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

信学技報
TECHNICAL REPORT OF IEICE.
IN2020-30 (2020-10)

移行先混在環境における自動オフロード方式の検討

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，少コアの CPU だけでなく，GPU，FPGA，メニーコア CPU 等のヘテロなデバイスを利用したシステムが増えている．しかし，これらの利用には，OpenMP，CUDA や OpenCL 等のハードウェアを意識した技術仕様の理解が必要であり，ハードルは高い．これらの背景から，私は，プログラマーが CPU 向けに開発したソースコードを，適用される環境に応じて，自動で変換し，リソース量等を設定して，高い性能で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いる．しかし，従来市中技術を含め，移行先環境が GPU，FPGA のように混在環境で適切に自動オフロードする研究は無かった．本稿では，環境適応ソフトウェアの新たな要素として，移行先が GPU，FPGA，メニーコア CPU が混在する環境で，アプリケーションをオフロードし高速化するための方式を検討する．提案手法を市中のアプリケーションで評価する．
キーワード　環境適応ソフトウェア, GPGPU，FPGA，自動オフロード，進化的計算，移行先混在環境

Study of Automatic Offloading Method in Mixed Offloading Destination Environment

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 recent years, utilization of heterogeneous hardware other than small core CPU such as GPU, FPGA or many core CPU is increasing. However, when using heterogeneous hardware, barriers of technical skills such as OpenMP, CUDA and OpenCL are high. Based on that, I have proposed environment-adaptive software that enables automatic conversion, configuration, and high performance operation of once written code, according to the hardware to be placed. However, including existing technologies, there has been no research to properly and automatically offload the mixed offloading destination environment such as GPU, FPGA and many core CPU. In this paper, as a new element of environment-adaptive software, I study a method for offloading applications properly and automatically in the environment where the offloading destination is mixed with GPU, FPGA and many core CPU. I evaluate the effectiveness of the proposed method in multiple applications.
Key words　Environment Adaptive Software, GPGPU, FPGA, Automatic Offloading, Evolutionary Computation, Mixed Offloading Destination Environment.

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，少コアの 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 等をクラウドのインスタンス（例えば, [2]- [12]）として提供している [13].

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）[14]，OpenCL（Open Computing Language）[15]，CUDA（Compute Unified Device Architecture）[16] 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．また，システムでは IoT 機器利用（例えば, [17]- [21]）や外部サービス連携（例えば, [22]- [28]）等も増えており，複雑

度は増している．

少コアの CPU 以外の GPU や FPGA，メニーコア 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．

Java [29] は 1995 年に登場し，一度記述したコードを，別メーカーの CPU を備える機器でも動作可能にし，環境適応に関するパラダイムシフトをソフト開発現場に起こした．しかし，移行先での性能については，適切であるとは限らなかった．そこで，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，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 [30] [31]．合わせて，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及び機能ブロックを，FPGA，GPU に自動オフロードする方式を提案評価している．

本稿は，GPU，FPGA，メニーコア CPU と移行先環境が多様な混在環境の場合に，アプリケーションをより高速に自動オフロードすることを目的とする．まず，GPU，FPGA，メニーコア CPU の単体移行先に対して自動オフロードする手法を，提案する．次に，多様な混在環境となった際に場合に，高速化するための手法を提案する．提案手法を既存アプリケーションで有効性を評価する．

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 [32]）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，メニーコア CPU，GPU 等のヘテロ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 [33] [34] も出てきている．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，指示行ベースで，並列処理等を行う箇所を指定して，指示行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 [35] や OpenMP 等，コンパイラとして PGI コンパイラ [36] 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，メニーコア 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，

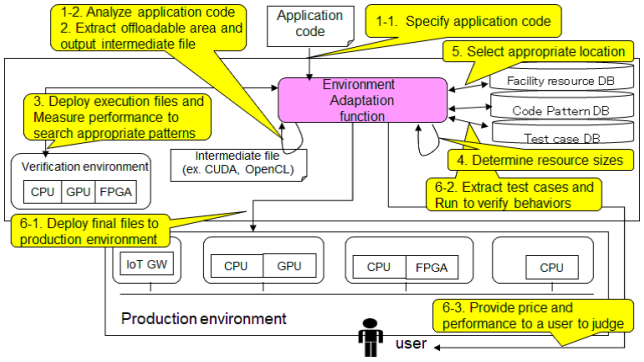


図 1　環境適応ソフトウェアのフロー

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 [37] 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．

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，著者は以前に図 1 の処理フローを提案している [31]．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，環境適応するために必要となる，コードの変換，リソース量の調整，配置場所の決定，自動検証（例えば [38] [39] 等を用いて），運用中の再構成を行うが，実施したい処理だけ切り出すこともできる．例えば，GPU，FPGA，メニーコア CPU 向けのコード変換だけ実施する場合は，Step 1-3 だけ処理すればよい．環境適応ソフトウェアの要素として，著者は並列処理箇所探索の試行錯誤を遺伝的アルゴリズム [40] を用いて自動化する GPU 自動オフロード等を提案している [30]．

本節を整理する．現状，ヘテロなデバイスに対するオフロードは手動での取組みが主流である．また，著者は環境適応ソフトウェアのコンセプトを提案し，自動オフロードを検討しているが，GPU 単体へのオフロードである等，GPU，FPGA，メニーコア CPU 等の多様なデバイスが混在している移行先環境は想定されていない．

3. 多様な移行先環境での自動オフロードの検討

3.1 多様移行先環境時の基本的考え

本稿で対象とする多様な移行先環境としては，GPU，FPGA，メニーコア CPU の 3 つとする．GPU，FPGA については CPU とは異なるヘテロジニアスなハードウェアとして歴史も深く，CUDA や OpenCL を用いた手動でのオフロードによる高速化事例も多く，市場も大きい．また，メニーコア CPU に関しては，近年 16 コア以上の多数のコアを搭載した CPU が千-数千ドルの低価格でも市場に出るようになり，OpenMP 等の技術仕様を用いて並列化を行い，手動チューニングすることで，高

速化する事例が出てきている．

移行先環境が単なる CPU だけでない場合で，自動で高速にオフロードするために，私は，検証環境の実機で性能測定し，進化計算手法等の手法と組み合わせて，徐々に高速なオフロードパターンを見つけるアプローチをとる．これは，今まで提案した GPU オフロード等の場合と同様である．理由として，性能に関しては，コード構造だけでなく，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，静的に予測する事が困難であり，動的な測定が必要だからと考えるからである．市中には，ループ文を見つけコンパイル段階で並列化する自動並列化コンパイラがあるが，並列化可能ループ文の並列化だけでは性能を測定すると低速になる場合も多いため，性能測定は必要と考える．

また，オフロードする対象については，私は，プログラムのループ文と機能ブロックとするアプローチをとる．これは，今まで検討した GPU，FPGA オフロード等の場合と同様である．ループ文については，処理時間がかかるプログラムの処理の大半はループで費やされているという現状から，ループ文がオフロードのターゲットとしてはまず考えられる．一方，機能ブロックについては，特定処理を高速化する際に，処理内容や処理ハードウェアに適したアルゴリズムを用いることが多いため，個々のループ文の並列処理等に比べ，大きく高速化できる場合がある．行列積算やフーリエ変換等の頻繁に使われる機能ブロック単位で，FPGA や GPU 等の処理デバイスに応じたアルゴリズムで実装された処理（IP コアや CUDA ライブラリ等）に置換することで高速化する．

3.2 個々の移行先環境への自動オフロード手法

3.1 の通り，GPU，FPGA，メニーコア CPU の 3 つの移行先環境に対して，ループ文，機能ブロックの 2 つの方法で，自動オフロードを検討する．

3.2.1 ループ文のメニーコア CPU 自動オフロード

ループ文のメニーコア CPU 自動オフロード手法については，今回新たに提案する．

メニーコア CPU も GPU と同様に，多数の計算コアを生かして，処理を並列化することで高速化を行う．GPU と異なる点は，メニーコア CPU の場合，メモリは共通であるため，GPU へのオフロードでしばしば問題となった CPU と GPU のメモリ間のデータ転送によるオーバヘッドは考慮する必要がない．また，メニーコア CPU でのプログラム処理の並列化には，OpenMP 仕様が頻繁に利用される．OpenMP は，#pragma omp parallel for 等の指示句でプログラムに対して，並列処理等を指定する仕様である．OpenMP での処理並列化は，OpenMP プログラマーが責任を持つこととなっており，並列化できない処理を並列化した場合には，コンパイラがエラー出力をするわけでなく，計算結果が誤って出力される．

これらを踏まえ，本稿では，ループ文のメニーコア CPU 向け自動オフロードは，ループの並列処理可否を，OpenMP の #pragma で指定するパターンを複数作成し，検証環境で実際の性能測定を繰り返すことで，徐々に高速化していく進化計算手法のアプローチをとる．ここで，GPU での自動オフロード

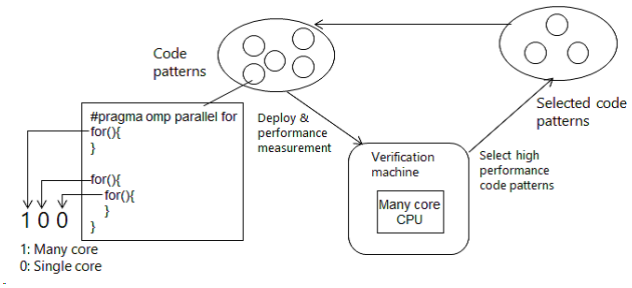


図 2　メニーコア CPU 向けの自動オフロード方式

で用いていた PGI コンパイラは，並列化不能時はコンパイラがエラーを出力していた．しかし，gcc 等の OpenMP コンパイラはそのようなエラーはプログラマの責任となる．そこで，自動化するために，OpenMP 指示句で行う処理は，ループ文をメニーコア CPU で並列処理するかどうかだけと単純化するとともに，並列処理した場合の最終計算結果が正しいかどうかのチェックも性能測定時に行うことで，正しい計算結果が出るパターンだけ進化計算の中で残る仕組みとする．

具体的には，コードが入力されたら Clang [41] 等で構文解析を行い，ループ文を判定する．ループ文に対して，#pragma omp parallel for の追加により，並列処理を指定した OpenMP コードを作成する．ここで，メニーコア CPU で並列処理の場合は 1，並列処理しない場合は 0 として，遺伝子パターンとする．準備した複数のパターンを gcc 等の OpenMP コンパイラでコンパイルし，メニーコア CPU を備えた検証環境マシンで性能測定をする．性能測定の結果高速処理のパターンを高い適応度に，低速処理のパターンを低い適応度に設定し，次世代のパターンを遺伝的アルゴリズムのエリート選択，交叉，突然変異等の処理により作成する．ここで，性能測定の際に，最終計算結果が並列処理しない場合と同じ結果であることを，オリジナルコードを通常 CPU で処理した場合と比較し，もし差分が許容できない程大きい場合はそのパターンの適応度は 0 として次世代には選ばれないようにする（図 2）．

3.2.2 ループ文の GPU 自動オフロード

ループ文の GPU 自動オフロード手法については，[31] [42] で提案を行っている．遺伝的アルゴリズムを用いた適切なループ文抽出と，CPU-GPU 転送の低減により，自動でのオフロードを可能としている．

3.2.3 ループ文の FPGA 自動オフロード

ループ文の FPGA 自動オフロード手法については，[43] で提案を行っている．算術強度やループ回数，リソース量を ROSE [44]，gcov [45] 等を用いて候補ループ文を絞り込んでから，検証環境での複数パターン性能測定を行い，自動でのオフロードを可能としている．

3.2.4 機能ブロックの自動オフロード

機能ブロックの自動オフロード手法については，[46] で提案を行っている．GPU，FPGA に対して方式検証を行ったが，処理としては，名前一致や Deckard [47] での類似性検出でオフロードできる機能ブロックを検出する処理で共通のため，メニーコア CPU でも同様に実施可能である．また，個々のルー

プ文のオフロードに比べ，複数のループ文を含む機能ブロック単位で処理するデバイスにアルゴリズム含めてチューンした機能ブロックオフロードは高速化度合いが高い．

3.3 多様な移行先での自動オフロード

3.1 の基本的考えと 3.2 の個々の移行先環境でのオフロード手法を元に，多様な移行先での自動オフロードについて検討する．

3.3.1 オフロード試行の検証順番

3.1 で述べたように，性能については，行う処理は同じでも，処理ハードウェアのスペック，処理内容（データサイズ，ループ回数等）により大きく変わるため，検証環境の実機で測定する事が必要だと考える．移行先環境 GPU，FPGA，メニーコア CPU の 3 つに対して，ループ文，機能ブロックの 2 つの方法でのオフロードがあるため，合計 3 × 2 の 6 つオフロードが考えられる．

まず，ループ文と機能ブロックに関しては，アルゴリズム含めて処理内容に合わせてオフロードする機能ブロックオフロードの方が高速化できる．

次に，仮想化等を行わない場合に GPU と FPGA とメニーコア CPU で，中心価格帯はメニーコア CPU ≒ GPU ＜ FPGA である．また，1 パターンの検証時間はメニーコア CPU ≒ GPU ＜ FPGA となる．FPGA は，回路設定に数時間必要で，性能測定には時間がかかるのが現状である．

これらの現況を踏まえ，6 つのオフロードで検証する順番として，メニーコア CPU 向け機能ブロックオフロード，GPU 向け機能ブロックオフロード，FPGA 向け機能ブロックオフロード，メニーコア CPU 向けループ文オフロード，GPU 向けループ文オフロード，FPGA 向けループ文オフロードを提案する．この順番でオフロード試行を行い，高性能となるパターンを探索していく．前半の 3 つと後半の 3 つは，対象とするコードが変わっても良いとする．具体的には，前半の 3 つで機能ブロックオフロードが可能だった場合は，後半の 3 つのループ文オフロードはオフロード可能だった機能ブロック部分を抜いたコードに対して試行する．なお，オフロード試行ではユーザが目標性能や価格を指定でき，ユーザが指定する範囲で十分高速で低価格なオフロードパターンが 6 つ試行の前方側で見つかっていれば，以降の試行はしなくても良い．

このようにする理由として，機能ブロックオフロードの方がループ文オフロードに比べ，オフロードできる対象は少ないが出来る場合はより高速化ができるため，順番的に先に検証する．

また，自動オフロードする際は，高速なパターンの探索には，できるだけ安価で短時間に行う事も期待される．そこで，検証時間がかかる FPGA は最後とし，前の段階で十分ユーザ要件を満足するパターンが見つかっていれば，FPGA 検証は行わない事とする．GPU とメニーコア CPU に関しては，価格的にも検証時間的にも大きな差はないが，メモリも別空間となりデバイス自体が異なる GPU に比して，メニーコア CPU の方が，通常 CPU との差は小さいため，検証順はメニーコア CPU を先として，メニーコア CPU で十分ユーザ要件を満足するパターンが見つかっていれば，GPU 検証は行わない事とする．CPU と GPU ではデバイスが異なるため，丸め誤差の違い等で，正しくオフロードできても計算結果が異なる場合がある

4. 評　　価

ループ文の GPU，FPGA への自動オフロード，機能ブロックの GPU，FPGA への自動オフロードは [31] 等で評価をしているため，本稿では，単体移行先へのオフロードの効果でなく，移行先混在環境でもアプリケーションを適切なデバイスにオフロードできていることを確認する．

4.1 評 価 条 件

4.1.1 評 価 対 象

評価対象は，行列計算，ブロック三重対角ソルバとする．

単純な行列計算は，機械学習分析の様々な場面で利用されている．サンプルとして 3 行列積算を，1000*1000 のサイズで計算を行う（STANDARD_DATASET : NI=1000, NJ=1000, NK=1000, NL=1000, NM=1000）[48]．IoT，AI の普及により，クラウドだけでなくデバイス等様々な場面で単純な行列計算が用いられることは多いため，既存アプリケーション含めて自動高速化のニーズはある．

ブロック三重対角ソルバは，偏微分方程式の数値解法である．数値計算には様々な種類，実装があるが，for 文が 100 を超える中規模以上の計算処理の一例として，NAS.BT [49] を用いる．パラメータは，grid size=64*64*64, number of iterations=200, time step=0.0008 の CLASS A 設定で行う．

4.1.2 評 価 手 法

対象のアプリケーションのコードを，多様移行先での自動オフロード機能に入力し，GPU，FPGA，メニーコア CPU の 3 つの移行先に対して，機能ブロック，ループ文のオフロードを試行し，6 つのオフロード試行結果を元に，高性能のパターンを作成し，その性能を測定する．オフロードせず全て通常 CPU で処理する場合に比べた改善度を評価する．合わせて，適切なデバイスを選択できていることを確認する．

実験の条件は以下で行う．

オフロード対象とループ文数 : 行列計算 3mm　18，ブロック三重対角ソルバ NAS.BT　120

メニーコア CPU，GPU でのループ文オフロードの遺伝的アルゴリズム条件は以下の通り．

個体数 M : ループ文数以下とする（3mm は 16，NAS.BT は 20）

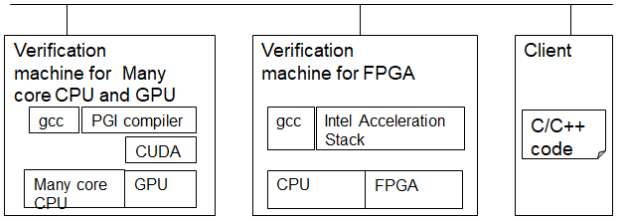
世代数 T : ループ文数以下とする（3mm は 16，NAS.BT は 20）

適合度 : $(\text{処理時間})^{-1/2}$　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．また，性能測定が一定時間（3 分）で終わらない際はタイムアウトさせ，処理時間∞秒として適合度計算する．

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



Name	Hardware	CPU	RAM	GPU	OS	gcc	CUDA toolkit	PGI Compiler
Verification Machine for Many core CPU and GPU	LEVEL-F039-LCR12W-XYVI	AMD Ryzen Threadripper 2990WX (32 Core)	32GB *2	NVIDIA GeForce RTX 2080 Ti (CUDA core: 4352, Memory: GDDR6 11GB)	Ubuntu 16.04.6 LTS	10.1	10.1	19.10
Client	HP ProBook 470 G3	Intel Core i5-6200U @2.3GHz	8GB		Windows 10 Pro			

Name	Hardware	CPU	RAM	FPGA	OS	gcc	Intel Acceleration Stack
Verification Machine for FPGA	Dell PowerEdge R740	Intel(R) Xeon Bronze 3104	32GB *2	Intel PAC with Intel Arria 10 GX FPGA	CentOS 7.4	10.1	1.2

図 3　性能測定環境

FPGA でのループ文オフロード試行の条件は以下の通り．

算術強度絞り込み：算術強度分析の上位 5 つのループ文に絞り込み

リソース効率絞り込み：リソース効率分析の上位 3 つのループ文に絞り込み（算術強度/リソース量が高い上位 3 つのループ文を選定）

実測オフロードパターン数：4（1 回目は上位 3 つのループ文オフロードパターンを測定し，2 回目は 1 回目で高性能だった 2 つのループ文オフロードの組み合わせパターンで測定．）

4.1.3　評 価 環 境

利用するメニーコア CPU として，AMD Ryzen Threadripper 2990WX (32 core) を用いる．OpenMP 処理は gcc10.1 を用いる．利用する GPU として GeForce RTX 2080 Ti (CUDA core: 4352, Memory：GDDR6 11GB) を用いる．Ryzen と GeForce は同一ノードに搭載されている．OpenACC 処理は PGI コンパイラ 19.10 と CUDA Toolkit 10.1 を用いる．FPGA は Intel PAC with Intel Arria10 GX FPGA を用いる．コンパイルするマシンは DELL EMC PowerEdge R740 で，Intel Acceleration Stack Version 1.2 で OpenCL をコンパイルする．評価環境とスペックを図 3 に示す．ここで，Client ノート PC から，ユーザが利用する C/C++言語アプリケーションコードを指定し，Verification Machine を用いてチューニングする．商用利用時は，チューニング後，実際のユーザが使う商用環境にデプロイする．

4.2　性 能 結 果

評価対象として，行列計算 3mm，ブロック三重対角ソルバ NAS.BT に対して，多様移行先での高速化を確認した．

図 4 は，3mm, NAS.BT に対する単コアでの処理時間，どのデバイスにどの方法でオフロードしたか，その処理時間，性能改善度及び別デバイスへのオフロード結果を示している．性能改善度は，オフロード先マシンにおいて，gcc もしくは PGI コンパイラで通常通りコンパイルし，単コアで処理した場合の処理時間を 1 としたときの，処理時間の改善度を表している．3mm に関しては，単コアでの処理時間が 51.3 秒であったが，ループ文オフロードによる GPU では，0.046 秒で処理し，1120 倍の改善度であった．ループ文をメニーコア CPU にオフロードした際は，1.05 秒で処理し，44.5 倍の改善度であり，より高性能の GPU が選択されている．NAS.BT に関しては，単コアでの処理時間が 130 秒であったが，ループ文オフロードによるメニーコア CPU では，24.1 秒で処理し，5.39 倍の改善度であった．ループ文を GPU にオフロードを試行した際は，処理時間が 150 秒を超えたため，全くオフロードせず単コアで処理する場合の 130 秒が結果となった．

	Processing time of single core	Appropriate offload device and method	Processing time with offload device	Performance improvement	Other offload device and method	Processing time with offload device	Performance improvement
3mm	51.3	GPU, loop offload	0.046	1120	Many core CPU, loop offload	1.05	44.5
NAS.BT	130	Many core CPU, loop offload	24.1	5.39	(GPU) (try loop offload)	130	1

図 4　移行先混在環境でのオフロード結果

単体測定では，6 つの測定にて，機能ブロックオフロードに関しては，探索は 1 分程度であるが，FPGA を使う場合は回路設定に 3 時間程度かかっている．ループ文に関しては，FPGA は 1 パターンに 3 時間程度かかるため 4 パターンで半日かかり，メニーコア CPU や GPU での遺伝的アルゴリズムでの探索は 6 時間程度ずつかかっている．結果的に，全ての測定で 1 日程度かかる．

5.　ま　と　め

本稿では，私が提案している，環境適応ソフトウェアの要素として，移行先が，GPU，FPGA，メニーコア CPU と混在している環境で高速化するための自動オフロード手法を提案した．

最初に事前準備として，多様な移行先環境の一つとして，メニーコア CPU に対するループ文の自動オフロード手法を，GPU での進化計算手法を参考に提案した．次に，移行先が多様な環境でのオフロードとして，移行先単体でのオフロード試行の順序検討を行った．具体的には，メニーコア CPU 向け機能ブロックオフロード，GPU 向け機能ブロックオフロード，FPGA 向け機能ブロックオフロード，メニーコア CPU 向けループ文オフロード，GPU 向けループ文オフロード，FPGA 向けループ文オフロードの順に検証する．これは機能ブロックの方がより高速化でき，また FPGA は性能測定に時間がかかるためである．

既存アプリケーションに対して，GPU，FPGA，メニーコア CPU の移行先混在環境における，提案手法での自動オフロードを行い，方式の有効性を確認した．今後は，移行先環境が混在の際に，CPU，GPU，FPGA の処理リソース量を調整し，コスト対効果を高めるための検討を行う．

文　　献

[1] A. Putnam, et al., "A reconfigurable fabric for accelerating large-scale datacenter services," ISCA'14, pp.13-24, 2014.
[2] O. Sefraoui, et al., "OpenStack: toward an open-source solution for cloud computing," International Journal of Computer Applications, Vol.55, 2012.
[3] Y. Yamato, "Server Selection, Configuration and Reconfiguration Technology for IaaS Cloud with Multiple Server Types," Journal of Network and Systems Management, Springer, Aug. 2017.

[4] Y. Yamato, "Cloud Storage Application Area of HDD-SSD Hybrid Storage, Distributed Storage and HDD Storage," IEEJ Transactions on Electrical and Electronic Engineering, Vol.11, Issue.5, pp.674-675, Sep. 2016.
[5] Y. Yamato, "Use case study of HDD-SSD hybrid storage, distributed storage and HDD storage on OpenStack," 19th International Database Engineering & Applications Symposium (IDEAS15), pp.228-229, July 2015.
[6] Y. Yamato, et al., "Development of Resource Management Server for Production IaaS Services Based on OpenStack," Journal of Information Processing, Vol.23, No.1, pp.58-66, Jan. 2015.
[7] Y. Yamato, et al., "Software Maintenance Evaluation of Agile Software Development Method Based on OpenStack," IEICE Transactions on Information & Systems, Vol.E98-D, No.7, pp.1377-1380, July 2015.
[8] Y. Yamato, et al., "Evaluation of Agile Software Development Method for Carrier Cloud Service Platform Development," IEICE Transactions on Information & Systems, Vol.E97-D, No.11, pp.2959-2962, Nov. 2014.
[9] Y. Yamato, et al., "Fast and Reliable Restoration Method of Virtual Resources on OpenStack," IEEE Transactions on Cloud Computing, Sep. 2015.
[10] Y. Yamato, "Performance-Aware Server Architecture Recommendation and Automatic Performance Verification Technology on IaaS Cloud," Service Oriented Computing and Applications, Springer, Nov. 2016.
[11] Y. Yamato, "Automatic verification technology of software patches for user virtual environments on IaaS cloud," Journal of Cloud Computing, Springer, 2015, 4:4, Feb. 2015.
[12] Y. Yamato, "OpenStack Hypervisor, Container and Baremetal Servers Performance Comparison," IEICE Communication Express, Vol.4, No.7, pp.228-232, July 2015.
[13] AWS web, https://aws.amazon.com/ec2/instance-types/
[14] T. Sterling, et al., "High performance computing : modern systems and practices," Cambridge, MA : Morgan Kaufmann, ISBN 9780124202153, 2018.
[15] 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.
[16] J. Sanders and E. Kandrot, "CUDA by example : an introduction to general-purpose GPU programming," Addison-Wesley, 2011
[17] M. Hermann, et al., "Design Principles for Industrie 4.0 Scenarios," Rechnische Universitat Dortmund. 2015.
[18] Y. Yamato, et al., "Predictive Maintenance Platform with Sound Stream Analysis in Edges," Journal of Information Processing, Vol.25, pp.317-320, Apr. 2017.
[19] Y. Yamato, et al., "Proposal of Lambda Architecture Adoption for Real Time Predictive Maintenance," 2016 Fourth International Symposium on Computing and Networking (CANDAR2016), pp.713-715, Nov. 2016.
[20] P. C. Evans and M. Annunziata, "Industrial Internet: Pushing the Boundaries of Minds and Machines," Technical report of General Electric (GE), Nov. 2012.
[21] Tron project web site, http://www.tron.org/
[22] Y. Yamato, "Ubiquitous Service Composition Technology for Ubiquitous Network Environments," IPSJ Journal, Vol.48, No.2, pp.562-577, Feb. 2007.
[23] Y. Yamato, et al., "Study of Service Processing Agent for Context-Aware Service Coordination," IEEE International Conference on Service Computing (SCC 2008), pp.275-282, July 2008.
[24] H. Sunaga, et al., "Ubiquitous Life Creation through Service Composition Technologies," World Telecommunications Congress 2006 (WTC2006), May 2006.
[25] M. Takemoto, et al., "Service-composition Method and Its Implementation in Service-provision Architecture for Ubiquitous Computing Environments," IPSJ Journal, Vol.46, No.2, pp.418-433, Feb. 2005. (in Japanese)
[26] Y. Yokohata, et al., "Service Composition Architecture for Programmability and Flexibility in Ubiquitous Communication Networks," IEEE International Symposium on Applications and the Internet Workshops (SAINTW'06), Jan. 2006.
[27] Y. Nakano, et al., "Method of creating web services from web applications," IEEE International Conference on Service-Oriented Computing and Applications (SOCA 2007), pp.65-71, June 2007.
[28] Y. Yokohata, et al., "Context-Aware Content-Provision Service for Shopping Malls Based on Ubiquitous Service-Oriented Network Framework and Authentication and Access Control Agent Framework," IEEE Consumer Communications and Networking Conference (CCNC 2006), pp.1330-1331, Jan. 2006.
[29] J. Gosling, et al., "The Java language specification, third edition," Addison-Wesley, 2005. ISBN 0-321-24678-0.
[30] Y. Yamato, et al., "Automatic GPU Offloading Technology for Open IoT Environment," IEEE Internet of Things Journal, Sep. 2018.
[31] 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.
[32] K. Shirahata, et al., "Hybrid Map Task Scheduling for GPU-Based Heterogeneous Clusters,"IEEE CloudCom, 2010.
[33] Altera SDK web site, https://www.altera.com/products/design-software/embedded-software-developers/opencl/documentation.html
[34] Xilinx SDK web site, https://japan.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/lyx1504034296578.html
[35] S. Wienke, et al., "OpenACC-first experiences with real-world applications," Euro-Par Parallel Processing, 2012.
[36] M. Wolfe, "Implementing the PGI accelerator model," ACM the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, pp.43-50, Mar. 2010.
[37] E. Su, et al., "Compiler support of the workqueuing execution model for Intel SMP architectures," In Fourth European Workshop on OpenMP, Sep. 2002.
[38] Jenkins web site, https://jenkins.io/
[39] Selenium web site, https://www.seleniumhq.org/
[40] J. H. Holland, "Genetic algorithms," Scientific american, Vol.267, No.1, pp.66-73, 1992.
[41] Clang website, http://llvm.org/
[42] Y. Yamato, "Improvement Proposal of Automatic GPU Offloading Technology," The 8th International Conference on Information and Education Technology (ICIET 2020), pp.242-246, Mar. 2020.
[43] Y. Yamato, "Proposal of Automatic FPGA Offloading for Applications Loop Statements," The 7th Annual Conference on Engineering and Information Technology (ACEAIT 2020), pp.111-123, 2020.
[44] ROSE compiler framework web site, http://rosecompiler.org/
[45] Gcov web site, http://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[46] 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.
[47] Deckard web site, http://github.com/skyhover/Deckard
[48] Polybench 3mm web site, https://web.cse.ohio-state.edu/pouchet.2/software/polybench/
[49] NAS.BT web site, https://www.nas.nasa.gov/publications/npb.html