GPU 制御用ファームウェア開発環境

藤 居 祐 輔† 安 積 卓 也†
西 尾 信 彦† 加 藤 真 平††

多くの GPU はオンチップのマイクロコントローラに実装されたファームウェアによって制御されている．このマイクロコントローラを効果的に使用することで，データ転送やカーネル実行に関する GPU 資源管理のさらなる改善が期待できる．しかし，コードの記述がアセンブリに限られるなど，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の生産性に問題がある．そのため本稿では，NVIDIA 社製 GPU マイクロコントローラ向けコンパイラおよびデバッグ支援ツールを開発し，開発環境として提供することで，GPU 資源管理の新たな方向性を示す．コンパイラの実装には，移植性の高いコンパイラ基盤として知られる LLVM を用いた．一方，デバッグ支援ツールは，マイクロコントローラ上でファームウェアを実行し，コマンドの送信や監視をおこなう．本開発環境を用いて NVIDIA 社製 GPU マイクロコントローラのファームウェアを開発し，評価をおこない，実行時間のオーバヘッドは 2.3%であった．さらに，メモリコピーなどを含めた全体の時間におけるオーバヘッドの割合は 0.01%と，許容範囲内に収まることを確認した．

Development Environment for GPU Control Firmware

Yusuke Fujii,† Takuya Azumi,† Nobuhiko Nishio†
and Shinpei Kato††

GPUs are often controlled by firmware running on the integrated on-chip microcontroller. This microcontroller is highly available to extend the functionality of GPU resource management relevant to GPU code executions and data transfers. However, GPU firmware development environments are not yet well organized in the literature. In this paper, we develop a compiler and debugging tool set for NVIDIA's GPU microcontrollers in order to enhance the productivity of GPU firmware development. In particular, we implement the compiler using the well-known portable LLVM compiler infrastructure, while together providing a debugging subsystem that can individually execute firmware on the microcontoroller, monitor its status, and send firmware commands. As a proof of concept, we develop new firmware using our compiler tool set and evaluate it on an NVIDIA's graphics card. Our experimental results demonstrate that the overhead of introducing our firmware is suppressed to within 2.3%, as compared to the native proprietary firmware. Furthermore, the overhead occupy the 0.01% of total time including memory copy and so forth.

1. はじめに

GPU（Graphic Processing Unit）は，大量のプロセッサコアが搭載されており，従来のプロセッサと比べ，大幅に性能やエネルギー効率に優れている．近年の CPU と GPU の性能比を**表 1** に示す．同時期に発売された，“Intel Core i7 3960X” と “NVIDIA GeForce GTX680” で比較する．前者は 158.4GFLOPS，後者は 3090GFLOPS と，GPU の方が FLOPS が圧倒的に高い．さらに，1watt あたりの FLOPS においても，前者は 0.57（GFLOPS/watt），後者は 15.85（GFLOPS/watt）とエネルギー効率に優れている．GPU はグラフィックス処理が主である認識が強いが，CUDA や OpenCL などの GPGPU（General Purpose-GPU）という GPU の優れた処理能力を，汎用的な処理に用いる方式が注目されている．これらの優れた処理能力や汎用的な処理に用いられることから，GPU はスーパーコンピュータなどの高性能なコンピュータや組込みシステムに利用され始めており，今後も利用範囲は広がると予測される．近年では，参考文献 [1, 2] といった GPU 資源管理に関する研究など，GPU に関する研究が盛んにおこなわれている．

GPU での処理には，その構成から様々な問題が存在する．多くの GPU は，PCI（Peripheral Component

† 立命館大学
Ritsumeikan University
†† 名古屋大学
Nagoya University

ⓒ 2012 Information Processing Society of Japan

表 1 Intel CPU と NVIDIA GPU の比較
Table 1 Comparison of the Intel CPU architectures and the NVIDIA GPU architectures

	Core i7 980XE	Core i7 3960X	GeForce GTX285	GeForce GTX480	GeForce GTX680
プロセッサコア数	6	6	240	480	1536
単精度性能（GFLOPS）	108.0	158.4	933.0	1350.0	3090.0
メモリ帯域幅（GB/sec）	37.55	51.2	159.0	177.0	192.2
消費電力（watt）	130	278	183	250	195
発売時期	2010/03	2011/11	2009/01	2010/04	2012/03

Interconnect）によってホストに接続され，デバイスとして動作する．そして，ホスト側に実装されるデバイスドライバがGPUに処理を依頼し，GPUが処理をおこなう．この時に起きる問題として，スケジューリングの問題がある．ホスト側は，GPUへ処理の依頼しかできないため，処理中のジョブに対しての指示ができない．そのためプリエンプションを含めたスケジューリングや，異常なジョブなどによって暴走するGPUを止める事ができない．さらに，ホスト側が指示をするためGPUでの処理にも関わらず，CPU負荷に影響を受けやすい問題もある．GPUでの処理の効率化として，TimeGraph [3] などを提案しているが，こちらはあくまでホスト側でのアプローチを用いているため上記問題は解決できない．そのため我々は，GPUにオンチップで搭載されるマイクロコントローラのファームウェア開発によって，GPUカーネル実行やデータ転送に関する，より粒度の細かい資源管理を可能にし，効率化を目指している．具体的には，マイクロコントローラ上でのスケジューリングや，参考文献 [5] で挙げられる，DMA（Direct Memory Access）の効率的な利用によるデータ転送の効率化などである．

本研究は，FARM（Firmware As Resource Manager）[6] プロジェクトとして進めている．FARMプロジェクトを進めるにあたって，ファームウェア開発に関する問題が発生している．現状では，マイクロコントローラのファームウェアの開発環境として公開されているものはアセンブリに限られ，デバッグ環境もないため，ファームウェア生産性が低い．

本稿では上記の問題を解決するため，GPU制御用ファームウェア開発環境を提供し，ファームウェアの生産性の向上を目指す．本開発環境は，NVIDIA社製GPUマイクロコントローラ向けのコンパイラ，および，デバッグ支援ツールが含まれる．コンパイラの実装には，移植性や最適化に優れたコンパイラ基盤のLLVMを用いる．デバッグ支援ツールはマイクロコントローラ上でファームウェアを実行し，コマンドの送信や監視をおこなうツールを実装する．そして，本開発環境を用いてNVIDIA社製GPUマイクロコントローラ向けのファームウェアを開発し，NVIDIA標準のファームウェアと，本稿で開発したファームウェアの性能を比較する．

本稿は，全6章で構成される．2章では，本稿において基盤となるプラットフォーム技術についてまとめる．3章では，実際の設計と実装について述べ，4章では本開発環境を用いて，開発したファームウェアを用いて評価をおこなう．5章では本稿と関連の深い研究を挙げ，比較をおこなう．そして最後に6章で結論を述べる．

2. 基盤プラットフォーム技術

本章では，本開発環境において，基盤となる技術についてまとめる．

2.1 GPUマイクロコントローラ向けアセンブラ

現在，GPUマイクロコントローラのファームウェアの開発環境には，アセンブラ，デバイスドライバが提供されている．これらは，PathScale [7] がオープンソースで提供している．このうち，アセンブラはenvytools [8] と呼ばれる，パッケージに含まれている．

2.2 GPUデバイスドライバ

GPUデバイスドライバは，CPU側の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．具体的には，デバイスドライバが，コマンドやデータをマイクロコントローラに送信し，それを受け取り，マイクロコントローラがGPU制御をおこなう．このインタフェースをランタイムライブラリに提供するため，アプリケーションがGPUを用いた処理が可能になる．

GPUマイクロコントローラのファームウェアは通常のマイクロコントローラと違い，ROMに焼く形ではなく，デバイスドライバに組み込み，OSが起動し，デバイスドライバが読み込まれた時点でロードされ，実行する形式である．したがって，開発したファームウェアを実際にGPU制御に用いるには，デバイスドライバに組み込む必要がある．そのためには，オープンソースのデバイスドライバが必要になる．現在オープンソースで提供されているデバイスドライバには，PathScaleの提供するPSCNV [9] と，Linuxに標準で組み込まれているX.Org Foundation [10] とFreedesktop.org [11] が進めているプロジェクトのNouveau [12] がある．

ⓒ 2012 Information Processing Society of Japan

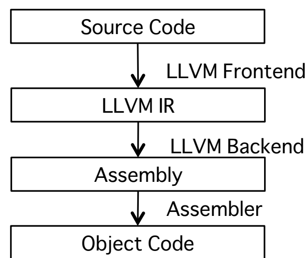

図 1 LLVM でのコンパイルの流れ
Fig. 1 Compilation stages of LLVM

表 2 GF100 に搭載されるマイクロコントローラの仕様
Table 2 Specification of GF100 Microcontroller

Name	HUB	GPC
Architecture	Fermi	Fermi
Number	1	4
Clock	270MHz	50MHz
TimerClock	135MHz	25MHz
Bit	32bit	32bit
Code size	16,384 byte	8,192 byte
Data size	4,096 byte	2,048 byte

2.3 LLVM

LLVM（Low Level Virtual Machine）は，現在も開発が進められているコンパイラ基盤であり，オープンソース☆で公開されている．LLVM の主な特徴は，コンパイルのすべての工程で最適化がおこなわれ，拡張性に優れており，さらに移植性に優れている．LLVM はコンパイラに必要な機能がモジュール化され，各機能を統合するドライバで構成されている．

LLVM では**図 1** のとおり，ソースコードから，LLVM IR という中間言語を生成する．そして中間言語から LLVM バックエンドを通して，アセンブリコードを生成する．最後にアセンブリコードからオブジェクトコードを生成する．LLVM を利用するメリットとしては，作業の効率化が挙げられる．高性能なコンパイラを構築する作業はあまりにも膨大な量である．しかし，LLVM ではすでに 10 年もの歳月をかけ，コンパイラ基盤を構築しており，これを利用し開発環境構築の効率化を目指す．

2.3.1 LLVM IR

LLVM IR（Intermediate Representation）は，LLVM アセンブリ言語，Bitcode などとも呼ばれ，LLVM で用いられる中間言語である．この中間言語は，表現力，拡張性，軽量かつ低レベルであることを追求しており，SSA（Static Single Assignment）ベースの表現である．SSA ベースの表現を用いており，様々なコンパイラ最適化アルゴリズムを実現できる．

2.3.2 LLVM フロントエンド

LLVM において，高級言語から LLVM 中間言語を生成する部分をフロントエンドと呼ぶ．C， C++，Ruby，Phython や D 言語など様々な高級言語に対応したフロントエンドが用意されている．フロントエンドは自由に選択でき，開発の幅が広がっている．我々は，普及性の高さや，Linux でのネイティブアプリケーション，カーネル，デバイスドライバ開発などに使われていることから，コード記述には C 言語を選択する．よって，LLVM のフロントエンドには，C 言語をサポートする Clang を用いる．

- **Clang** C， C++， Objective-C， Objective-C++言語を対象として開発されている LLVM のフロントエンドである．2012 年 7 月時点では，C 言語に関しては開発完了しているが，残りの C++ や Objective-C などはまだ開発中である．ただしアップル社が支援しているとともに，FreeBSD では 2012 年 5 月時点で GCC を Clang と LLVM に置き換えると発表しており，今後の発展に多いに期待できるツールチェーンである．

2.3.3 LLVM バックエンド

LLVM において，LLVM IR から対象とするマシンの命令セットに合わせたアセンブリコード，バイナリコードを生成する部分をバックエンドと呼ぶ．コードの生成部は対象とするマシン仕様に依存しない形になっており，対象マシンの仕様が記述されたライブラリを読み込み，様々な命令セットのコードの生成が可能としている．ライブラリには，MIPS や X86， ARM などが標準で添付されている．我々は，静的にコードを生成する LLVM バックエンドの LLC（LLVM static Compiler）を用いて，NVIDIA 社製マイクロコントローラに合わせたコードを生成する．

3. GPU 制御用ファームウェア開発環境の設計・実装

本章では，GPU マイクロコントローラのファームウェア開発の生産性向上を実現するための，開発環境の設計，実装について述べる．

3.1 GPU マイクロコントローラ

本研究では，NVIDIA の Fermi アーキテクチャを採用した GPU に搭載されているマイクロコントローラを用いる．GF100（GeForce GTX480）を例に解説する．GF100 では，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 基で一つの GPC（Graphics Processing Cluster）が構成され，ROP（Rendering Output Pipeline, Raster Output Processor）という，コアで処理したデータをビデオメモリに書き込む部分

☆ http://llvm.org/releases/3.1/LICENSE.TXT

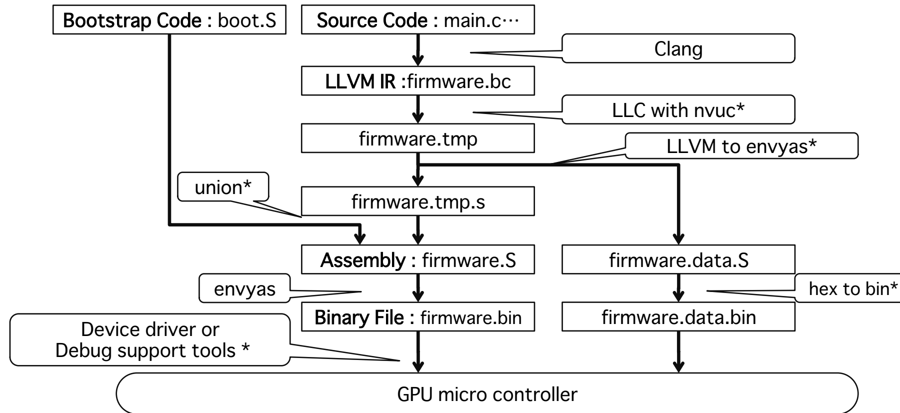

図 2 GPU マイクロコントローラ向けコンパイラ全体図
Fig. 2 Overview of Compiler Implementation

が，4 基の GPC で共用されている．この GPC に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 GPC と呼んでいる．一方，この GPC を統括しているマイクロコントローラが搭載されており，これを HUB と呼ぶ．この HUB と GPC によって CUDA プログラム等の実行に利用される GPU コンテキストを制御する．HUB と GPC 以外にも，様々なマイクロコントローラがあり，命令セットは同一である．このマイクロコントローラのレジスタは MMIO（Memory-mapped I/O） で，CPU 側のメモリ空間にマッピングされている．**表 2** で表すとおり，マイクロコントローラのファームウェアのコードサイズは非常に限られるため，ファームウェアはサイズを考慮した設計をおこなわなければならない．

3.2 GPU マイクロコントローラ向けコンパイラ

GPU マイクロコントローラ向けコンパイラは，C 言語コードから，GPU マイクロコントローラの命令セットに合わせたバイナリコードを生成する．

3.2.1 全体の流れ

GPU マイクロコントローラ向けコンパイラは LLVM で実装した．GPU マイクロコントローラ向けコンパイラの全体図を**図 2** に示す．大きな流れとしては，Clang で C コードから LLVM IR の生成をおこない，LLC で LLVM IR からアセンブリコードを生成する．そしてアセンブリコードをコード部とデータ部に分割し，コード部に起動用のコードを結合し，ラベルとアドレスの置換をおこなった後に，envyas で実行ファイルを生成する．実行ファイルは，デバイスドライバかデバッグ支援ツールを用いて実行できる．本開発環境を利用し，開発をおこなう場合，開発者は C コードを記述するのみでよい．各名前の横の “ * ” は我々が追加した部分を示す．追加した規模はターゲットマシンの追加に C++言語で 4,270 行であり，そのほかのスクリプトが Perl 言語で 384 行である．

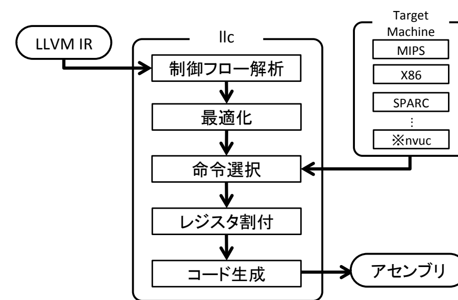

図 3 LLC のコード生成の流れ
Fig. 3 Code generation stages of LLC

(1) Clang Clang は 2.3.2 項で解説したとおり，LLVM のフロントエンドであり，C コードから LLVM IR を生成する．

(2) LLC with nvuc* LLC は 2.3.3 項で解説した LLVM のバックエンドである．LLC の流れを**図 3** に示す．LLC では LLVM IR を入力として受け取り，制御フロー解析，最適化，命令選択，レジスタ割付，コード生成という流れで実行し，アセンブリコードを出力する．この流れは共通化されており，ターゲットマシンの違いにはとらわれない．命令選択の時点で，ターゲットマシンの設定を読み込み，各々の仕様に合わせた命令，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コードを出力できる．このターゲットマシンの設定に，NVIDIA 社製 GPU マイクロコントローラを追加した．この追加した設定を nvuc （NVidia MICRO Controller） としている．

(3) LLVM to envyas* （2）で生成されたアセンブリコードは，コード部（firmware.tmp）と，データ部 （firmware.data.S） で別ファイルに分割する．これは，デバイスドライバの仕様に合わせている．ここで分割したコード部に対して，図 2

ⓒ 2012 Information Processing Society of Japan

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 4 C 言語のソースコードと生成されるコード，左：C 言語，中央：LLVM IR，右：アセンブリ
Fig. 4 C source code and generated code. Left : C, Center : LLVM IR, Right : Assembly

における union で BootstrapCode を結合する．BootstrapCode には main 関数の呼び出し，割込みハンドラの設定などが記述されている．次の（4）envyas では，コード部に用いているラベルのデータが別ファイル存在するため，アドレス先が参照できない．そのためこの時点でコード部のラベルをデータ部のアドレスに置換する．

(4) envyas envyas は 2.1 節で解説した envytools に含まれる，GPU マイクロコントローラ向けアセンブラである．(3) で生成されたコード部を入力とし，実行ファイルを出力する．

(5) hex to bin* （3）で分割したデータ部を実行ファイルへと変換する．

(6) 実行* 実行は，デバイスドライバに実行ファイルを組み込むか，デバッグ支援ツールを用いる．デバイスドライバに実行ファイルを組み込む際には，オープンソースである PSCNV を用いる．デバイスドライバ，デバッグ支援ツールは起動時にファームウェアのバイナリファイルを読み込み，マイクロコントローラに書き込み実行する．

3.2.2 生成されるコード

C コードの例と，C コードから生成される LLVM IR，アセンブリコードを**図 4** に示す．左から順に，C コード，3.2.1 項（1）で生成される LLVM IR，3.2.1 項（3）で生成されるアセンブリのコード部である．

3.3 デバッグ支援ツール

デバッグ支援ツールは，マイクロコントローラへのファームウェアの書き込み，コマンド・データの送信，レジスタの値の表示をおこなう．**図 5** に本ツールの流れを示し，各機能について解説する．

(1) ファームウェアの書き込み HUB と GPC ともに，コンパイラで生成したファームウェアのバイナリコードを，MMIO でマッピングされたアドレス空間に書き込む．書き込みが完了した後に，指定されたレジスタにフラグをセットすると，このファームウェアが実行される．

(2) コマンド・データの送信 通常マイクロコントローラは，実行開始し，初期化完了後，コマンドを受け付けるまで待機状態になる．このコマンド・データの送信は，指定されたレジスタにコマンドとデータを書き込み，割込みを発生させ，コマンド・データの送信がおこなわれ，マイクロコントローラの処理が再開される．

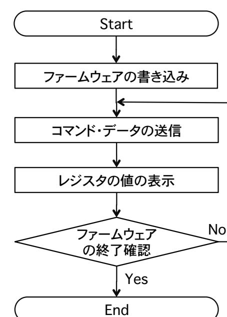

図 5 デバッグ支援ツールの流れ
Fig. 5 Flowchart of our debugging support tool

(3) レジスタの値の表示 マイクロコントローラのレジスタの値を表示し，実行中の状態を把握する．

3.4 ファームウェア開発

本節では，本開発環境を用いて開発した，GPU マイクロコントローラのファームウェアのうち，HUB のファームウェアについてまとめる．HUB のファームウェアのフローチャートを**図 6** に示す．ファームウェア実行のフラグがセットされると，ファームウェアは実行される．ファームウェアの大まかな流れは，START から開始，initialize，sleep，ihbody，work の順におこない，sleep から work までを繰り返す．

(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おこなわれる．そして，全 GPC の初期化完了のフラグを確認し，HUB 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．コマンドが送信されると，マイクロコントローラは割込みを発生させ，“ihbody” を起動する．

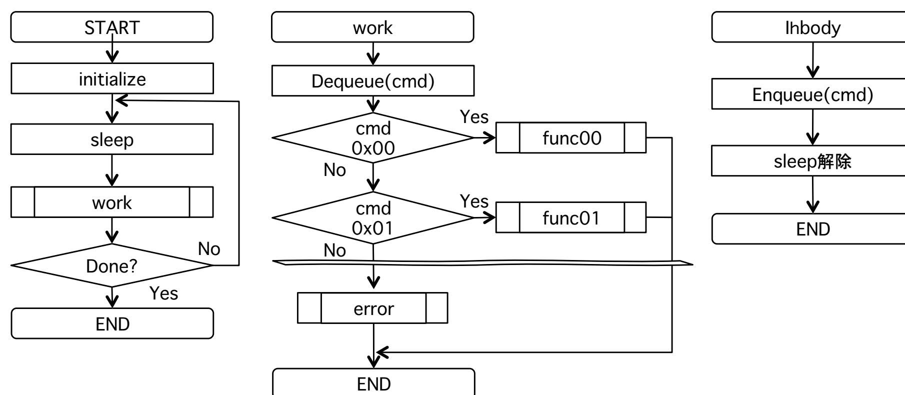

図 6 開発したファームウェアのフローチャート
Fig. 6 Flowchart of our basic firmware

表 3 評価環境
Table 3 Experimental setup

CPU	Intel core i7 2600
GPU	NVIDIA GeForce GTX480
Memory	8GB
Kernel	Linux 2.6.42.12-1.fc15.x86_64
Device driver	PSCNV

(3) ihbody ihbody では送信されたコマンドをエンキューし，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，デキューをおこない，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．

以上のとおり，ファームウェアは，コマンドを受け付け，それに合わせた機能を実行し，GPU の制御をおこなっている．本 HUB ファームウェアの規模は，C 言語コードで 2,374 行あり，生成後のアセンブリコードで 6,911 行である．

4. 評　価

本稿における評価は，NVIDIA のファームウェアと，我々のファームウェアの性能の比較をおこない，本開発環境の有効性を示す．評価環境は**表 3** のとおりである．

4.1 実行時間計測

本評価ではオープンソースとして開発されている CUDA のランタイム，および，資源管理エンジンのセットである Gdev [4] [13] を用いて，NVIDIA のファームウェアと開発したファームウェアとで Gdev のサンプルプログラムを実行した時の，実行時間を計測し性能を比較する．本評価における実行時間は，ホスト側からデバイス側へのデータのコピー，処理の実行，デバイス側からホスト側へのデータのコピーにかかった時間である．

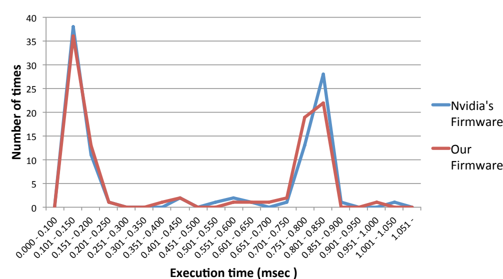

図 7 実行時間の分布
Fig. 7 Execution time distribution

Gdev サンプルプログラムは基本的な計算をする，madd，mmul，loop，fmadd の 4 つを利用する．madd は行列の加算，mmul は行列の乗算，loop は繰り返し処理，fmadd は小数値を成分として持つ行列の加算である．まず，madd プログラムの 100 回の計測結果の分布を**図 7** に示す．縦軸は，発生回数を表し，横軸は実行時間を表す．この図から，実行時間のばらつきが見られ，2ms 以下，7ms 以上に集中している．NVIDIA のファームウェアと我々のファームウェアとでの発生比率もほぼ同じであった．これは GPU のプログラムは実行時間に関して，影響を与えるものが CPU で実行するものに比べ多数存在するため，ばらつきが発生すると考えられる．具体的に影響を与えるものにはホストメモリ，デバイスドライバ，データバス，GPU のキャッシュやメモリなどが挙げられる．そのため比較するデータを，2ms 以下はケース A，7ms 以上はケース B とし，それぞれで平均を取り比較する．ケース A の結果を**図 8** に，ケース B の結果を**図 9** に示す．横軸は，サンプルプログラム名を表し，縦軸は，実行時間（ms）を表す．図 8, 図 9 から見れるように，

ⓒ 2012 Information Processing Society of Japan

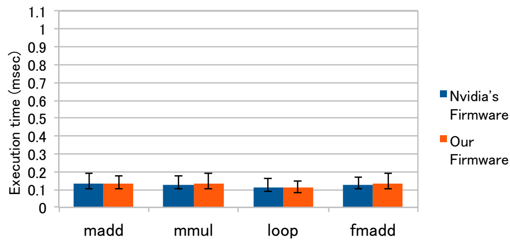

図 8　Gdev サンプルプログラムの実行時間：ケース A
Fig. 8　Execution time of microbenchmark programs : Case A

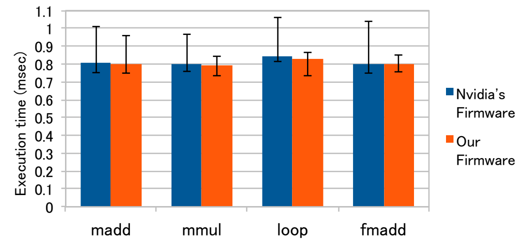

図 9　Gdev サンプルプログラムの実行時間：ケース B
Fig. 9　Execution time of microbenchmark programs : Case B

オーバヘッドは少ない．オーバヘッドが最大なのは，ケース A における，madd の 0.003 ms で，2.31%であった．一方でケース B の loop では，0.002 ms の速度の向上が見られた．さらに，どちらのファームウェアにおいても実行時間に対して誤差が大きい．

さらに，アプリケーションとして，GPU を使用する場合には，GPU 自体が処理をおこなう時間が増えるため，ファームウェアにおける実行時間のオーバヘッドは，影響が少なくなる．たとえば，NVIDIA のファームウェアにおける，madd の全体の時間は 21.842ms であった．ここでの全体の時間は，ホスト側でプログラムを実行し，終了するまでの時間である．全体の時間に対して，ファームウェアにおける実行時間のオーバヘッドは 0.01%と比較的小さい．我々のファームウェアにおける，madd の全体の時間についても，20.543ms となり，NVIDIA との差異はあまり見受けられなかった．したがって次の理由により，我々のファームウェアのオーバヘッドは許容範囲内である．

- 今回観測された実行時間の差異は，誤差よりもはるかに小さい．
- 実行時間の差異は全体の時間に対してはるかに小さくなる．

4.2　GPU コンテキストの生成時間

前述の全体の時間には，GPU コンテキストの生成，メモリ確保やメモリ解放などにかかる時間が含まれている．NVIDIA 標準ファームウェアにおける madd の，1 回目実行時における全体の時間は 214.056ms であり，2 回目以降の実行時における全体の時間は 21.842ms と，大幅に差がある結果が出た．さらに，我々の開発したファームウェアにおいても同様の現象が見られた．この要因としては，1 回目の実行時に GPU コンテキストを生成しており，2 度目以降は 1 度目に生成した GPU コンテキストを用いるためである．GPU コンテキストは，CPU のスレッドに関連付けられ，GPU コンテキストに関連付けて，メモリ割り当てや GPU での処理をおこない，マルチスレッディングを実現しているもので，アプリケーションの最初の実行時に生成される．つまり，GPU コンテキストの生成に時間がかかっており，マルチスレッドな処理をおこなうためには，無視できない量であり，今後，考慮していく必要がある．この問題は，実際にファームウェアを開発し，評価をしなければ得られない知見である．

5. 関 連 研 究

本章では，本稿と関連の深い研究について紹介し，比較をおこなう．

5.1　Helios: Heterogeneous Multiprocessing with Satellite Kernels

Helios [14] は，異なるアーキテクチャや異なる特性を持った CPU に大して透過的に利用できる OS を提供する取り組みである．Helios では，GPU や NIC といった，プログラマブルデバイスが出てきつつあるとし，高性能なベクター処理や，高速通信にはこうしたプログラマブルデバイスを用いる必要があるとしている．GPU や NIC では，デバイスドライバを経由してその機能を活用しており，この形式では CPU とデバイス間の通信によって転送可能なデータ量は限られてしまう．さらに，デバイスドライバ自体の複雑化や，デバイス上で，あるタスクを動作させるためのインタフェースの提供もされていない問題点が挙げられている．解決手法として，Helios と呼ばれる OS を提案している．Helios はサテライトカーネルと呼ばれるマイクロカーネルを持ち，サテライトカーネルは，スケジューラ，メモリマネージャ，ネームスペースマネージャ，カーネル間コミュニケーションなどの規則を持っている．Helios は，我々が目指すものと方向性は同じであり，Helios が NIC に対してアプローチをおこなったのに対し，我々は，GPU に対してアプローチをおこなう．上記の CPU とデバイス間の通信量の制限，デバイスドライバ自体の複雑化，デバイス上でのタス

ク動作のためのインタフェースといった問題は，本開発環境により，GPU マイクロコントローラのファームウェア上に実装が可能になり，解決可能になる．

5.2 メニーコア並列計算機における通信機構の設計

DCFA（Direct Communication Facility for many-core based Accelerators [15] は，ホストを介さず，デバイス同士が直接通信し，デバイス ⟩ ホスト ⟩ 別ホスト ⟩ 別デバイスといった，ホストを介する無駄を省いている．しかし，現状では GPU においてはデバイスアドレスが分からないため適用できないとしている．

我々の開発環境を利用し，マイクロコントローラがデータ転送制御をすると GPU においても，ホストを介さない通信がおこなえる．

6. 結　　論

本稿では，既存研究よりも粒度の細かい資源管理をおこなうために，GPU マイクロコントローラを用いることを提案．開発環境として，NVIDIA 社製の GPU を制御するオンチップのマイクロコントローラ向けのコンパイラとデバッグ支援ツールを開発した．さらに，ファームウェアを開発し，NVIDIA のファームウェアと比較した．Gdev のサンプルプログラムでの最大のオーバヘッドで 2.31%以下に収まり，誤差や全体の実行時間との比較と併せて，オーバヘッドが許容範囲内であることを示した．これにより本開発環境によって開発するファームウェアは有効であり，コンパイラやデバッグ支援ツールによって開発の手間が削減できる事から，本開発環境は有効である．加えて，本開発環境をオープンソースで開発，公開 [6] し，生産性の向上，研究の活発化についても期待する．さらに，ファームウェアの開発，評価をおこなうことで，GPU コンテキスト生成にかかるオーバヘッドが大きいことを発見した．

今後，我々は，本開発環境を用いて，1 章で述べたマイクロコントローラを用いた GPU 資源管理を進めていく．さらに，4 章で得られた，GPU コンテキストの生成にかかるオーバヘッドに関しても，ファームウェアを改良すると良い結果が得られると予想されるため，取り組むべき課題である．

本開発環境の提供によって，将来的な環境を想定した研究がおこなえる点も考慮できる．GPU は現在，CPU とは別のデバイスに搭載されている．しかし，Intel 社の Ivybridge，Haswell や Larrabee などを見ると，将来的に GPU は CPU にオンチップになると予想される．この時，GPU の制御は CPU でおこなうと考えられる．マイクロコントローラを CPU と想定すると，GPU が CPU にオンチップになった環境を想定した研究もおこなえる．

謝辞 envytools や PSCNV という優れたソフトウェアをオープンソースで公開し，本論において多大な貢献を与えてくれた PathScale，LLVM という優れたインフラストラクチャを提供する LLVM Team に謹んで感謝の意を表する．

参 考 文 献

1) Bautin, M., Dwarakinath, A. and Chiueh, T.: Graphic engine resource management, *Proceedings of SPIE*, Vol. 6818, p. 68180O (2008).
2) Dwarakinath, A.: A Fair-Share Scheduler for the Graphics Processing Unit, Master's thesis, Computer Science , Stony Brook University (2008).
3) Kato, S., Lakshmanan, K., Rajkumar, R. and Ishikawa, Y.: TimeGraph: GPU scheduling for real-time multi-tasking environments, 2011 USENIX Annual Technical Conference (USENIX ATC ' 11), p. 17 (2011).
4) Kato, S., McThrow, M., Maltzahn, C. and Brandt, S.: Gdev: First-class GPU resource management in the operating system, *USENIX ATC*, Vol. 12 (2012).
5) Kato, S., Brandt, S., Ishikawa, Y. and Rajkumar, R. R.: Operating Systems Challenges for GPU Resource Management, *OSPERT 2011*, p. 21 (2011).
6) Fujii, Y., Azumi, T. and Kato, S.: Farm Project., `http://sys.ertl.jp/farm/` (2012).
7) PathScale: ENZO, `http://www.pathscale.com`.
8) PathScale: Envytools, `https://github.com/pathscale/envytools`.
9) PathScale: PSCNV GPU Device Driver, `http://www.pathscale.com`.
10) Foundation, X.: X.org, `http://www.x.org/`.
11) Freedesktop.org: Freedesktop.org, `http://www.freedesktop.org/`.
12) Freedesktop.org and Foundation, X.: Nouveau, `http://nouveau.freedesktop.org/`.
13) Kato, S.: Gdev Project., `http://sys.ertl.jp/gdev/` (2012).
14) 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).
15) Si, M. and Ishikawa, Y.: Design of Direct Communication Facility for Manycore-based Accelerators, Technical Report 16, CASS2012 in conjunction with IPDPS2012 (2012).

ⓒ 2012 Information Processing Society of Japan