用して、作成するホスト（例えば、CPU）側のプログラムによりデバイス（例えば、FPGA）で実行される。カーネル関数hello()をホスト側から呼び出す部分は、OpenCLランタイムAPIの一つであるclEnqueueTask()を呼び出すことである。

ホストコードで記述するOpenCLの初期化、実行、終了の基本フローは、下記ステップ1～13である。このステップ1～13のうち、ステップ1～10がカーネル関数hello()をホスト側から呼び出すまでの手続（準備）であり、ステップ11でカーネルの実行となる。

[0235] 1. プラットフォーム特定

OpenCLランタイムAPIで定義されているプラットフォーム特定機能を提供する関数clGetPlatformIDs()を用いて、OpenCLが動作するプラットフォームを特定する。

[0236] 2. デバイス特定

OpenCLランタイムAPIで定義されているデバイス特定機能を提供する関数clGetDeviceIDs()を用いて、プラットフォームで使用するGPU等のデバイスを特定する。

[0237] 3. コンテキスト作成

OpenCLランタイムAPIで定義されているコンテキスト作成機能を提供する関数clCreateContext()を用いて、OpenCLを動作させる実行環境となるOpenCLコンテキストを作成する。

[0238] 4. コマンドキュー作成

OpenCLランタイムAPIで定義されているコマンドキュー作成機能を提供する関数clCreateCommandQueue()を用いて、デバイスを制御する準備であるコマンドキューを作成する。OpenCLでは、コマンドキューを通して、ホストからデバイスに対する働きかけ（カーネル実行コマンドやホストーデバイス間のメモリコピーコマンドの発行）を実行する。

[0239] 5. メモリオブジェクト作成

OpenCLランタイムAPIで定義されているデバイス上にメモリを確保する

機能を提供する関数`clCreateBuffer()`を用いて、ホスト側からメモリオブジェクトを参照できるようにするメモリオブジェクトを作成する。

[0240] 6. カーネルファイル読み込み

デバイスで実行するカーネルは、その実行自体をホスト側のプログラムで制御する。このため、ホストプログラムは、まずカーネルプログラムを読み込む必要がある。カーネルプログラムには、OpenCLコンパイラで作成したバイナリデータや、OpenCL C言語で記述されたソースコードがある。このカーネルファイルを読み込む（記述省略）。なお、カーネルファイル読み込みでは、OpenCLランタイムAPIは使用しない。

[0241] 7. プログラムオブジェクト作成

OpenCLでは、カーネルプログラムをプログラムプロジェクトとして認識する。この手続きがプログラムオブジェクト作成である。

OpenCLランタイムAPIで定義されているプログラムオブジェクト作成機能を提供する関数`clCreateProgramWithSource()`を用いて、ホスト側からメモリオブジェクトを参照できるようにするプログラムオブジェクトを作成する。カーネルプログラムのコンパイル済みバイナリ列から作成する場合は、`clCreateProgramWithBinary()`を使用する。

[0242] 8. ビルド

ソースコードとして登録したプログラムオブジェクトを OpenCL Cコンパイラ・リンカを使いビルドする。

OpenCLランタイムAPIで定義されているOpenCL Cコンパイラ・リンカによるビルドを実行する関数`clBuildProgram()`を用いて、プログラムオブジェクトをビルドする。なお、`clCreateProgramWithBinary()`でコンパイル済みのバイナリ列からプログラムオブジェクトを生成した場合、このコンパイル手続は不要である。

[0243] 9. カーネルオブジェクト作成

OpenCLランタイムAPIで定義されているカーネルオブジェクト作成機能を提供する関数`clCreateKernel()`を用いて、カーネルオブジェクトを作成す

る。1つのカーネルオブジェクトは、1つのカーネル関数に対応するので、カーネルオブジェクト作成時には、カーネル関数の名前(hello)を指定する。また、複数のカーネル関数を1つのプログラムオブジェクトとして記述した場合、1つのカーネルオブジェクトは、1つのカーネル関数に1対1に対応するので、`clCreateKernel()`を複数回呼び出す。

[0244] 10. カーネル引数設定

OpenCLランタイムAPIで定義されているカーネルへ引数を与える（カーネル関数を持つ引数へ値を渡す）機能を提供する関数`clSetKernel()`を用いて、カーネル引数を設定する。

以上、上記ステップ1～10で準備が整い、ホスト側からデバイスでカーネルを実行するステップ11に入る。

[0245] 11. カーネル実行

カーネル実行（コマンドキューへ投入）は、デバイスに対する働きかけとなるので、コマンドキューへのキューイング関数となる。

OpenCLランタイムAPIで定義されているカーネル実行機能を提供する関数`clEnqueueTask()`を用いて、カーネルhelloをデバイスで実行するコマンドをキューイングする。カーネルhelloを実行するコマンドがキューイングされた後、デバイス上の実行可能な演算ユニットで実行されることになる。

[0246] 12. メモリオブジェクトからの読み込み

OpenCLランタイムAPIで定義されているデバイス側のメモリからホスト側のメモリへデータをコピーする機能を提供する関数`clEnqueueReadBuffer()`を用いて、デバイス側のメモリ領域からホスト側のメモリ領域にデータをコピーする。また、ホスト側からクライアント側のメモリへデータをコピーする機能を提供する関数`clEnqueueWriteBuffer()`を用いて、ホスト側のメモリ領域からデバイス側のメモリ領域にデータをコピーする。なお、これらの関数は、デバイスに対する働きかけとなるので、一度コマンドキューへコピーコマンドがキューイングされてからデータコピーが始まることになる。

[0247] 13. オブジェクト解放

最後に、ここまでに作成してきた各種オブジェクトを解放する。

以上、OpenCL C言語に沿って作成されたカーネルの、デバイス実行について説明した。

[0248] ・リソース量算出機能

P L D 処理パターン作成部 2 1 5 は、リソース量算出機能として、作成したOpenCLをプレコンパイルして利用するリソース量を算出する（「1回目のリソース量算出」）。P L D 処理パターン作成部 2 1 5 は、算出した算術強度およびリソース量に基づいてリソース効率を算出し、算出したリソース効率をもとに、各ループ文で、リソース効率が所定の値より高いc個のループ文を選ぶ。

P L D 処理パターン作成部 2 1 5 は、組み合わせたオフロードOpenCLでプレコンパイルして利用するリソース量を算出する（「2回目のリソース量算出」）。ここで、プレコンパイルせず、1回目測定前のプレコンパイルでのリソース量の和でもよい。

[0249] <性能測定部 1 1 8>

性能測定部 1 1 8 は、作成されたP L D 処理パターンのアプリケーションをコンパイルして、検証用マシン 1 4 に配置し、P L D にオフロードした際の性能測定用処理を実行する。

[0250] 性能測定部 1 1 8 は、配置したバイナリファイルを実行し、オフロードした際の性能を測定するとともに、性能測定結果を、オフロード範囲抽出部 2 1 3 a に戻す。この場合、オフロード範囲抽出部 2 1 3 a は、別のP L D 処理パターン抽出を行い、中間言語ファイル出力部 2 1 3 b は、抽出された中間言語をもとに、性能測定を試行する（図 2 の符号 a 参照）。

[0251] 性能測定部 1 1 8 は、バイナリファイル配置部 (Deploy binary files) 1 1 8 a を備える。バイナリファイル配置部 1 1 8 a は、G P U を備えた検証用マシン 1 4 に、中間言語から導かれる実行ファイルをデプロイ(配置)する。

[0252] 性能測定の具体例について述べる。

PLD処理パターン作成部215は、高リソース効率のループ文を絞り込み、実行ファイル作成部119が絞り込んだループ文をオフロードするOpenCLをコンパイルする。性能測定部118は、コンパイルされたプログラムの性能を測定する（「1回目の性能測定」）。

[0253] そして、PLD処理パターン作成部215は、性能測定された中でCPUに比べ高性能化されたループ文をリスト化する。PLD処理パターン作成部215は、リストのループ文を組み合わせてオフロードするOpenCLを作成する。PLD処理パターン作成部215は、組み合わせたオフロードOpenCLでプレコンパイルして利用するリソース量を算出する。

なお、プレコンパイルせず、1回目測定前のプレコンパイルでのリソース量の和でもよい。実行ファイル作成部119は、組み合わせたオフロードOpenCLをコンパイルし、性能測定部118は、コンパイルされたプログラムの性能を測定する（「2回目の性能測定」）。

[0254] <実行ファイル作成部119>

実行ファイル作成部119は、所定回数繰り返された、処理時間の測定結果をもとに、複数のPLD処理パターンから最高評価値のPLD処理パターンを選択し、最高評価値のPLD処理パターンをコンパイルして実行ファイルを作成する。

[0255] 以下、上述のように構成されたオフロードサーバ1Aの自動オフロード動作について説明する。

[自動オフロード動作]

本実施形態のオフロードサーバ1Aは、環境適応ソフトウェアの要素技術としてユーザアプリケーションロジックのFPGA自動オフロードに適用した例である。

図2に示すオフロードサーバ1Aの自動オフロード処理を参照して説明する。

図2に示すように、オフロードサーバ1Aは、環境適応ソフトウェアの要素技術に適用される。オフロードサーバ1Aは、制御部（自動オフロード機

能部) 11 と、テストケースDB 131 と、中間言語ファイル 133 と、検証用マシン 14 と、を有している。

オフロードサーバ 1A は、ユーザが利用するアプリケーションコード (Application code) 125 を取得する。

[0256] ユーザは、例えば、各種デバイス (Device) 151、CPU-GPU を有する装置 152、CPU-FPGA を有する装置 153、CPU を有する装置 154 を利用する。オフロードサーバ 1A は、機能処理を CPU-GPU を有する装置 152、CPU-FPGA を有する装置 153 のアクセラレータに自動オフロードする。

[0257] 以下、図 2 のステップ番号を参照して各部の動作を説明する。

〈ステップ S21 : Specify application code〉

ステップ S21 において、アプリケーションコード指定部 111 (図 13 参照) は、ユーザに提供しているサービスの処理機能 (画像分析等) を特定する。具体的には、アプリケーションコード指定部 111 は、入力されたアプリケーションコードの指定を行う。

[0258] 〈ステップ S12 : Analyze application code〉

ステップ S12 において、アプリケーションコード分析部 112 (図 13 参照) は、処理機能のソースコードを分析し、ループ文や FFT ライブラリ呼び出し等の特定ライブラリ利用の構造を把握する。

[0259] 〈ステップ S13 : Extract offload able area〉

ステップ S13 において、PLD 処理指定部 213 (図 13 参照) は、アプリケーションのループ文 (繰り返し文) を特定し、各繰り返し文に対して、FPGA における並列処理またはパイプライン処理を指定して、高位合成ツールでコンパイルする。具体的には、オフロード範囲抽出部 213a (図 13 参照) は、ループ文等、FPGA にオフロード可能な処理を特定し、オフロード処理に応じた中間言語として OpenCL を抽出する。

[0260] 〈ステップ S14 : Output intermediate file〉

ステップ S14 において、中間言語ファイル出力部 213b (図 13 参照

）は、中間言語ファイル 133 を出力する。中間言語抽出は、一度で終わりでなく、適切なオフロード領域探索のため、実行を試行して最適化するため反復される。

[0261] <ステップ S 15 : Compile error>

ステップ S 15 において、PLD 処理パターン作成部 215（図 13 参照）は、コンパイルエラーが出るループ文に対して、オフロード対象外とするとともに、コンパイルエラーが出ない繰り返し文に対して、FPGA 処理するかしないかの指定を行う PLD 処理パターンを作成する。

[0262] <ステップ S 21 : Deploy binary files>

ステップ S 21 において、バイナリファイル配置部 118a（図 13 参照）は、FPGA を備えた検証用マシン 14 に、中間言語から導かれる実行ファイルをデプロイする。バイナリファイル配置部 118a は、配置したファイルを起動し、想定するテストケースを実行して、オフロードした際の性能を測定する。

[0263] <ステップ S 22 : Measure performances>

ステップ S 22 において、性能測定部 118（図 13 参照）は、配置したファイルを実行し、オフロードした際の性能と電力使用量を測定する。

オフロードする領域をより適切にするため、この性能測定結果は、オフロード範囲抽出部 213a に戻され、オフロード範囲抽出部 213a が、別パターンの抽出を行う。そして、中間言語ファイル出力部 213b は、抽出された中間言語をもとに、性能測定を試行する（図 2 の符号 a 参照）。性能測定部 118 は、検証環境での性能・電力使用量測定を繰り返し、最終的にデプロイするコードパターンを決定する。

[0264] 図 2 の符号 a に示すように、制御部 21 は、上記ステップ S 12 乃至ステップ S 22 を繰り返し実行する。制御部 21 の自動オフロード機能をまとめると、下記である。すなわち、PLD 処理指定部 213 は、アプリケーションのループ文（繰り返し文）を特定し、各繰り返し文に対して、FPGA における並列処理またはパイプライン処理を OpenCL（中間言語）で指定して、高

位合成ツールでコンパイルする。そして、P L D 処理パターン作成部 2 1 5 は、コンパイルエラーが出るループ文を、オフロード対象外とし、コンパイルエラーが出ないループ文に対して、P L D 処理するかしないかの指定を行う P L D 処理パターンを作成する。そして、バイナリファイル配置部 1 1 8 a は、該当 P L D 処理パターンのアプリケーションをコンパイルして、検証用マシン 1 4 に配置し、性能測定部 1 1 8 が、検証用マシン 1 4 で性能測定用処理を実行する。実行ファイル作成部 1 1 9 は、所定回数繰り返された、性能測定結果をもとに、複数の P L D 処理パターンから最高評価値（例えば、 $\text{評価値} = (\text{処理時間})^{-1/2}$ が最も高いもの）のパターンを選択し、選択パターンをコンパイルして実行ファイルを作成する。

[0265] <ステップ S 2 3 : Deploy final binary files to production environment >

ステップ S 2 3 において、本番環境配置部 1 2 0 は、最終的なオフロード領域を指定したパターンを決定し、ユーザ向けの本番環境にデプロイする。

[0266] <ステップ S 2 4 : Extract performance test cases and run automatically>

ステップ S 2 4 において、性能測定テスト抽出実行部 1 2 1 は、実行ファイル配置後、ユーザに性能を示すため、性能試験項目をテストケース D B 1 3 1 から抽出し、抽出した性能試験を自動実行する。

[0267] <ステップ S 2 5 : Provide price and performance to a user to judge>

ステップ S 2 5 において、ユーザ提供部 1 2 2 は、性能試験結果を踏まえた、価格・性能等の情報をユーザに提示する。ユーザは、提示された価格・性能等の情報をもとに、サービスの課金利用開始を判断する。

[0268] 上記ステップ S 2 1 ～ステップ S 2 5 は、ユーザのサービス利用のバックグラウンドで行われ、例えば、仮利用の初日の間に行う等を想定している。また、コスト低減のためにバックグラウンドで行う処理は、G P U ・ F P G A オフロードのみを対象としてもよい。

[0269] 上記したように、オフロードサーバ 1 A の制御部（自動オフロード機能部

） 21 は、環境適応ソフトウェアの要素技術に適用した場合、機能処理のオフロードのため、ユーザが利用するアプリケーションのソースコードから、オフロードする領域を抽出して中間言語を出力する（ステップS 12～ステップS 15）。制御部21は、中間言語から導かれる実行ファイルを、検証用マシン14に配置実行し、オフロード効果を検証する（ステップS 21～ステップS 22）。検証を繰り返し、適切なオフロード領域を定めたのち、制御部21は、実際にユーザに提供する本番環境に、実行ファイルをデプロイし、サービスとして提供する（ステップS 26）。

[0270] なお、上記では、環境適応に必要な、コード変換、リソース量調整、配置場所調整を一括して行う処理フローを説明したが、これに限らず、行いたい処理だけ切出すことも可能である。例えば、FPGA向けにコード変換だけ行いたい場合は、上記ステップS 21～ステップS 25の、環境適応機能や検証環境等必要な部分だけ利用すればよい。

[0271] [FPGA自動オフロード]

上述したコード分析は、Clang等の構文解析ツールを用いて、アプリケーションコードの分析を行う。コード分析は、オフロードするデバイスを想定した分析が必要になるため、一般化は難しい。ただし、ループ文や変数の参照関係等のコードの構造を把握したり、機能ブロックとしてFFT処理を行う機能ブロックであることや、FFT処理を行うライブラリを呼び出している等を把握することは可能である。機能ブロックの判断は、オフロードサーバが自動判断することは難しい。これもDeckard等の類似コード検出ツールを用いて類似度判定等で把握することは可能である。ここで、Clangは、C/C++向けツールであるが、解析する言語に合わせたツールを選ぶ必要がある。

[0272] また、アプリケーションの処理をオフロードする場合には、GPU、FPGA、IoT GW等それぞれにおいて、オフロード先に合わせた検討が必要となる。一般に、性能に関しては、最大性能になる設定を一回で自動発見するのは難しい。このため、オフロードパターンを、性能測定を検証環境で何度か繰り返すことにより試行し、高速化できるパターンを見つけることを行

う。

[0273] 以下、アプリケーションソフトウェアのループ文のFPGA向けオフロード手法について説明する。

[フローチャート]

図14は、オフロードサーバ1Aの動作概要を説明するフローチャートである。

ステップS201でアプリケーションコード分析部112は、アプリケーションのオフロードしたいソースコードの分析を行う。アプリケーションコード分析部112は、ソースコードの言語に合わせて、ループ文や変数の情報を分析する。

[0274] ステップS202でPLD処理指定部213は、アプリケーションのループ文および参照関係を特定する。

[0275] 次に、PLD処理パターン作成部215は、把握したループ文に対して、FPGAオフロードを試行するかどうか候補を絞っていく処理を行う。ループ文に対してオフロード効果があるかどうかは、算術強度が一つの指標となる。

ステップS203で算術強度算出部214は、算術強度分析ツールを用いてアプリケーションのループ文の算術強度を算出する。算術強度は、計算数が多いと増加し、アクセス数が多いと減少する指標であり、算術強度が高い処理はプロセッサにとって重い処理となる。そこで、算術強度分析ツールで、ループ文の算術強度を分析し、密度が高いループ文をオフロード候補に絞る。そこで、算術強度分析ツールで、ループ文の算術強度を分析し、密度が高いループ文をオフロード候補に絞る。

[0276] 高算術強度のループ文であっても、それをFPGAで処理する際に、FPGAリソースを過度に消費してしまうのは問題である。そこで、高算術強度ループ文をFPGA処理する際のリソース量の算出について述べる。

FPGAにコンパイルする際の処理としては、OpenCL等の高位言語からハードウェア記述のHDL等のレベルに変換され、それに基づき実際の配線処

理等がされる。この時、配線処理等は多大な時間がかかるが、HDL等の途中状態の段階までは時間は分単位でしかかからない。HDL等の途中状態の段階であっても、FPGAで利用するFlip FlopやLook Up Table等のリソースは分かる。このため、HDL等の途中状態の段階をみれば、利用するリソース量はコンパイルが終わらずとも短時間でわかる。

[0277] そこで、本実施形態では、PLD処理パターン作成部215は、対象のループ文をOpenCL等の高位言語化し、まずリソース量を算出する。また、ループ文をオフロードした際の算術強度とリソース量が決まるため、算術強度／リソース量または算術強度×ループ回数／リソース量をリソース効率とする。そして、高リソース効率のループ文をオフロード候補として更に絞り込む。

[0278] 図14のフローに戻って、ステップS204でPLD処理パターン作成部215は、gcov、gprof等のプロファイリングツールを用いてアプリケーションのループ文のループ回数を測定する。

ステップS205でPLD処理パターン作成部215は、ループ文のうち、高算術強度で高ループ回数のループ文を絞り込む。

[0279] ステップS206でPLD処理パターン作成部215は、絞り込まれた各ループ文をFPGAにオフロードするためのOpenCLを作成する。

[0280] ここで、ループ文のOpenCL化（OpenCLの作成）について、補足して説明する。すなわち、ループ文をOpenCL等によって、高位言語化する際には、2つの処理が必要である。一つは、CPU処理のプログラムを、カーネル（FPGA）とホスト（CPU）に、OpenCL等の高位言語の文法に従って分割することである。もう一つは、分割する際に、高速化するための技法を盛り込むことである。一般に、FPGAを用いて高速化するためには、ローカルメモリキャッシュ、ストリーム処理、複数インスタンス化、ループ文の展開処理、ネストループ文の統合、メモリアンターリーブ等がある。これらは、ループ文によっては、絶対効果があるわけではないが、高速化するための手法として、よく利用されている。

- [0281] 次に、高リソース効率のループ文が幾つか選択されたので、それらを用いて性能を実測するオフロードパターンを実測する数だけ作成する。FPGAでの高速化は、1個の処理だけFPGAリソース量を集中的にかけて高速化する形もあれば、複数の処理にFPGAリソースを分散して高速化する形もある。選択された単ループ文のパターンを一定数作り、FPGA実機で動作する前段階としてプレコンパイルする。
- [0282] ステップS207でPLD処理パターン作成部215は、作成したOpenCLをプレコンパイルして利用するリソース量を算出する（「1回目のリソース量算出」）。
- [0283] ステップS208でPLD処理パターン作成部215は、高リソース効率のループ文を絞り込む。
- [0284] ステップS209で実行ファイル作成部119は、絞り込んだループ文をオフロードするOpenCLをコンパイルする。
- [0285] ステップS210で性能測定部118は、コンパイルされたプログラムの性能を測定する（「1回目の性能測定」）。候補ループ文が幾つが残るため、性能測定部118は、それらを用いて性能を実測する（詳細については、図15のサブルーチン参照）。
- [0286] ステップS211でPLD処理パターン作成部215は、性能測定された中でCPUに比べ高性能化されたループ文をリスト化する。
- [0287] ステップS212でPLD処理パターン作成部215は、リストのループ文を組み合わせてオフロードするOpenCLを作成する。
- ステップS213でPLD処理パターン作成部215は、組み合わせたオフロードOpenCLでプレコンパイルして利用するリソース量を算出する（「2回目のリソース量算出」）。なお、プレコンパイルせず、1回目測定前のプレコンパイルでのリソース量の和でもよい。このようにすれば、プレコンパイル回数を削減することができる。
- [0288] ステップS214で実行ファイル作成部119は、組み合わせたオフロードOpenCLをコンパイルする。

[0289] ステップS 2 1 5で性能測定部 1 1 8は、コンパイルされたプログラムの性能を測定する（「2回目の性能測定」）。性能測定部 1 1 8は、選択された単ループ文に対してコンパイルして測定し、更に高速化できた単ループ文に対してはその組み合わせパターンも作り2回目の性能測定を行う（詳細については、図 1 5 のサブルーチン参照）。

[0290] ステップS 2 1 6で本番環境配置部 1 2 0は、1回目と2回目の測定の中で最高性能のパターンを選択して本フローの処理を終了する。測定された複数パターンの中で、短時間のパターンを解として選択する。

[0291] このように、ループ文のFPGA自動オフロードは、算術強度とループ回数が高くリソース効率が高いループ文に絞って、オフロードパターンを作り、検証環境で実測を通じて高速なパターンの探索を行う（図 1 4 参照）。

[0292] 図 1 5 は、性能測定部 1 1 8の性能・電力使用量測定処理を示すフローチャートである。本フローは、図 1 4 のステップS 2 1 0またはステップS 2 1 5のサブルーチンコールにより呼び出され、実行される。

[0293] ステップS 3 0 1で、性能測定部 1 1 8は、FPGAオフロード時に必要となる処理時間を測定する。

[0294] ステップS 3 0 2で、性能測定部 1 1 8は、測定した処理時間をもとに評価値を設定する。

[0295] ステップS 3 0 3で、性能測定部 1 1 8は、評価値が高い個体ほど適合度が高くなるように評価された評価値の高いパターンの性能を測定し、図 1 4 のステップS 2 1 0またはステップS 2 1 5に戻る。

[0296] [オフロードパターンの作成例]

図 1 6 は、PLD処理パターン作成部 2 1 5の探索イメージを示す図である。

制御部（自動オフロード機能部）2 1（図 1 3 参照）は、ユーザが利用するアプリケーションコード（Application code）1 2 5（図 2 参照）を分析し、図 1 6 に示すように、アプリケーションコード 1 2 5のコードパターン（Code patterns）2 4 1からfor文の並列可否をチェックする。図 1 6 の符

号 r に示すように、コードパターン 2 4 1 から 4 つの for 文が見つかった場合、各 for 文に対してそれぞれ 1 桁、ここでは 4 つの for 文に対し 4 桁の 1 または 0 を割り当てる。ここでは、FPGA 処理する場合は 1、FPGA 処理しない場合（すなわち CPU で処理する場合）は 0 とする。

[0297] [CコードからOpenCL最終解の探索までの流れ]

図 1 7 の手順 A－F は、CコードからOpenCL最終解の探索までの流れを説明する図である。

アプリケーションコード分析部 1 1 2（図 1 3 参照）は、図 1 7 の手順 A に示す「Cコード」を構文解析し（〈構文解析〉：図 1 7 の符号 s 参照）、PLD 処理指定部 2 1 3（図 1 3 参照）は、図 1 7 の手順 B に示す「ループ文、変数情報」を特定する（図 1 7 の符号 t 参照）。

[0298] 算術強度算出部 2 1 4（図 1 3 参照）は、特定した「ループ文、変数情報」に対して、算術強度分析ツールを用いて算術強度分析（Arithmetic Intensity analysis）する（図 1 7 の符号 u 参照）。PLD 処理パターン作成部 2 1 5 は、算術強度が高いループ文をオフロード候補に絞る。さらに、PLD 処理パターン作成部 2 1 5 は、プロファイリングツールを用いてプロファイリング分析（Profiling analysis）を行って、高算術強度で高ループ回数のループ文をさらに絞り込む。

[0299] そして、PLD 処理パターン作成部 2 1 5 は、絞り込まれた各ループ文を FPGA にオフロードするための OpenCL を作成（OpenCL 化）する（図 1 7 の符号 v 参照）。

さらに、OpenCL 化時にコード分割と共に展開等の高速化手法を導入する（後記）。

[0300] <「高算術強度，OpenCL化」具体例（その 1）：手順 C>

例えば、アプリケーションコード 1 3 0 のコードパターン 2 4 1（図 1 6 参照）から 4 つの for 文（4 桁の 1 または 0 の割り当て）が見つかった場合、算術強度分析で 3 つが絞り込まれる（選ばれる）。すなわち、図 1 7 の符号 u に示すように、4 つの for 文から、3 つの for 文のオフロードパターン「100

0」「0010」「0001」が絞り込まれる。

[0301] <OpenCL化時にコード分割と共に実行する「展開」例>

FPGAからCPUへのデータ転送する場合の、CPUプログラム側で記述されるループ文 [k=0; k<10; k++] {
}

において、このループ文の上部に、`¥pragma unroll`を指示する。すなわち、

```
¥pragma unroll  
for(k=0; k<10; k++){  
}
```

と記述する。

[0302] `¥pragma unroll`等のIntelやXilinx（登録商標）のツールに合った文法で`unroll`を指示すると、上記展開例であれば、`i=0, i=1, i=2`と展開してパイプライン実行することができる。このため、リソース量は10倍使うことになるが、高速になる場合がある。

また、`unroll`で展開する数は全ループ回数個でなく5個に展開等の指定もでき、その場合は、ループ2回ずつが、5つに展開される。

以上で、「展開」例についての説明を終える。

[0303] 次に、PLD処理パターン作成部215は、オフロード候補として絞り込まれた高算術強度のループ文を、リソース量を用いてさらに絞り込む。すなわち、PLD処理パターン作成部215は、リソース量を算出し、PLD処理パターン作成部215は、高算術強度のループ文のオフロード候補の中から、リソース効率（＝算術強度／FPGA処理時のリソース量、または、算術強度×ループ回数／FPGA処理時のリソース量）分析して、リソース効率の高いループ文を抽出する。

[0304] 図17の符号vでは、PLD処理パターン作成部215は、絞り込んだループ文をオフロードするためのOpenCLをコンパイル（<プレコンパイル>）する。

[0305] <「高算術強度，OpenCL化」具体例（その2）>

図17の符号uに示すように、算術強度分析で絞り込まれた4つのオフロードパターン「1000」「0100」「0010」「0001」の中から、上記リソース効率分析により3つのオフロードパターン「1000」「0010」「0001」に絞り込む。

以上、図17の手順Cに示す「高算術強度、OpenCL化」について説明した。

[0306] 図17の手順Dに示す「リソース効率の高いループ文」に対して、性能測定部118は、コンパイルされたプログラムの性能を測定する（「1回目の性能測定」）。

そして、PLD処理パターン作成部215は、性能測定された中でCPUに比べ高性能化されたループ文をリスト化する。以下、同様に、リソース量を算出、オフロードOpenCLコンパイル、コンパイルされたプログラムの性能を測定する。

[0307] <「高算術強度、OpenCL化」具体例（その3）>

図17の符号wに示すように、3つのオフロードパターン「1000」「0010」「0001」について1回目測定を行う。その3つの測定の中で、「1000」「0010」の2つの性能が高くなったとすると、「1000」と「0010」の組合せについて2回目測定を行う。

[0308] 図17の符号xでは、実行ファイル作成部119は、絞り込んだループ文をオフロードするためのOpenCLをコンパイル（<本コンパイル>）する。

[0309] 図17の手順Eに示す「組合せパターン実測」は、候補ループ文単体、その後、その組合せで検証パターン測定することをいう。

[0310] <「高算術強度、OpenCL化」具体例（その4）>

図17の符号yに示すように、「1000」と「0010」の組合せである「1010」について2回目測定する。2回測定し、その結果、1回目測定と2回目測定の中で最高速度の「0010」が選択された。このような場合、「0010」が最終の解となる。ここで、組合せパターンがリソース量制限のため測定できない場合がある。この場合、組合せについてはスキップして、単体の結果から

最高速度のものを選ぶだけでもよい。

[0311] 図 17 の符号 z では、性能測定部 118 は、1 回目測定と 2 回目測定の中で最高速度の良い「0010」を選択（＜選択＞）する。

[0312] 以上により、図 17 の手順 F に示す「OpenCL 最終解」の「0010」（図 17 の符号 a a 参照）が選択された。

[0313] ＜デプロイ（配置）＞

OpenCL 最終解の、最高処理性能の PLD 処理パターンで、本番環境に改めてデプロイして、ユーザに提供する。

[0314] [実装例]

実装例を説明する。

FPGA は Intel PAC with Intel Arria10 GX FPGA 等が利用できる。

FPGA 処理は、Intel Acceleration Stack (Intel FPGA SDK for OpenCL、Quartus Prime Version) 等が利用できる。

Intel FPGA SDK for OpenCL は、標準 OpenCL に加え、Intel 向けの #pragma 等を解釈する高位合成ツール (HLS) である。

実装例では、FPGA で処理するカーネルと CPU で処理するホストプログラムを記述した OpenCL コードを解釈し、リソース量等の情報を出力し、FPGA の配線作業等を行い、FPGA で動作できるようにする。FPGA 実機で動作できるようにするには、100 行程度の小プログラムでも 3 時間程の長時間がかかる。ただし、リソース量オーバーの際は、早めにエラーとなる。また、FPGA で処理できない OpenCL コードの際は、数時間後にエラーを出力する。

[0315] 実装例では、C/C++ アプリケーションの利用依頼があると、まず、C/C++ アプリケーションのコードを解析して、for 文を発見するとともに、for 文内で使われる変数データ等のプログラム構造を把握する。構文解析には、LLVM/Clang の構文解析ライブラリ等が利用できる。

[0316] 実装例では、次に、各ループ文の FPGA オフロード効果があるかの見込みを得るため、算術強度分析ツールを実行し、計算数、アクセス数等で定ま

る算術強度の指標を取得する。算術強度分析には、ROSEフレームワーク等が利用できる。算術強度上位個のループ文のみ対象とするようにする。

次に、gcov等のプロファイリングツールを用いて、各ループのループ回数を取得する。算術強度×ループ回数が上位a個のループ文を候補に絞る。

[0317] 実装例では、次に、高算術強度の個々のループ文に対して、FPGAオフロードするOpenCLコードを生成する。OpenCLコードは、該当ループ文をFPGAカーネルとして、残りをCPUホストプログラムとして分割したものである。FPGAカーネルコードとする際に、高速化の技法としてループ文の展開処理を一定数bだけ行ってもよい。ループ文展開処理は、リソース量が増えるが、高速化に効果がある。そこで、展開する数は、一定数bに制限してリソース量が膨大にならない範囲で行う。

[0318] 実装例では、次に、a個のOpenCLコードに対して、Intel FPGA SDK for OpenCLを用いて、プレコンパイルをして、利用するFlip Flop、Look Up Table等のリソース量を算出する。使用リソース量は、全体リソース量の割合で表示される。ここで、算術強度とリソース量または算術強度とループ回数とリソース量から、各ループ文のリソース効率を計算する。例えば、算術強度が10、リソース量が0.5のループ文は、 $10 / 0.5 = 20$ 、算術強度が3、リソース量が0.3のループ文は $3 / 0.3 = 10$ がリソース効率となり、前者が高い。また、ループ回数をかけた値をリソース効率としてもよい。各ループ文で、リソース効率が高いc個を選定する。

[0319] 実装例では、次に、c個のループ文を候補に、実測するパターンを作る。例えば、1番目と3番目のループが高リソース効率であった場合、1番をオフロード、3番をオフロードする各OpenCLパターンを作成して、コンパイルして性能測定する。複数の単ループ文のオフロードパターンで高速化できている場合（例えば、1番と3番両方が高速化できている場合）は、その組合せでのOpenCLパターンを作成して、コンパイルして性能測定する（例えば1番と3番両方をオフロードするパターン）。

[0320] なお、単ループの組み合わせを作る際は、利用リソース量も組み合わせに

なる。このため、上限値に納まらない場合は、その組合せパターンは作らない。組合せも含めてd個のパターンを作成した場合、検証環境のFPGAを備えたサーバで性能測定を行う。性能測定には、高速化したいアプリケーションで指定されたサンプル処理を行う。例えば、フーリエ変換のアプリケーションであれば、サンプルデータでの変換処理をベンチマークに性能測定をする。

実装例では、最後に、複数の測定パターンの高速なパターンを解として選択する。

[0321] 第2実施形態でも第1実施形態で述べたと同様の「リソース量決定と配置決定」を実行する（説明省略）。

[0322] [評価]

評価を説明する。

第2実施形態の[ループ文のFPGA自動オフロード]では、第1実施形態の[ループ文のGPU自動オフロード]と同様に評価できる。

[0323] <評価対象>

評価対象は、第2実施形態の[ループ文のFPGA自動オフロード]では、MRI（Magnetic Resonance Imaging）画像処理のMRI-Qとする。

MRI-Qは、非デカルト空間の3次元MRI再構成アルゴリズムで 사용되는スキャナー構成を表す行列Qを計算する。MRI-Qは、C言語で記述されており、性能測定中に3次元MRI画像処理を実行し、Large(最大)の $64 \times 64 \times 64$ サイズのデータで処理時間を測定する。CPU処理は、C言語を用い、FPGA処理はOpenCLに基づき処理される。

[0324] <評価手法>

対象となるアプリケーションのコードを入力し、移行先のGPUやFPGAに対して、Clang等で認識されたループ文オフロードを試行してオフロードパターンを決める。この際に、処理時間と電力使用量を測定する。最終オフロードパターンについて、電力使用量の時間変化を取得し、全てCPUで処理する場合に比した低電力化を確認する。

第2実施形態の「ループ文のFPGA自動オフロード」では、GAは行わず、算術強度等を用いて、測定パターンが4パターンとなるまで絞り込む。

オフロード対象ループ文：MRI-Q 16

パターン適合度：処理時間が低い程、評価値が高くなり、高適合度になる。第2の実施形態のMRI-Qでも前述の図12のような形で、単純に安さ優先や応答時間優先の配置に比べて、コストや応答時間が改善できる。

[0325] [ハードウェア構成]

第1および第2の実施形態に係るオフロードサーバは、例えば図18に示すような構成の物理装置であるコンピュータ900によって実現される。

図18は、オフロードサーバ1、1Aの機能を実現するコンピュータの一例を示すハードウェア構成図である。コンピュータ900は、CPU901、RAM902、ROM903、HDD904、アクセラレータ905、入出力インターフェイス（I/F）906、メディアインターフェイス（I/F）907、および通信インターフェイス（I/F：Interface）908を有する。

[0326] アクセラレータ905は、通信I/F908からのデータ、または、RAM902からのデータの少なくとも一方のデータを高速に処理するアクセラレータ（デバイス）である。例えば、アクセラレータ905は、図2の各種デバイス（Device）151、CPU-GPUを有する装置152、CPU-FPGAを有する装置153、CPUを有する装置154のアクセラレータである。

なお、アクセラレータ905として、CPU901またはRAM902からの処理を実行した後にCPU901またはRAM902に実行結果を戻すタイプ（look-aside型）を用いてもよい。一方、アクセラレータ905として、通信I/F908とCPU901またはRAM902との間に入って、処理を行うタイプ（in-line型）を用いてもよい。

[0327] アクセラレータ905は、通信I/F908を介して外部装置915と接続される。入出力I/F906は、入出力装置916と接続される。メディ

アイ／Ｆ９０７は、記録媒体９１７からデータを読み書きする。

[0328] CPU 901は、ROM 903またはHDD 904に記憶されたプログラムに基づき作動し、RAM 902に読み込んだプログラム（アプリケーションや、その略のアプリとも呼ばれる）を実行することにより、図１、図１３に示すオフロードサーバ１，１Ａの各処理部による制御を行う。そして、このプログラムは、通信回線を介して配布したり、CD-ROM等の記録媒体９１７に記録して配布したりすることも可能である。

ROM 903は、コンピュータ９００の起動時にCPU 901によって実行されるブートプログラムや、コンピュータ９００のハードウェアに依存するプログラム等を格納する。

[0329] CPU 901は、入出力Ｉ／Ｆ９０６を介して、マウスやキーボード等の入力部、および、ディスプレイやプリンタ等の出力部からなる入出力装置９１６を制御する。CPU 901は、入出力Ｉ／Ｆ９０６を介して、入出力装置９１６からデータを取得するとともに、生成したデータを入出力装置９１６へ出力する。なお、プロセッサとしてCPU 901とともに、GPU（Graphics Processing Unit）等を用いてもよい。

[0330] HDD 904は、CPU 901により実行されるプログラムおよび当該プログラムによって使用されるデータ等を記憶する。通信Ｉ／Ｆ９０８は、通信網（例えば、NW（Network））を介して他の装置からデータを受信してCPU 901へ出力し、また、CPU 901が生成したデータを、通信網を介して他の装置へ送信する。

[0331] メディアＩ／Ｆ９０７は、記録媒体９１７に格納されたプログラムまたはデータを読み取り、RAM 902を介してCPU 901へ出力する。CPU 901は、目的の処理に係るプログラムを、メディアＩ／Ｆ９０７を介して記録媒体９１７からRAM 902上にロードし、ロードしたプログラムを実行する。記録媒体９１７は、DVD（Digital Versatile Disc）、PD（Phase change rewritable Disk）等の光学記録媒体、MO（Magneto Optical disk）等の光磁気記録媒体、磁気記録媒体、導体メモリーテープ媒体又は半導体

メモリ等である。

[0332] 例えば、コンピュータ 900 が第 1 および第 2 の実施形態に係るオフロードサーバ 1, 1 A として機能する場合、コンピュータ 900 の CPU 901 は、RAM 902 上にロードされたプログラムを実行することによりオフロードサーバ 1, 1 A の機能を実現する。また、HDD 904 には、RAM 902 内のデータが記憶される。CPU 901 は、目的の処理に係るプログラムを記録媒体 912 から読み取って実行する。この他、CPU 901 は、他の装置から通信網を介して目的の処理に係るプログラムを読み込んでもよい。

[0333] [効果]

以上説明したように、第 1 実施形態に係るオフロードサーバ 1 (図 1 参照) は、アプリケーションプログラムの特定処理をアクセラレータにオフロードするオフロードサーバであって、アプリケーションプログラムのソースコードを分析するアプリケーションコード分析部 112 と、アプリケーションプログラムのループ文の中で用いられる変数の参照関係を分析し、ループ外でデータ転送してよいデータについては、ループ外でのデータ転送を明示的に指定する明示的指定行を用いたデータ転送指定を行うデータ転送指定部 113 と、アプリケーションプログラムのループ文を特定し、特定した各ループ文に対して、アクセラレータにおける並列処理指定文を指定してコンパイルする並列処理指定部 114 と、コンパイルエラーが出るループ文に対して、オフロード対象外とするとともに、コンパイルエラーが出ないループ文に対して、並列処理するかしないかの指定を行う並列処理パターンを作成する並列処理パターン作成部 117 と、並列処理パターンのアプリケーションプログラムをコンパイルして、アクセラレータ検証用装置に配置し、アクセラレータにオフロードした際の性能測定用処理を実行する性能測定部 118 と、変換したアプリケーションプログラムを、ユーザの指定するコストまたは応答時間の条件に応じて、ネットワーク上の、クラウドサーバ、キャリアエッジサーバ、ユーザエッジサーバのいずれかに配置する際、デバイスおよび

リンクのコスト、計算リソース上限、帯域上限を制約条件とし、かつ計算リソースのコストまたは応答時間を目的関数とした線形計画式に基づいて、アプリケーションプログラムの配置場所を計算して設定する配置設定部 170 と、を備える。

[0334] このようにすることにより、GPU、FPGA等のアクセラレータに自動オフロードしたアプリケーションに対して、ユーザのリクエストする価格条件や応答時間条件、アプリケーションの配置数等を変更して、ネットワーク上の、クラウドサーバ、キャリアエッジサーバ、ユーザエッジサーバのいずれかに配置する適正配置を計算する。これにより、変換したアプリケーションを計算リソースのコストまたは応答時間の要求を満たして、ユーザ要望に従った最適配置を実現することができる。

[0335] 第2実施形態に係るオフロードサーバ1A（図13参照）は、アプリケーションプログラムの特定処理をPLDにオフロードするオフロードサーバであって、アプリケーションプログラムのソースコードを分析するアプリケーションコード分析部112と、アプリケーションプログラムのループ文を特定し、特定した各ループ文に対して、PLDにおけるパイプライン処理および並列処理をOpenCLで指定した複数のオフロード処理パターンにより作成してコンパイルするPLD処理指定部213と、アプリケーションプログラムのループ文の算術強度を算出する算術強度算出部214と、算術強度算出部214が算出した算術強度をもとに、算術強度が所定の閾値より高いループ文をオフロード候補として絞り込み、PLD処理パターンを作成するPLD処理パターン作成部215と、作成されたPLD処理パターンのアプリケーションプログラムをコンパイルして、アクセラレータ検証用装置に配置し、PLDにオフロードした際の性能測定用処理を実行する性能測定部118と、変換したアプリケーションプログラムを、ユーザの指定するコストまたは応答時間の条件に応じて、ネットワーク上の、クラウドサーバ、キャリアエッジサーバ、ユーザエッジサーバのいずれかに配置する際、デバイスおよびリンクのコスト、計算リソース上限、帯域上限を制約条件とし、かつ計算リ

ソースのコストまたは応答時間を目的関数とした線形計画式に基づいて、アプリケーションプログラムの配置場所を計算して設定する配置設定部 170 と、を備える。

[0336] このようにすることにより、実際に性能測定するパターンを絞ってから検証環境に配置し、コンパイルして PLD（例えば、FPGA）実機で性能測定することで、性能測定する回数を減らすことができる。これにより、PLD への自動オフロードにおいて、アプリケーションのループ文の自動オフロードを高速で行うことができる。そして、変換したアプリケーションに対して、ユーザのリクエストする価格条件や応答時間条件、アプリケーションの配置数等を変更して、ネットワーク上の、クラウドサーバ、キャリアエッジサーバ、ユーザエッジサーバのいずれかに配置する適正配置を計算する。これにより、変換したアプリケーションについて、計算リソースのコストまたは応答時間の要求を満たして、ユーザ要望に従った最適配置を実現することができる。

[0337] 第 1 および第 2 実施形態に係るオフロードサーバ 1, 1A において、配置設定部 170 は、サーバにアプリケーションプログラムを配置した際に、計算リソースのコストを極小化する配置、または、応答時間を極小化する配置を計算することを特徴とする。

[0338] このようにすることにより、変換したアプリケーションを計算リソースのコストまたは応答時間の要求を満たして最適に配置することができる。

[0339] 第 1 および第 2 実施形態に係るオフロードサーバ 1, 1A において、配置設定部 170 は、サーバにアプリケーションプログラムを配置した際に、計算リソースのコストを極小化する配置を〔数 1〕に示す線形計画式に従って計算することを特徴とする。

[0340] このようにすることにより、例えば、一月いくら以内での配置が必要な要求の場合は、〔数 1〕の式（1）の応答時間の最小化が目的関数となり、〔数 1〕の式（2）の計算リソースのコストがいくら以内であるかが制約条件の一つとなる。よって、変換したアプリケーションを計算リソースのコスト

の要求を満たして最適に配置することができる。

[0341] 第1および第2実施形態に係るオフロードサーバ1, 1Aにおいて、配置設定部170は、サーバにアプリケーションプログラムを配置した際に、応答時間を極小化する配置を〔数2〕に示す線形計画式に従って計算することを特徴とする。

[0342] このようにすることにより、例えば、アプリケーションの応答時間が何秒以内での配置が必要な要求の場合は、〔数2〕の式(5)のコストの最小化が目的関数となり、〔数2〕の式(6)の応答時間が何秒以内であるかが制約条件の一つとなる。よって、変換したアプリケーションをユーザの応答時間の要求を満たして最適に配置することができる。

[0343] 本発明は、コンピュータを、上記オフロードサーバとして機能させるためのオフロードプログラムとした。

[0344] このようにすることにより、一般的なコンピュータを用いて、上記オフロードサーバ1, 1Aの各機能を実現させることができる。

[0345] また、上記各実施形態において説明した各処理のうち、自動的に行われるものとして説明した処理の全部又は一部を手作業で行うこともでき、あるいは、手作業で行われるものとして説明した処理の全部又は一部を公知の方法で自動的に行うこともできる。この他、上述文書中や図面中に示した処理手順、制御手順、具体的名称、各種のデータやパラメータを含む情報については、特記する場合を除いて任意に変更することができる。

また、図示した各装置の各構成要素は機能概念的なものであり、必ずしも物理的に図示の如く構成されていることを要しない。すなわち、各装置の分散・統合の具体的形態は図示のものに限られず、その全部又は一部を、各種の負荷や使用状況などに応じて、任意の単位で機能的又は物理的に分散・統合して構成することができる。

[0346] また、上記の各構成、機能、処理部、処理手段等は、それらの一部又は全部を、例えば集積回路で設計する等によりハードウェアで実現してもよい。また、上記の各構成、機能等は、プロセッサがそれぞれの機能を実現するプ

プログラムを解釈し、実行するためのソフトウェアで実現してもよい。各機能を実現するプログラム、テーブル、ファイル等の情報は、メモリや、ハードディスク、SSD (Solid State Drive) 等の記録装置、又は、IC (Integrated Circuit) カード、SD (Secure Digital) カード、光ディスク等の記録媒体に保持することができる。

[0347] また、本実施形態では、組合せ最適化問題を、限られた最適化期間中に解を発見できるようにするため、遺伝的アルゴリズム (GA) の手法を用いているが、最適化の手法はどのようなものでもよい。例えば、local search (局所探索法)、Dynamic Programming (動的計画法)、これらの組み合わせでもよい。

[0348] また、本実施形態では、C/C++向けOpenACCコンパイラを用いているが、GPU処理をオフロードできるものであればどのようなものでもよい。例えば、Java lambda (登録商標) GPU処理、IBM Java 9 SDK (登録商標) でもよい。なお、並列処理指定文は、これらの開発環境に依存する。

例えば、Java (登録商標) では、Java 8よりlambda形式での並列処理記述が可能である。IBM (登録商標) は、lambda形式の並列処理記述を、GPUにオフロードするJITコンパイラを提供している。Javaでは、これらを用いて、ループ処理をlambda形式にするか否かのチューニングをGAで行うことで、同様のオフロードが可能である。

[0349] また、本実施形態では、繰り返し文 (ループ文) として、for文を例示したが、for文以外のwhile文やdo-while文も含まれる。ただし、ループの継続条件等を指定するfor文がより適している。

符号の説明

- [0350] 1, 1A オフロードサーバ
- 11, 21 制御部
 - 12 入出力部
 - 13 記憶部
 - 14 検証用マシン (アクセラレータ検証用装置)

- 1 1 1 アプリケーションコード指定部
- 1 1 2 アプリケーションコード分析部
- 1 1 3 データ転送指定部
- 1 1 4 並列処理指定部
- 1 1 4 a, 2 1 3 a オフロード範囲抽出部
- 1 1 4 b, 2 1 3 b 中間言語ファイル出力部
- 1 1 5 リソース比決定部
- 1 1 6 リソース量設定部
- 1 1 7 並列処理パターン作成部
- 1 1 8 性能測定部
- 1 1 8 a バイナリファイル配置部
- 1 1 9 実行ファイル作成部
- 1 2 0 本番環境配置部
- 1 2 1 性能測定テスト抽出実行部
- 1 2 2 ユーザ提供部
- 1 2 5 アプリケーションコード
- 1 3 1 テストケースDB
- 1 3 2 設備リソースDB
- 1 3 3 中間言語ファイル
- 1 5 1 各種デバイス
- 1 5 2 CPU-GPUを有する装置
- 1 5 3 CPU-FPGAを有する装置
- 1 5 4 CPUを有する装置
- 1 7 0 配置設定部
- 2 1 3 PLD処理指定部
- 2 1 4 算術強度算出部
- 2 1 5 PLD処理パターン作成部
- 9 0 5 アクセラレータ

請求の範囲

[請求項1]

アプリケーションプログラムの特定処理をアクセラレータにオフロードするオフロードサーバであって、

前記アプリケーションプログラムのソースコードを分析するアプリケーションコード分析部と、

前記アプリケーションプログラムのループ文の中で用いられる変数の参照関係を分析し、ループ外でデータ転送してよいデータについては、ループ外でのデータ転送を明示的に指定する明示的指定行を用いたデータ転送指定を行うデータ転送指定部と、

前記アプリケーションプログラムのループ文を特定し、特定した各前記ループ文に対して、前記アクセラレータにおける並列処理指定文を指定してコンパイルする並列処理指定部と、

コンパイルエラーが出るループ文に対して、オフロード対象外とするとともに、コンパイルエラーが出ないループ文に対して、並列処理するかしないかの指定を行う並列処理パターンを作成する並列処理パターン作成部と、

前記並列処理パターンの前記アプリケーションプログラムをコンパイルして、アクセラレータ検証用装置に配置し、前記アクセラレータにオフロードした際の性能測定用処理を実行する性能測定部と、

変換した前記アプリケーションプログラムを、ユーザの指定するコストまたは応答時間の条件に応じて、ネットワーク上の、クラウドサーバ、キャリアエッジサーバ、ユーザエッジサーバのいずれかに配置する際、デバイスおよびリンクのコスト、計算リソース上限、帯域上限を制約条件とし、かつ計算リソースのコストまたは応答時間を目的関数とした線形計画式に基づいて、アプリケーションプログラムの配置場所を計算して設定する配置設定部と、

を備えることを特徴とするオフロードサーバ。

[請求項2]

アプリケーションプログラムの特定処理を P L D (Programmable L

ogic Device) にオフロードするオフロードサーバであって、

前記アプリケーションプログラムのソースコードを分析するアプリケーションコード分析部と、

前記アプリケーションプログラムのループ文を特定し、特定した各前記ループ文に対して、前記P L Dにおけるパイプライン処理および並列処理をOpenCLで指定した複数のオフロード処理パターンにより作成してコンパイルするP L D処理指定部と、

前記アプリケーションプログラムのループ文の算術強度を算出する算術強度算出部と、

前記算術強度算出部が算出した算術強度をもとに、前記算術強度が所定の閾値より高いループ文をオフロード候補として絞り込み、P L D処理パターンを作成するP L D処理パターン作成部と、

作成された前記P L D処理パターンの前記アプリケーションプログラムをコンパイルして、アクセラレータ検証用装置に配置し、前記P L Dにオフロードした際の性能測定用処理を実行する性能測定部と、

変換した前記アプリケーションプログラムを、ユーザの指定するコストまたは応答時間の条件に応じて、ネットワーク上の、クラウドサーバ、キャリアエッジサーバ、ユーザエッジサーバのいずれかに配置する際、デバイスおよびリンクのコスト、計算リソース上限、帯域上限を制約条件とし、かつ計算リソースのコストまたは応答時間を目的関数とした線形計画式に基づいて、アプリケーションプログラムの配置場所を計算して設定する配置設定部と、

を備えることを特徴とするオフロードサーバ。

[請求項3]

前記配置設定部は、サーバにアプリケーションプログラムを配置した際に、計算リソースのコストを極小化する配置、または、応答時間を極小化する配置を計算する

ことを特徴とする請求項1または請求項2に記載のオフロードサーバ。

[請求項4] 前記配置設定部は、サーバにアプリケーションプログラムを配置した際に、応答時間を極小化する配置を次式に示す線形計画式に従って計算する

[数1]

$$R_k = \sum_{i \in Device} (A_{i,k}^d \cdot B_{i,k}^p) + \sum_{j \in Link} (A_{j,k}^l \cdot \frac{C_k}{B_k^l}) \quad \dots(1)$$

$$\sum_{i \in Device} a_i (\frac{A_{i,k}^d \cdot B_k^d}{C_i^d}) + \sum_{j \in Link} b_j (\frac{A_{j,k}^l \cdot B_k^l}{C_j^l}) \leq P_k \quad \dots(2)$$

$$\sum_{k \in App} (A_{i,k}^d \cdot B_k^d) \leq C_i^d \quad \dots(3)$$

$$\sum_{k \in App} (A_{j,k}^l \cdot B_k^l) \leq C_j^l \quad \dots(4)$$

a_i : デバイス使用コスト

b_j : リンク使用コスト

C_i^d : i 番目のデバイスの計算リソース上限

C_j^l : j 番目のリンクの帯域上限

C_k : k 番目のアプリで使うデータ容量

$A_{i,k}^d$: i 番目のデバイスで k 番目のアプリ使用有無

$A_{j,k}^l$: j 番目のリンクで k 番目のアプリ使用有無

B_k^d : k 番目のアプリが使用する計算リソース量

B_k^l : k 番目のアプリが使用する帯域

$B_{i,k}^p$: i 番目のデバイスでの k 番目アプリの処理時間

ことを特徴とする請求項 1 または請求項 2 に記載のオフロードサーバ。

[請求項5] 前記配置設定部は、サーバにアプリケーションプログラムを配置した際に、計算リソースのコストを極小化する配置を次式に示す線形計画式に従って計算する

[数2]

$$P_k = \sum_{i \in Device} a_i \left(\frac{A_{i,k}^d \cdot B_k^d}{C_i^d} \right) + \sum_{j \in Link} b_j \left(\frac{A_{j,k}^l \cdot B_k^l}{C_j^l} \right) \quad \dots(5)$$

$$\sum_{i \in Device} (A_{i,k}^d \cdot B_{i,k}^p) + \sum_{j \in Link} (A_{j,k}^l \cdot \frac{C_k}{B_k^l}) \leq R_k \quad \dots(6)$$

$$\sum_{k \in App} (A_{i,k}^d \cdot B_k^d) \leq C_i^d \quad \dots(3)$$

$$\sum_{k \in App} (A_{j,k}^l \cdot B_k^l) \leq C_j^l \quad \dots(4)$$

a_i : デバイス使用コスト

b_j : リンク使用コスト

C_i^d : i 番目のデバイスの計算リソース上限

C_j^l : j 番目のリンクの帯域上限

C_k : k 番目のアプリで使うデータ容量

$A_{i,k}^d$: i 番目のデバイスで k 番目のアプリ使用有無

$A_{i,k}^l$: j 番目のリンクで k 番目のアプリ使用有無

B_k^d : k 番目のアプリが使用する計算リソース量

B_k^l : k 番目のアプリが使用する帯域

$B_{i,k}^p$: i 番目のデバイスでの k 番目アプリの処理時間

ことを特徴とする請求項1または請求項2に記載のオフロードサーバ。

[請求項6] アプリケーションプログラムの特定処理をアクセラレータにオフロ

ードするオフロードサーバのオフロード制御方法であって、

前記オフロードサーバは、

前記アプリケーションプログラムのソースコードを分析するステップと、

前記アプリケーションプログラムのループ文の中で用いられる変数の参照関係を分析し、ループ外でデータ転送してよいデータについては、ループ外でのデータ転送を明示的に指定する明示的指定行を用いたデータ転送指定を行うステップと、

前記アプリケーションプログラムのループ文を特定し、特定した各前記ループ文に対して、前記アクセラレータにおける並列処理指定文を指定してコンパイルするステップと、

コンパイルエラーが出るループ文に対して、オフロード対象外とするとともに、コンパイルエラーが出ないループ文に対して、並列処理するかしないかの指定を行う並列処理パターンを作成するステップと、

前記並列処理パターンの前記アプリケーションプログラムをコンパイルして、アクセラレータ検証用装置に配置し、前記アクセラレータにオフロードした際の性能測定用処理を実行するステップと、

変換した前記アプリケーションプログラムを、ユーザの指定するコストまたは応答時間の条件に応じて、ネットワーク上の、クラウドサーバ、キャリアエッジサーバ、ユーザエッジサーバのいずれかに配置する際、デバイスおよびリンクのコスト、計算リソース上限、帯域上限を制約条件とし、かつ計算リソースのコストまたは応答時間を目的関数とした線形計画式に基づいて、アプリケーションプログラムの配置場所を計算して設定するステップと、を実行する

ことを特徴とするオフロード制御方法。

[請求項7]

アプリケーションプログラムの特定処理をPLD (Programmable Logic Device) にオフロードするオフロードサーバのオフロード制御

方法であって、

前記オフロードサーバは、

前記アプリケーションプログラムのソースコードを分析するステップと、

前記アプリケーションプログラムのループ文を特定し、特定した各前記ループ文に対して、前記P L Dにおけるパイプライン処理および並列処理をOpenCLで指定した複数のオフロード処理パターンにより作成してコンパイルするステップと、

前記アプリケーションプログラムのループ文の算術強度を算出する算術強度算出部と、

前記算術強度算出部が算出した算術強度をもとに、前記算術強度が所定の閾値より高いループ文をオフロード候補として絞り込み、P L D処理パターンを作成するステップと、

作成された前記P L D処理パターンの前記アプリケーションプログラムをコンパイルして、アクセラレータ検証用装置に配置し、前記P L Dにオフロードした際の性能測定用処理を実行するステップと、

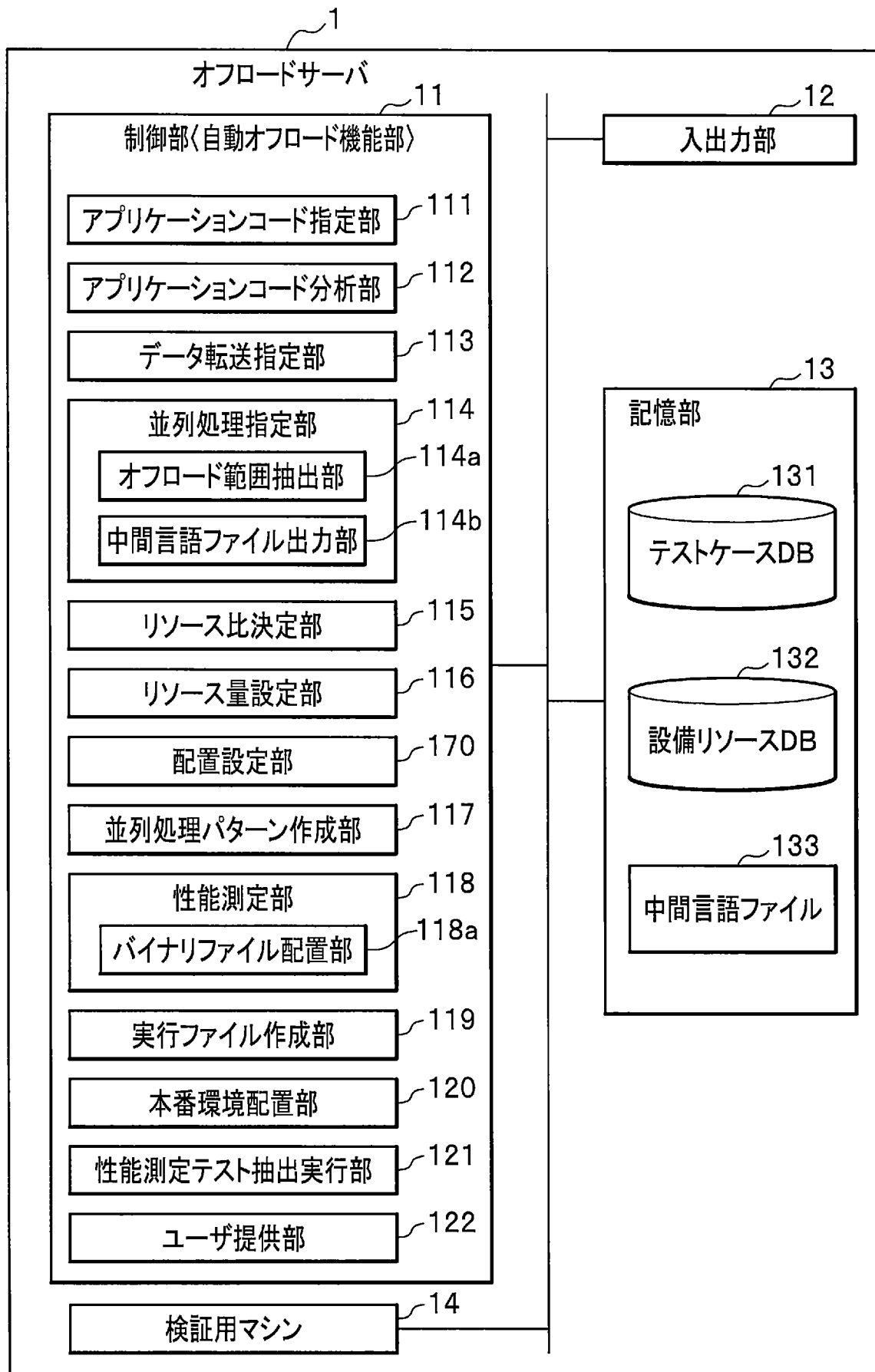
変換した前記アプリケーションプログラムを、ユーザの指定するコストまたは応答時間の条件に応じて、ネットワーク上の、クラウドサーバ、キャリアエッジサーバ、ユーザエッジサーバのいずれかに配置する際、デバイスおよびリンクのコスト、計算リソース上限、帯域上限を制約条件とし、かつ計算リソースのコストまたは応答時間を目的関数とした線形計画式に基づいて、アプリケーションプログラムの配置場所を計算して設定するステップと、を実行する

ことを特徴とするオフロード制御方法。

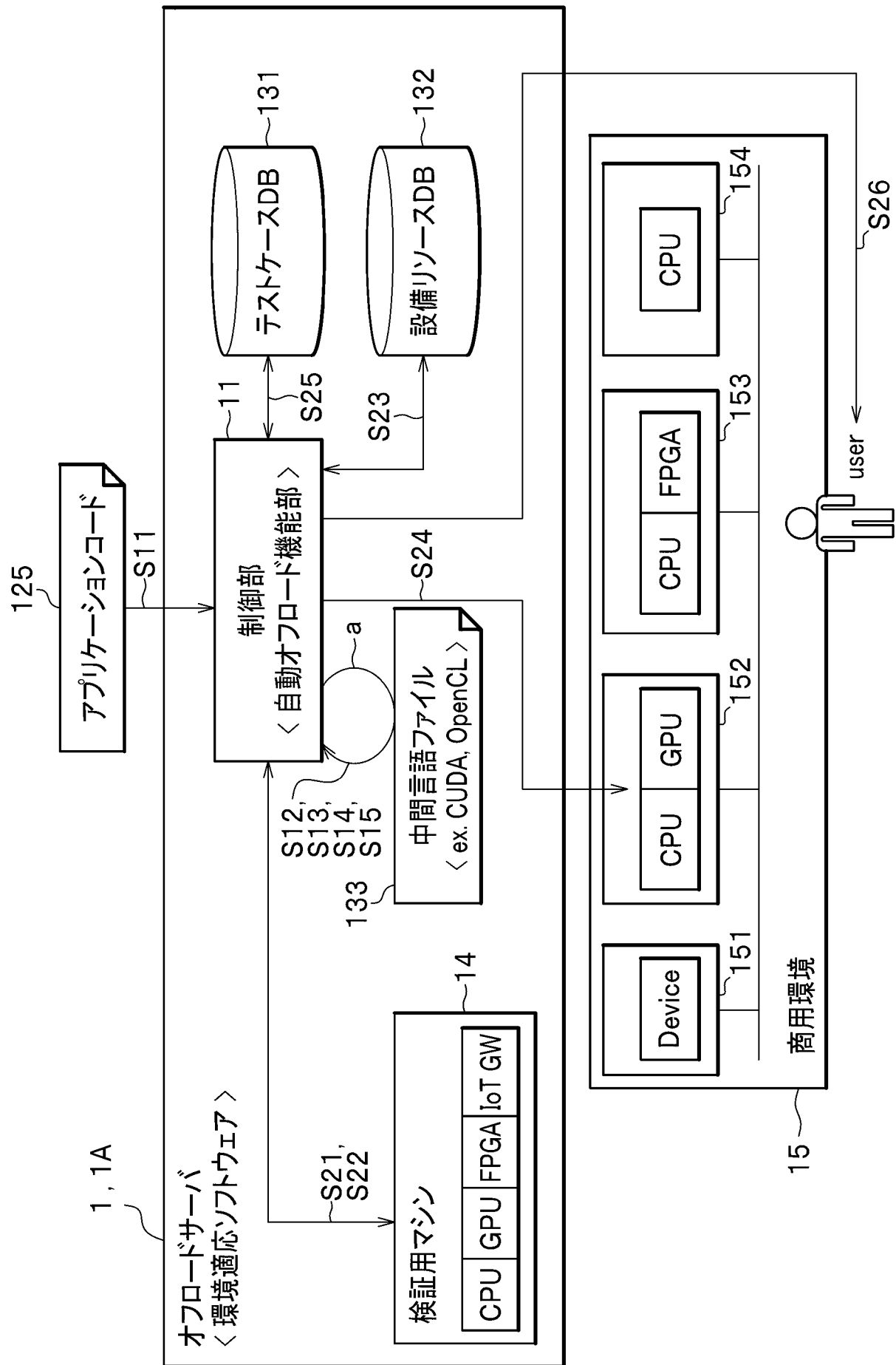
[請求項8]

コンピュータを、請求項1乃至請求項5のいずれか一項に記載のオフロードサーバとして機能させるためのオフロードプログラム。

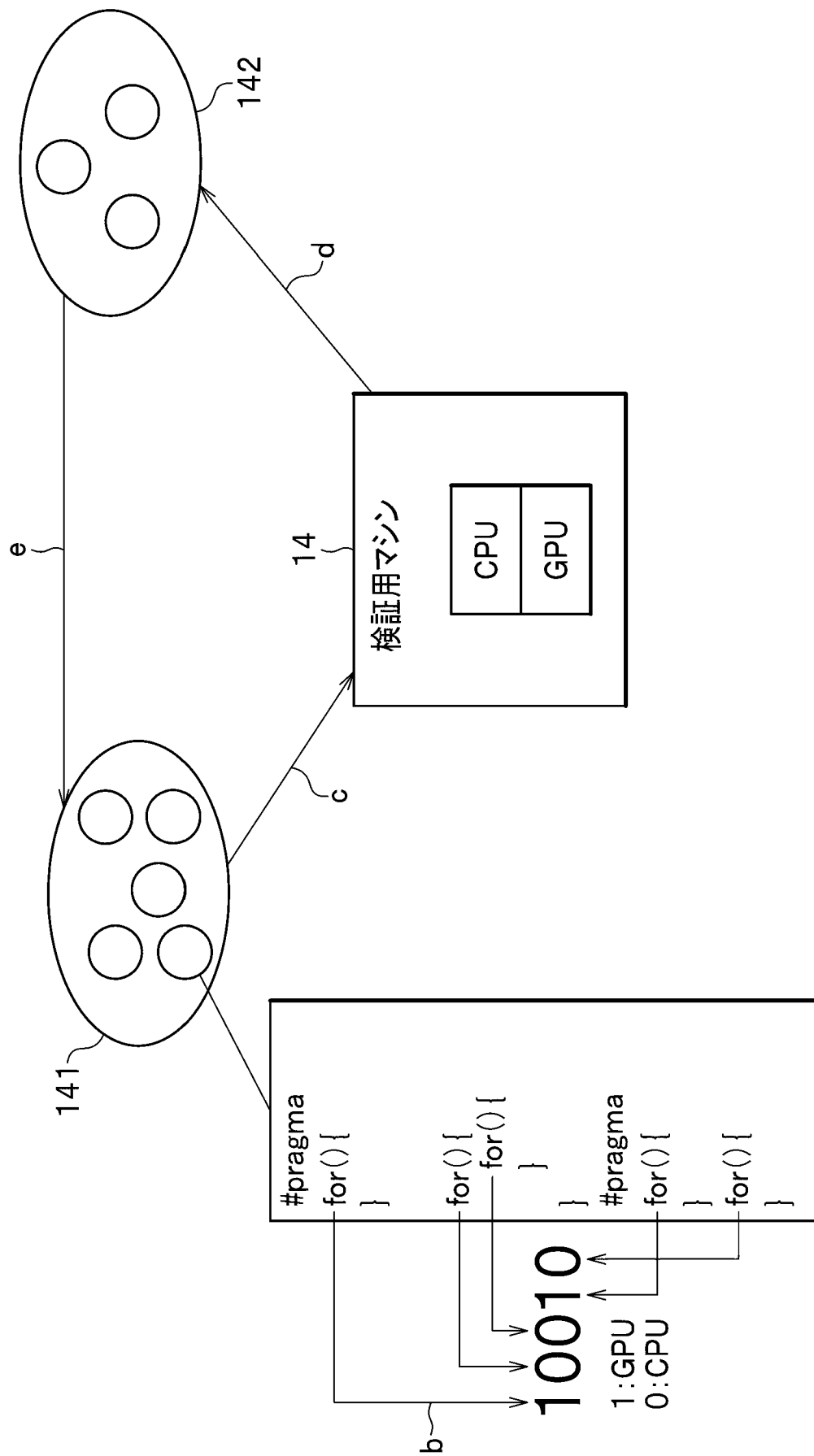
[図1]



[図2]



[図3]



[図4]

CPU

〈1〉 for(i=0; i<10; i++){

 〈2〉 for(j=0; j<20; j++){

 a, b ← f

 }

}

〈3〉 for(k=0; k<30; k++){

 c, d ← g

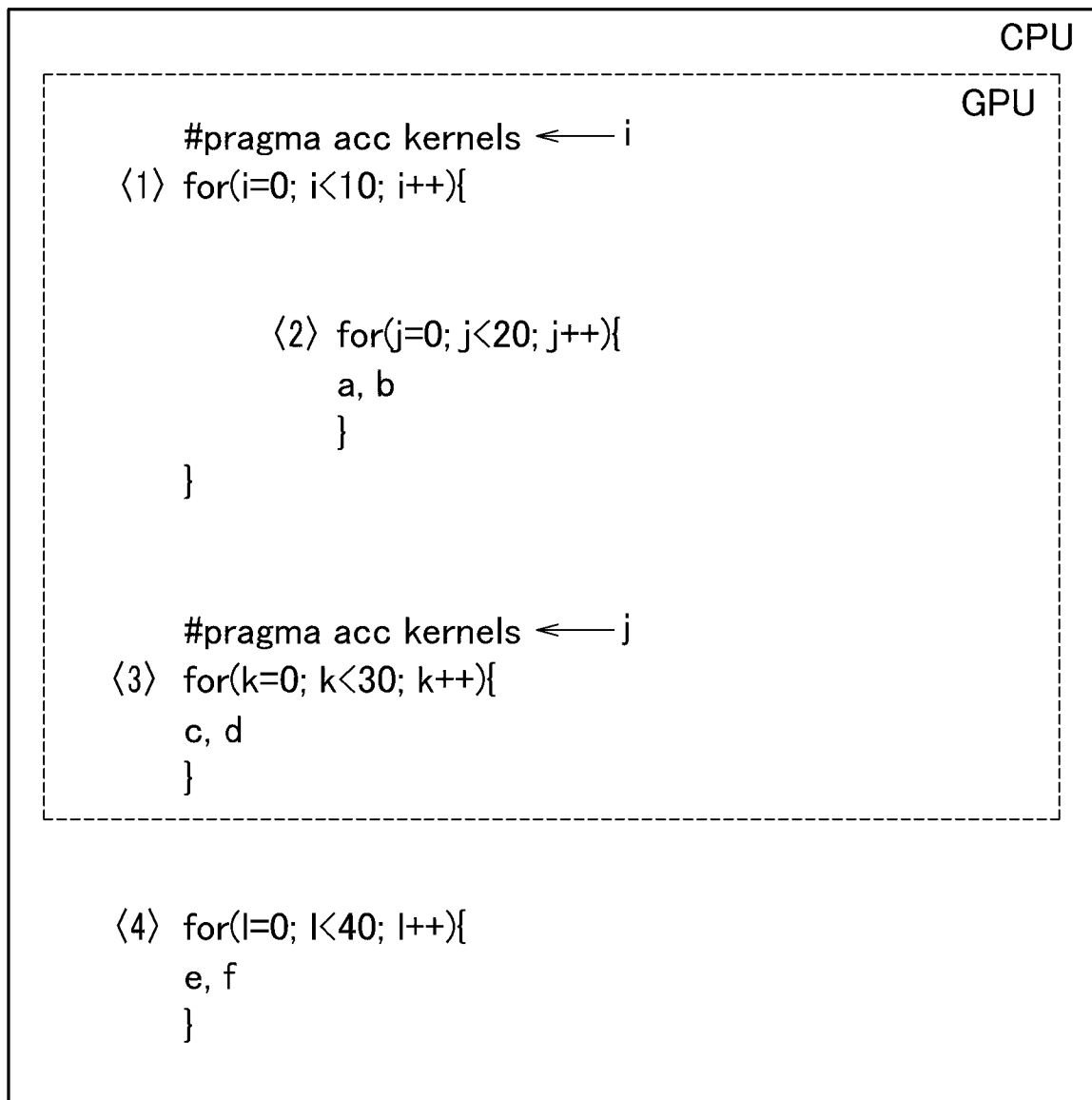
}

〈4〉 for(l=0; l<40; l++){

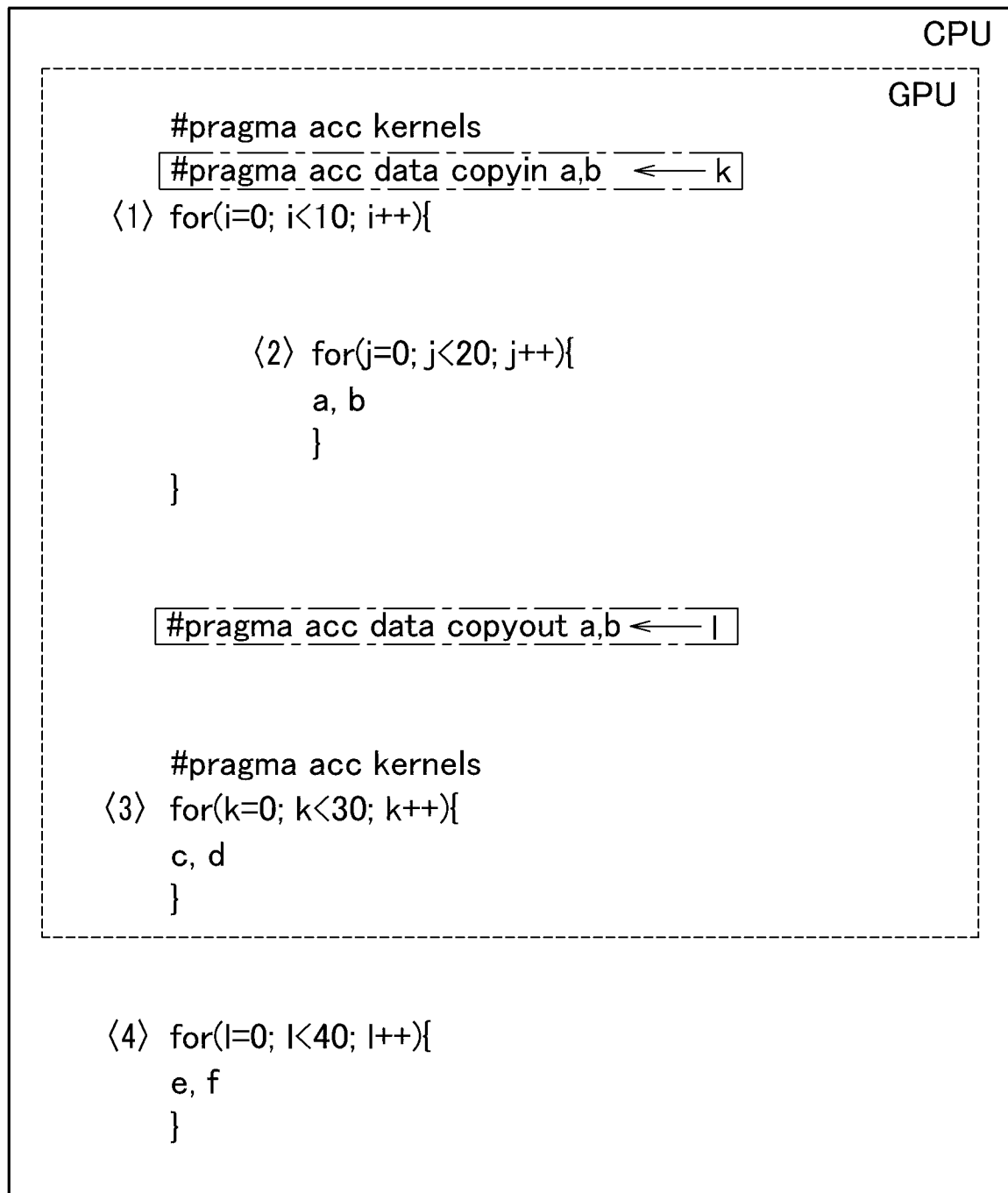
 e, f ← h

}

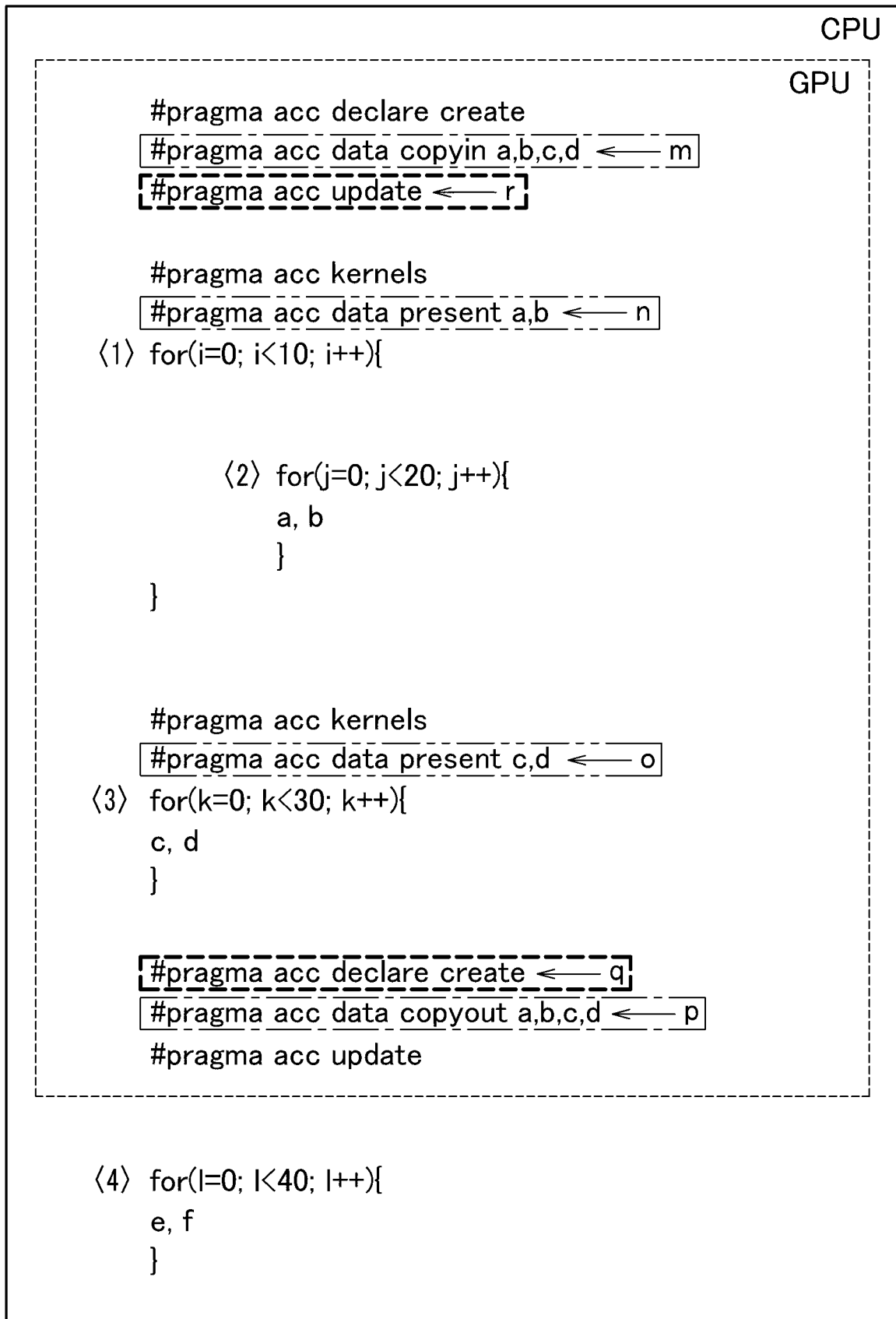
[図5]



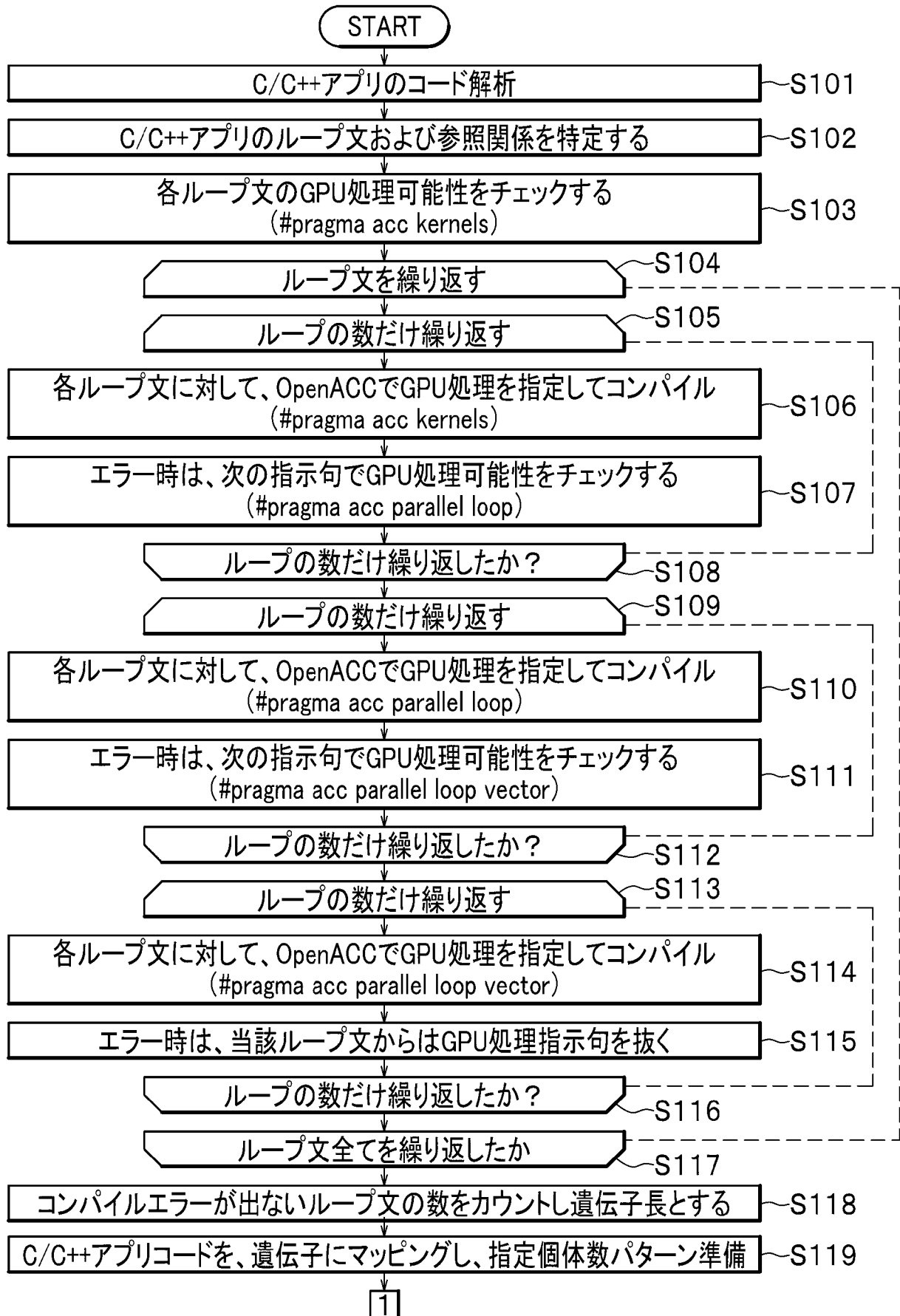
[図6]



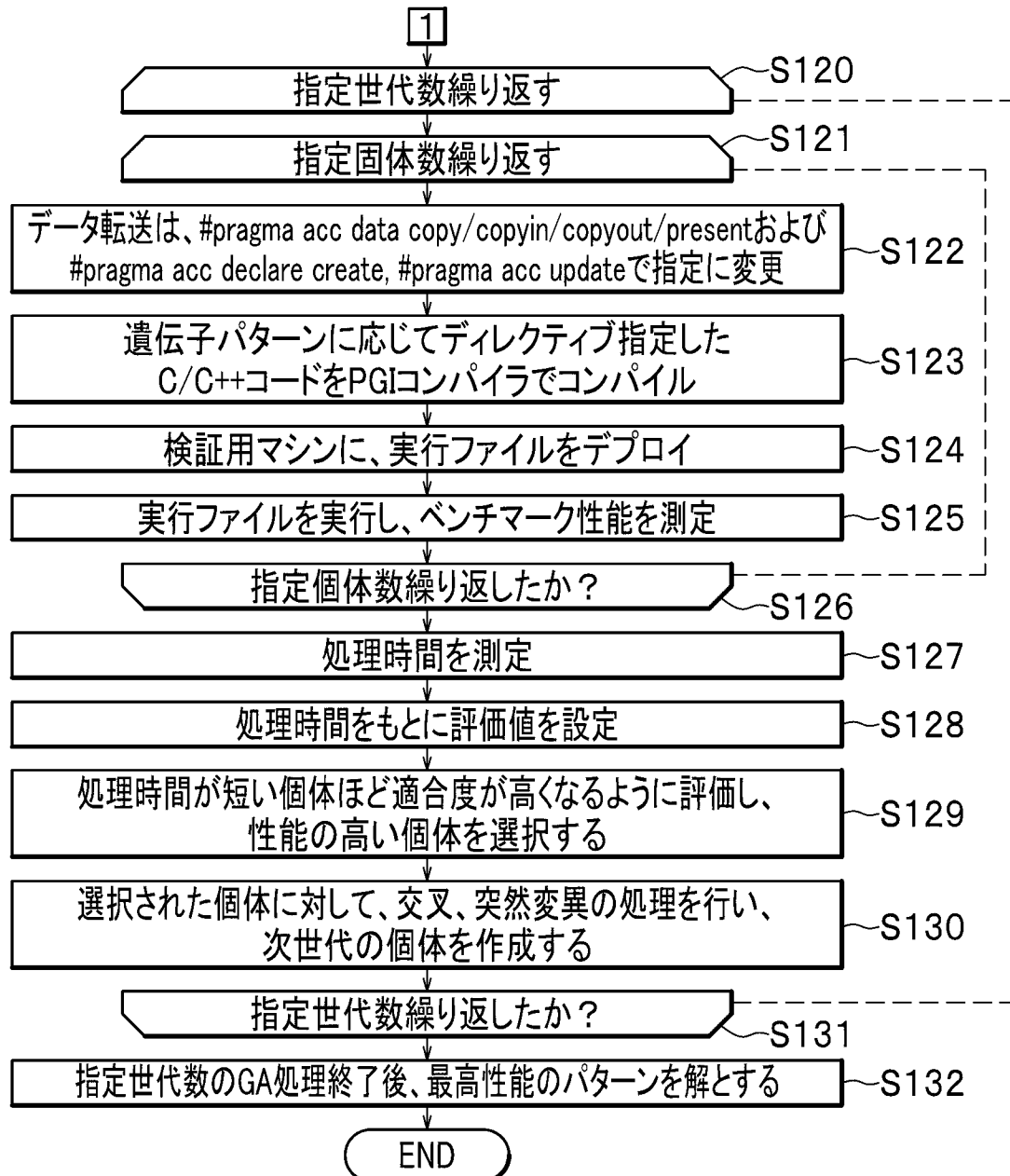
[図8]



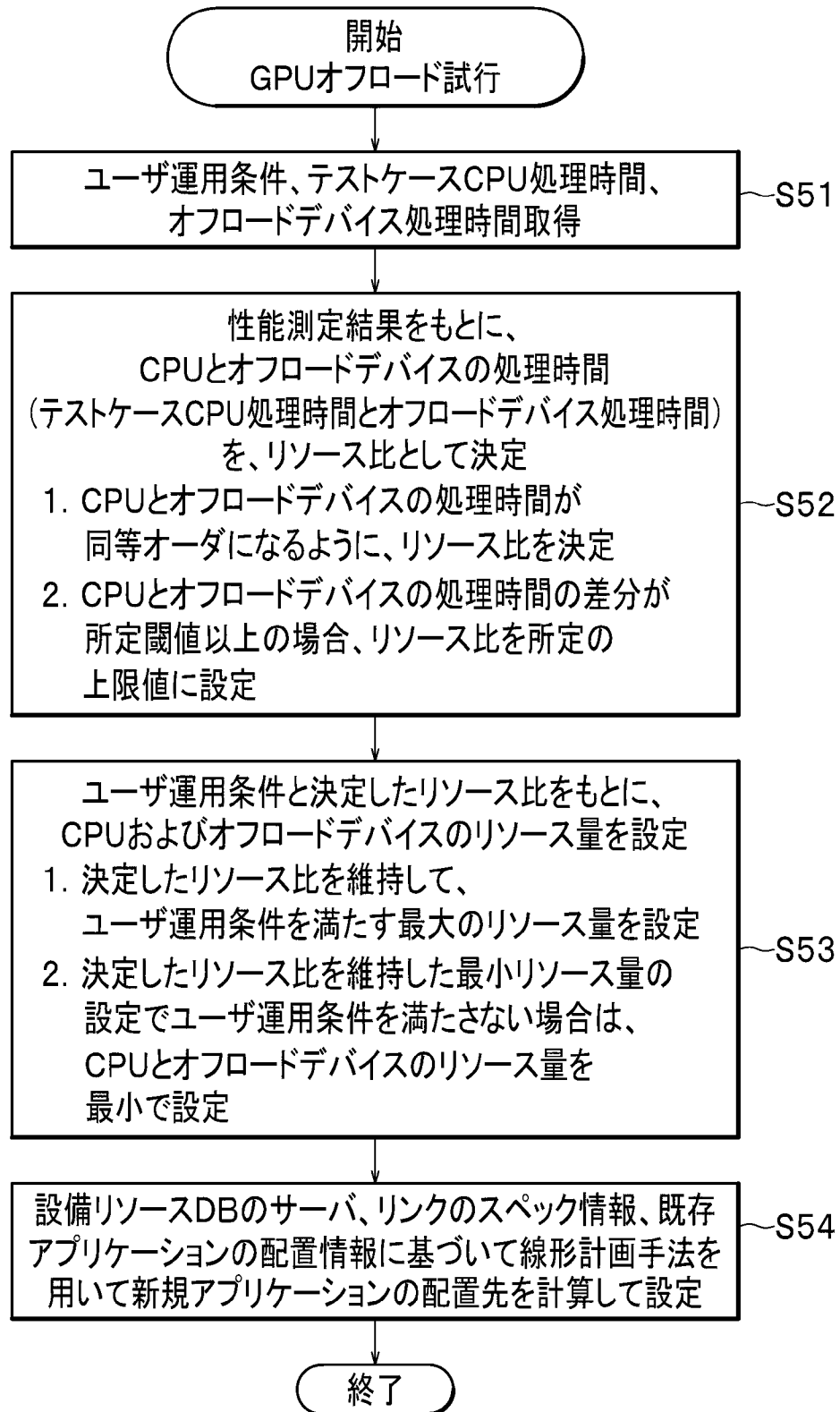
[図9A]



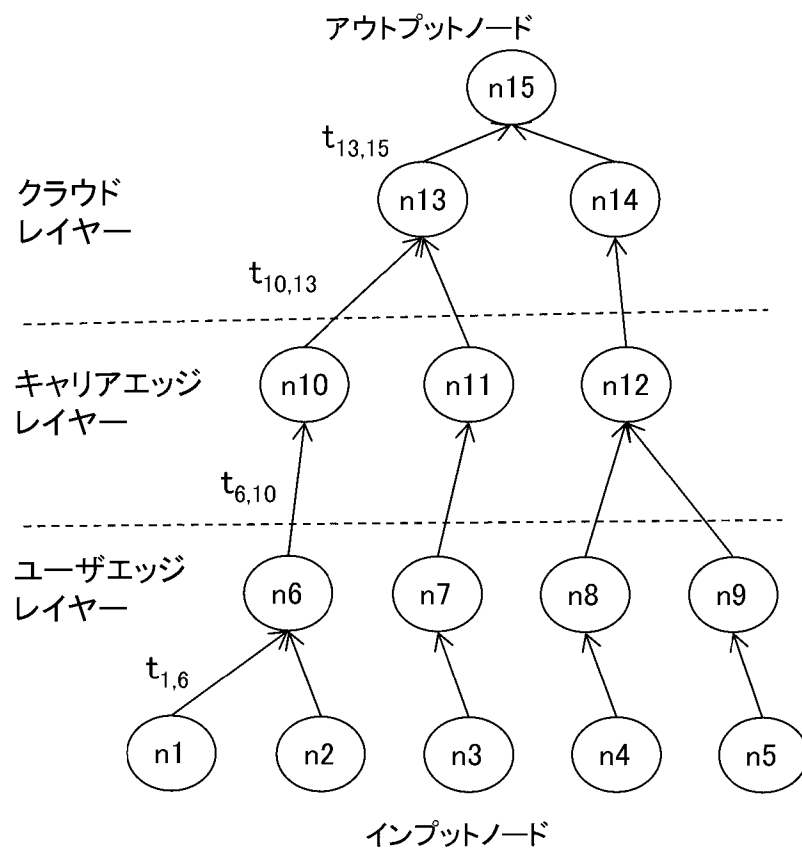
[図9B]



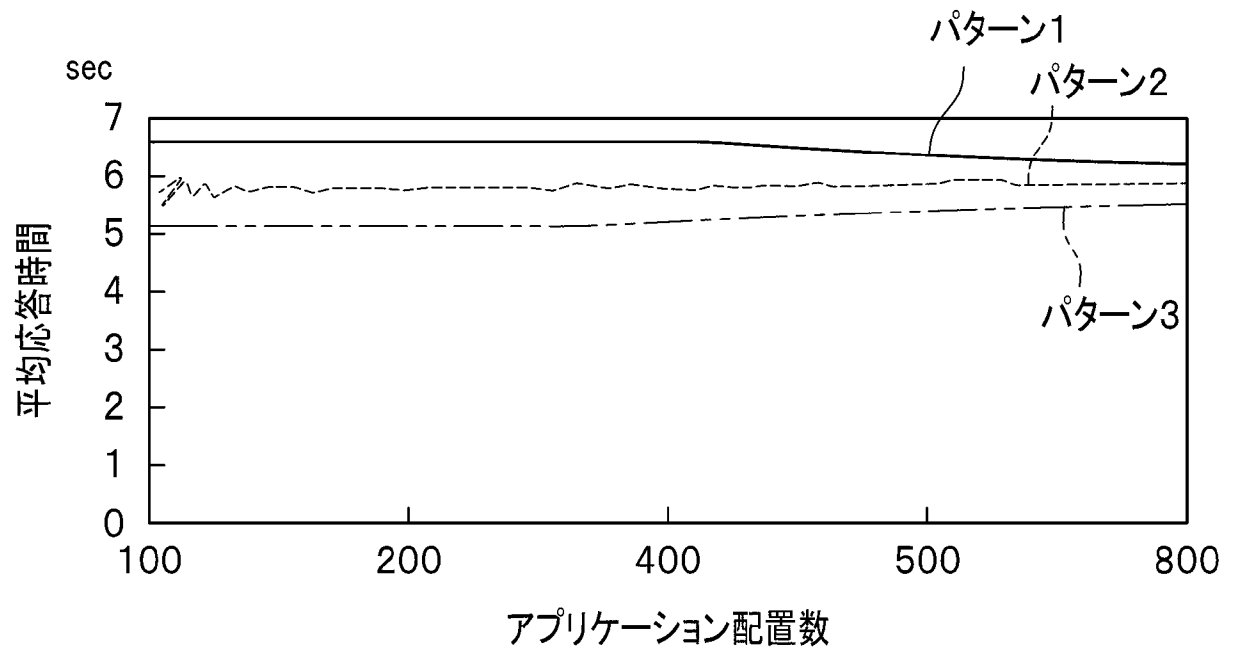
[図10]



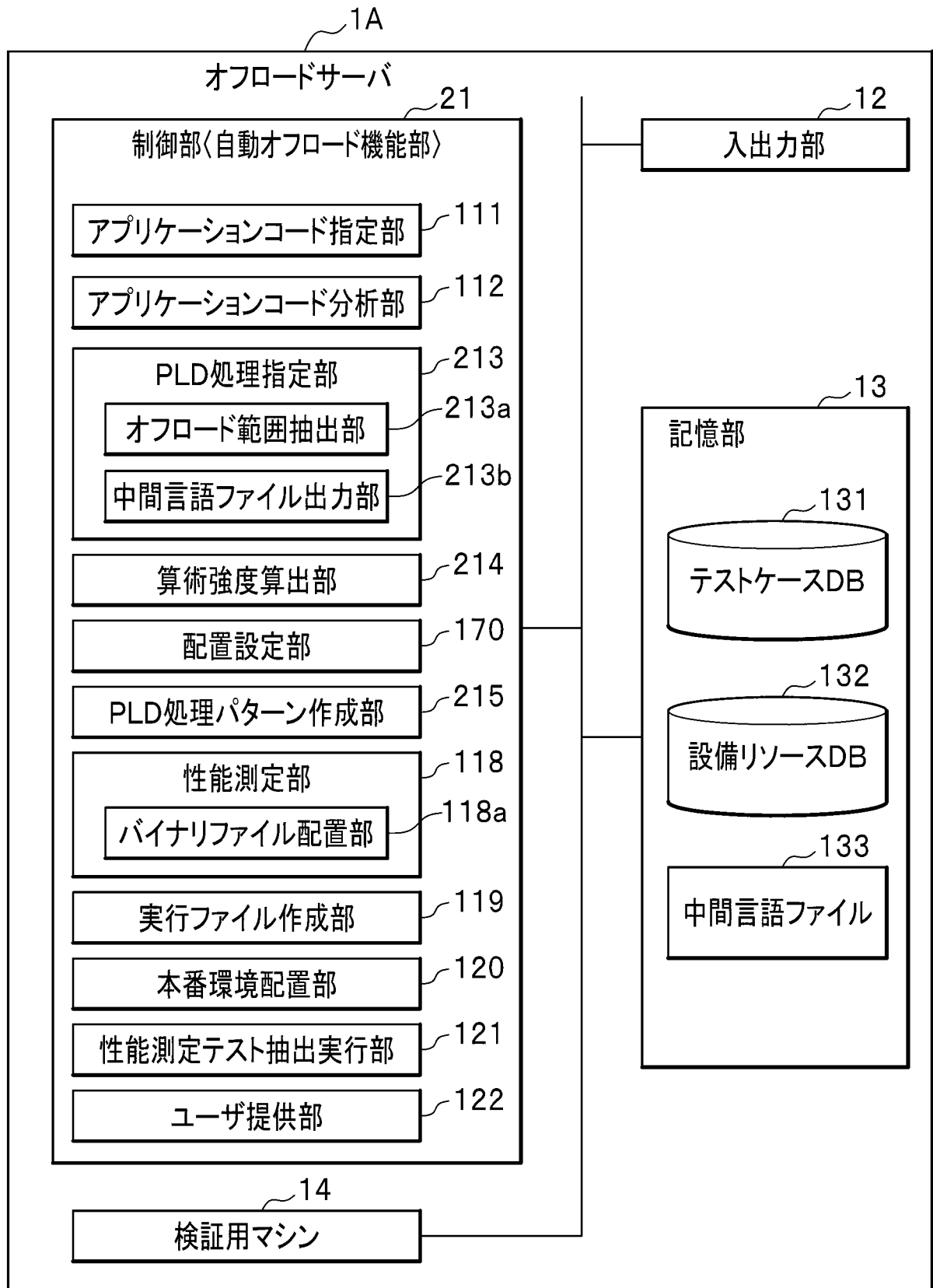
[図11]



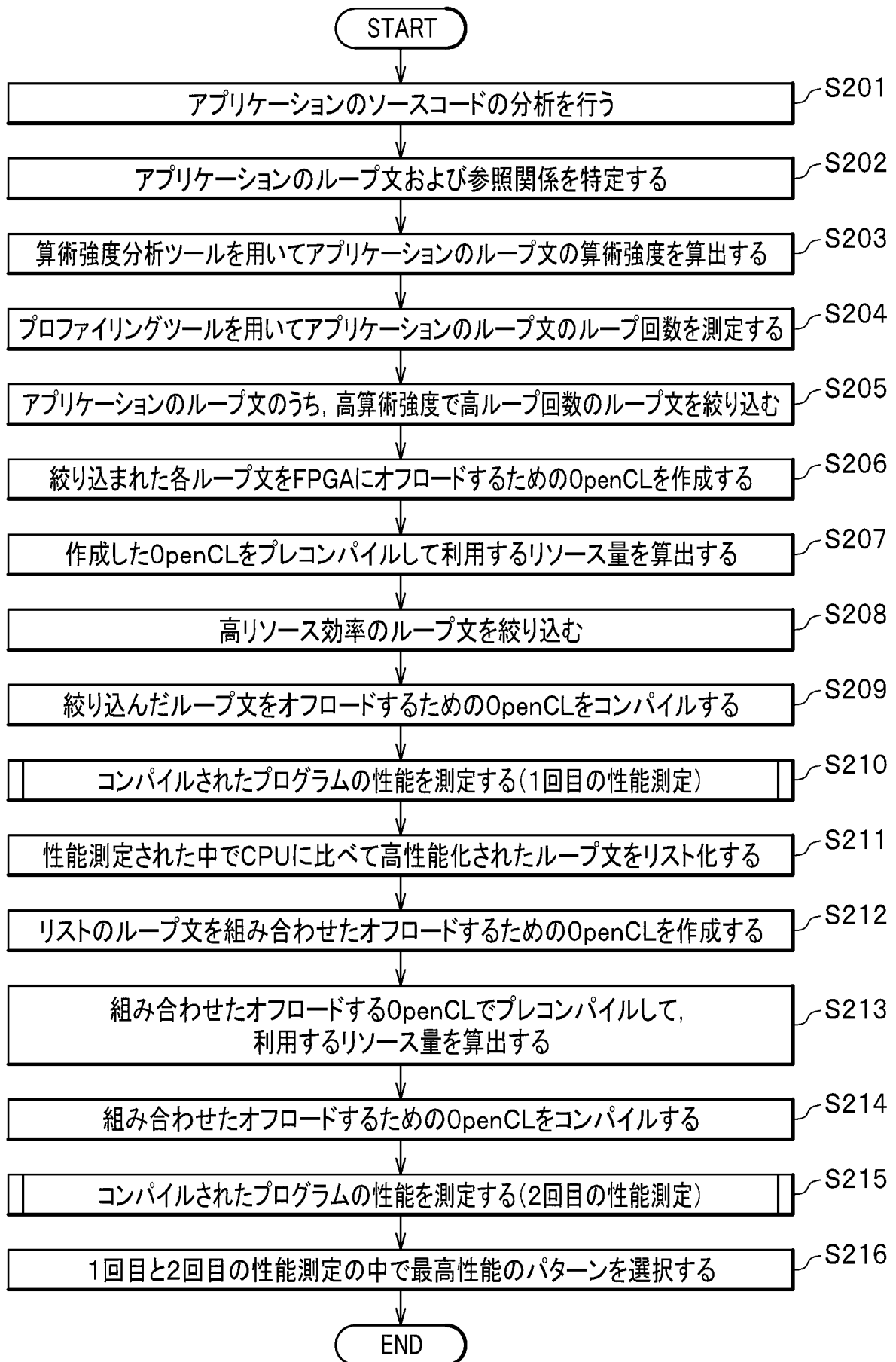
[図12]



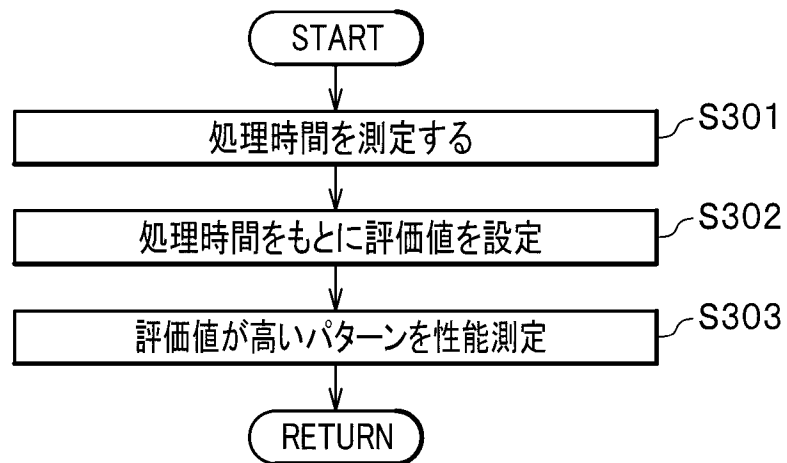
[図13]



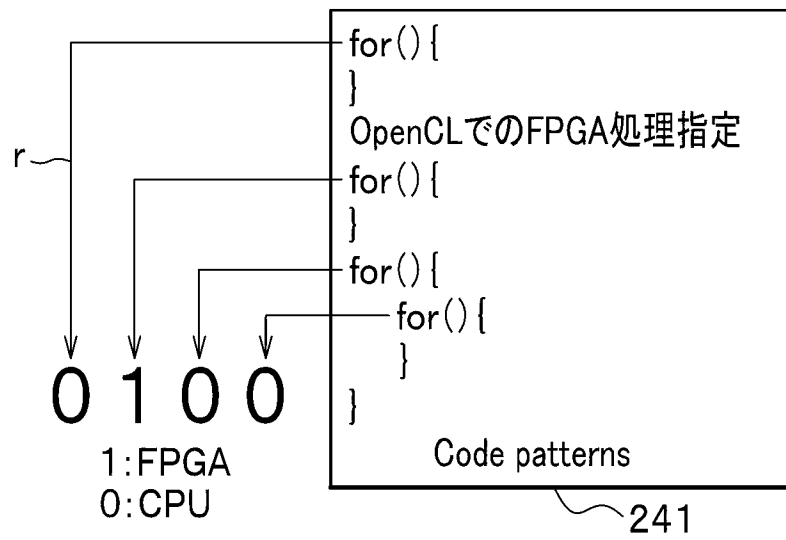
[図14]



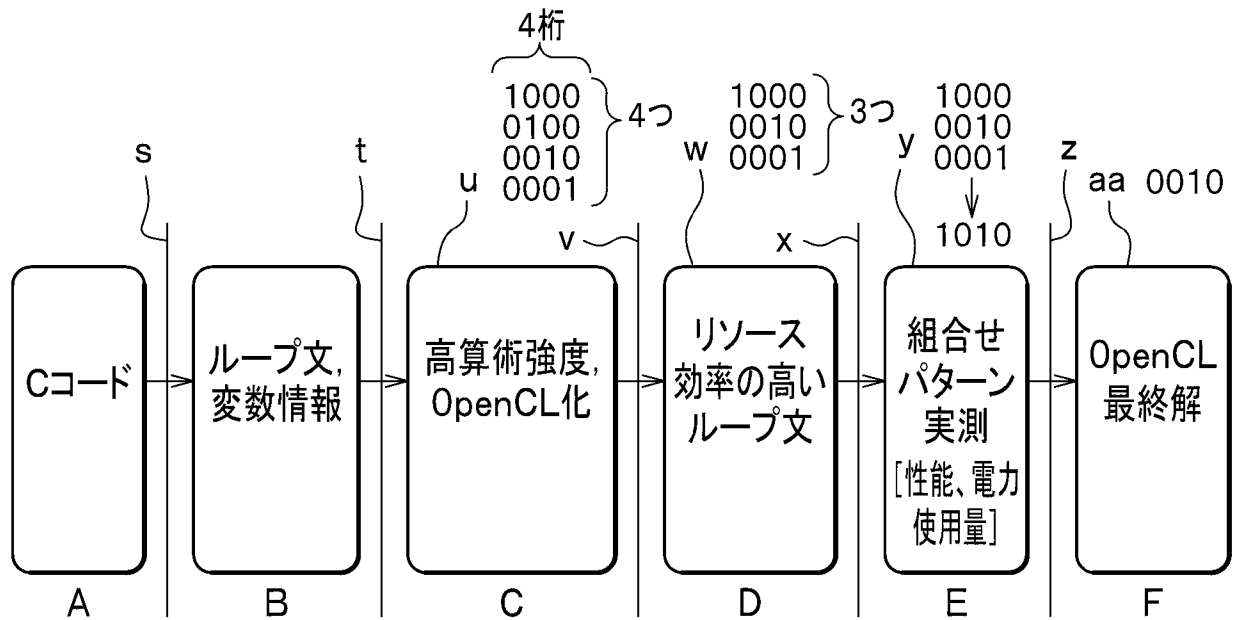
[図15]



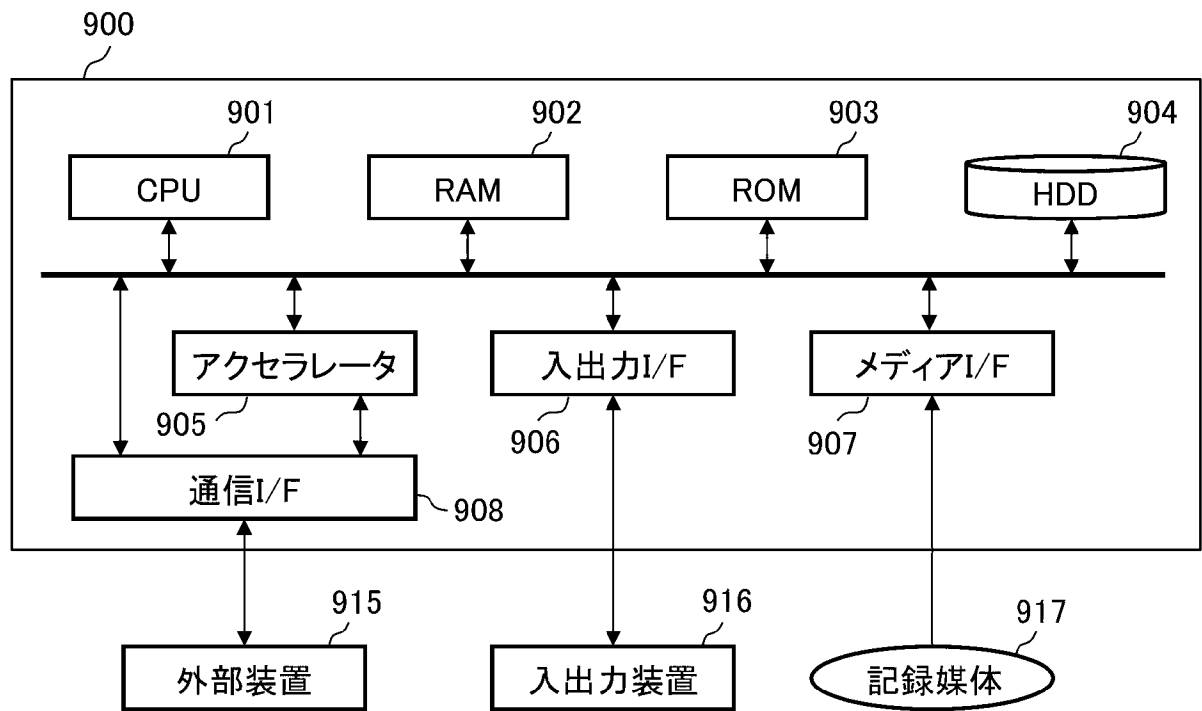
[図16]



[図17]



[図18]



INTERNATIONAL SEARCH REPORT

International application No.

PCT/JP2022/002880

A. CLASSIFICATION OF SUBJECT MATTER**G06F 11/34**(2006.01)i; **G06F 8/41**(2018.01)i; **G06F 8/60**(2018.01)i; **G06F 9/50**(2006.01)i

FI: G06F8/60; G06F8/41 130; G06F9/50 150A; G06F11/34 142

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

G06F11/34; G06F8/00-9/54

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Published examined utility model applications of Japan 1922-1996
 Published unexamined utility model applications of Japan 1971-2022
 Registered utility model specifications of Japan 1996-2022
 Published registered utility model applications of Japan 1994-2022

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	山登庸次, 環境適応アプリケーション配置適正化の初期検討, engrxiv.org [online], 2021, [retrieved on 31 March 2022], Internet:<URL:https://engrxiv.org/preprint/view/1770>, <DOI: https://doi.org/10.31224/osf.io/vdn8z>, non-official translation (YAMATO, Yoji, Initial study of appropriate placement of environment-adaptive applications)	1-3, 6-8
A	chapters 1-3	4-5
Y	WO 2020/175411 A1 (NIPPON TELEGRAPH AND TELEPHONE CORPORATION) 03 September 2020 (2020-09-03)	1-3, 6-8
A	paragraphs [0029]-[0051], fig. 1	4-5
Y	WO 2021/156955 A1 (NIPPON TELEGRAPH AND TELEPHONE CORPORATION) 12 August 2021 (2021-08-12)	1, 3, 6, 8
	claim 1	
Y	WO 2021/156956 A1 (NIPPON TELEGRAPH AND TELEPHONE CORPORATION) 12 August 2021 (2021-08-12)	2-3, 7-8
	claim 1	

☒ Further documents are listed in the continuation of Box C.☒ See patent family annex.

* Special categories of cited documents:

“A” document defining the general state of the art which is not considered to be of particular relevance

“E” earlier application or patent but published on or after the international filing date

“L” document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

“O” document referring to an oral disclosure, use, exhibition or other means

“P” document published prior to the international filing date but later than the priority date claimed

“T” later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

“X” document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

“Y” document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

“&” document member of the same patent family

Date of the actual completion of the international search

31 March 2022

Date of mailing of the international search report

12 April 2022

Name and mailing address of the ISA/JP

Japan Patent Office (ISA/JP)
 3-4-3 Kasumigaseki, Chiyoda-ku, Tokyo 100-8915
 Japan

Authorized officer

Telephone No.

INTERNATIONAL SEARCH REPORT

International application No.

PCT/JP2022/002880**C. DOCUMENTS CONSIDERED TO BE RELEVANT**

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	CN 111405569 A (CHINA THREE GORGES UNIVERSITY) 10 July 2020 (2020-07-10) entire text, all drawings	1-8

INTERNATIONAL SEARCH REPORT
Information on patent family members

International application No.

PCT/JP2022/002880

Patent document cited in search report			Publication date (day/month/year)	Patent family member(s)	Publication date (day/month/year)
WO	2020/175411	A1	03 September 2020	US 2022/0058009 A1 paragraphs [0031]-[0049], fig. 1	
WO	2021/156955	A1	12 August 2021	(Family: none)	
WO	2021/156956	A1	12 August 2021	(Family: none)	
CN	111405569	A	10 July 2020	(Family: none)	

A. 発明の属する分野の分類（国際特許分類（IPC）） G06F 11/34(2006.01)i; G06F 8/41(2018.01)i; G06F 8/60(2018.01)i; G06F 9/50(2006.01)i FI: G06F8/60; G06F8/41 130; G06F9/50 150A; G06F11/34 142														
B. 調査を行った分野														
調査を行った最小限資料（国際特許分類（IPC）） G06F11/34; G06F8/00-9/54														
最小限資料以外の資料で調査を行った分野に含まれるもの <table border="0"> <tr> <td>日本国実用新案公報</td> <td>1922-1996年</td> </tr> <tr> <td>日本国公開実用新案公報</td> <td>1971-2022年</td> </tr> <tr> <td>日本国実用新案登録公報</td> <td>1996-2022年</td> </tr> <tr> <td>日本国登録実用新案公報</td> <td>1994-2022年</td> </tr> </table>			日本国実用新案公報	1922-1996年	日本国公開実用新案公報	1971-2022年	日本国実用新案登録公報	1996-2022年	日本国登録実用新案公報	1994-2022年				
日本国実用新案公報	1922-1996年													
日本国公開実用新案公報	1971-2022年													
日本国実用新案登録公報	1996-2022年													
日本国登録実用新案公報	1994-2022年													
国際調査で使用した電子データベース（データベースの名称、調査に使用した用語）														
C. 関連すると認められる文献														
引用文献の カテゴリー*	引用文献名 及び一部の箇所が関連するときは、その関連する箇所の表示	関連する 請求項の番号												
Y	山登庸次, 環境適応アプリケーション配置適正化の初期検討, engrxiv.org [online], 2021, [検索日 2022.03.31], インターネット:<URL: https://engrxiv.org/preprint/view/1770>, <DOI: https://doi.org/10.31224/osf.io/vdn8z> 第1章-第3章	1-3, 6-8												
A		4-5												
Y	WO 2020/175411 A1 (日本電信電話株式会社) 03.09.2020 (2020-09-03) 段落[0029]-[0051], 図1	1-3, 6-8												
A		4-5												
Y	WO 2021/156955 A1 (日本電信電話株式会社) 12.08.2021 (2021-08-12) [請求項1]	1, 3, 6, 8												
Y	WO 2021/156956 A1 (日本電信電話株式会社) 12.08.2021 (2021-08-12) [請求項1]	2-3, 7-8												
A	CN 111405569 A (CHINA THREE GORGES UNIVERSITY) 10.07.2020 (2020-07-10) 全文, 全図	1-8												
☐ C欄の続きにも文献が列挙されている。 ☒ パテントファミリーに関する別紙を参照。														
<table border="0"> <tr> <td>* 引用文献のカテゴリー</td> <td>"T" 国際出願日又は優先日後に公表された文献であって出願と抵触するものではなく、発明の原理又は理論の理解のために引用するもの</td> </tr> <tr> <td>"A" 特に関連のある文献ではなく、一般的な技術水準を示すもの</td> <td>"X" 特に関連のある文献であって、当該文献のみで発明の新規性又は進歩性がないと考えられるもの</td> </tr> <tr> <td>"E" 国際出願日前の出願または特許であるが、国際出願日以後に公表されたもの</td> <td>"Y" 特に関連のある文献であって、当該文献と他の1以上の文献との、当業者にとって自明である組合せによって進歩性がないと考えられるもの</td> </tr> <tr> <td>"L" 優先権主張に疑義を提起する文献又は他の文献の発行日若しくは他の特別な理由を確立するために引用する文献（理由を付す）</td> <td>"&" 同一パテントファミリー文献</td> </tr> <tr> <td>"O" 口頭による開示、使用、展示等に言及する文献</td> <td></td> </tr> <tr> <td>"P" 国際出願日前で、かつ優先権の主張の基礎となる出願の日の後に公表された文献</td> <td></td> </tr> </table>			* 引用文献のカテゴリー	"T" 国際出願日又は優先日後に公表された文献であって出願と抵触するものではなく、発明の原理又は理論の理解のために引用するもの	"A" 特に関連のある文献ではなく、一般的な技術水準を示すもの	"X" 特に関連のある文献であって、当該文献のみで発明の新規性又は進歩性がないと考えられるもの	"E" 国際出願日前の出願または特許であるが、国際出願日以後に公表されたもの	"Y" 特に関連のある文献であって、当該文献と他の1以上の文献との、当業者にとって自明である組合せによって進歩性がないと考えられるもの	"L" 優先権主張に疑義を提起する文献又は他の文献の発行日若しくは他の特別な理由を確立するために引用する文献（理由を付す）	"&" 同一パテントファミリー文献	"O" 口頭による開示、使用、展示等に言及する文献		"P" 国際出願日前で、かつ優先権の主張の基礎となる出願の日の後に公表された文献	
* 引用文献のカテゴリー	"T" 国際出願日又は優先日後に公表された文献であって出願と抵触するものではなく、発明の原理又は理論の理解のために引用するもの													
"A" 特に関連のある文献ではなく、一般的な技術水準を示すもの	"X" 特に関連のある文献であって、当該文献のみで発明の新規性又は進歩性がないと考えられるもの													
"E" 国際出願日前の出願または特許であるが、国際出願日以後に公表されたもの	"Y" 特に関連のある文献であって、当該文献と他の1以上の文献との、当業者にとって自明である組合せによって進歩性がないと考えられるもの													
"L" 優先権主張に疑義を提起する文献又は他の文献の発行日若しくは他の特別な理由を確立するために引用する文献（理由を付す）	"&" 同一パテントファミリー文献													
"O" 口頭による開示、使用、展示等に言及する文献														
"P" 国際出願日前で、かつ優先権の主張の基礎となる出願の日の後に公表された文献														
国際調査を完了した日 31.03.2022	国際調査報告の発送日 12.04.2022													
名称及びあて先 日本国特許庁(ISA/JP) 〒100-8915 日本国 東京都千代田区霞が関三丁目4番3号	権限のある職員（特許庁審査官） 多賀 実 5B 9367 電話番号 03-3581-1101 内線 3545													

国際調査報告
パテントファミリーに関する情報

国際出願番号

PCT/JP2022/002880

引用文献			公表日	パテントファミリー文献	公表日
WO	2020/175411	A1	03.09.2020	US 2022/0058009 A1 段落[0031]-[0049], 図1	
WO	2021/156955	A1	12.08.2021	(ファミリーなし)	
WO	2021/156956	A1	12.08.2021	(ファミリーなし)	
CN	111405569	A	10.07.2020	(ファミリーなし)	