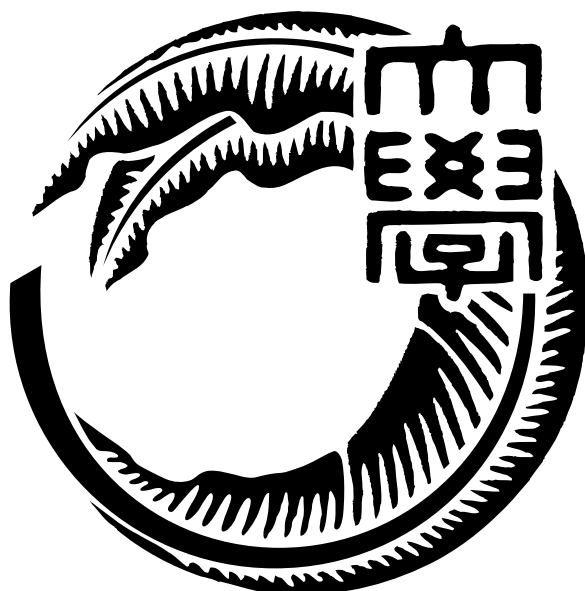


平成27年度 卒業論文

Code Gear Data Gear による GPGPU 処理  
実装



琉球大学工学部情報工学科

135704C 東恩納 琢偉

指導教員 河野 真治

# 目次

第1章	GearsOSでのGPU実行	1
第2章	GearsOS	2
2.1	Continuation Based C . . . . .	2
2.2	並列性 . . . . .	2
2.3	柔軟性 . . . . .	3
2.4	Gears での Meta computation の実行 . . . . .	3
第3章	Gears OSの構成	4
3.1	TaskManager . . . . .	4
3.2	Worker . . . . .	4
第4章	GPGPU	5
4.1	GPGPUとは . . . . .	5
4.2	CUDAでの実行 . . . . .	6
第5章	GPU実装	7
5.1	GPUWokerの実装 . . . . .	7
5.2	CUDATwiceの実装 . . . . .	7
5.3	CMake . . . . .	7
5.4	性能評価 . . . . .	7
5.5	比較 . . . . .	7
第6章	今後の課題	8
6.1	GPU並列実行 . . . . .	8
第7章	結論	9
7.1	まとめ . . . . .	9
7.2	今後の課題 . . . . .	9

# 目 次

2.1 goto による Code Segment 間の接続 . . . . .	2
4.1 Gears OS による GPGPU . . . . .	6

# 表 目 次

# 第1章 GearsOSでのGPU実行

CPUの処理速度の向上のためクロック周波数の増加は発熱や消費電力の増大により難しくなっている。

そのため、クロック周波数を上げる代わりにCPUのコア数を増やす傾向にある。マルチコアCPUの性能を発揮するには、処理をできるだけ並列化しなければならない。また、PCの処理性能を上げるためにマルチコアCPU以外にもGPUやCPUとGPUを複合したヘテロジニアスなプロセッサが登場している。並列処理を行う上でこれらのリソースを無視することができない。

しかし、これらのプロセッサで性能を出すためにはこれらのアーキテクチャに合わせた並列プログラミングが必要になる。

並列プログラミングフレームワークではこれらのプロセッサを抽象化し、CPUと同等に扱えるようにすることも求められる。

そのためには並列処理を行うプログラムの依存関係を解決することや、本研究室で開発しているGears OSはCode GearとData Gearによって構成される。Code Gearは処理の単位、Data Gearはデータの単位となる。Gears OSではCode/Data Gearを用いて記述することでプログラム全体の並列度を高めて、効率的に並列処理することが可能になることを目的とする。

また、Gears OSの実装自体がCode/Data Gearを用いたプログラミングの指針となるように実装する。

Gears OSにおけるTaskは実行するCode Gearと実行に必要なInput Data Gear, 出力されるOutput Data Gearの組で表現される。Input/Output Data Gearによって依存関係が決定し、それに沿って並列実行する。

依存関係の解決などのMeta Computationの実行はMeta Code Gearで行われる。Meta Code GearはCode Gearに対応しており、Code Gearが実行した後にそれに対応したMeta Code Gearが実行される。これらのことから、並列プログラミングで必要とされるGPUやマルチコアCPUをCPUと同等に扱うためには依存関係を明確に出来るGears OSでCUDAによるGPU実装を行い、CbCによるGPU処理での有用性を発見することが本研究の目的である。

## 第2章 GearsOS

### 2.1 Continuation Based C

Gears OS の実装は本研究室で開発している CbC(Continuation based C)[?] を用いて行われている。

CbC は処理を Code Segment を用いて分割して記述することを基本としている。Gears OS の Code Gear は CbC を元に記述されている。CbC のプログラムでは C の関数の単位で Code Segment を用いて分割し、処理を記述している。Code Segment は C の関数と異なり戻り値を持たない。Code Segment の宣言は C の関数の構文と同様に行い、型に `_code` を使い宣言している。Code Segment から Code Segment への移動は `goto` の後に移動先の Code Segment 名と引数を並べた記述する構文を用いて行う。この `goto` による処理の遷移を継続と呼び、C での関数呼び出しにあたり、C では関数の引数の値がスタックに積まれていくが、Code Segment の `goto` では戻り値を持たないため、スタックに値を積んでいく必要がなくスタックを変更する必要がない。このようなスタックに積まない継続を軽量継続と呼び、呼び出し元の環境を持たない。図 2.1 は Code Segment 間の接続関係を表している。

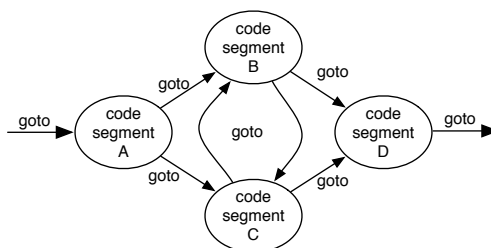


図 2.1: goto による Code Segment 間の接続

### 2.2 Code Gear と Data Gear

Code Gear はプログラムの実行コードそのものであり、OpenCL[?]/CUDA[?] の kernel に相当する。

Code Gear は処理の基本として、Input Data Gear を参照し、一つまたは複数の Output Data Gear に書き込む。また、接続された Data Gear 以外には参照を行わない。Input

Data Gear と Output Data Gear の 2 つによって、Code Gear の Data に対する依存関係を解決し、Code Gear の並列実行を可能とする。

Code Gear は CbC を元に記述されており、処理の移行は function call ではないので、呼び出し元に戻る概念はない。その代わりに、次に実行する Code Gear を軽量継続の goto で指定する。

Data Gear は、int や文字列などの Primitive Data Type の組み合わせ (struct) である。Data Gear は様々な型を持つ union として定義される。

Gear の特徴の一つはその処理が Code Gear, Data Gear に閉じていることにある。これにより、Code Gear の実行時間、メモリ使用量を予測可能なものにする。

## 2.3 並列性

Gears OS は Code Gear Data Gear によってプログラムを Gear という単位によって分けられ、Gear 毎に依存関係が明確化されている。そのため並列化を行う場合、並列処理を行う Gear の依存関係に沿って容易に並列化することが出来る。

## 2.4 柔軟性

Code Gear Data Gear は CbC を用いて記述されており、その特徴としてプログラムの処理が Gear という単位で分けられて記述されていることである。

また Gear の処理の遷移には goto を用いており、この goto 先を変更することで容易に処理の順番を変更、もしくは新しい処理を追加することが出来る。図??は Code Gear C と Code Gear D を入れ替えたものであり、この入れ替えを行う場合 Code Gear C を指していた goto の先を Code Gear D に書き換えるだけで済む。

## 2.5 Gears での Meta computation の実行

Gears OS では通常の処理を Computation、Computation のための Computation を Meta Computation として扱う。Meta Computation の例として並列処理の依存関係の解決や、OS が行うネットワーク管理、メモリ管理等の資源制御などが挙げられる。Gears OS では Meta Computation を Meta Code Gear、Meta Data Gear で表現する。Meta Code Gear は通常の Code Gear 直後に遷移され、Meta Computation を実行する。Meta Computation の実行後は通常の Code Gear で指定した Code Gear を実行する。つまり Code Gear の実行後は何かしらの Meta Code Gear を実行する。

## 第3章 Gears OSの構成

Code ?? に CAS を使用した Task 挿入を示している。Code ?? は 2つの Code Gear を定義しており、putQueue3 は Queue に要素がある場合、putQueue4 は Queue に要素がない場合の Task 挿入を示している。

### 3.1 TaskManager

Gears OS の TaskManager は WaitTaskQueue に入っている Task の依存関係を解決する。Task には Input/Output Data Gear の情報が格納されている。Input Data Gear は Task に必要な Data Gear で揃ったら Task は実行可能な状態になる。Output Data Gear は Task が Persistent Data Tree に書き出す Data Gear である。この Input と Output の関係が依存関係となる。TaskManager は Persistent Data Tree を監視しており、WaitTaskQueue に入っている Task の Input Data Gear が揃っているのを確認したら実行可能な Task として AcitiveTaskQueue へ移動させる。

### 3.2 Worker

Worker は TaskQueue から Task を取得し、実行する。Task には実行する Code Gear と実行に必要な Code Gear の key が格納されている。実行に必要な Code Gear は Persistent Data Tree から key を使って取得する。

各 Worker は個別の Context を参照しており、メモリ空間も独立しているのでメモリを確保する処理で他の Thread を止めることはない。ただし、Persistent Data Tree への書き出しは競合する可能性があるので CAS を利用してデータの一貫性を保証する必要がある。

Worker が TaskQueue から Task の取得を行う Code Gear を Code ?? に示す。Task Queue から取得した Task から実行する Code Gear と必要な Data Gear の key を Worker Context に書き込むことで実行される。

Worker から取得された Task の Code Gear は並列実行される。並列実行される Code Gear と言っても他の Code Gear と同じである。これは Gears OS 自体が Code Gear によって構成されていることに起因する。つまり、Gears OS を利用して書かれたプログラムで定義されている Code Gear に依存関係がないとき、全て並列に実行することができる。



# 第4章 GPGPU

## 4.1 GPGPUとは

GPGPU とは、元々は画像出力や画像編集などの画像処理に用いられる GPU を画像処理以外に利用する技術の事である。

画像の編集はピクセル毎に行われるため多大な数の処理を行う必要があるが、GPU は CPU に比べコア数が多数あり、多数のコアで同時に計算することによって CPU よりも多数の並列な処理を行う事が出来る。

これによって GPU は画像処理のような多大な処理を並列処理することで、CPU で処理するよりも高速に並列処理することが出来る。しかし、GPU のコアは CPU のコアに比べ複雑な計算は出来ない構造であるため単純計算しか出来ない、また一般的にユーザーから GPU 単体に直接命令を書き込むことも出来ないなどの問題点も存在する。GPGPU は CPU によって単純計算の Task を GPU に振り分ける事によって、GPU の問題点を解決しつつ、高速な並列処理を行うことである。また Data Gear へのアクセスは接続された Code Gear からのみであるから、処理中に変数が書き変わる事がない。図 4.1 では以下の流れで処理が行われる。

- Data Gear を Persistent Data Tree に挿入。
- TasMannager で実行する Code Gear と実行に必要な Data Gear への Key を持つ Task を生成。
- 生成した Task を TaskQueue に挿入。
- Worker の起動。
- Worker が TskQueue から Task を取得。
- 取得した Task を元に必要な Data Gear を Persistent Data Tree から取得。
- 並列処理される Code Gear を実行。

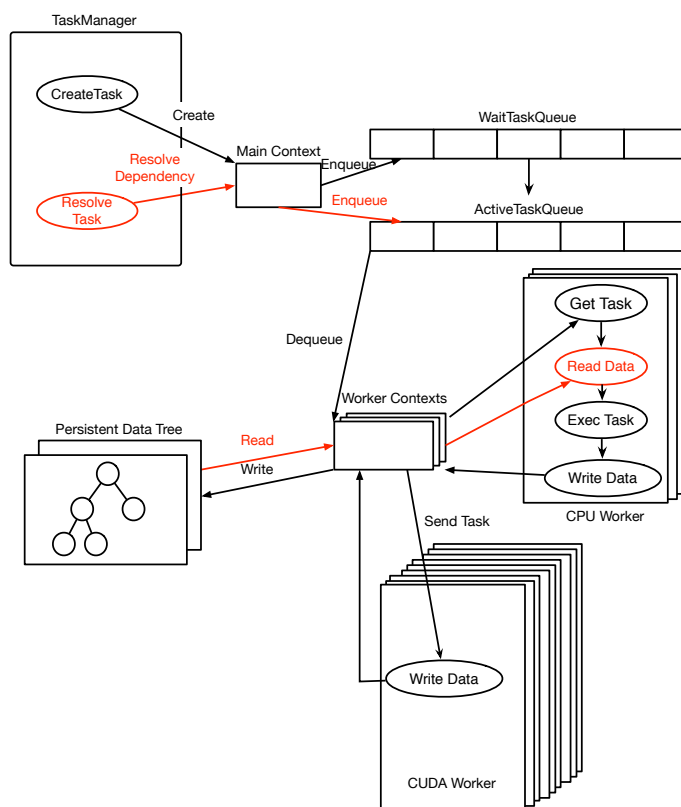


図 4.1: Gears OS による GPGPU

## 4.2 CUDA での実行

CUDA とは NVIDIA が提供している並列コンピューティングアーキテクチャで

## 第5章 GPU 実装

5.1 GPUWoker の実装

5.2 CUDATwice の実装

5.3 CMake

5.4 性能評価

5.5 比較

## 第6章 今後の課題

### 6.1 GPU 並列実行

## 第7章 結論

7.1 まとめ

7.2 今後の課題

# 謝辭

2017年3月