Vol.2017-OS-140 No.5
2017/5/16

Gears OS における並列処理

東恩納 琢偉[†1]　伊 波　立 樹[†2]　河 野 真 治[†1]

Gears OS は継続を中心とした言語で記述されており、メタ計算をノーマルレベルと分けて記述することができる。並列処理はメタ計算によって記述されており、CbC 自体には並列処理の機能はない。Gears OS のプログラムは Code Gear と Data Gear の集まりである interface によって行われる。Gears OS でのスレッドは interface の集合で出来ており、code gear data gear を接続する context という meta data gear を持つ。 並行実行する場合は新しく context を生成し、それを時分割または、物理的な CPU に割り当てることによって実現される。 つまり、context そのものがスレッドとなる。

Gears OS での同期機構は data gear を待ち合わせることによって行われる。例えば、GPU 上で実行する場合は必要な data gear を GPU 内部に転送し、それらが揃った時点で並列実行される。data gear の待ち合わせはメモリ上の data gear の meta data gear に待ち合わせ用のキューを作ることによって行われる。キューには Gears OS のスレッドつまり context meta data gear が入る。

本論文では Gears OS での並列処理の構成方法について述べる。並列処理をメタレベルで行うことにより、並列処理で重要なチューニングや性能測定あるいはデバッグをメタ計算を切り替えることにより、ノーマルレベルの計算を変更することなく行うことができることを示す。

Takui HIGASHIONNA ,[†1] Tatsuki IHA [†2] and Shinji KONO [†1]

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 1 は Code Segment 間の接続関係を表している。

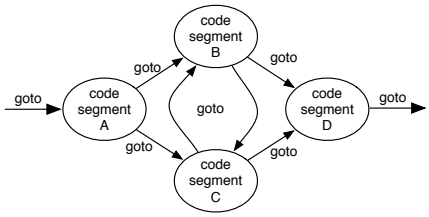

図 1　goto による Code Segment 間の接続

†1 琉球大学工学部情報工学科
Information Engineering, University of the Ryukyus.
†2 琉球大学大学院理工学研究科情報工学専攻
Interdisciplinary Information Engineering, Graduate School of Engineering and Science, University of the Ryukyus.

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。

3. 並 列 性

Code Gear が処理するのに必要な Input Data Gear と処理の実行後に出力される Output Data Gear の組を Task と呼び、Data Gear の入出力関係は Input Data Gear と Output Data Gear によって表せられるため、図**??**のようになっており、Code Gear を実行するのに必要なデータの依存関係を明確にする。
また、依存関係の無い Code Gear は自動で並列に実行される。並列実行の際には Meta Code Gear で記述された Task を Worker に投げることで行われる。

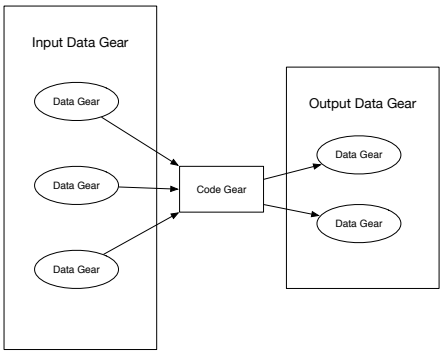

図 2 codeGear$_{d}ataGear$

4. 柔 軟 性

G 通常の処理を Computation、 Computation のための Computation を Meta Computation として扱う。
例として、 Code Gear が次に実行する Code Gear を goto で名前指定する。この継続処理に対して Meta Code Gear が名前を解釈して、処理を対応する Code Gear に引き渡す。これらは、従来の OS の Dynamic Loading Library や Command 呼び出しに対応する。名前と Code Gear へのポインタの対応は Meta Data Gear に格納される。この Meta Data Gear を Context と呼び、これは従来の OS の Process や Thread を表す構造体に対応する。Meta Computation を使用することで以下のことが可能になる。

- 元の計算を保存したデータ拡張や機能の追加
- GPU 等のさまざまなアーキテクチャでの動作
- 並列処理や分散処理の細かいチューニングや信頼性の制御
- Meta Computation は 通常の Computaiton の間に挟まれる

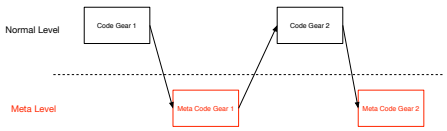

図 3 $\mathrm{Meta}_{c}ode_{g}ear$

Gears OS の構成

5. 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。

6. 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 1 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。

```
// Dequeue
__code getQueue(struct Context* context, struct Queue
     * queue, struct Node* node) {
    if (queue->first == 0)
        return;

    struct Element* first = queue->first;
    if (__sync_bool_compare_and_swap(&queue->first,
          first, first->next)) {
        queue->count--;

        context->next = GetQueue;
        stack_push(context->code_stack, &context->next
             );

        context->next = first->task->code;
        node->key = first->task->key;

        goto meta(context, Get);
    } else {
        goto meta(context, GetQueue);
    }
}
```

Code 1 $\mathrm{sync}_{d}equeue.$

GPGPU

7. 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 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 Tas k 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。

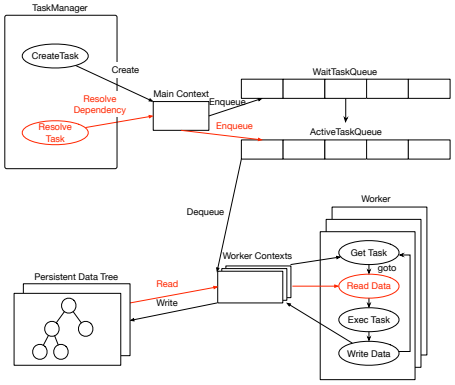

図 4 Gears OS による GPGPU

8. CUDA とは

CUDA とは NVIDIA 社が提供している並列コンピューティング用の統合開発環境で、コンパイラ、ライブラリなどの並列コンピューティングを行うのに必要なサポートを提供している。
一般的にも広く使われている GPU の開発環境である。Task(kernel) は .ptx という GPU 用のアセンブラに変換され、プログラム内部から直接 kernel を呼び出す構文を持つ API である。さらにメモリ転送と kernel 呼び出しを自分で制御する DriverAPI の 2 種類をもつ。

GPU 実装

9. CPUWoker

Worker thread で動く Task スケジューラーである。synchronized queue から Task の List を読み込み実行する。Data Gear の待ち合わせの管理を行う。CPU-Worker は receive Task という API を持ち、Task がなくなるまで繰り返す。

10. CUDAWorker の実装

CPUWorker を再利用して作成する Task スケジューラー。
CUDA ライブラリの初期化を行う以外の動作は CUDAWorker と全く同じになる。
GPU へのデータ転送及び GPU 側での Task の実行は Task の Meta Code Gear で行われる。

11. CMake

CMake はビルド自動化ツールであり、プログラムをコンパイル行う際にライブラリや動作環境によって様々な設定を行う必要がありますが、 CMake にはそれらのファイルやライブラリを探し出して実行することが出来ます。
CMake のビルドは 2 段階からなり、 CMake の制御ファイルの CMakeList.txt によってビルドするファイルを指定でき、指定されたファイルをビルドする際にはまずビルドに必要なライブラリやファイルを探し、通常のビルド環境用のビルドファイルを作成し、次にビルドするファイルにあったビルドを行います。
CMake は Windows や Linux 等、複数の OS の環境に対応している他、様々なコンパイルオプションがあり、クロスコンパイルや、ユーザーがコンパイルのルールを追加することで特殊なコンパイラや OS にも対応することが出来る。
Code Gear/Data Gear による GPGPU 処理の実装には CUDA 専用コンパイラである nvcc と Code Gear Data Gear のコンパイルを両立させる必要があるが、CMake は同時に 1 つのコンパイラしか扱えない。`|add_costom_command`—を使うことで CUDA のコンパイラ nvcc を呼び出し、マクロを用いて、Code Gear Data Gear のコンパイルを両立させた。

結論

12. ま と め

Code Gear Data Gear を用いて CUDA を利用した並列処理プログラムを記述した。CUDA 専用のコンパイラの nvcc と Code Gear Data Gear のコンパイラを CMake を用いる事
で両立させた。Gears OS での GPU の基本的な実行を確認することができた

13. 今後の課題

今後は Meta computation 部分の自動生成、GPGPU の Meta computation によるチューニングなどを行い、Gears OS における GPGPU のサポートを広げる。