The University of Osaka
Institutional Knowledge Archive

Title	GPUコンピューティングの特徴と応用事例
Author(s)	安福, 健祐
Citation	サイバーメディアHPCジャーナル. 2012, 2, p. 13-16
Version Type	VoR
URL	https://doi.org/10.18910/70529
rights	
Note	

The University of Osaka Institutional Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

The University of Osaka

GPU コンピューティングの特徴と応用事例

安福 健祐
大阪大学 サイバーメディアセンター サイバーコミュニティ研究部門

1. はじめに

ここ数年プログラムの処理速度を向上させるための並列コンピューティング手法として、GPU の利用が広がっている。GPU は、スマートフォン、タブレット、PC からスーパーコンピュータに至るまで様々なデバイスに搭載されており、本来のグラフィックス処理から汎用的な数値計算（GPU コンピューティング）も可能となっている。大阪大学サイバーメディアセンターでは、GPU コンピューティングに対応した大規模計算機システムの運用は行われていないが、本稿では近年 HPC 分野でも注目される技術として GPU コンピューティングを取り上げ、GPU の変遷と GPU コンピューティングの開発環境である CUDA の解説および応用事例について紹介する。

2. GPU の変遷

GPU は 1980 年代に登場したグラフィックスワークステーションに搭載されている「ジオメトリエンジン」に端を発し、1990 年代からは、PC やビデオゲーム機にも普及していく。GPU は元々3D-CG のリアルタイムレンダリングを実現するための専用ハードウェアであり、GPU を通した計算過程は「グラフィックスパイプライン」と呼ばれ、現在までに大きな進化を遂げてきた（図 1 参照）。

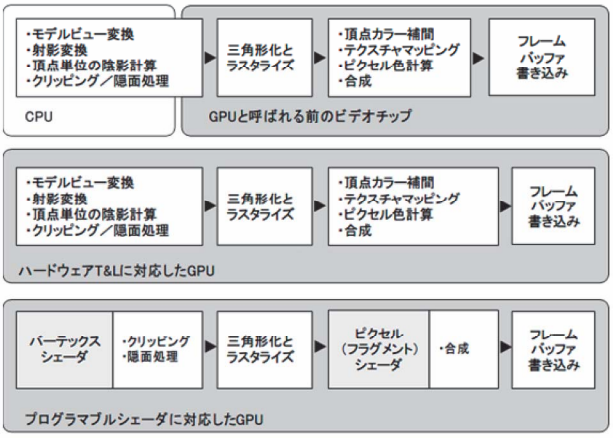

図１：グラフィックスパイプラインの変化

特に 2001 年に発表された NVIDIA 社の GPU GeForce3 には、グラフィックスパイプラインで行う処理を自由にプログラムできる「プログラマブルシェーダ」という仕組みが搭載され話題となった。プログラマブルシェーダとは GPU 上の演算プロセッサの汎用性を高めたものであり、従来の固定機能ではなく、ソフトウェアとして機能を実装できることにより、多彩なグラフィックス表現が可能になっている。その結果、GPU は安価ながら高い演算性能を持つ並列計算ハードウェアとしても注目されはじめる。プログラマブルシェーダを駆使すれば、グラフィックス処理以外の計算も GPU で実質可能となったのである。実際、科学計算分野ではタンパク質折り畳みシミュレーション、金融分野ではストックオプション価格決定シミュレーション、医療分野では MRI の画像再構成などに GPU が応用されている。またこのようにグラフィックス計算以外に汎目的で GPU を利用することは、GPGPU(General-Purpose computing on GPUs)と呼ばれるようになった。

2006 年 NVIDIA 社が発表した GPU アーキテクチャ G80 では、本格的に GPU を汎用並列計算ハードウェアとして利用するための機能がサポートされる。本来の 3D グラフィックス計算には必要のない仕組みも導入されており、例えば、汎用の統合シェーダの演算ユニットは IEEE の単精度浮動小数点演算の要件を満たすようになり、GPU 上の演算ユニットには、ビデオメモリへの任意の読み取り、書き込みアクセスに加えて、プログラムで管理できる共有メモリと呼ばれるキャッシュへのアクセスもできるようになった。また、プログラム開発環境として CUDA の提供が開始される。これまで GPGPU のプログラム開発にはシェーダと呼ばれるグラフィックス言語を駆使する必要があったが、CUDA では C 言語とほぼ同じ言語仕様になっている。NVIDIA 社は

CUDA を使って GPU を汎用計算に使うことを GPU コンピューティングと呼ぶようになり、従来のシェーディング言語で汎用計算を行っていたときの GPGPU とは区別している。ただ一般的には CUDA を使ったプログラムも GPGPU と呼ばれることが多いのが現状である。

2008 年には、新しい NVIDIA の GPU アーキテクチャとして GT200 が発表される。GT200 では、汎用並列プロセッサとしての機能がさらに強化されている。特に科学計算分野においては待望の倍精度浮動小数点演算がサポートされた。世界で初めてスーパーコンピュータに GPU が登載されたのも、GT200 アーキテクチャのものである。そのパフォーマンスは 1GPU で 1 TFLOPS に達しており、GPU が 1997 年のスーパーコンピュータの演算性能に追いついたことになる。さらに「Fermi (2010 年)」「Kepler (2012 年)」とアーキテクチャが刷新されるごとに、GPU コンピューティングとしての性能強化が図られている。

3. CUDA の特徴

ここからは GPU コンピューティングの開発環境である CUDA について簡単な解説を行い、その特徴をみていきたい。

3.1 ホストコードとデバイスコード

CUDA には「ホスト」と「デバイス」という概念がある。ホストが CPU と主記憶装置(RAM)のことを指しており、デバイスが GPU とビデオメモリ(VRAM)を指す。CUDA のプログラムは、ホスト上で動作する「ホストコード」と、デバイス上で動作する「デバイスコード」を組み合わせたものになっており、GPU だけではプログラムを動かすことはできない。ホストコードは、CPU によって逐次処理が行われるので、従来の C/C++で書かれたプログラムが動作する。一方のデバイスコードは、GPU によって並列処理を行うため専用の関数を定義する。この関数のことを「カーネル関数」と呼ぶ。CUDA のプログラムの流れは、ホストコードからデバイスコード（カーネル関数）を呼び出すことで、GPU 上で並列計算を行う仕組みになっている（図 2 参照）。

次にカーネル関数を呼び出す具体的な方法について説明する。カーネル関数の実行単位は「グリッド」と呼ばれる。その「グリッド」はいくつかの「ブロック」で構成され、「ブロック」もいくつかの「スレッド」で構成されるという階層構造になっている。そのためホストコードからカーネル関数を実行するときは、「ブロック」の数と「スレッド」の数を指定する必要がある。図 3 に実際のプログラム例を示す。関数を呼び出す際、関数名とカッコ「()」の間に三重カッコ「<<<>>>」を付けて、ブロック数とスレッド数を指定している。

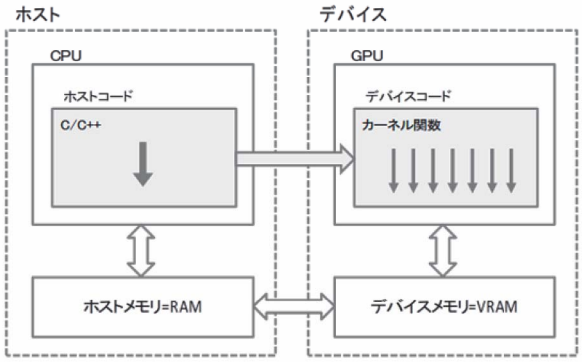

図２：ホストとデバイス

```
__global__ void kernel() {
    printf("blockID:%d, threadID:%d¥n", blockIdx.x,
threadIdx.x);
}
int main( void ) {
    kernel <<<2, 3>>>();
    cudaThreadSynchronize();
    return 0;
}
```

図３：カーネル関数を呼び出すプログラム例

CUDA の「デバイス」「グリッド」「ブロック」「スレッド」という階層は、GPU のハードウェア構成が反映されたものである。「デバイス」が一つの GPU に対応する。グラフィックボードを複数挿すことも可能なので、デバイスが複数ある場合もある。デバイスの下位概念として「グリッド」はある。1 デバイス（1GPU）に 1 グリッドが対応する。

GPU の階層構造を図 4 に示す。GPU には、「スト

リーミング・プロセッサ(SP)」という最小単位の演算プロセッサがある。SP は「CUDA コア」とも呼ばれる。この SP がいくつかまとまって「ストリーミング・マルチプロセッサ(SM)」が構成されている。このプロセッサの階層構造が、CUDA の「スレッド」と「ブロック」に対応している。つまり、ブロックが SM、スレッドが SP に対応している。ハードウェア的に見れば、同一 SM 内の SP 同士のみがスケジューラ、シェアードメモリを共有することになる。

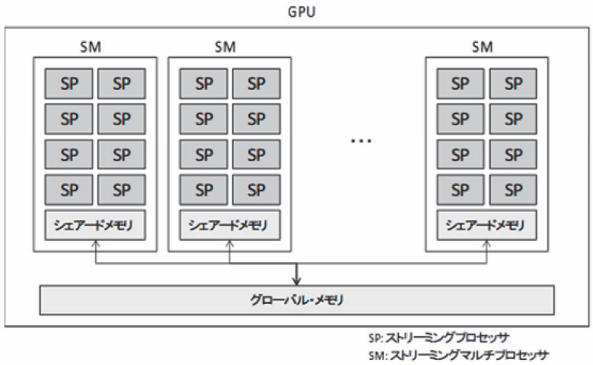

図４：GPU の階層構造

また、CUDA には「ウォープ(Warp)」という概念がある。ウォープとは、ストリーミング・マルチプロセッサ(SM)がスレッドを一つのグループにして管理する単位のことである。SM 内には、一つしか命令管理機能が登載されていないため、同じウォープ内のスレッドは同じ命令が実行される。これは SIMD(Single Instruction Multiple Data)と呼ばれるプログラミング・モデルである。一方、異なる SM 間のスレッド同士では異なる命令を実行することができ、スレッド全体でみれば、同じプログラムが実行される。これは SPMD(Single Program Multiple Data)と呼ばれるプログラミング・モデルである。SIMD がすべてのスレッドが同じ命令を実行するのに対し、SPMD は、プログラムが同じでもすべてのスレッドで同じ命令が実行されるとは限らない。例えば、プログラム内に if 文のような条件分岐があれば、スレッド間で異なる命令が実行される。しかしながら、SIMD となる同一ウォープ内のスレッドでもプログラムで条件分岐は扱える。CUDA ではその場合ウォープ単位でプログラムの分岐を判断し、ウォープ内のスレッド同士で分岐方向が異なるときは、両方の分岐先の命令が実行される。ただし、単純に二つの命令を実行してしまうと、計算結果がおかしくなるため、スレッドは命令を実行しながら、その命令を有効にするか、無効にするのかを切り替えている。このように分岐先の命令がすべて実行されるということは、分岐が多くなれば、実行する命令数が増え、その結果、実行速度の低下につながる。このような状態は「ウォープ・ダイバージェント」と呼ばれ、それを避けるため、プログラムではウォープの単位を意識することも必要となる。

3.2　CUDA のメモリモデル

CUDA のスレッドが扱えるデータは「デバイスメモリ」上のデータのみである。ホストコードとデバイスコードではそれぞれメモリ空間が異なっているため、ホストコードでホストメモリ上のデータにアクセスし、デバイスコードでデバイスメモリ上のデータにアクセスする。またホストコード側からデバイスメモリの確保やデータのコピーなど制御を行う必要があり、そのための API が用意されている。

一般的に、CPU でも GPU でもメモリへのアクセス速度が性能上のボトルネックになるケースが多い。CPU がアクセスするホストメモリ(RAM)は、その間にキャッシュメモリと呼ばれる高速小容量のメモリを介して、ホストメモリへのアクセスを減らす仕組みがある。ただプログラマがキャッシュメモリを意識するのは相当パフォーマンスを要求される一部のプログラムに限られている。GPU がアクセスするデバイスメモリ(VRAM)も、GPU がデータを読み書きする際には、GPU 上の高速小容量のオンチップメモリによってデータ転送効率を向上させる階層的なメモリモデルになっている。ただし、デバイスコードはホストコードと異なり、GPU のメモリの種類とその特徴を意識してプログラミングすることで、パフォーマンスが大きく影響される。

CUDA のメモリの種類を図 5 に示す。この中でも「シェアードメモリ」は同一ブロック内のスレッドでデータを共有することができる高速小容量のオンチップメモリであり、CUDA プログラミングを最適

化するのに非常に重要な役割を果たす。ただし、GPUのアーキテクチャが刷新されるごとに、キャッシュメモリが登載されるなど、上記のようなハードウェア構造を意識しなくても、ある程度のパフォーマンスが出るようになってきている。

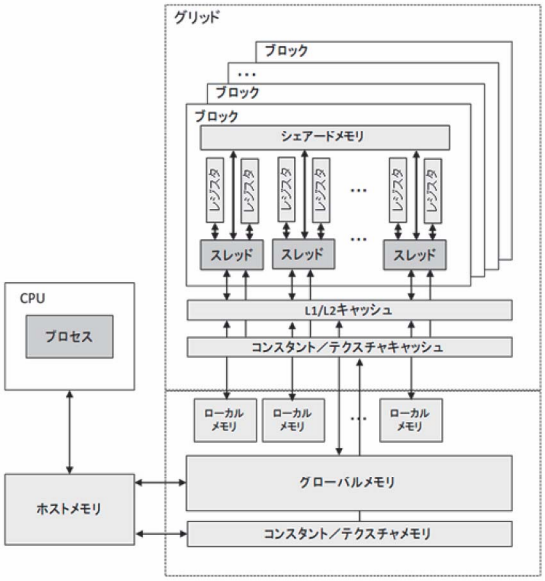

図５：CUDA のメモリモデル

4. 応用事例（群集シミュレーション）

CUDA の具体的な応用事例として群集シミュレーションを取り上げ、CPU のみの逐次処理との性能比較を行った。群れを表現するためのアルゴリズムとしては Boids[1]を使用する。Boids とは、Separation、Alignment、Cohesion の三つの行動ルール（図 6）によって鳥や魚の群れを表現するものであり、各個体の動きを運動方程式によって記述し、タイムステップごとにその位置を更新していくものである。

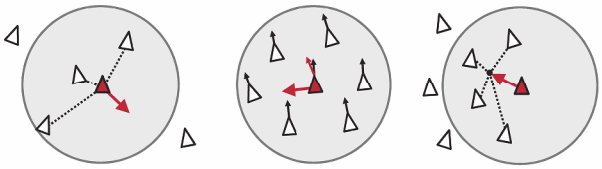

図６：Boids モデルの三つの行動ルール

個体の数を変化させながら、1 ステップにおけるカーネル関数を実行するのに要した時間を計測した結果を図 8 に示す。CPU の計算では、個体の数が増えると、処理時間が指数関数的に増加しているのに対し、GPU のほうはほぼ線形となっており、スケーラビリティに優れているといえる。個体の数が少ない段階、特に個体が 10 程度であれば CPU のほうがかえって高速であるが、個体数が増えていくごとに、GPU のほうが高速に処理できていることがわかる。

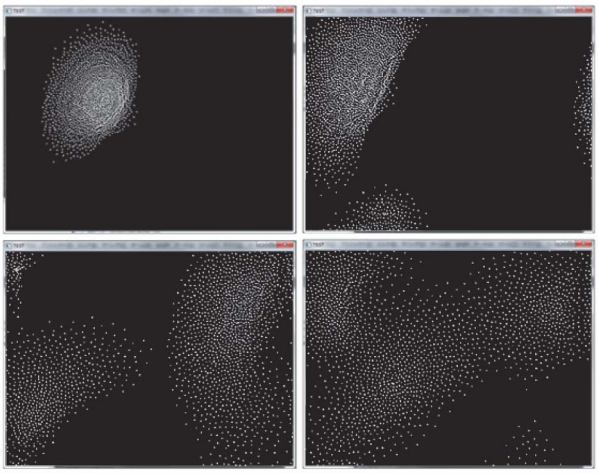

図７：シミュレーションの実行画面

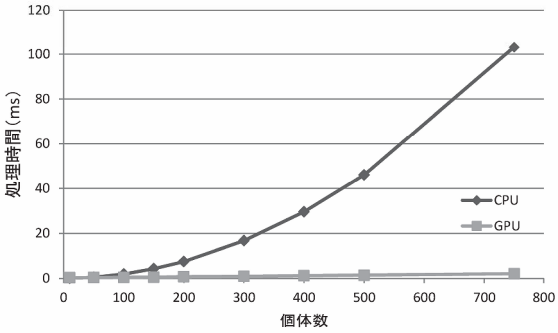

図８：CPU と GPU の比較

5. おわりに

GPU を活用した並列コンピューティングによって、従来リアルタイムでは困難だった汎用計算をインタラクティブにシミュレーションすることができれば、数値計算だけで解析を行っていた研究が飛躍的に進む可能性がある。また、GPU コンピューティングは、本来 GPU が性能を発揮する可視化との連携が取りやすいのも特徴であろう。

参考文献

(1) Craig W. Reynolds: Flocks, Herds, and Schools: A Distributed Behavioral Model, Siggraph’87, pp.25-34, 1987.7