情報処理学会論文誌 Vol.55 No.2 1048–1058 (Feb. 2014)

GPUオンチップマイクロコントローラへのデータ転送処理オフロード

藤居 祐輔[1,a)] 安積 卓也[2,b)] 西尾 信彦[2,c)] 加藤 真平[3,d)]

受付日 2013年5月13日, 採録日 2013年11月1日

概要：サイバーフィジカルシステム（CPS）が注目される中，その技術基盤として，GPU などのデバイスが利用され始めている．GPU はデバイスドライバを経由して利用されているが，CPS のように短い周期で繰り返し多くの処理が行われると，ホストへの負担が増えるとともに，デバイス制御や処理の同期によってレイテンシが発生する．さらに GPU 処理では，データをデバイスメモリへと転送する必要があり，上記問題を悪化させ，データ転送処理自体にも影響を与える．そのため我々は，GPU 制御処理の一部を GPU マイクロコントローラ上で動作するファームウェアへオフロードし，GPU 処理の効率化をめざす．本論文では，オフロード基盤としてコンパイラ，デバッグ支援ツールを含んだ GPU 制御ファームウェア開発環境と，既存の NVIDIA 社製ファームウェアと同等の機能を持つファームウェアを開発する．次に，オフロード基盤を用いて，制御処理の一部である DMA 転送処理をファームウェアに追加実装することで，オフロードを実現し GPU 処理を効率化する．我々は，実装したファームウェアと既存のファームウェアを比較し，性能低下がないことを示すことで，オフロード基盤の有効性を確認した．オフロードしたデータ転送処理では，既存のデータ転送処理と比べ，一部のデータサイズにおいて約 1.5 倍の転送速度を実現し，さらに既存データ転送処理へのオーバラップ転送を実現した．

キーワード：GPU，GPGPU，GPU データ転送，マイクロコントローラ，開発環境

Data Transfer Processing Offload to the GPU On-chip Microcontrollers

Yusuke Fujii[1,a)] Takuya Azumi[2,b)] Nobuhiko Nishio[2,c)] Shinpei Kato[3,d)]

Received: May 13, 2013, Accepted: November 1, 2013

Abstract: Cyber-Physical Systems (CPS) aim to control complex real-world phenomenon. The computational cost and real-time constraints could be a major challenge of CPS. GPUs have come to be used as base technology for CPS. CPS applications have many short periodic tasks. Therefore latency is occurred and CPU load is increased because of synchronizations and control processes when the CPS application runs on a GPU. In this paper, we present an offloading data transfer process to GPU microcontrollers. We aim at effective GPU processing which reduce the CPU load and improve the data transfer. First of all, we prepared the process-offloading infrastructure, which has the GPU microcontrollers firmware development environment and the no-extensible firmware. Secondly, we provide to data transfer methods by implementing an extensible firmware; transferring data by microcontrollers and transferring data by overlapping to the other data transfers and microcontrollers. Our extensible firmwares do not have performance degradation from the performance of existing firmwares. In addition, we compared our data transfer methods with existing methods. As a result, our transfer methods are one and a half times as fast as transfer speed of existing method when the data size band of part.

Keywords: GPU, GPGPU, GPU data transfer, microcontrollers, development environment

1. はじめに

近年，サイバーフィジカルシステム（CPS）に注目が集まっている．CPS は，綿密に実世界の現象を制御するために，情報処理と物理的要素が相まった，次世代のネットワークや組込みシステムなどを指す．これらの制御アルゴリズムは複雑化しており，従来のセーフクリティカルな組込みシステムから区別され，処理速度が重要視される．Rath らの HBT-EP Tokamak システム [1] は，核融合の制御を行う CPS アプリケーションの一例である．このシステムでは，規定時間内に実行を完了しなければならないという，RTOS の基本的な制約を持つとともに，その規定時間がきわめて短く，既存のシングルプロセッサや，マルチプロセッサでは処理速度という面から制約を守ることができない．本問題へのアプローチとして，メニーコアプロセッサの利用が考えられており，特に，メニーコアプロセッサのなかでもコストや，CUDA [2] などによる普及度の面から，GPU が注目されている．

現在の OS では，GPU はデバイスドライバを介して利用される．これでは，CPS のように短い周期で繰り返し多くの処理を行う場合，GPU の処理が増え，ホスト側への負担が増加し，その制御や同期によってレイテンシが発生する．さらに，GPU はコプロセッサであるため，実行時に必要なデータを GPU のデバイスメモリへと転送する必要があり，これによって，さらに，GPU の処理数は増えるため，上記問題を悪化させ，データ転送自身にも影響を与える．

本論文では，上記の問題を解決するために，GPU マイクロコントローラへのデバイス制御処理のオフローディングを提案する．GPU にはオンチップにマイクロコントローラが搭載され，その上で動作するファームウェアによって電圧管理や CUDA アプリケーションの管理が行われていることが GPU デバイスドライバを開発しているオープンソースコミュニティによって示されている [3]．このマイクロコントローラに制御処理をオフロードし有効に利用することで，問題の解決とともに．GPU 処理の様々な改善が期待できる．しかしながら，コードの記述がアセンブリ言語に限られるなど，現状では開発環境が整備されておらず，ファームウェア開発による処理のオフロードは困難である．

そのため，処理オフロードに必要とされるファームウェア開発環境と，NVIDIA が提供するものと同等の機能を持つファームウェア（ベーシックファームウェア）を開発し，処理オフローディング基盤として提供する．そして，提供する処理オフローディング基盤を用いて，制御処理の一部であるデータ転送処理のオフロードを実現し，その応用として並列データ転送の性能を評価することで，処理オフロードによる GPU 処理の効率化の有用性，可能性を示す．

本論文は全 7 章で構成される．次章では，関連する研究と現状の課題点を述べ，3 章ではアプローチについて述べる．4 章では，本研究において基盤となる技術について述べ，5 章で設計実装について述べる．6 章ではオフロード基盤と，オフロードされたデータ転送処理について評価を行い，7 章でまとめと今後の展望を述べる．

2. 関連研究

本章では，処理オフロードとデータ転送について，それぞれ関連する研究を述べる．

2.1 処理オフロード

Helios: Heterogeneous Multiprocessing with Satellite Kernels [4]

プログラマブルデバイスを用いた異種プラットフォーム環境においては，それぞれのプロセッサで異なる処理が行われる．この不均一性によって同期処理が必要となり，レイテンシの発生がみられる．さらに処理頻度の増加などによって，デバイスドライバなどへの割込みが頻発し，ホスト側への負担の向上がみられる．そこで Nightingale らは，Helios を提案している．Helios は，透過的なプロセッサ間通信，異種アーキテクチャのカプセル化，開発・チューニングの簡素化を実現しており，これらが動作するカーネルをサテライトカーネルと呼ぶ．NIC ドライバなどを XScale や NUMA ドメインのプログラマブルデバイスへとオフロードすることで，サービスの効率化を達成したサテライトカーネルが動作する環境条件として，割込みハンドラ，タイマ，トラップの受け取りといった機能を持つことをあげており，これはサテライトカーネルに限らず，OS 処理のオフロードにはこれらの機能が必要になると考えられる．しかし，GPU はこれらの機能を持っておらず，Helios では，サテライトカーネルによるアプローチが不可能であるとしている．

GPU マイクロコントローラは，サテライトカーネルの動作環境条件を満たしているが，既存の処理を実行しつつ，サテライトカーネルを動作させることは困難である．

2.2 GPU データ転送

Gdev: First-Class GPU Resource Management

[1] 立命館大学大学院情報理工学研究科
Graduate School of Information Science and Engineering, Ritsumeikan University, Kusatsu, Shiga 525–8577, Japan
[2] 立命館大学情報理工学部
College of Information Science and Engineering, Ritsumeikan Uniersity, Kusatsu, Shiga 525–8577, Japan
[3] 名古屋大学大学院情報科学研究科
Graduate School of Information Science, Nagoya Uniersity, Nagoya, Aichi 464–8601, Japan
a) yukke@ubi.cs.ritsumei.ac.jp
b) takuya@cs.ritsumei.ac.jp
c) nishio@cs.ritsumei.ac.jp
d) shinpei@is.nagoya-u.ac.jp

in the Operating System [5]
NVIDIA 社製 GPU における GPGPU 実行環境には CUDA があり，その実行環境が NVIDIA 社より提供されている．しかし，実行環境はオープンソースではないため，研究や開発が困難であり，より汎用的な GPU 利用をめざすための基盤が形成されていない．そのため Gdev は，GPGPU 実行環境をオープンソースで実装することで基盤を形成し，それに資源管理機能などを加えることで，より汎用的な第 1 級の GPGPU 実行環境をめざしている．Gdev で用いられているデータ転送手法は，メモリマッピングを用いた方式と DMA を用いた方式がある．これらの方式は転送するデータのサイズごとに特性が異なり，使い分けられている．メモリマッピングを用いた方式では 256 B を超えると速度の低下が大きくみられ，DMA を用いた方式では，64 KB までは，低速であるという結果がでている．DMA 転送がデータサイズが小さいときに低速で，一定のデータ転送時間を保つ理由は，DMA コントローラへの指示や同期に時間がかかり，それが大半を占めているためと予想される．以上の特性から現状では，256 B から 64 KB まではそれぞれ手法において，不得意とするデータサイズ帯であるといえる．さらに，GPU の仕様上，Gdev においても，1 つの GPU 実行単位（GPU Channel）に対し，DMA コントローラは 1 つのみ割当てが可能となっており，複数の DMA コントローラを利用することができないという問題がある．

Zero-Copy I/O Processing for Low-Latency GPU Computing [6]
1 章にあげた CPS アプリケーションにおいて有効なデータ転送として，メモリマッピングを用いた，I/O デバイスからのデータの利用手法がある．しかし，メモリマッピングを用いる手法は，デバイスからホストへの読み込みが低速であり，システムの周期によっては処理結果の確認が困難である．そのため，読み込み時のみ，既存のデータ転送を組み合わせたハイブリッド転送を提案している．対象としている CPS アプリケーションでは，データ入力が 2 KB となっており，このデータサイズ帯においてさらなる高速化を求められる．

3. アプローチ

本論文では，ホストの負担の軽減や一部データサイズ帯における既存データ転送手法の改善などを含めた GPU 処理の効率化のために，GPU マイクロコントローラへのデータ転送処理オフローディングを提案する．GPU マイクロコントローラのファームウェアコードの記述は，アセンブリ言語に限られるなど，現状では開発環境が整備されておらず，ファームウェア開発による，処理のオフロードは困難である．そのため，処理オフロードに必要とされるファームウェア開発環境と，NVIDIA が提供するものと同等の機能を持つファームウェア（ベーシックファームウェア）を開発し，処理オフローディング基盤として提供する．ファームウェア開発環境は，移植性の高さで知られる LLVM を用いた GPU マイクロコントローラ向けコンパイラとデバッグ支援ツールで構成される．ベーシックファームウェアは GPU 処理においてコンテキスト管理を行うためのマイクロコントローラのものであり，GPU 処理に密接した制御処理のオフローディングが期待できる．

次に，オフロード基盤を用いてファームウェアに制御処理を実装することで，制御処理オフロードを実現する．マイクロコントローラには外部メモリ（デバイスメモリ，ホストメモリ）に存在するデータの読み込み/書き込みを可能とする機能が搭載されている．この機能は本来 GPU コンテキストの読み込み・書き込みを行うためのものであるが，自身の RAM 領域を介することでデバイスメモリ・ホストメモリ間のデータ転送を実現する．

最後に，オフロードしたデータ転送処理を既存のデータ転送処理にオーバラップさせることで，並列データ転送処理を実現し，性能評価によってオフロードによる GPU 処理の効率化の有用性，可能性を示す．

4. 基盤プラットフォーム技術

4.1 GPU マイクロコントローラ

本論文では，NVIDIA の Fermi アーキテクチャ [7] を採用した GPU に搭載されるマイクロコントローラを用いる．**図 1** にホストコンピュータに接続される GPU の構成を示す．GPU は PCIe バスに接続され，ホストからのアクセスは，I/O バス，PCI ブリッジ，PCIe バスを通じて行う．Fermi アーキテクチャの GeForce GTX480 [8] に搭載される GF100 チップには，SM（Streaming Multiprocessor）が 16 基搭載されており，SM1 基は 32 個の CUDA コアで構成される．この SM4 基で 1 つの GPC（Graphics Processing Cluster）が構成される．GPC にはそれぞれマイクロコントローラが搭載され，このマイクロコントローラを本論文では「GPC マイコン」と呼ぶ．加えて，GPC を統括しているマイクロコントローラも搭載されており，これを本論文では「HUB マイコン」と呼ぶ．

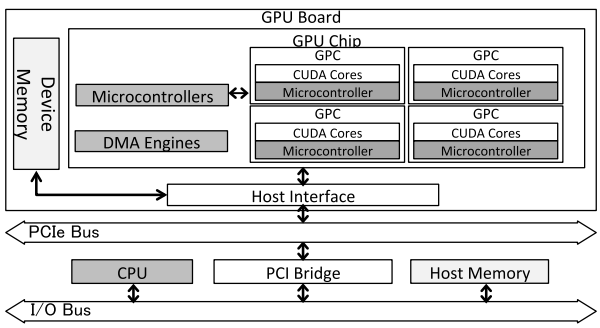


図 1 NVIDIA GPU の構成図
Fig. 1 Architecture of NVIDIA's GPU.

NVIDIA の GPU マイクロコントローラの命令セットは，独自のものであり，仕様はすべて非公開である．しかし，Nouveau Project [9] が GPU を解析し，これらの情報をまとめて公開している．これには GPU マイクロコントローラの情報も含まれており，解析ツールとともにオープンソースで Envytools [3] として配布されている．Envytools には，マイクロコントローラ用のアセンブラも含められている．現在の Envytools にまとめられている情報は Fermi アーキテクチャ以前のものが多く，GeForce GTX680 [10] などに搭載されている Kepler アーキテクチャ [11] の情報は少ない．これは本論文で Fermi を用いた理由の 1 つである．

4.2 GPU ランタイム

一般的に GPU のための API はランタイムライブラリが提供する．その一方で GPU の資源管理はデバイスドライバや OS（Operating System）モジュール [5], [12], [13] によってサポートされる．ファームウェアコードは，デバイスドライバに組み込まれており，OS 起動時に GPU 上のマイクロコントローラへ書き込まれる．そのため，独自のファームウェアを利用する際は，4.1 節であげた Nouveau プロジェクトが開発するデバイスドライバを用いる．さらに，CUDA の実行には CUDA ランタイムエンジンが必要であり，こちらもオープンソースの Gdev を用いる．Gdev は，GPU デバイスドライバと密接な関係を持ち，OS モジュールとして動作しており，ユーザ空間で動作するランタイムライブラリが GPU を制御するのに必要なインタフェースを提供する．

5. 設計・実装

本論文における，オフロード基盤と，データ転送処理のオフロードの設計・実装について述べる．オフロード基盤は開発環境と，ベーシックファームウェアで構成される．データ転送処理オフロードでは，ファームウェアの拡張設計・実装について述べ，最後に，オフロードしたデータ転送を用いた，オーバラップ転送について述べる．

5.1 GPU マイクロコントローラ向けファームウェア開発環境

本開発環境は，GPU マイクロコントローラ向けコンパイラ（GUC，GPU Microcontrollers Compilers）とデバッグ支援ツールで構成される．

コンパイラは，GPU マイクロコントローラのアーキテクチャ変更に対応するために，移植性の高さを実現する LLVM インフラストラクチャを用いる．**図 2** に LLVM のコンパイルフローを示す．LLVM はソースコードから LLVM 中間言語を生成する部と，中間言語からオブジェクトコードを生成する部がモジュール化され区別されており，

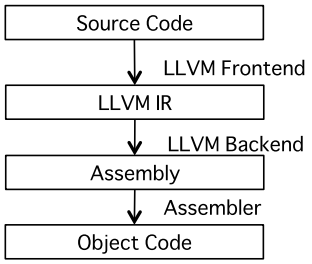


図 2 LLVM のコンパイルフロー
Fig. 2 LLVM compile flow.

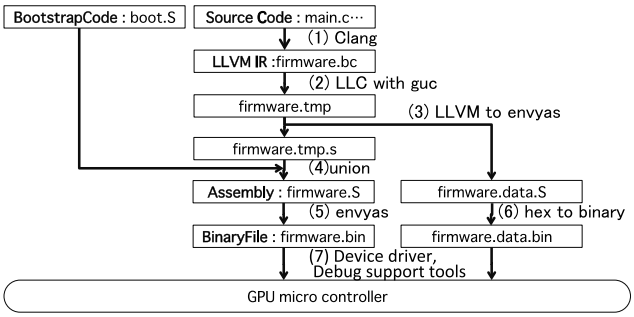


図 3 GUC のコンパイルフロー
Fig. 3 GUC compile flow.

それぞれフロントエンド，バックエンドと呼ぶ．このバックエンドはコード生成部と命令セット仕様が分離されており，対象とするアーキテクチャに依存しない形になっているため，命令セットの追加が容易である．さらに，LLVM を用いることでコンパイラ開発自体に関しても，効率化が望める．

デバッグ支援ツールは，デバッグが困難なファームウェア開発を支援するためのツールである．本来ファームウェアは，OS カーネル起動時に GPU デバイスドライバによってマイクロコントローラに読み込まれ，実行される．さらにその後の制御も，デバイスドライバによって行われており，ファームウェア実行中の，マイクロコントローラの状態の把握は困難である．これらの制御はすべて，MMIO を通して行う．そのため，ユーザ空間から PCI の MMIO 空間にアクセスすることで，簡易的なデバイスドライバとして，ファームウェアの読み込みや，実行，レジスタ状態の読み込みなどを行うことでデバッグを支援する．

5.1.1 GUC

GUC の全体図を**図 3** に示す．GUC は，**(1) Clang**，**(2) LLC with guc**，**(3) LLVM to envyas**，**(4) union**，**(5) envyas**，**(6) hex to binary**，**(7) GPU Device Driver**，**Debug support tools** で構成される．

大まかな流れとして，C 言語コードから LLVM 中間言語を生成 **(1)** し，LLVM 中間言語からアセンブリ言語コードの生成を行う **(2)**．そして生成されたアセンブリ言語コードをコード部とデータ部に分割 **(3)** し，コード部に起動用コードを結合 **(4)** する．その後，コード部からアセンブラ

によってバイナリコードを生成し (5)，データ部をバイナリコードに変換 (6) する．開発したファームウェアの実行は GPU デバイスドライバや，本論文で提供するデバッグ支援ツールによってマイクロコントローラへの書き込み・実行 (7) が可能である．それぞれの詳細を以下で解説する．

(1) Clang：Clang は LLVM のフロントエンドであり，C 言語系列のコード (main.c, . . .) から LLVM 中間言語コード (firmware.bc) を生成する．GUC では，Linux でのネイティブアプリケーション，デバイスドライバ，カーネルの開発に使われている実績から，対象とするフロントエンドの言語を C 言語としている．

(2) LLC with guc：LLC（LLvm Static Compiler）は LLVM のバックエンドである．LLC のコード生成の流れを**図 4** に示す．LLC は LLVM 中間言語を入力とし，制御フロー解析，最適化，命令選択，レジスタ割付けといった流れでコンパイルされ，アセンブリ言語コードを出力する．この流れは共通化されており，ターゲットマシンには依存していない．ターゲットマシン依存部は命令選択時に追加モジュールとして読み込まれ，各々の仕様に沿った命令，レジスタを選択する．これによって，ターゲットマシンの仕様に合わせたアセンブリ言語コードが出力可能となる．LLC with guc ではターゲットマシンのモジュールとして NVIDIA 社製 Fermi アーキテクチャのマイクロコントローラを追加することで，LLVM 中間言語コードから GPU マイクロコントローラのアセンブリ言語コード (firmware.tmp) を生成可能としている．本モジュール名は guc としている．

(3) LLVM to envyas：マイクロコントローラへのファームウェアの書き込みはコード部とデータ部が分割されている必要があるため，LLVM to envyas によって (2) で生成されたアセンブリ言語コードを分割しコード部 (firmware.tmp.s) とデータ部 (firmware.data.S) を生成する．その後，コード部のデータラベルをマイクロコントローラで利用可能な相対アドレスへと変換する．

(4) union：(3) で分割されたコード部には起動用のブートストラップコードが含まれていない．そのため，事前に用意したブートストラップコードをコード部に結合し，アセンブラでアセンブル可能なコード (firmware.S) を生成する．ブートストラップコードには，レジスタの初期化，割込みハンドラの初期設定，main 関数の呼び出しが含まれる．

(5) envyas：envyas は 4.1 節で述べた Envytools に含まれる NVIDIA 社製 GPU マイクロコントローラ向けアセンブラである．(4) で生成されたコードを入力とし，バイナリコード（firmware.bin）を出力する．

(6) hex to binary：データ部をバイナリコード (firmware.data.bin) へと変換する．

(7) Device driver, Debug support tool：生成したファームウェアのバイナリコードは，デバイスドライバへの組み込み，もしくはデバッグ支援ツールを用いて実行する．

C 言語コードと，C 言語コードから生成される LLVM 中間言語，アセンブリ言語コードを**図 5** に示す．

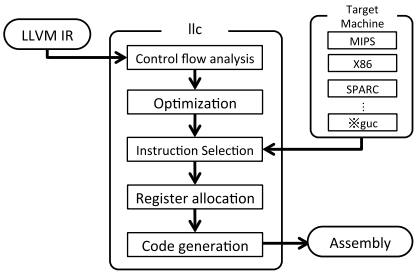


図 4 LLC のコード生成フロー
Fig. 4 LLC code generate flow.

5.1.2 デバッグ支援ツール

マイクロコントローラのレジスタは PCI の BAR（Base Address Register）空間に MMIO でマッピングされている．デバッグ支援ツールでは libpciaccess ライブラリ [14] を用いて，BAR 空間にアクセスし，マイクロコントローラの IO レジスタへの書き込み・読み込みを実現している．本ツールではファームウェアの GPU マイクロコントローラへのロード，開始指示，コマンド送信，レジスタの読み込みを行う．**図 6** に本ツールのフローを示し各項目ごとに解説する．

(1) Load Firmware：HUB マイコンと GPC マイコンともに，コンパイラで生成したバイナリコードをデバッグ支援ツールが読み込み，マッピングされたデータ

C

```
int max(int a,int b){
   return a > b ? a : b;
}
```

LLVM IR

```
define i32 @max(i32 %a, i32 %b) nounwind
readnone {
 %1 = icmp sgt i32 %a, %b
 %2 = select i1 %1, i32 %a, i32 %b
  ret i32 %2
}
```

Assembly

```
max:
      cmp b32 $r14 $r15
      bra l    #LBB1_2
      sub b32 $r15 $r14 0
LBB1_2:
      ret
```

図 5 GUC によるコード生成例
Fig. 5 Code generation example by the GUC.

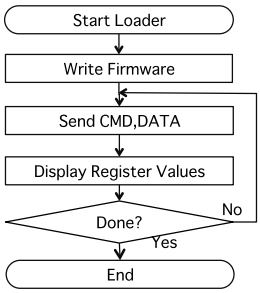


図 6 デバッグ支援ツールのフロー
Fig. 6 Debugging support tools flow.

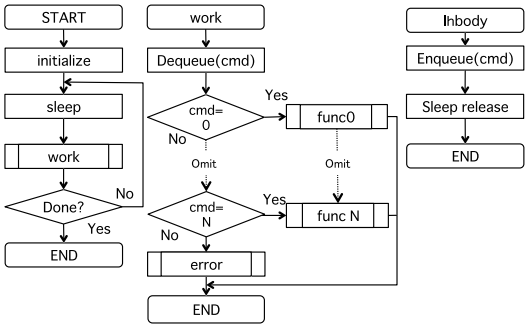


図 7 ベーシックファームウェアのフロー
Fig. 7 Basic firmware flow.

ポートレジスタに読み込んだバイナリコードを書き込むことで，ファームウェアのマイクロコントローラへ書き込みを行う．書き込みが完了した後に，実行開始レジスタにフラグをセットすることで，ファームウェアが実行される．

(2) Send Command, Data：マイクロコントローラはコマンドドリブンである．そのため動作にはコマンドの送信，コマンド実行に利用するデータの送信を行う必要がある．コマンド・データの送信は，コマンド受信レジスタに書き込むことで行われる．

(3) Display Register Values：BAR 空間にマッピングされたマイクロコントローラのレジスタを参照し表示する．

5.1.3 ベーシックファームウェア

本項では，GUC を用いて開発した HUB マイコン，GPC マイコンのファームウェアについてまとめる．本ファームウェアは NVIDIA の提供するファームウェアと同等の機能を持ち，これをオフロード実装の基盤とすることを想定しているためベーシックファームウェアと呼ぶ．

図 7 にファームウェアのフローチャートを示す．ファームウェアの大まかな流れとして START から開始し，initialize，sleep，ihbody，work の順に行い sleep，work を繰り返す．

(1) initialize：ファームウェアが開始されると，割込みハンドラ（ihbody）の設定，制御する GPC の個数などの必要なデータの取得が行われる．そして，GPC マイコンの初期化完了のフラグを確認し，HUB マイコンの初期化完了フラグをセットし，(2) へと移行する．

(2) sleep：(1) 終了後，ファームウェアは待機状態に移り，デバイスドライバ，デバッグ支援ツールからのコマンド送信を待つ．コマンドを受信すると割込みが発生し，ihbody 関数が立ち上がる．

(3) ihbody：ihbody では送信されたコマンドをコマンドキューにエンキューし，ファームウェアの待機状態を解除する．

(4) work：ファームウェアの待機状態が解除されると，work 関数が呼び出され，デキューを行い，条件分岐によってコマンドにあった機能を呼び出す．機能実行後は，Done（ファームウェア終了フラグ）を確認し，フラグが立っていなければ，(2) へと移行し，再度待機状態でコマンドの送信を待つ．

以上のようにファームウェアの動作はコマンドドリブンになっており，外部からの割込みによってコマンドを受信し．コマンドにそった機能を実行する．この機能（func 0, . . . , func N）はそれぞれのマイクロコントローラによって異なり，HUB マイコン，GPC マイコンでは CUDA での GPU コンテキストスイッチなどがその機能にあたる．GPU コンテキストとは，CUDA において各種処理を行うためのハンドルであり，デバイスメモリのベースアドレスやマイクロコントローラの状態が含まれている．CPU のプロセスごとに作成するのが一般的である．本ベーシックファームウェアではこのコンテキストスイッチを実装しており，図 7 の func 部に位置している．

5.2 ファームウェア拡張によるデータ転送処理オフロード

本論文では，GPU マイクロコントローラのファームウェアに機能を追加実装することで，GPU マイクロコントローラへのデータ転送処理オフロードを実現する．データ転送機能は図 7 の func 部に実装している．GPU マイクロコントローラは，MMIO 空間への直接のアクセスは不可能であるが，自身のデータセクションと，外部のメモリとの DMA データ転送を行う DMA コントローラが搭載されているため，これを用いて実装する．ただし，この DMA コントローラには，利用時に制約が存在する．転送元，転送先のデータアドレスはどちらも UVA（Unified Virtual Addressing）で管理されたアドレスでなければならない．さらに，一度に行えるデータ転送のサイズは 2^n（$3 \leq n \leq 8$）B で指定され，必ず自身のデータセクションを介さなければならない．そのため，データサイズが 256 B を超えた際には，複数回データ転送機能を呼び出さなければならない．

データ転送部の実装には 3 種の手法があり，HUB を用いるもの，GPC を用いるもの，GPC を 4 つ用いて並列にデータ転送するものがある．単一のマイクロコントローラを用いた実装は，本データ転送機能を順序的に呼び出す．

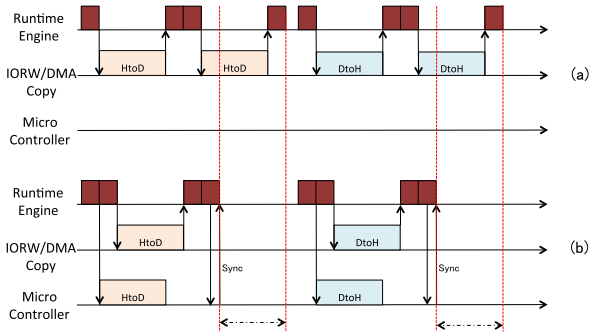


図 8 データ転送のオーバラップ
Fig. 8 Overlaped data transfer.

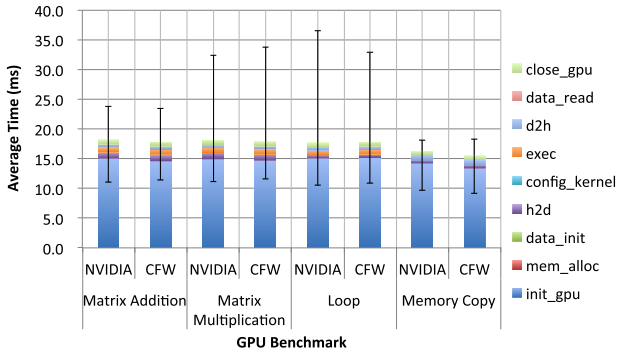


図 9 CUDA ベンチマーク計測結果
Fig. 9 Result of CUDA benchmark in the Gdev.

マイクロコントローラに搭載される DMA コントローラの動作には，各設定レジスタへのパラメータの設定，さらに転送開始レジスタへのフラグセットによって行われる．複数のマイクロコントローラを用いた実装は，本データ転送機能を複数のマイクロコントローラで呼び出すことで並列データ転送を行う．

GPU マイクロコントローラによるデータ転送は，ホストの処理に依存しない形で行うことができる．さらに既存の GPU とホスト間でのデータ転送では，1 つの Channel が同時に 2 つの DMA 転送を行うことはできないが．GPU マイクロコントローラを用いることで並列データ転送が可能になる利点がある．

5.3 データ転送のオーバラップ

オフロードされたデータ転送処理を用いると，既存のデータ転送へマイクロコントローラのデータ転送をオーバラップさせることが可能である．本論文における実装の，処理タイムラインを**図 8** に示す．図 8 の (a) は従来の順次データ転送を呼ぶ出した場合で，(b) は今回提案するデータ転送のオーバラップを行った場合である．データ転送はランタイムエンジンの指示によって開始され，終了時に同期を行う．データ転送のオーバラップでは，オフロードされたデータ転送処理を非同期で実装し，従来のデータ転送を呼び出した後に同期することで，データ転送にかかる時間が削減できると考えられる．

6. 評価

実験環境は，Intel core i7 2600，NVIDIA Geforce GTX480，8 GB の RAM で，Linux カーネルは，ファームウェア性能評価では 2.6.42.12-1.fc15.x86_64，データ転送評価では 3.10.0-rc6 を用いている．CUDA のランタイムエンジンは Gdev で，GPU デバイスドライバは Nouveau を用い，実行環境はすべてオープンソースのものを利用する．

6.1 オフロード基盤

オフロード基盤の有用性を示すため，CUDA プログラムを用いてベーシックファームウェアと NVIDIA のファームウェアの性能評価を行い，開発効率についての考察を行う．今回利用する CUDA ランタイムエンジンである Gdev は GPU デバイスドライバと，ユーザアプリケーションの間に位置しており，CUDA API をユーザアプリケーションに提供することで，ユーザへと GPGPU の実行環境を提供している．

Gdev には，ベンチマークが含まれており，行列和，行列積，データ転送，繰返し処理などの基本的な処理に関するテストが可能である．

本評価では上記のベンチマークを用いて，アプリケーション実行時間，コンテキスト切替え時間を，NVIDIA のファームウェアと比較することで，同等の性能が実現できたことを示す．

まず，ベンチマーク単体の実行時間の 100 回の計測結果の平均値を**図 9** に示す．ユーザアプリケーション内で各処理ごとにかかった時間を計測しておりそれぞれ凡例に示す．NVIDIA は NVIDIA 公式のファームウェアを用いた場合であり，CFW（Compatible FirmWare）は我々のファームウェアを利用した場合である．グラフのひげは最大と最小の誤差を示している．

我々のファームウェアは，NVIDIA のファームウェアを用いた場合との実行時間（exec）の差はほぼ見られず，オーバヘッドは最大でも繰返し処理の 0.0787 ms で，ベンチマーク全体の時間の 0.4%程度である．CUDA アプリケーションなどの GPU での処理は，いまだ洗練されておらず，PCIe バスやデバイスメモリ，キャッシュなど実行時間に影響を与えるものが多く存在しているため，実行時間の誤差は大きくなる．加えてオーバヘッドが見られないベンチマークも多いことから，ベンチマーク単体では，同等の性能が実現できている．

次に，ベンチマークを 2 個並行に実行し，実行時間を比較することでコンテキスト切替え時間を含めた性能の比較を行う．ここでは，行列和のアプリケーションを利用する．GPU 上ではアプリケーションは同時に 1 つしか処理されず，2 個のアプリケーションは発行された順に処理さ

れる．先に処理されるアプリケーションが終了し，次のアプリケーションが処理されるときに，コンテキストスイッチが発生する．そのため本論文では，コンテキストスイッチを含めた時間として，2 個のアプリケーション全体にかかった時間を計測することで，性能の比較を行う．

上記実験によって計測した 100 回の計測結果の平均値を**図 10** に示す．NVIDIA のファームウェアの場合と，我々の開発したファームウェアとで，合計時間の差はほぼ見られなかった．さらに，合計時間の誤差が大きいことからも，この合計時間の差は誤差の範囲内だといえ，ファームウェアの性能低下はないといえる．

したがってアプリケーション単体，アプリケーション複数の場合での性能差が見られないことから，NVIDIA が提供するファームウェアと同等の性能を持つファームウェアが実現できた．

一方の開発効率については，従来のアセンブリ言語での開発と比べ，C 言語での記述が可能であるため，記述コード数の削減，レジスタ割付けなどの手間の削減などによって向上することは自明である．今回開発したベーシックファームウェアの行数は HUB マイコンで 2,164 行，GPC マイコンで 1,206 行であった．オフロード処理の追加は 3 種の転送を可能にする実装で 49 行であり，開発者への負荷は少ないといえる．さらにアセンブリ言語でなく高級言語であるため，コードの可読性は高いことからも負荷を抑えることができると期待できる．

以上の基盤としての拡張のないファームウェア，そして開発効率の向上によって，オフロード基盤として有効であるといえる．

6.2 データ転送性能

本節では，従来の CUDA で用いられているデータ転送手法と，本論文で提案するオフロードしたデータ転送手法を用いた手法で転送にかかる時間を比較することで，我々の手法の性能を評価する．

本評価の比較対象は，従来のデータ転送手法として Gdev で用いられている 2 種と，今回オフロードした転送の 3 種を比較する．それぞれ以下のとおりである．

- **DMA**：標準的な，DMA 転送手法
- **IORW**：標準的な，メモリマッピングされたデータの読み取り/書き取りによる転送
- **HUB**：HUB マイコンによるデータ転送
- **GPC1**：GPC マイコンを 1 個利用したデータ転送
- **GPC4**：GPC マイコンを 4 個利用した並列データ転送

計測に用いる CUDA アプリケーションはホストメモリとデバイスメモリ間でデータ転送を行うのみであり，GPU カーネルは実行しない．アプリケーション内のデータ転送に用いる API（cuMemcpyHtoD/cuMemcpyDtoH）の呼び出し前，呼び出し後の時間を測定することでデータ転送にかかった時間を計測する．ここでのデータ転送時間には API 呼び出しや DMA コントローラへの指示にかかる時間についても含まれるが，実利用時には必ず API を通して利用するため，本評価では含むほうが好ましい．加えて CUDA アプリケーションは Linux の SCHED_RR（リアルタイムラウンドロビン型）のスケジューリングポリシを用いて，優先度を最高に設定し実行する．

上記実験により計測した各サイズ，各手法のデータ転送の平均時間を**図 11** に示す．図 11 (a) はホストメモリからデバイスメモリへのデータ転送時間，図 11 (b) はデバイス

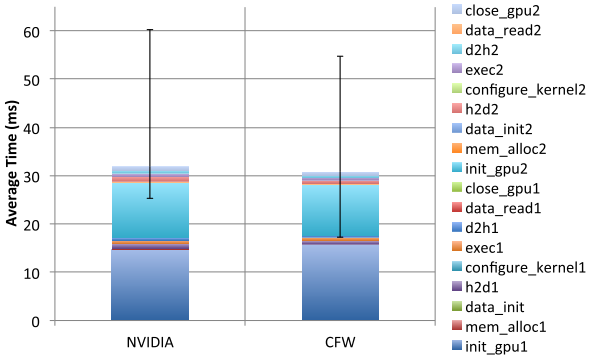
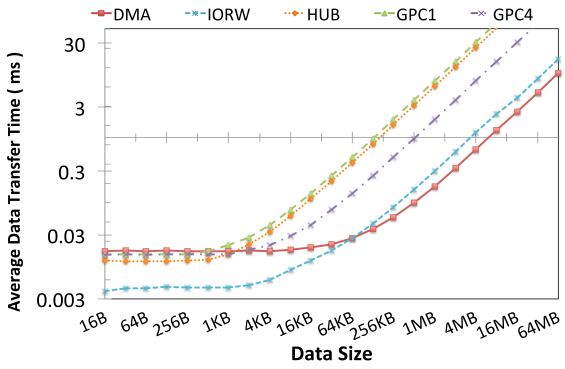
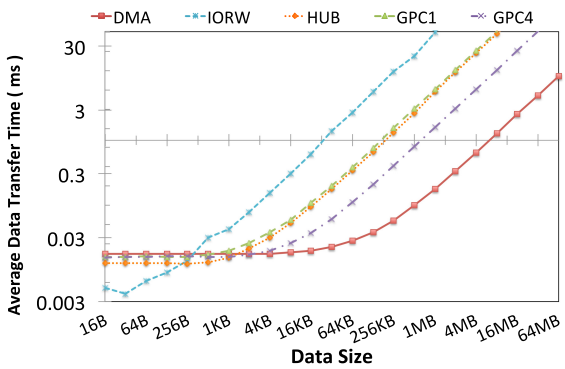


図 10 コンテキストスイッチ時間の計測結果
Fig. 10 Result of measuring the context switch time.



(a) Host to Device



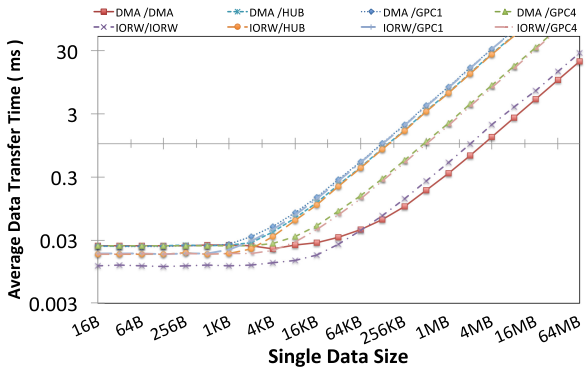
(b) Device to Host

図 11 それぞれの手法によるデータ転送時間
Fig. 11 Result of the data transfer time by each method.

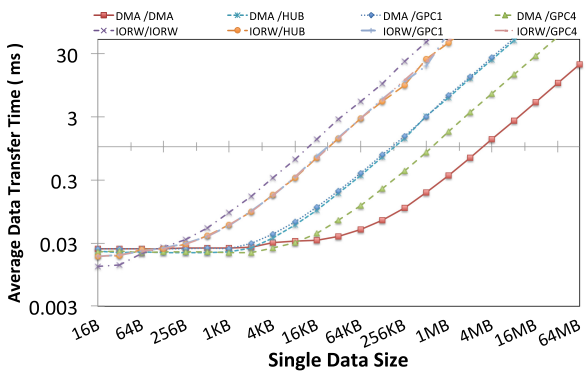
メモリからホストメモリへのデータ転送時間を指す．

ホストメモリからデバイスメモリへの転送の図 11 (a) では，128 KB までは IORW が最も速く，それ以降は DMA が最も速いという結果が出た．デバイスメモリからホストメモリへの転送の図 11 (b) では，IORW が低速になり，256 B から 4 KB までの区間では HUB が最も速く，それ以降は DMA が最も速いという結果が出た．HUB マイコンは GPC マイコンに比べクロックが高いため，GPC1，GPC4 よりも高速であった．GPC4 はそれぞれの GPC マイコン 4 個へ指示を送る部分に時間がかかっているため，低データサイズ帯では遅い結果がでたが，データサイズが増えるにつれ，並列なデータ転送の恩恵を受け GPC4 が高速になっていった．

GPU アプリケーションでは，転送するサイズは静的に決定されるため，各データサイズごと・各転送先ごとに手法を使い分ければよく，Rath らの GPU を用いた核融合制御システム [1] のような 2 KB から 4 KB のデータを扱う事例もあるため，本手法は有用であるといえる．加えて，DMA と同様にデータ転送自体は CPU を利用しないため負荷の削減についても期待ができる．本実験に用いたプロトタイプ実装は，従来のデータ転送と同様に API を提供しホスト側からのコマンド送信をトリガとしている．そのためカーネル実行をトリガとするなど，ファームウェアに閉じることでさらなる負荷の削減や同期の削減を期待できる．



(a) Host to Device



(b) Device to Host

図 12 データ転送のオーバラップの計測結果
Fig. 12 Result of the overlapping data transfer.

6.3 データ転送のオーバラップ

本評価は，6.2 節と同一の環境下で，CUDA アプリケーションを API を 2 度呼び出す形に変更することで計測を行う．DMA，IORW を順次に 2 回行う場合と，DMA，IORW にオフロードしたデータ転送を被せた場合の，計 8 手法でデータ転送時間を比較する．上記実験によって計測した結果 100 回の平均値を**図 12** に示す．

図 12 (a) はホストメモリからデバイスメモリへのデータ転送時間，図 12 (b) はデバイスメモリからホストメモリへのデータ転送時間を指す．

ホストからデバイス間では，単一のデータ転送と同様に，IORW が最速であった．デバイスからホスト間においても単一のデータ転送時と同様に，256 B から 16 KB の区間においてオーバラップしたデータ転送手法が優れた転送速度を発揮した．一部の区間ではあるが高速な転送が可能であるため制御処理のオフロードは GPU 処理の効率化に有効であるといえる．

7. おわりに

本論文では，GPU 処理の効率化をめざし，GPU マイクロコントローラへのデバイス制御処理のオフロードを提案し，オフロード基盤と，制御処理としてデータ転送のオフロード，オフロードしたデータ転送を利用したオーバラップ転送を提供した．評価の結果，我々のファームウェアは既存のファームウェアと比べオーバヘッドは見られず，オフロード基盤として有効であることを確認した．加えて，オフロードしたデータ転送，オーバラップした転送のどちらも既存手法の不得意とするデータサイズ帯において，優れた性能を示した．

本論文での成果の GPU 制御用ファームウェア開発環境は，https://github.com/CS005/guc でオープンソースとして提供しており，ベーシックファームウェアについても公開に向けて準備を進めており，さらに我々が得た知識をドキュメント化することで，ファームウェア開発における負荷を軽減する必要がある．今後は，次世代アーキテクチャへと対応するとともに，さらなる GPU 制御処理のオフロードについても取り組む．今回対象とした GPU 制御処理とは別の処理をファームウェアに追加実装することで，電圧管理や，GPU のカーネル実行などへのアプローチも可能であると考えられる．既存の GPU 資源管理の研究 [5], [12], [13], [15], [16] では，ホスト側による管理のみに焦点をあてているため，同期が発生することで細粒度な資源管理が難しい．したがってマイクロコントローラを用いた細粒度な資源管理について取り組む．

謝辞 本研究の一部は公益財団法人堀科学芸術振興財団と，総務省の「戦略的情報通信研究開発推進制度 (SCOPE)」

（受付番号 132307011）の支援を受けて実施された．

参考文献

[1] Rath, N., Bialek, J., Byrne, P., DeBono, B., Levesque, J., Li, B., Mauel, M., Maurer, D., Navratil, G. and Shiraki, D.: High-speed, Multi-input, Multi-output Control Using GPU Processing in the HBT-EP Tokamak, *Fusion Engineering and Design*, Vol.87, No.12, pp.1895–1899 (2012).

[2] NVIDIA: CUDA, available from ⟨http://www.nvidia.com/⟩.

[3] Koscielnicki, M.: Envytools, available from ⟨https://0x04.net/envytools.git⟩.

[4] Nightingale, E.B., Hodson, O., McIlroy, R., Hawblitzel, C. and Hunt, G.C.: Helios: Heterogeneous Multiprocessing with Satellite Kernels, *SOSP'09*, pp.221–234 (2009).

[5] Kato, S., McThrow, M., Maltzahn, C. and Brandt, S.: Gdev: First-class GPU Resource Management in the Operating System, *Proc. 2012 USENIX Conference on Annual Technical Conference, USENIX ATC'12*, p.37, USENIX Association (2012).

[6] Kato, S., Aumiller, J. and Brandt, S.: Zero-Copy I/O Processing for Low-Latency GPU Computing, *Proc. 4th ACM/IEEE International Conference on Cyber-Physical Systems*, pp.170–178 (2013).

[7] NVIDIA: Fermi architecture, available from ⟨http://www.nvidia.com/object/fermi-architecture.html⟩.

[8] NVIDIA: GeForce GTX480 Specifications, available from ⟨http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-480/specifications⟩.

[9] Nouveau: Accelerated Open Source driver for nVidia cards., available from ⟨http://nouveau.freedesktop.org/⟩.

[10] NVIDIA: GeForce GTX680 Specifications, available from ⟨http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-680⟩.

[11] NVIDIA: Kepler architecture, available from ⟨http://www.nvidia.com/object/nvidia-kepler.html⟩.

[12] Kato, S., Lakshmanan, K., Kumar, A., Kelkar, M., Ishikawa, Y. and Rajkumar, R.: RGEM: A Responsive GPGPU Execution Model for Runtime Engines, *RTSS*, pp.57–66, IEEE Computer Society (2011).

[13] Bautin, M., Dwarakinath, A. and cker Chiueh, T.: Graphic Engine Resource Management, *Proc. SPIE*, Vol.6818, p.68180O (2008).

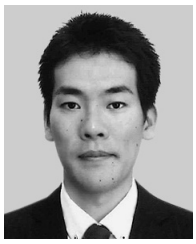
[14] X.org Foundation: X.org, available from ⟨http://cgit.freedesktop.org/xorg/lib/libpciaccess/⟩.

[15] Basaran, C. and Kang, K.-D.: Supporting Preemptive Task Executions and Memory Copies in GPGPUs, *Proc. Euromicro Conference on Real-Time Systems*, pp.287–296 (2012).

[16] Elliott, G. and Anderson, J.: Robust Real-Time Multiprocessor Interrupt Handling Motivated by GPUs, *Proc. Euromicro Conference on Real-Time Systems*, pp.267–276 (2012).

藤居 祐輔 （学生会員）

2013 年立命館大学情報システム学科卒業．同年立命館大学大学院情報理工学研究科博士前期課程計算機科学コース入学，現在に至る．CPS，組込みシステム，マルチ・メニーコアのシステム基盤に興味を持つ．2013 年山下記念研究賞．IEEE 会員．



安積 卓也 （正会員）

立命館大学情報理工学部助教．2009 年名古屋大学大学院情報科学研究科情報システム学専攻博士後期課程修了．2008～2010 年日本学術振興会特別研究員．2011～2012 年カリフォルニア大学アーバイン校客員研究員．2009 年より現職．リアルタイムシステム，組込みシステム向けのコンポーネントシステムの研究に従事．博士（情報科学）．IEEE，日本ソフトウェア科学会，電子情報通信学会各会員．

西尾 信彦 （正会員）

1986 年東京大学工学部計数工学科数理工学コース卒業．1988 年同大学大学院理学系研究科情報学専攻修士課程修了．同博士課程単位取得退学後，1992 年より（有）アクセス研究開発室，1993 年より慶応義塾大学環境情報学部および政策・メディア研究科に勤務．博士（政策・メディア）．2003 年より立命館大学に勤務．現在，同大学情報理工学部教授．2000～2004 年 JST さきがけ研究 21「協調と制御」領域研究者．2007～2008 年 Google Inc. Visiting Scientist．自立分散協調システム，ユビキタス・コンピューティングとセンシングネットワークの研究開発に従事．1994 山下記念研究賞．ACM，IEEE 各会員．

加藤 真平 （正会員）

2004 年慶應義塾大学理工学部情報工学科卒業．2008 年同大学大学院理工学研究科開放環境科学専攻後期博士課程修了．博士（工学）．同年慶應義塾大学理工学部特別研究員．2009 年東京大学大学院情報理工学系研究科特別研究員．同年カーネギーメロン大学訪問研究員．2011 年カリフォルニア大学研究員．2012 年より名古屋大学に勤務．現在，同大学大学院情報科学研究科准教授．オペレーティングシステム，サイバーフィジカルシステム，並列分散システムの研究開発に従事．