MPI-OpenCL ハイブリッド並列を実現するプログラミング・フレームワークの設計と実装

土居意弘[†] 村瀬正名[††] 前田久美子[††]
小松秀昭[††] 野田茂穂[†††] 姫野龍太郎[†††]

異種のプロセッサが混在するヘテロジニアス計算システムにおけるプログラム開発は，アルゴリズムの並列化，アーキテクチャごとの最適化やデータ通信などのプログラミング上の課題がある．これらの課題に対し，我々はヘテロジニアス・システム向け並列プログラミング・フレームワークを提案している．本フレームワークでは，サブルーチンを特定アーキテクチャに最適化実装した「部品」を，ドラッグ・アンド・ドロップ操作で接続するだけで複雑なアプリケーションを記述し，実際のシステムに合わせた部品の並列化，演算リソースの割り当ておよび MPI 通信コードの生成をコンパイラが自動的に実現する．本稿では，リソース・バンドルという概念を導入し，OpenCL や OpenMP のような自動並列化ライブラリによる並列化と我々のフレームワークによるコンパイラの自動リソース割り当ての共存を可能にした．IBM POWER7，Intel Core i7 X980，NVIDIA Tesla C2050 の 3 種類のプロセッサが混在したヘテロジニアス・システムにおけるレイトレーシング法による並列レンダラーに適用し，1920 × 1080 サイズのレンダリングを 23.5 フレーム/秒で実行できた．

Design and Implementation of a Programming Framework achieving MPI-OpenCL Hybrid Parallelization

Munehiro Doi,[†] Masana Murase,[††] Kumiko Maeda,[††]
Hideaki Komatsu,[††] Shigeho Noda[†††] and Ryutaro Himeno[†††]

We have difficulties in programming parallel applications for heterogeneous systems, because it requires parallelization of algorithms, optimizations for each architecture, and network programming to exploit the performance from such systems. To address these problems, we have proposed a parallel programming framework for heterogeneous systems, where an application programmer constructs workflows of “components” that are subroutines optimized for each architecture. Our compiler determines a data decomosition policy, assigns computational resources to each component, and generates network code. This paper extends our framework to handle both conventional MPI-based parallelization and language-level parallelizaiton such as OpenCL or OpenMP. We implement a ray-tracing OpenCL component running in parallel on a heterogeneous system with IBM POWER7, Intel Core i7 X980, and NVIDIA Tesla C2050. With our framework, the compiler decomposes a rendering area and determines workloads to each architecture. As a result, we achieved 23.5 fps for rendering a 1920×1080 image.

1. はじめに

マルチコア・システムのさらなる性能向上のため，近年は特定の演算で高い性能を発揮するアクセラレータを組み合わせたヘテロジニアス・システムと呼ばれる形態が，注目を集めている．例えば RIKEN Integrated Cluster of Clusters （RICC）[1] では，Intel Xeon，NVIDIA Tesla，MDGRAPE3 など多様なプロセッサを統合している．ヘテロジニアス・システムでは，アプリケーション開発者はノード間，ノード内において異なった並列プログラミングを強いられる．特に，ノード間はネットワーク通信を行い，ノード内ではスレッド並列を行うハイブリッド並列と呼ばれる手法の場合，例えば，ノード間は MPI，ノード内ではマルチコア CPU 向けには pthread による明示的な並列化または OpenMP による自動並列化，GPU 向けには CUDA，OpenCL と多岐に渡り，ヘテロジニ

† 日本アイ・ビー・エム株式会社 システムズ&テクノロジー・グループ
Japan STG Laboratory, IBM Japan, Ltd.
†† 日本アイ・ビー・エム株式会社 東京基礎研究所
IBM Research - Tokyo, IBM Japan, Ltd.
††† 理化学研究所 情報基盤センター
Advanced Center for Computing and Communication, RIKEN

© 2012 Information Processing Society of Japan

アス・システムの利用には，多様な並列プログラミング技術に習熟する必要があった．

こうした背景から，我々は計算資源の配分を自動決定し，通信コードを自動生成するプログラミング・フレームワークを提案している[2),3)]．本フレームワークの 2-レベル・プログラミング・モデルでは，熟練開発者が再利用可能なアルゴリズムを単一のマイクロアーキテクチャに最適化された部品としてプログラミングし，一般開発者はドラッグ・アンド・ドロップ操作で部品をつなぎあわせることで，より複雑なアプリケーションを記述する．事前プロファイリングによって得られた部品の特性から，各部品をどのマイクロアーキテクチャで実行するか，部品をどの程度データ並列させて実行させるか，部品間のデータ通信（MPI，メモリコピー，ポインタ渡し，PCIe 通信）には何を使うかを我々のコンパイラが自動決定する．

しかし，我々のフレームワークでは部品はシングル・スレッド用に設計されていると想定しているため，部品のコードを例えば OpenMP を利用して書いた場合，フレームワークが想定しないスレッドが生成されて，本来利用可能なハードウェア・スレッドの数よりも多くなってしまうという課題があった．

本稿では，我々が開発してきたフレームワークにリソース・バンドルという概念を導入し，部品の実装手段として OpenCL の適用を試みる．リソース・バンドルは複数の計算リソースをまとめたもので，部品には常にリソース・バンドルの単位で計算リソースが割り当てられる．ハードウェア・コンフィグレーションで計算リソースをリソース・バンドルとして定義しておくことで，たとえば OpenCL を呼び出すルーチンに複数の CPU コアを一括して割り当てることが可能になる．実際にスレッドにどのリソース・バンドルが割り当てられるかは，ハードウェア・コンフィグレーションでの定義，プログラム部品の要求，ノードの処理能力，部品に与えるデータサイズおよび計算時間によって，我々が開発中のコンパイラが自動決定する．さらに，部品開発者は，リソース・バンドルの情報を実行時に取得できる．これにより，例えば OpenCL カーネル分割パラメータの，アーキテクチャ毎の調整が実現される．

本フレームワーク上にレイトレーシングレンダラーを OpenCL を用いて実装し，IBM POWER7，Intel x86，NVIDIA Tesla の 3 種類のプロセッサが混在したヘテロジニアス・システム上において，リソース・バンドルを用いた並列化とその性能を評価した．

本稿の構成は以下の通りである．第 2 章で従来技術について述べる．第 3 章でフレームワークの概要とプログラミング手順について解説し，第 4 でリソース・バンドルの導入について説明する．第 5 章でフレームワークをレイトレーシング法による並列レンダラーに適用した実験結果を報告し，第 6 章で本稿の総括と今後の課題を述べる．

2. 従来技術

並列プログラミング，特にネットワークプログラミングを簡便にする研究が行われている．UPC[4)]，Chapel[5)]，Fortress[6)]，X10[7)] は，Partitioned Global Address Space （PGAS）と呼ばれる並列プログラミングモデルに基づいたプログラミング言語である．PGAS プログラミングでは，複数の演算ユニット，メモリユニットを有するシステム全体のグローバルメモリ空間を仮定し，このグローバルメモリ空間を複数の論理メモリ空間に分割したアドレス空間を定義する．個々の論理メモリ空間が個別の物理プロセッサに関連付けられ，この論理メモリ空間に対して，単一あるいは複数の スレッドを割り付けることで，論理メモリ空間内の並列プログラミングおよび論理メモリ空間をまたがった並列プログラミングの両者が実現される．論理メモリ空間をまたがったデータアクセスについては，PGAS プログラミング言語が自動的にネットワークコードを生成するため，ユーザは MPI などのホスト間通信コードを明示的に記述する必要はない．しかし，各論理メモリ空間に，どの計算スレッドをいくつ割り付けるかは明示的に指定する必要がある．一方，我々のアプローチでは，事前のプロファイリング情報に基づいた各コンポーネントへのリソース・スケジューリングおよび通信コードの自動生成を行うため，ユーザはネットワーク通信およびリソース・マッピングの両プログラミングの習熟および実装が不要となる．

設樂[8)] らは，複数のホストをあたかも一台のホストであるかのように仮想化するレイヤを設けることで，単一の OpenCL プログラムを複数のホスト上で実行可能にしている．本仮想化レイヤ（仮想 OpenCL API）は，OpenCL のタスクキューを仮想化し，ホストプログラムは，ローカルキューと同じようにリモートホスト上のタスクキューにタスクを投入できるが，PGAS プログラミング言語同様に，ユーザはどのリモートホスト上でどのタスクを実行するかは，仮想 OpenCL API を通じて明示的に記述する必要がある．Virtual OpenCL[9)] では，リモートホストにタスクを投入するレイヤを OpenCL ライブラリとして実装したことにより，既存の OpenCL アプリケーションを変更せず

ⓒ 2012 Information Processing Society of Japan

にリモート実行できる．また，複数デバイスを並列実行するためには，各デバイスを別の OpenCL コンテキストとして扱うように書きなおすことが必要になるが，書き換えを容易にするため，C++ライブラリと OpenMP を拡張した Many GPUs Package（MGP）を提供している[10]．Virtual OpenCL はカーネルコードを実際にどのノードで実行するかはランタイムに決定されるが，我々のフレームワークでは通信相手を考慮してコンパイル時に決定する．

ストリーム・アプリケーション向けのプログラミング言語として，StreamIt[11] や SPADE[12] などがあるが，特定のハードウェア・アーキテクチャを前提としている．我々のフレームワークは，一般的なクラスタへの適用を目的としている．

3. フレームワークの概要

我々がこれまで提案してきた 2-レベル・プログラミング・モデルでは，特定ハードウェア向けの最適化ができる少数の熟練開発者によってアルゴリズムを部品化し，多くのアプリケーション開発者はその部品をドラッグ・アンド・ドロップ操作でつなぎ合わせてストリーム・グラフを作り，より複雑なアプリケーションを構築する[2),3)]（図 1）．構築されたアプリケーション・プログラムは，本フレームワークが提供するコンパイラによってリソース・スケジューリングおよび通信コードの生成が行われ，最終的に実行形式に変換される．以下，本フレームワークを用いたプログラムの開発手順について順に解説する．

3.1 2－レベル・プログラミング

プログラム開発の最初のステップで，プログラム部品となる UDOP（User Define OPerator）の定義と計算ルーチンを実装する．UDOP は特定のマイクロアーキテクチャには依存しない，機能とデータ入出力を定義しただけの抽象コンポーネントで，ストリーム・グラフの記述に用いる．UDOP を特定のマイクロアーキテクチャ向けに実装したものを KERNEL（カーネル）と呼ぶ．いずれの定義文も///で始まる数行の指示文からなる（表 1）．例えば，入出力データについてデータ型，データサイズ，ポート名を定義する必要がある．KERNEL の定義ではそれらに加えてアーキテクチャや使用するアクセラレータの指定等も行う．KERNEL の実装は C 言語，C++または FOTRAN などで作成可能で，定義されたデータ入出力ポート名と同じ名前の引数を持つ．1 つの UDOP に対して複数の KERNEL を実装することもできる．プログラム部品の粒度に制約はないが，機能的に完結していて再利用しやすいことが望ましい．KERNEL には，オプションとして初期化ルーチンと終了処理ルーチンを記述することもでき，割り当てられたリソースに合わせたパラメータ調整や，作業用メモリの取得・開放などを行うことができる．

表 1 UDOP および KERNEL 定義アノテーション
Table 1 UDOP and KERNEL Annotation

Annotation		Description
UDOP Annotation		
UDOP	[*udop_name*]	The name of a UDOP
IN (or '+')	[*type*] [*port_name*]	An input port for this UDOP
OUT (or '-')	[*type*] [*port_name*]	An output port for this UDOP
KERNEL Annotation		
KERNEL	[*kernel_prototype*]	The prototype of a kernel function
BASE (or '>')	[*udop_name*]	The base UDOP for this kernel
IN (or '+')	[*type*] [*port_name*] [*memory_type*]	Override port name, Specify memory type
OUT (or '-')	[*type*] [*port_name*] [*memory_type*]	Override port name, Specify memory type
ARCH	[*host_architecture*]	The host machine architecture to run this kernel
ACCEL	[*accelerator*]	The acceleration feature of this kernel
INIT	[*init_prototype*]	The prototype of a inititialization function
CLEAN	[*clean_prototype*]	The prototype of a cleaning-up function

二番目のステップでは，一般開発者がドラッグ・アンド・ドロップ操作を主体としたビジュアル・プログラミング環境を利用したアプリケーションの開発を行う．具体的には，UDOP をパレットから選んで画面に置き，UDOP のペアを，データ・フローを表す線で接続したストリーム・グラフと呼ばれるアプリケーション・プログラムを作成する．

3.2 リソース・スケジューリングと通信コード生成

ストリーム・グラフとして書かれたアプリケーション・プログラムは，我々が開発しているコンパイラによって並列動作する実行形式群に変換される．コンパイルの第 1 段階では，利用できる演算リソースの中で，アプリケーションのスループットを最大化するように，部品を配置する方法を決定する．第 2 段階では，部品間のデータ通信コードを同期やメモリ管理などと併せて自動的に生成する．

ストリーム・グラフの各部品は，ソフトウェア・パイプラインのステージとして動作する．パイプラインを構成する各ステージの実行時間が短いほど，全体のスループットは向上するため，コンパイラは，ヘテロジニアス・システムのハードウェア・コンフィグレー

ⓒ 2012 Information Processing Society of Japan

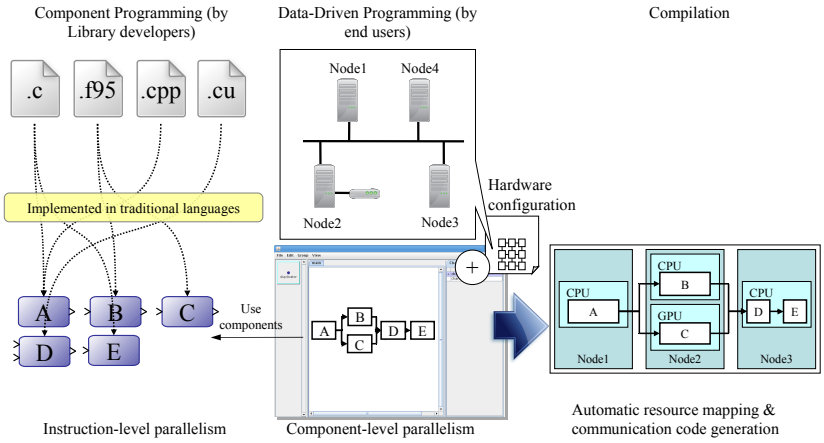

図 1 2-レベル・プログラミング・モデル.
Fig. 1 2-Level Programming Model.

ションで与えられたリソース制約の中で，ストリーム・グラフ上の各 UDOP の実行時間の最大値を最小化するように，リソース・スケジューリングを行う．

並列化可能属性を持つ UDOP の場合，演算量に応じて複数のスレッドによる実行に並列化される場合がある．KERNEL 性能と利用できるリソースによっては同じ UDOP を実装した 2 個以上の KERNEL に割り当てられる場合もある．各 KERNEL 性能はダミーデータによる実行時間をコンパイルに先立って計測してデータベースに記憶しておく．連続した UDOP をグループにし，ループ属性を持たせて繰り返し処理を行うことも可能である．繰り返し部分は逐次処理しかできないためループ内の各 UDOP に個別のスレッドを割り当てるのではなく，1 つのスレッド中でループ内のすべての UDOP を連続実行する．

UDOP から KERNEL へのマッピングの決定後，各 KERNEL を物理的な演算リソースへの割り当ておよびネットワーク・エンベッディング処理を行い通信方式の決定を行う．

割り当てが完了したら部品間のデータ通信部分を含むソースコードを MPMD（Multiple Program Multiple Data）モデルで生成する．KERNEL の入出力およびデータ通信で使われるメモリの管理はフレームワークが行うため，KERNEL では与えられたメモリ領域に対する読み書きをするだけでよい．生成されたソースコードは，gcc や nvcc などのネイティブ・コンパイラによって実行形式に変換される．さらに，アプリケーションの実行開始と終了待機を行うコントロール・プログラムと，実行形式の各ノードへの配置と起動を行うためのスクリプトも自動生成する．

```
<node id="node" num="1">
  <processor id="node_cpu_bundle"
                          architecture="x86_64*11" num="1"/>
  <processor id="node_cpu" architecture="x86_64" num="1"/>
  <processor id="node_gpu" architecture="oclgpu" num="1"/>
</node>
```

図 2 リソース・バンドルを利用する場合のハードウェア定義（一部）.
Fig. 2 A hardware configuration using Resource Bundle.

先行研究[2),3)] では，ステンシル・アプリケーションを用いて評価を行なってきたが，我々のフレームワークはそれに特化したものではなく，一般の通信にも適用できる．

4. リソース・バンドルモデル

本稿では，我々が提案してきたフレームワークに新たにリソース・バンドルという概念を導入する．リソース・バンドルは複数の計算リソースをまとめたもので，これまでは 1 つの KERNEL（またはループ内のため統合された KERNEL 群）に対し 1 つのハードウェア・スレッドを割り当てるように並列化を行なっていたが，代わりに 1 つのリソース・バンドルを割り当てるように拡張する．リソース・バンドルが実際に利用されるかどうかは，ハードウェア・スレッドの演算能力，UDOP のデータサイズおよび KERNEL の計算時間によって，我々のコンパイラが自動的に決定する．

リソース・バンドルに含まれるハードウェア・スレッドの数は，最小 1 からノードに含まれる演算リソースの数までで，ハードウェア・コンフィグレーションによって指定する（図 4）．CPU コアを複数使う指定の他，CPU と GPU の混在も許される．リソース・バンドルを利用する KERNEL は，定義で利用したいリ

ⓒ 2012 Information Processing Society of Japan

```
(a)
/// KERNEL conventional_kernel
/// ARCH x86_64

(b)
/// KERNEL OpenCL_kernel
/// ARCH x86_64*11
```

図 3 従来の KERNEL 定義（a）とリソース・バンドルを要求する KERNEL 定義（b）.
Fig. 3 A conventional KERNEL definition (a) and a KERNEL definition which requires a Resource Bundle. (b)

ソース・バンドルのコンフィグレーションを ARCH 指示文で指定する．図 3（a）はこれまでの KERNEL 定義である．アーキテクチャ（ARCH）にマイクロプロセッサ・アーキテクチャが指定されているのみである．図 3（b）はリソース・バンドルを指定する場合の KERNEL 定義である．例えば x86_64*11 は x86_64 を 11 個利用するリソース・バンドルに付けられた名前である．KERNEL は，リソース・バンドルのサイズ情報を実行時に取得することができ，KERNEL 内部スレッド数の決定などのパラメータ最適化に利用することができる．

リソース・バンドルを設定することで KERNEL が要求する演算リソースをスケジューラが正しく把握することができるため，利用可能なハードウェア・スレッド数を超えて過剰にスレッドが生成されることを防止できる．同様に，並列化されたコード資産を部品として用いる場合にも，我々のフレームワークによる最適化を両立することが可能になる．

5. アプリケーションへの適用例

5.1 Julia 集合のレイトレーシング

本稿では，リソース・バンドルの適用例として，IBM OpenCL Lounge[13] にて提供されている julia プログラムを，我々のフレームワーク上に移植し，IBM POWER7，Intel Core i7 X980，NVIDIA Tesla C2050 の 3 種類のプロセッサが混在したヘテロジニアス・システム上で実行した結果を評価した．julia プログラムは，クオータニオン上に定義された Julia 集合の断面をレイトレーシング法によって描画するものである[14]（図 4）．レイトレーシング法はデータ並列性が高く，特に julia プログラムの場合は初期化時に，ノード間で予めデータを共有しておけばレンダリング処理でのデータ交換は発生しない．

5.2 本フレームワークでの実装

移植は主に以下の作業からなる．まず，主要なサブルーチンについて，データ入出力のフローを形成するように切り出し，KERNEL のアノテーションを加える．同時に UDOP 定義も作成し，ビジュアル・プログラミング・ツールを用いてストリーム・グラフを作成する．最後に並列度などのパラメータとハードウェア・コンフィグレーションを指定してコンパイルを行い，実行形式を生成する．

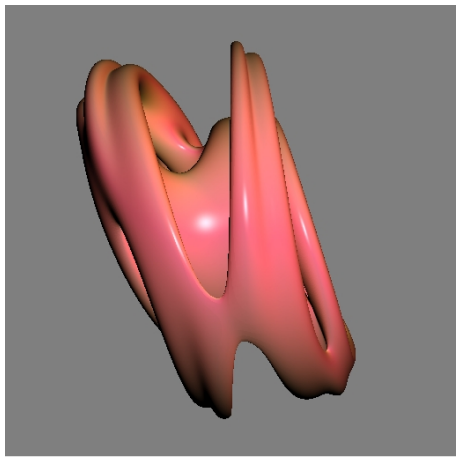

図 4 Julia 集合のレンダリング例.
Fig. 4 A renderered image of Julia set.

表 2 評価に使用したマシン環境.
Table 2 A system used for our experiments.

	Node 1	Node 2A	Node 2B
CPU	IBM POWER7	Intel Core i7 X980	
CPU Clock	3.3GHz	3.33GHz	
Physical Cores	8	6	
Hadware Threads	32	12	
Host Memory	64GB	12GB	
Network Interface	10Gb Ethernet	1Gb Ethernet	
GPGPU	N/A	NVIDIA Tesla C2050	
Compiler	GCC 4.3.2	GCC 4.4.5	
CUDA Version	N/A	4.1RC2	
OpenCL Version	1.1	1.0	
Display	None	None	Yes

使用した機器を表 2 に示す．各ホストには OpenCL 環境が導入されており，NVIDIA Tesla C2050 では GPU による並列演算，IBM POWER7 および Intel Core i7 X980 では CPU による並列演算をサポートする．Tesla は Node 2A と Node 2B の拡張カードとして導入されているため，これらホストでは二種類のアーキテクチャで OpenCL カーネルを実行することができる．なお，Node 2A と Node 2B は全く同じ構成であるが，DECODE が割り当てられたノードを Node 2B と呼ぶこととし，描画結果の画面表示は Node 2B で行うこととする．

図 5 は julia プログラムのストリーム・グラフである．並列レンダリングした結果を，GUI デスクトッ

© 2012 Information Processing Society of Japan

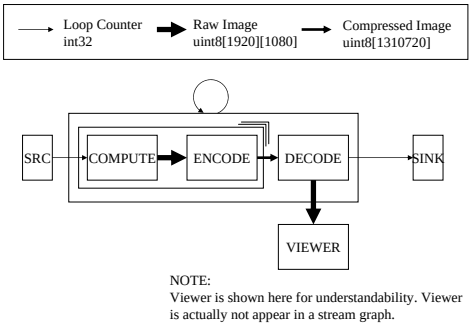

図 5 Julia 集合レンダラーアプリケーションのストリームグラフ.
Fig. 5 Stream graph for the Julia set renderer.

```
/* UDOP definition is an interface definition */
/// UDOP UDOP_JULIA_COMPUTE
/// +int32 counter_in
/// -uint8[][] out

/// UDOP UDOP_JPEG_ENCODE
/// +uint8[][] counter_in
/// -int32 counter_out
/// -uint8[] out

/// UDOP UDOP_JPEG_DECODE
/// +uint8[] in
/// +int32 counter_in
/// -int32 counter_out

/*  KERNEL definition */
/// KERNEL julia_kernel(counter_in, out$$)
/// +int32 counter_in
/// -uint8[1920][1080] out
/// > UDOP_JULIA_COMPUTE
/// INIT julia_init()
/// CLEAN julia_cleanup()
/// ARCH x86_64[12],ppc64[32]
/// ACCEL oclcpu,oclgpu
void julia_kernel(
        rv_int32_t counter_in,
        rv_uint8_t* out,
        const rv_int64_t* out_dim) {

snipped...

}
```

図 6 Julia 集合レンダラーアプリケーションの UDOP 定義と KERNEL 定義.
Fig. 6 UDOP and kernel definition for the Julia set renderer.

プ上のウィンドウに統合して表示する．図 6 のように UDOP および KERNEL を定義した．

UDOP_JULIA_COMPUTE がレイトレーシング処理の UDOP である．KERNEL の実装は OpenCL を用いて記述した一種類である．初期化ルーチンを使用し，割り当てられたリソースを判断して，OpenCL のローカルワークサイズパラメータを調整する処理を行なっている．CPU の場合は 1×1，GPU の場合は OpenCL ランタイムによる自動決定としている．

```
<dppx>
  <configuration>
    <node id="node1" num="1">
      <processor id="node1_cpu_bundle"
                          architecture="ppc64*32" num="1"/>
    </node>
    <node id="node2" num="2">
      <processor id="node2_cpu_bundle"
                          architecture="x86_64*11" num="1"/>
      <processor id="node2_cpu" architecture="x86_64" num="1"/>
      <processor id="node2_gpu" architecture="gpu" num="1"/>
    </node>
    <connect architecture="mpi">
      <peer id="node1"/>
      <peer id="node2"/>
    </connect>
  </configuration>
</dppx>
```

図 7 使用したハードウェア定義ファイル.
Fig. 7 A hardware configuration file for for our experiments.

UDOP_JPEG_ENCODE はレンダリング結果の JPEG 圧縮を行う UDOP で，ビューアノードまで画像データを送信する通信コストを減らすために用いる．圧縮後のデータサイズは画像の内容によって増減するが，本フレームワークの現在の実装ではランタイムに転送サイズが変化するポートはサポートしていないため，圧縮データを収めるのに十分と見込まれる量の固定サイズの 1 次元配列を宣言している．本 UDOP も並列化が可能である．

UDOP_JPEG_DECODE は並列レンダリングされた結果の画像を統合してビューアプログラムに転送する．入力データは JPEG 圧縮された画像を複数個含む 1 次元配列で，伸長・統合された画像は再び圧縮され，TCP/IP 通信を利用してビューアプログラムに送られる．ビューアプログラムは，GLUT を用いて書かれているが，無限ループでイベントを待機するイベントドリブン設計になっている．無限ループは，我々のフレームワークのソフトウェア・パイプライン構造に配置できないため，本フレームワークによるアプリケーションとは独立したプログラムとした．

UDOP と KERNEL およびストリームグラフの作成が終わったら，システム構成をリソース・バンドルを含むハードウェア・コンフィグレーションとして表現し（図 7），作成したストリーム・グラフと共にコンパイラに与えてリソース割り当て，通信コードの生成および実行形式への変換までを実施した．最終的に生成されたプログラムの構成を図 8 に示す．

プログラムは，3 つの各ホストに 1 つずつ実行形式が生成され，それぞれ 1 つまたは 2 つのスレッドを実行する（図 8）．計 5 つのスレッドはそれぞれ UDOP_JULIA_COMPUTE を実装した COMPUTE KERNEL（図 8（a））を実行する．CPU のみを使う

ⓒ 2012 Information Processing Society of Japan

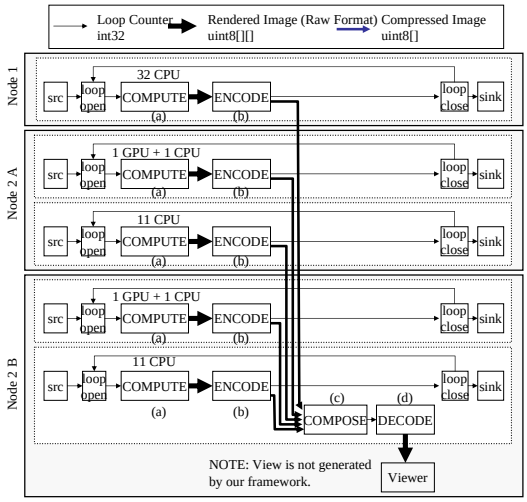

図 8 生成されたプログラムの構成.
Fig. 8 Structure of the generated program.

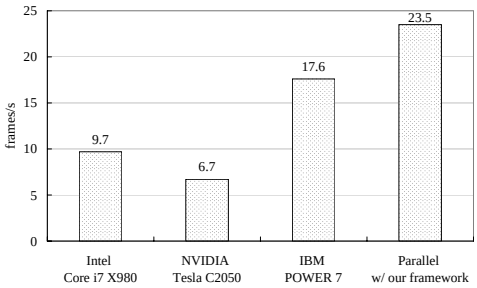

図 9 フレームレートの比較.
Fig. 9 A comparison of frame rates.

表 3 JPEG 圧縮・伸長オーバーヘッド（msec）.
Table 3 Overhead for JPEG codec (msec).

Node	ENCODE KERNEL	DECODE KERNEL
Node 1	63.9	NA
Node 2A	14.1	NA
Node 2B	14.1	26.8

KERNEL には 11 コアまたは 32 コアのリソース・バンドルが割り当てられており，KERNEL の中で OpenCL が複数のハードウェア・スレッドを利用することができるようになっている．また，GPGPU を利用する KERNEL は，GPGPU の呼び出しのために CPU リソースも 1 つ割り当てられている．これらの KERNEL はすべて，OpenCL で記述された単一のソースコードを使用している．各ノードのレンダリング結果は，各ノードから Node 2B に送られるが，ENCODE KERNEL（図 8（b））が並列化されたため，DECODE KERNEL（図 8（d））の前に，ストリーム・グラフにはなかったデータ統合のための COMPOSE KERNEL（図 8（c））が挿入されている．

5.3 ベンチマーク結果と考察

我々のコンパイラによって生成された並列レンダリング・アプリケーションと，比較対象としてオリジナルの julia プログラムを Intel Core i7 X980，IBM POWER7，NVIDIA Tesla C2050 の各ノード単体で実行した場合の，フレームレートを表 9 に示す．各ノード単体は 6.7〜17.6frames/s であったがが，本フレームワークを用いて並列化された場合は 23.5frames/s という結果を得た．以下，結果について考察する．

各ノード単体で実行する場合に比べ，並列実行する場合は，データ集約のためのネットワーク転送が必要となる．また，本稿ではデータ転送量を減らすため，送受信の前後に JPEG 圧縮・伸長処理を挿入している．これらにより，レンダリングノードの ENCODE KERNEL での圧縮処理，ビューアノードの DECODE KERNEL での伸長処理，および ENCODE KERNEL から COMPOSE KERNEL へのデータ転送処理が，並列化に伴うオーバーヘッドとなる．データ集約を行う Node 2B への，レンダリング結果 1 フレーム分の転送時間は，Node 2A と Node 2B の間の netperf ツールによる実測値 106.7MB/s から，1.25/106.7 = 11.7msec と見積もれる．ただし，圧縮しない場合は 7.91/106.7 = 74.1msec となる．また，1 フレーム分の JPEG 圧縮・伸長の処理時間は，表 3 に示したとおりである．

各オーバーヘッドの合計は，Node 2A から Node 2B では 14.1 + 11.7 + 26.8 = 52.6msec となり，圧縮しない場合よりも小さいが，並列化による ENCODE KERNEL 部分の減少を考慮してもデータ通信がボトルネックになっていると判断できる．また，Node 1 から Node 2B は 63.9 + 11.7 + 26.8 = 102.4msec となり圧縮しない場合よりも長くなってしまう．現状では圧縮・伸長処理は UDOP としてストリーム・グラフ上に記述されているので常に処理が実行されるが，UDOP として書かずに，コンパイラによる最適化として，ネットワーク通信時間を見積もって圧縮の有無を自動選択するよう拡張するなどの対策が必要と考えられる．

6. まとめと今後の課題

本稿ではこれまで提案してきたヘテロジニアス・シ

ⓒ 2012 Information Processing Society of Japan

ステム向けのプログラミング・フレームワークを拡張し，リソースバンドルの導入を新たに提案した．同一ノードに含まれる複数のハードウェア・スレッドを一括で 1 つの部品に与えられるようになり，スレッド並列化された既存のコード資産や，OpenCL や OpenMP などの独自に並列化を行う言語もプログラム部品としてスケジューリングが可能となった．

提案手法を用いて Julia 集合のレンダリングを並列実行するアプリケーションを作成し，IBM POWER7, Intel Xeon, NVIDIA Tesla の 3 種類のプロセッサが混在したシステム上に自動並列化する実験を行った．OpenCL で記述したプログラム部品に対し，スケジューラがリソースバンドルを割り当てることで，OpenCL が自動的に複数のハードウェア・スレッドを利用した並列処理を実行できる．また，リソースバンドル情報をランタイムに取得して，OpenCL の分割パラメータとして反映させた．3 台のマシンによる並列演算で，1920×1080 のサイズのレイトレーシングを 23.5fps で実行することができた．

今後の課題として，現在はユーザーが行なっているリソース・バンドルへの分割のコンパイラによる自動化，通信時間やメモリ容量なども考慮したスケジューリングの導入が挙げられる．また，実用アプリケーションへの適用と評価も行っていきたい．

参 考 文 献

1) RIKEN: RICC, http://accc.riken.jp/ricc.html.
2) Murase, M., Komatsu, H., Maeda, K., Noda, S., Doi, M. and Himeno, R.: A parallel programming framework orchestrating multiple languages and architectures., *Conf. Computing Frontiers* (Cascaval, C., Trancoso, P. and Prasanna, V. K.(eds.)), ACM, p. 22 (2011).
3) 土居意弘, 村瀬正名, 前田久美子, 小松秀昭: ハイブリッド・システム向け並列プログラミング・フレームワーク, *IBM PROVISION*, Vol. 67, pp. 79 – 86 (2010).
4) Wang, L., Huang, M., Narayana, V.K. and El-Ghazawi, T.: Scaling scientific applications on clusters of hybrid multicore/GPU nodes, *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, New York, NY, USA, ACM, pp. 6:1–6:10 (2011).
5) Callahan, D., Chamberlain, B. and Zima, H.: The cascade high productivity language, *High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings. Ninth International Workshop on*, pp. 52 – 60 (2004).
6) Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., SteeleJr., G.L. and Tobin-Hochstadt, S.: The Fortress Language Specification Version 1.0, http://research.sun.com/projects/plrg/Publications/fortress.1.0.pdf.
7) Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C. and Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing, *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, ACM, pp. 519–538 (2005).
8) 設樂明宏, 鎌田俊昭, 山田昌弘, 西川由理, 吉見真聡, 天野英晴: OpenCL 互換アクセラレータのマルチノード環境における開発負担軽減のためのミドルウェアの実装, 情報処理学会研究報告. [ハイパフォーマンスコンピューティング], Vol. 2010, No. 22, pp. 1–8 (2010-12-09).
9) MOSIX.org: The Virtual OpenCL (VCL) Cluster Platform, http://www.MOSIX.org.
10) Barak, A., Ben-Nun, T., Levy, E. and Shiloh, A.: A package for OpenCL based heterogeneous computing on clusters with many GPU devices, *Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on*, pp. 1 –7 (2010).
11) Thies, W., Karczmarek, M. and Amarasinghe, S. P.: StreamIt: A Language for Streaming Applications, *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, London, UK, Springer-Verlag, pp.179–196 (2002).
12) Gedik, B., Andrade, H., Wu, K.-L., Yu, P. S. and Doo, M.: SPADE: the system s declarative stream processing engine, *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, ACM, pp. 1123–1134 (2008).
13) IBM Corporation: OpenCL Lounge, https://www.ibm.com/developerworks/mydeveloperworks/groups/service/html/communityview?communityUuid=80367538-d04a-47cb-9463-428643140bf1.
14) Hart, J. C., Sandin, D. J. and Kauffman, L. H.: Ray tracing deterministic 3-D fractals, *SIGGRAPH Comput. Graph.*, Vol. 23, pp. 289–296 (1989).

© 2012 Information Processing Society of Japan