A-011

Python を拡張した GPGPU 向け言語開発環境の設計
Design of a Language Development Environment for GPGPU based on Python

鎌田 知也† 佐々木 晃‡
Tomoya Kamata Akira Sasaki

1. はじめに

近年 Graphics Processing Unit(GPU)を画像処理以外に用いられる General-purpose computing on GPU(GPGPU)が行われている。GPGPU には、CPU と比較すると多数のコアを持つ GPU を用いた高い並列処理を活かしたプログラムを作成することが出来るという特徴がある。しかしながら、GPGPU を用いた開発にはスレッドやブロックなどの計算資源やデバイスメモリやシェアードメモリといったメモリの種類と構造などの GPU アーキテクチャの理解や、GPU ベンダである NVIDIA が提供している C 言語の拡張言語の CUDA C などといった専用言語の学習があり、比較的敷居が高いものである。

本研究では Python を拡張し GPGPU 向けの言語を設計した。本提案言語は、Python を拡張した言語設計になっており GPGPU を行う際に新たな言語を習得する場合より低い学習コストでプログラムを作成することが出来る。また、本提案言語ではクラスの宣言によってオブジェクトを定義することで、複雑なデータ構造を持つプログラムの作成することが可能となっている。本提案環境では競合処理や同期処理といった並列プログラミング特有の問題についての知識を有する開発者が利用することを想定している。

既存手法としてバインディングツールを用いた開発がある。バインディングツールは高レベルな処理系である Java をはじめ、Perl や Python などに提供されている。これを用いることでホスト言語の特徴を活かしたプログラムの記述ができる。しかしこれらのツールで、カーネル関数内の命令を CUDA C で記述しなければならず、ホスト言語に加えて CUDA C の学習と移植作業、実行時の計算資源の設定が必要となる。そのためソースコード全体を移植せずに行うことができるが、様々な学習が必要となるデメリットが存在する。

2. 提案開発環境

2.1 開発環境概要

本提案環境はコード解析、カーネルコード生成、実行部の 3 つによって構成されており、各部分について説明する。

2.1.1 コード解析

コード解析部では、開発者が生成したソースコードから情報を取得するための解析を行う。解析には Python の標準ライブラリ AST を用いて変数や定数、関数などを静的に解析する。これによりプログラム言語において意味のある情報のみを得ることができる。

2.1.2 カーネルコード生成

カーネルコード生成では、解析によって得られた情報を用いてカーネルコードを生成する。生成される言語は CUDA C である。CUDA C では Python と同一の構文であっても処理結果が異なる場合がある。この場合は本提案言語仕様に沿って同等の処理を行うコードを生成する。

2.1.3 実行

実行部では、生成されたカーネルコードをコンパイルし、データの転送、計算資源の設定を行い、カーネルコードを実行する。まず生成したカーネルコードを実行できるよう JIT(Just-In-Time)コンパイラを用いてコンパイルする。その後、開発者が指定した引数のデータを GPU のメモリへ転送しカーネル処理を行う。また、コンパイルや GPU のメモリへデータを転送する機能等は PyCUDA を用いて実現している。

2.2 カーネル関数の記述言語の仕様

2.2.1 対応命令

本提案言語仕様は Python の構文のサブセット言語を採用している。カーネル関数内で利用できる命令は基本的な命令や式および関数呼び出しと宣言に対応できるよう設計している。条件分岐や繰り返し、繰り返しの制御命令などを利用することができる。式として論理演算や比較、ビット演算、ビットシフト、リストなどに対応しており、さらに関数呼び出しに対応している。またクラス関数の定義や宣言が可能となっており、複雑なデータ構造を利用したプログラムの作成も行うことができる。

2.2.2 記述法

本提案環境によるスクリプトの記述例を図 1 に示す。この例は、関数 kernel が GPU で処理を行うカーネル関数である。本提案環境の呼び出しは関数 gpu_run であり引数として、カーネル処理を行うエントリーメソッド、スレッド番号を保持する変数の文字列、実行するスレッドの長さ、エントリーメソッドの引数、をとる。通常の GPGPU にお

```
def kernel(a, b, result):
  “””
  a:int*
  b:int*
  result:int*
  “””
  result[thread_id] = a[thread_id] + b[thread_id]


/a, b, result の初期化/
import tools
tools.gpu_run(kernel, ”thread_id”, N, a, b, result)
```

図 1 記述例

† 法政大学大学院情報科学研究科
‡ 法政大学情報科学部

いて、並列処理の際にはスレッド番号を取得する命令を記述して各スレッドを制御するが、本手法ではこのような命令を自動で生成されたカーネルコードに挿入するため、開発者はカーネル関数内に記述する必要がない。

2.2.3 予約語

本提案環境ではいくつかの予約語を利用する。これらは先に述べた制約上必要となる関数や、並列処理で必要となる同期処理を行う関数がある。予約語は以下の 2 つがある。

range():繰り返し命令で利用する

__synchronized():全スレッドで同期処理を行う

2.2.4 制約

カーネル関数内において Python の構文すべての命令を実行することはできない。例として、繰り返しの記述法について述べる。Python ではイテレータを用いて繰り返し回数の制御をしているが、CUDA C ではそれに加えて数値を用いて制御が可能である。そこで本提案言語仕様では Python において標準的に用いられる range 関数を記述可能としている。しかし、カーネル関数内での range 関数はイテレータとなるリストを生成するわけではない。また、繰り返し変数の終点を指定するためにリストの長さを返す len 関数がよく利用されるが、本提案言語では len 関数に対応しておらずこれに注意が必要である。

カーネル関数内では Python のように動的型付きではなく、静的型付きとしてプログラミングする必要があり、本提案環境ではヒアドキュメント内に型情報の記述しコード生成に用いる。関数内で新たに宣言された変数は浮動小数となり任意の型情報を持った変数を宣言したい場合、引数同様にヒアドキュメント内への記述をしなければならない。また関数の返り値を設定する場合も同様にヒアドキュメントへの記述が必要である。ただし、最初に呼び出されるエントリーメソッドにおいては返り値を設定できず、記述した場合であっても適用されない。

3. 評価

本研究を評価するにあたり実際にプログラムを作成し記述性に加え、通常の Python とバインディングツールである PyCUDA[3]との処理時間を比較する。評価用として作成したプログラムはジュリア集合演算で、測定環境として CPU を Core i7 860、GPU を Quadro K2200、主記憶 8GB メモリを搭載しているコンピュータを用いた。

ジュリア集合演算では通常の四則演算に加え複素数演算が必要となる。今回作成したプログラムではユーザが定義した複素数クラスを利用している。

処理時間では、Python の処理時間と比較して本提案環境を利用した場合では 70 倍以上の処理時間短縮に成功し、Python では解像度に比例し処理時間が大きくなるが、本提案環境では解像度を変更した場合であっても処理時間は大きく変化せず、実行効率が高くなった。しかし、PyCUDA と比較すると本提案環境には 2 割のオーバーヘッドが見られた。

4. 考察

実験から、本提案環境では Python と比較し数十倍以上の高速化を得ることができ GPGPU による効果を引き出せたことがわかる。また PyCUDA と同等の処理能力を実現できるが、ソースコードの規模が大きくなるにつれオーバーヘッドが増大する。そこで本提案環境にはキャッシュ機構を備え、2 回目以降の実行速度向上に用いた。その結果、オーバーヘッドを約 5 割削減できた。[1]

本提案環境では Python と同様の記法で記述するが、完全に同一の動作を行うものではない。しかし、一般的な制御文や関数の定義に加えて、クラス宣言を用いたオブジェクトベースのプログラミングも可能となっている。先に述べた通り、実験で作成したジュリア集合演算では複素数を表すためにクラスを用いて複素数の構造及び演算を定義している。これらの言語機能を用意することで本提案環境では CUDA 入門書[2]のサンプルコードの約半数で同等のプログラムによる記述が可能である。これらのことから、単純な配列を利用したプログラムからクラスを利用した複雑なデータ構造を持ったプログラムまで作成することができる。

実行速度においても Python と比較して速度の向上がみられ、PyCUDA と比較した場合であっても前述のキャッシュ機構を用いることで 1 割前後の速度差に収めることができた。この速度差は、PyCUDA ではカーネル関数を CUDA C での記述が必要であるとなるため、言語移植を行う前のプロトタイピングやテストを目的として本提案環境を用いる場合には許容される速度差だと考えられる。

5. おわりに

本論文では、Python を拡張し GPGPU 開発を行える環境を設計するとともに、実装例を示した。これにより新たな言語の学習、GPU へのデータ転送命令、計算資源の管理といったプログラミングを行う際の負担が軽減され、GPGPU への高い敷居が下がることが期待される。また本提案環境を利用することで GPU の高い並列処理能力を活かしたプログラムの開発が容易に行えることも期待される。

今後の課題として、対応構文の拡充と制約の削減があげられる。本提案環境でのカーネル関数における Python の構文は一部の対応となっており、開発者が本環境を用いる際に言語仕様に合わせた記述への変更が必要となる。また対応構文の一部に利用上の制約があり、開発者はこれを意識しつつ開発を行う必要がある。そのため構文を拡充させ、制約を削減することでさらに自由度の高い開発を行うことができると考える。また、GPGPU の知識を有している開発者へのオプションとして計算資源や転送命令を直接設定することができる機能を追加することで、CUDA C に近いプログラムの作成でき実際の環境への移植が容易となり、さらに高速なプログラムを作成することができると考える。

これらに加えて、型推論の対応が課題として残っている。本提案環境では開発者が記述したヒアドキュメントをアノテーションとして利用しカーネル関数内で宣言された変数の型情報を設定している。しかし、型推論を導入することでアノテーションを省略することも可能となり、開発者に引数や変数の型を提案することができる。

参考文献

[1] Andreas Klöckner, Nicolas Pinto, Bryan Catanzaro, Yunsup Lee, Paul Ivanov, Ahmed Fasih, GPU Scripting and Code Generation with PyCUDA, GPU Computing Gems, edited by Wen-mei Hwu, Elsevier Chapter 27 (2011).

[2] Jason Sanders, Edward Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley Professional, 2010

[3] PyCUDA: http://documen.tician.de/pycuda/ 2015