一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
IN2022-91(2023-03)

環境適応処理における運用開始後GPU構成変更の提案

山登庸次†

† NTT ネットワークサービスシステム研究所，東京都武蔵野市緑町 3-9-11
E-mail: †yoji.yamato.wa@hco.ntt.co.jp

あらまし　近年，ヘテロジニアスハードウェア利用が増えているが，これらの十分な活用には，CUDA 等のハードウェアを意識した技術理解が必要であり，壁は高いのが現状である．これらの背景から，私は，エンジニアが通常 CPU 向けに記述したアプリケーションコードを，配置される環境に応じて，自動で変換やリソース量設定等をして，高性能に運用可能とする環境適応ソフトウェアのコンセプトを提案してきた．本稿では，今まで運用開始前の変換や設定しか検討していなかったが，運用中の利用特性に応じて，すでに動作している GPU ロジックを再構成することを検証する．運用中 GPU 再構成方式を提案し，GPU に自動オフロードしたアプリケーションを，運用中再構成により別ループ文や別アプリケーションにロジックを再構成する処理を通じて，性能改善と断時間を確認し，有効性を示す．
キーワード　環境適応ソフトウェア, 自動オフロード，GPGPU，運用中再構成，コストパフォーマンス

Proposal of GPU configuration change after service launch for environment-adaptive software

Yoji YAMATO†

† Network Service Systems Laboratories, NTT Corporation, 3-9-11, Midori-cho, Musashino-shi, Tokyo
E-mail: †yoji.yamato.wa@hco.ntt.co.jp

Abstract　In order to make full use of heterogeneous hardware, it is necessary to have a technical skill of hardware such as CUDA, and the current situation is that the barrier is high. Based on this background, I have proposed environment-adaptive software that enables high-performance operation by automatically converting application code written for normal CPUs by engineers according to the deployed environment and setting appropriate amount of resources. Until now, I only considered conversions and settings before operation. In this paper, I verify that the logic is reconfigured according to the usage characteristics during operation. I confirm that the application running on the GPU is reconfigured into other loops or applications offloading according to the usage trends.
Key words　Environment Adaptive Software, Automatic Offloading, GPGPU, Reconfiguration during Operation, Cost Performance

1. はじめに

近年，CPU の半導体集積度が 1.5 年で 2 倍になるというムーアの法則が減速するのではないかと言われている．そのような状況から，CPU だけでなく，FPGA（Field Programmable Gate Array）や GPU（Graphics Processing Unit）等のデバイスの活用が増えている．例えば，Microsoft 社は FPGA を使って Bing の検索効率を高めるといった取り組みをしており [1]，Amazon 社は，FPGA, GPU 等をクラウド（例えば, [2]- [7]）のインスタンスとして提供している [8]．また，ヘテロジニアスなハードウェアとして，IoT 機器（例えば, [9]- [18]）等の小型デバイスの利用も増えている．

しかし，少コアの CPU 以外のデバイスをシステムで適切に活用するためには，デバイス特性を意識した設定やプログラム作成が必要であり，OpenMP（Open Multi-Processing）[19]，OpenCL（Open Computing Language）[20]，CUDA（Compute Unified Device Architecture）[21] といった知識が必要になってくるため，大半のプログラマーにとっては，スキルの壁が高い．

少コアの CPU 以外の GPU や FPGA，マルチコア CPU 等のデバイスを活用するシステムは今後ますます増えていくと予想されるが，それらを最大限活用するには，技術的壁が高い．そこで，そのような壁を取り払い，少コアの CPU 以外のデバイスを十分利用できるようにするため，プログラマーが処理

ロジックを記述したソフトウェアを，配置先の環境（FPGA，GPU，メニーコア CPU 等）にあわせて，適応的に変換，設定し，環境に適合した動作をさせるような，プラットフォームが求められている．

私は，一度記述したコードを，配置先の環境に存在する GPU や FPGA，マルチコア 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に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た．合わせて，環境適応ソフトウェアの要素として，C 言語プログラムコードのループ文及び機能ブロックを，GPU，FPGA に自動オフロードする方式を提案評価している [22]- [27]．

しかし，私の検証はこれまで，運用開始前に変換等の環境処理を行うことだけを行っており，運用開始後に商用環境での実際の利用特性に応じて再適応することは検討していなかった．例えば，運用開始前は，画像処理の中でクラス分類が多い処理が多い前提で GPU 処理を行うようにロジックを組んでいたが，運用開始 3 か月後の実際のリクエスト数では，画像処理の中でオブジェクト検知の処理が多くなっていたため，GPU 処理を行うロジックを変更した方が良い場合等である．

本稿は，運用開始後に実際の利用特性に応じて再構成する中で，特に GPU ロジックの再構成を対象とする．GPU をアプリケーションサーバのアクセラレートに利用する運用で，運用中に，実際の利用特性に応じて GPU ロジックを再構成している例は，市中クラウド（AWS の GPU インスタンス等）の利用で聞いたことがなく，インパクトが大きいためである．まず，既存アプリケーションを GPU に自動オフロードして運用開始し，利用特性を分析して，別のオフロード形態に GPU ロジックを再構成することを提案し，断時間短く変更する方式を提案する．提案方式の有効性を，運用中 GPU 再構成を通じて，性能改善度と断時間で評価する．

2. 既存技術

2.1 市中技術

GPU の並列計算パワーを画像処理でないものにも使う GPGPU（General Purpose GPU）（例えば [28]）を行うための環境として CUDA が普及している．CUDA は GPGPU 向けの NVIDIA 社の環境だが，FPGA，マルチコア CPU，GPU 等のヘテロジニアスなデバイスを同じように扱うための仕様として OpenCL が出ており，その開発環境 [29] も出ている．CUDA，OpenCL は，C 言語の拡張を行いプログラムを行う形だが，プログラムの難度は高い（FPGA 等のカーネルと CPU のホストとの間のメモリデータのコピーや解放の記述を明示的に行う等）

CUDA や OpenCL に比べて，より簡易にヘテロなデバイスを利用するため，ディレクティブ（指示行）ベースで，並列処理等を行う箇所を指定して，ディレクティブに従ってコンパイラが，GPU，メニーコア CPU 等に向けて実行ファイルを作成する技術がある．仕様としては，OpenACC [30] や OpenMP 等，コンパイラとして PGI コンパイラ [31] や gcc 等がある．

CUDA，OpenCL，OpenACC，OpenMP 等の技術仕様を用いることで，FPGA や GPU，マルチコア CPU へオフロードすることは可能になっている．しかしデバイス処理自体は行えるようになっても，高速化することには課題がある．例えば，マルチコア CPU 向けに自動並列化機能を持つコンパイラとして，Intel コンパイラ [32] 等がある．これらは，自動並列化時に，コードの中のループ文中で並列処理可能な部分を抽出して，並列化している．しかし，メモリ処理等の影響で単に並列化可能ループ文を並列化しても性能がでないことも多い．FPGA や GPU 等で高速化する際には，OpenCL や CUDA の技術者がチューニングを繰り返したり，OpenACC コンパイラ等を用いて適切な並列処理範囲を探索し試行することがされている．

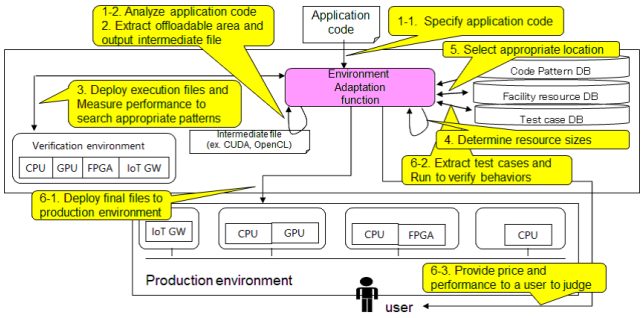


図 1　環境適応ソフトウェアのフロー

このため，技術スキルが乏しいプログラマーが，FPGA や GPU，マルチコア CPU を活用してアプリケーションを高速化することは難しいし，自動並列化技術等を使う場合も並列処理箇所探索の試行錯誤等の稼働が必要だった．並列処理箇所探索の試行錯誤を自動化する取り組みとして，著者は進化計算手法を用いた GPU 自動オフロードを提案している．

2.2 環境適応処理のフロー

ソフトウェアの環境適応を実現するため，著者は図 1 の処理フローを提案している．環境適応ソフトウェアは，環境適応機能を中心に，検証環境，商用環境，テストケース DB，コードパターン DB，設備リソース DB の機能群が連携することで動作する．

ここで，Step1-6 は，運用開始前に必要となる，コードの変換，リソース量の設定，配置場所の設定，動作確認を行うが，Step7 は，運用開始後に，実際の利用データを分析して，変更した方が良い際に再構成する．再構成する内容は，運用開始前の処理と同じ，コードの変換，リソース量の設定，配置場所の設定である．

課題を整理する．ヘテロジニアスなデバイスでのアクセラレートは手動が主流である．私は環境適応ソフトウェアを提案し，GPU や FPGA 自動オフロード方式を提案しているが，それらは運用開始前の話であり，運用開始後の再構成については考慮されていない．そこで，本稿は，運用中再構成に取り組み，特に GPU オフロードロジックの，実際の利用データに応じた再構成を対象とする．再構成する内容は，GPU オフロードロジック，リソース量，配置場所等，多々あるが，利用データに応じて GPU ロジックを再構成する例は，Amazon 等の市中ク

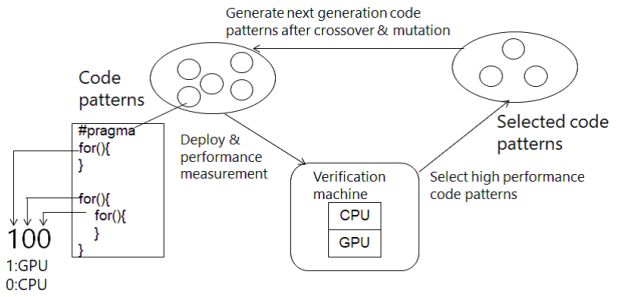


図 2　ループ文の GPU 自動オフロード方式

ラウドでも例がなく，難度，効果が大きいため，GPU ロジックを対象とする．

3. 運用中 GPU 再構成

3.1 運用開始前 GPU 自動オフロード方式の簡易紹介

著者の以前論文で検証している GPU 自動オフロード方式を簡易紹介する．自動化のために大きく 2 つ特徴がある．GPU に適したループ文抽出を遺伝的アルゴリズム（GA）[33] を用いて行う点を [22] で（図 2），GPU-CPU データ転送を抑えるため，ネストループ文で使われる変数をできるだけ上位ループで行う点を [27] で提案している．この 2 つのアイデアを元に，ループ文が 100 を超える中規模なアプリケーションでも数倍以上の自動高速化を確認している．

3.2 運用中 GPU 再構成に向けた基本方針

3.1 節の手法で，ユーザが指定したアプリケーションで，GPU に適したループ文部分を GPU に自動オフロードすることができる．ユーザが使う商用環境にオフロード後，商用環境での実際の性能と価格を確認し，ユーザはアプリケーションを利用開始する．ただし，3.1 節でのオフロードで用いる，性能最適化用テストケース（複数のオフロードパターンで性能比較する際に性能測定する項目）は，ユーザが指定した，運用開始前の想定利用データを利用しており，運用開始後に実利用されるデータから大きく離れる可能性がある．

そのため，3.2 節では，運用開始後の利用形態が，最初の想定と異なり，GPU には別ロジックをオフロードした方が性能が向上する等の場合に，GPU ロジックをユーザ影響低く再構成することを検討する．再構成は，同じアプリケーションでも異なるループ文オフロードに変える場合もあれば，異なるアプリケーションのオフロードに変える場合もある．

GPU ロジック再構成は，実行 OpenACC を変える必要があるが，やり方は複数ある．まず，実行マシンで，再構成前 OpenACC を停止し，再構成後 OpenACC を起動するやり方がある．OpenACC の停止，起動は短時間で，断時間も秒のオーダである．次に，再構成後 OpenACC を実行するマシン自体を新たに起動し，再構成前 OpenACC 処理が全て終わった等のタイミングで新しいリクエストは新マシンにルーティングを切替え，旧マシンはその後停止するやり方がある．ルーティング切替えは短時間で，断時間はほぼない．ユーザ影響度によって，GPU ロジック再構成のやり方は選択すればよいが，どちらでも若干は断時間が発生することや，別ロジックへの変更は動作確認試験が必要なことから，頻繁に再構成するべきとは考えておらず，効果が閾値以上の場合だけ提案する等制限を設ける．

検討する再構成は，1 か月等，一定期間のリクエスト傾向の分析から始まる．リクエスト傾向を分析し，現在オフロードしているアプリケーションより処理負荷が高いか同等のものがあるかを把握する．次に，処理負荷が高いリクエストを，想定利用データでなく実際に商用で利用されているデータを使って，GPU オフロードの最適化試行を検証環境で行う．ここで，検証により見つかった新しいオフロードパターンが現在のオフロードパターンより十分改善効果が高いかを閾値以上かそれ以下かで判定する．閾値を上回る場合は，ユーザに再構成を提案する．ユーザ了承後，商用環境を再構成するが，できるだけユーザ影響を抑えて再構成する．

3.3 運用中 GPU 再構成方式の提案

3.2 節の基本方針に基づいて，3.3 節では具体的な運用中再構成方式を提案する．再構成方式は，6 ステップからなり，各ステップを詳細説明する．

1. 一定期間（長期間）の，商用リクエストデータ履歴を分析し，処理時間負荷上位の複数アプリケーションを特定し，そのアプリケーション利用時の代表データを取得する．

1-1. 一定期間の各アプリケーション利用履歴から，実処理時間と利用回数合計を計算．ただし，GPU オフロードされているアプリケーションでは，オフロードされなかった場合の処理時間を仮に計算．運用開始前の想定利用データでの試験履歴から，(CPU 処理のみの際の実処理時間)/(GPU オフロードされた際の実処理時間) で改善度係数を求めておく．次に，実処理時間に改善度係数をかけた値の合計を比較に用いる処理時間合計とする．

1-2. 実処理時間合計を全アプリケーションで比較．

1-3. 実処理時間合計順に並べかえ，処理時間負荷上位の複数アプリケーションを特定．

1-4. 負荷上位アプリケーションの一定期間（短期間）のリクエストデータを取得し，データサイズを一定サイズごとに整列させ度数分布を作成．

1-5. データサイズ度数分布の最頻値 Mode に該当する実リクエストデータから，どれか一つデータを選び，代表データに選定．

2. 複数の負荷上位アプリケーションで，商用代表データのテストケースを高速化する，オフロードパターンを検証環境測定を通じて抽出．

2-1. 負荷上位アプリケーションで，Clang [34] 等での構文解析により for 文の数をカウントし，GPU 処理できない for 文を，コンパイラ機能で見つけて取り除く，

2-2. 残った for 文数を遺伝子長にして，一定数の遺伝子パターンを作成し，1 は GPU 実行，0 は非実行に対応させ，OpenACC ディレクティブを追加する．ディレクティブは#pragma acc kernels 等の GPU 計算および#pragma acc data copy 等のデータ処理の両方である．

2-3. 遺伝子パターンに対応する OpenACC ファイルを検証

環境機でコンパイルして，商用代表データのテストケースで性能測定し，高速なパターン程高い適応度を設定する．

2-4. 性能適応度に応じて，交叉等の処理を行い，次世代遺伝子パターンを作成する，遺伝的アルゴリズム処理を一定世代数繰返し行い，最終の最高性能パターンを解とする．

3. 現オフロードパターンと抽出した複数の新オフロードパターンの処理時間を測定し，商用利用頻度に基づく性能改善効果を求める．商用代表データでのテストケースで，

3-1. 現オフロードパターンで (検証環境実処理時間削減)*(商用環境利用頻度) 計算.

3-2. 複数の新オフロードパターンで (検証環境実処理時間削減)*(商用環境利用頻度) 計算.

4. 新オフロードパターンの性能改善度効果が現オフロードパターンのそれの閾値以上であるかで，再構成提案を判断.

4-1. 複数の (3-2) ÷ (3-1) を計算し，閾値以上かを確認し，以上なら再構成提案，以下なら何もしない.

5. 契約ユーザに GPU 再構成実施を提案し，OK/NG の返答を得る．

6. 商用環境で別 OpenACC を起動して再構成を実施．再構成実施時は，サーバの動作ファイルを変えても，別サーバに切り替えても良い.

6-1. 旧 OpenACC 停止し，新 OpenACC を起動.

6-1'. 新 OpenACC を処理するサーバを新たに構築し，新たな処理は新サーバに流すようルーティング変更.

4. 評　　価

4.1 評 価 条 件

4.1.1 評 価 対 象

評価対象は，多くのユーザが GPU で利用すると想定されるニューラルネットワークとフーリエ変換と流体計算を中心にする．

Darknet [35] は, C 言語のニューラルネットフレームワークであり，画像処理以外にも，classification, detection，nightmare 等様々な処理が出来るが，今回は，基本処理である，オブジェクト検知や画像加工の高速化を確認する．画像処理は，様々な種類があり GPU 向けライブラリも多いが，1 つの例として C 言語で利用可能な CPU 向けの Darknet で検証する．運用開始前オフロード検証では，Darknet に備え付けのサンプルアプリで detection 処理（画像検知）を利用する．

フーリエ変換処理は，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 [36] 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．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を考えた際に，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は想定される．運用開始前オフロード検証では NAS.FT に付属のサンプルテストケースを用いる．基本に使う Class A のグリッドサイズは 256*256*128 で，イテレーション数は 6 である．

姫野ベンチマーク [37] は，非圧縮流体解析の性能測定に用いられるベンチマークソフトで，ポアッソン方程式解法をヤコビ反復法で解いている．GPU での手動高速化に頻繁に利用されており，自動でも高速化できることの確認のため新たに利用する．データサイズは，Large(512*256*256) を基本に利用する．

この他は，ブロック対角ソルバ計算の NAS.BT [38]，3 次元 MRI 画像処理の MRI-Q [39] を，同じサーバ上で動かし，実行リクエストを受けるとする．

4.1.2 評 価 手 法

2 人のユーザを想定し，運用開始前に，ユーザ A は Darknet の detection 処理を指定し，ユーザ B は NAS.FT を指定し，自動オフロードする．商用環境には，それぞれ，一つづつオフロードし，残りの 4 アプリケーションは CPU のみの処理で動作させる．一定期間，商用環境サーバに，ユーザ毎に異なるリクエスト負荷をかけて，その結果を分析し，性能改善効果が高いオフロードパターンへの再構成を提案し，ユーザ承認後再構成されることを確認する．

GPU オフロード時の条件は以下で行う．

ループ文数：Darknet 171，NAS.FT 81，姫野 13，NAS.BT 120，MRI-Q 16

個体数 M：ループ文数以下（Darknet 30，NAS.FT 30，姫野 10，NAS.BT 30，MRI-Q 10）

世代数 T：ループ文数以下（Darknet 20，NAS.FT 20，姫野 10，NAS.BT 20，MRI-Q 10）

適合度：(処理時間)$^{-1/2}$　処理時間が短い程高適合度になる．また，(-1/2) 乗とすることで，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．

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 Pc：0.9

突然変異率 Pm：0.05

GPU 再構成に向けた，運用の条件は以下で行う．

リクエスト負荷：ユーザ A：Darknet detection 16 req/h, Darknet nightmare 20 req/h, NAS.FT 4 req/h, 姫野 3 req/h, NAS.BT 2 req/h，MRI-Q 1 req/h．Darknet 利用はサンプルデータの detection（画像検知）と nightmare（画像加工）処理とする．Darknet は，サンプルデータの 111 KB，170 KB，374 KB の写真を，3:5:2 の比でリクエストする．NAS.FT，姫野，NAS.BT，MRI-Q はサンプルデータのままとする．

ユーザ B：Darknet detection 4 req/h，Darknet nightmare 3 req/h, NAS.FT 20 req/h, 姫野 30 req/h, NAS.BT 2 req/h, MRI-Q 1 req/h．NAS.FT は，サンプルデータの Class W, A, B のサイズで，3:5:2 の比でリクエストする．姫野は，サンプルデータの M, L, XL のサイズで，2:5:3 の比でリクエストする．Darkent detection, nightmare，NAS.BT，MRI-Q はサンプルデータのままとする．

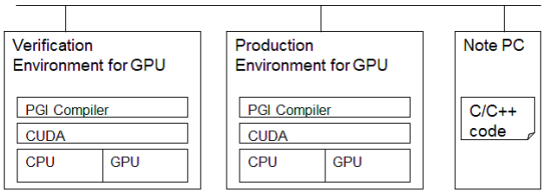
負荷分析時の長期間：1 時間

代表データ選定時の短期間：1 時間

負荷上位アプリケーションの数：2

性能改善効果閾値：2.0

再構成実施の中で，再構成に伴う，性能改善効果と各ステッ



Name	Hardware	CPU	RAM	GPU	OS	CUDA toolkit	PGI Compiler
Verification Environment for GPU	LEVEL-F039-LCRT2W-XYVI	AMD Ryzen Threadripper 2990WX	64GB	NVIDIA GeForce RTX 2080 Ti (CUDA core: 4352, Memory: GDDR6 11GB)	Ubuntu 16.04.6 LTS	10.1	19.10
Production Environment for GPU	LEVEL-F039-LCRT2W-XYVI	AMD Ryzen Threadripper 2990WX	64GB	NVIDIA GeForce RTX 2080 Ti (CUDA core: 4352, Memory: GDDR6 11GB)	Ubuntu 16.04.6 LTS	10.1	19.10
Client	HP ProBook 470 G3	Intel Core i5-6200U@2.3GHz	8GB		Windows 10 Pro		

図 3　性能測定環境

プの処理時間を取得する．

4.1.3　評 価 環 境

評価用 GPU として NVIDIA GeForce RTX 2080 Ti を用いる．GeForce RTX 2080 Ti 搭載サーバは，Iiyama LEVEL-F039-LCRT2W-XYVI である．GPU の制御は，PGI コンパイラ 19.10 と CUDA toolkit 10.1 を用いる．OpenACC の文法に従い，C 言語プログラムに，#pragma ディレクティブを追記することで，GPU オフロード処理がされ，別 OpenACC プログラムへの再構成も，PGI コンパイラで処理される．

評価環境とスペックを図 3 に示す．ここで，ノート PC が，オフロードするアプリケーションコードを指定し，検証環境での性能測定を通じてオフロードパターンを抽出後，商用環境にデプロイされる．商用環境アプリケーション群には，ノート PC から定期的にリクエストがされる．商用環境リクエスト履歴を分析して，検証環境を用いて新たなオフロードパターンを抽出し，ユーザ確認後，商用環境を新パターンに再構成する．

4.2　結　　果

図 4 は，ユーザ A と B の再構成前と再構成後のオフロードの処理時間改善度とそれに関連する一定期間の処理時間合計（改善度係数補正済み）を示している．

ユーザ A の再構成前は，Darknet の 4 個の for 文がオフロードされており，運用開始前想定データでの改善度は 2.92 であり，運用開始後は detection 処理に 16 req/h, nightmare 処理に 20 req/h 負荷がかかる．リクエストの実処理時間合計*2.92 の 2,980 秒が補正済処理時間合計である．計算した結果，Darknet と NAS.FT の 2 つが負荷上位アプリケーションとなる．この中で，運用開始後，Darknet は nightmare 処理が商用利用主体だったため，代表データを使ってオフロードパターン探索し，13 個の for 文をオフロードするパターンが見つかり，商用利用回数をかけることで，処理時間削減改善度は，再構成前 Darkenet で 96 秒，再構成後 Darknet で 1,850 秒となる．ユーザ B の再構成前は，NAS.FT がオフロードされており，運用開始前想定データでの改善度は 2.54 である．リクエストの実処理時間合計*2.54 の 1,210 秒が補正済処理時間合計である．計算した結果，姫野と NAS.FT の 2 つが負荷上位アプリケーションとなる．運用開始後の代表データを使ってオフロードパターンを探索し，商用利用回数をかけることで，処理時間削減改善度は，再構成前 NAS.FT で 308 秒，再構成後姫野で 1,180 秒となる．

User A	Offload application	Improvement of processing time	Summation of processing time
Before reconfiguratoin	Darknet (4 loops)	96.0 sec/h	2,980 sec
After reconfiguratoin	Darknet (13 loops)	1,850 sec/h	2,980 sec
User B	Offload application	Improvement of processing time	Summation of processing time
Before reconfiguratoin	NAS.FT	308 sec/h	1,210 sec
After reconfiguratoin	Himeno benchmark	1,180 sec/h	1,400 sec

図 4　提案方式の再構成を通じた処理時間改善比較

図 4 から，改善度閾値 2.0 をチェックし，ユーザ A の場合は，Darknet のオフロードループ文を変更した再構成が提案され，ユーザ B の場合は，NAS.FT から姫野のオフロードアプリケーションを変更した再構成が提案される．

リクエスト分析は今回 1 時間分のデータしか行っていないためサイズは小さいが，サイズに比例して長時間化すると考える．今回，リクエスト分析と代表データ選定に 1 秒程度，改善効果計算に一日弱，再構成実施に 1 秒未満がかかっている．運用開始前のオフロード試行や，運用中の新オフロードパターン試行については，数十世代の GA の性能測定に 7 時間程度かかることから，負荷上位アプリケーションが 3 つの場合は 1 日弱かかる．しかし，新オフロードパターン試行を含め，分析等殆どの処理は，商用環境のアプリケーション運用中のバックグラウンドで行われるため，ユーザ影響はない．唯一，商用での再構成実施については，アプリケーションの断時間が発生するが，OpenACC の停止，起動による再構成でも 1 秒未満であり，殆ど影響がないことを確認できた．もし，より，短い断時間が必要な際は，新マシンに新 OpenACC を準備してルーティングを切り替える等も考えられる．

運用中 GPU 処理アプリケーションを，ユーザ毎の利用特性に応じて，異なるループ文オフロードに再構成したり，異なるアプリケーションオフロードに再構成したりを確認した．再構成を通じ，性能改善度が閾値以上に上がり，断時間が十分短くできる事を示した．

5.　ま　と　め

本稿では，アプリケーション運用開始後に，利用特性に応じて，適切な GPU ロジックに運用中に再構成する，運用中 GPU 再構成方式を提案した．

運用開始前に，アプリケーションループ文を GPU に自動オフロードさせておく．提案方式は，一定期間毎に，実リクエストデータから，CPU 処理時間の負荷が上位のアプリケーションを取得し，該当する代表テストケースを把握する．次に，複数の代表テストケースを高速化するオフロードパターンを検証環境での試行測定を通じ抽出する．これは，運用開始前オフロードとほぼ同様である．次に，現在の高速化パターンと抽出した新しい高速化パターンの処理時間を測定し，商用利用頻度に基

づく処理時間改善を求める．ここで，新しい高速化パターンが現在パターンの閾値以上効果の場合，再構成実施をユーザに提案する．ユーザ了承が得られたら，商用環境で OpenACC の停止，新 OpenACC 起動による再構成を行う．GPU に自動オフロードしたアプリケーションを，利用特性に応じた運用中再構成により，別ループ文のオフロードや別アプリケーションのオフロードに GPU ロジックを再構成した．再構成により処理時間削減を改善し，併せて，1 秒未満の短い断時間で再構成を行い，方式有効性を確認した．

文　献

[1] A. Putnam, et al., "A reconfigurable fabric for accelerating large-scale datacenter services," Proceedings of the 41th Annual International Symposium on Computer Architecture (ISCA'14), pp.13-24, June 2014.

[2] O. Sefraoui, et al., "OpenStack: toward an open-source solution for cloud computing," International Journal of Computer Applications, Vol.55, No.3, 2012.

[3] Y. Yamato, "Automatic system test technology of virtual machine software patch on IaaS cloud," IEEJ Transactions on Electrical and Electronic Engineering, Vol.10, Issue.S1, pp.165-167, Oct. 2015.

[4] Y. Yamato, et al., "Fast Restoration Method of Virtual Resources on OpenStack," IEEE Consumer Communications and Networking Conference (CCNC2015), Las Vegas, pp.607-608, Jan. 2015.

[5] Y. Yamato, "Automatic verification for plural virtual machines patches," The 7th International Conference on Ubiquitous and Future Networks (ICUFN 2015), pp.837-838, Sapporo, July 2015.

[6] Y. Yamato, "Proposal of Optimum Application Deployment Technology for Heterogeneous IaaS Cloud," 2016 6th International Workshop on Computer Science and Engineering (WCSE 2016), pp.34-37, June 2016.

[7] Y. Yamato, et al., "Fast and Reliable Restoration Method of Virtual Resources on OpenStack," IEEE Transactions on Cloud Computing, DOI: 10.1109/TCC.2015.2481392, Sep. 2015.

[8] AWS EC2 web site, https://aws.amazon.com/ec2/instance-types/

[9] M. Hermann, et al., "Design Principles for Industrie 4.0 Scenarios," Rechnische Universitat Dortmund. 2015.

[10] Y. Yamato and M. Takemoto, "Method of Service Template Generation on a Service Coordination Framework," 2nd International Symposium on Ubiquitous Computing Systems (UCS 2004), Nov. 2004.

[11] Y. Yamato, et al., "Proposal of Real Time Predictive Maintenance Platform with 3D Printer for Business Vehicles," International Journal of Information and Electronics Engineering, Vol.6, No.5, pp.289-293, Sep. 2016.

[12] H. Noguchi, et al., "Distributed Search Architecture for Object Tracking in the Internet of Things," IEEE Access, DOI: 10.1109/ACCESS.2018.2875734, Oct. 2018.

[13] Y. Yamato, et al., "Security Camera Movie and ERP Data Matching System to Prevent Theft," IEEE Consumer Communications and Networking Conference (CCNC 2017), pp.1021-1022, Jan. 2017.

[14] Y. Yamato, et al., "Analyzing Machine Noise for Real Time Maintenance," 2016 8th International Conference on Graphic and Image Processing (ICGIP 2016), Oct. 2016.

[15] H. Noguchi, et al., "Device Identification Based on Communication Analysis for the Internet of Things," IEEE Access, DOI: 10.1109/ACCESS.2019.2910848, pp.52903-52912, Apr. 2019.

[16] H. Noguchi, et al., "Autonomous Device Identification Architecture for Internet of Things," 2018 IEEE 4th World Forum on Internet of Things (WF-IoT 2018), pp.407-411, Feb. 2018.

[17] Y. Yamato, "Experiments of posture estimation on vehicles using wearable acceleration sensors," The 3rd IEEE International Conference on Big Data Security on Cloud (BigDataSecurity 2017), pp.14-17, May 2017.

[18] P. C. Evans and M. Annunziata, "Industrial Internet: Pushing the Boundaries of Minds and Machines," Technical report of General Electric (GE), Nov. 2012.

[19] T. Sterling, et al., "High performance computing : modern systems and practices," Cambridge, MA : Morgan Kaufmann, ISBN 9780124202153, 2018.

[20] J. E. Stone, et al., "OpenCL: A parallel programming standard for heterogeneous computing systems," Computing in science & engineering, Vol.12, No.3, pp.66-73, 2010.

[21] J. Sanders and E. Kandrot, "CUDA by example : an introduction to general-purpose GPU programming," Addison-Wesley, 2011.

[22] Y. Yamato, et al., "Automatic GPU Offloading Technology for Open IoT Environment," IEEE Internet of Things Journal, DOI: 10.1109/JIOT.2018.2872545, Sep. 2018.

[23] Y. Yamato, "Study and Evaluation of Automatic GPU Offloading Method from Various Language Applications," International Journal of Parallel, Emergent and Distributed Systems, Taylor and Francis, DOI: 10.1080/17445760.2021.1971666, Sep. 2021.

[24] Y. Yamato, "Study and Evaluation of Improved Automatic GPU Offloading Method," International Journal of Parallel, Emergent and Distributed Systems, Taylor and Francis, DOI: 10.1080/17445760.2021.1941010, June 2021.

[25] Y. Yamato, "Automatic Offloading Method of Loop Statements of Software to FPGA," International Journal of Parallel, Emergent and Distributed Systems, Taylor and Francis, DOI: 10.1080/17445760.2021.1916020, Apr. 2021.

[26] Y. Yamato, "Proposal of Automatic Offloading for Function Blocks of Applications," The 8th IIAE International Conference on Industrial Application Engineering 2020 (ICIAE 2020), pp.4-11, Mar. 2020.

[27] 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, 2019.

[28] J. Fung and M. Steve, "Computer vision signal processing on graphics processing units," 2004 IEEE International Conference on Acoustics, Speech, and Signal Processing, Vol. 5, pp.93-96, 2004.

[29] Xilinx SDK web site, https://japan.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/lyx1504034296578.html

[30] S. Wienke, et al., "OpenACC-first experiences with real-world applications," Euro-Par 2012 Parallel Processing, pp.859-870, 2012.

[31] M. Wolfe, "Implementing the PGI accelerator model," ACM the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, pp.43-50, Mar. 2010.

[32] E. Su, et al., "Compiler support of the workqueuing execution model for Intel SMP architectures," In Fourth European Workshop on OpenMP, Sep. 2002.

[33] J. H. Holland, "Genetic algorithms," Scientific american, Vol.267, No.1, pp.66-73, 1992.

[34] Clang website, http://llvm.org/

[35] Darknet website, https://pjreddie.com/darknet/

[36] NAS.FT website, https://www.nas.nasa.gov/publications/npb.html

[37] Himeno benchmark web site, http://accc.riken.jp/en/supercom/

[38] NAS.BT website, https://www.nas.nasa.gov/publications/npb.html

[39] MRI-Q website, http://impact.crhc.illinois.edu/parboil/