|  |
| --- |
| 博士課程教育リーディングプログラム  実世界データ循環学リーダー人材育成プログラム |
| データツールファースト |
| CUDA入門 |

|  |
| --- |
| 名古屋大学  2014/03/02 |

* NVIDIA、CUDA、GeForceは、米国におけるNVIDIA社の登録商標または商標です。
* Linuxは、Linus Torvaldsの米国およびその他の国における登録商標あるいは商標です。
* その他の、記載されている会社名、製品名は、各社の登録商標または商標です。

目次

[1. はじめに 1](#_Toc381565335)

[1.1 背景 1](#_Toc381565336)

[1.2 本書について 1](#_Toc381565337)

[2. CUDA 3](#_Toc381565338)

[2.1 CUDAとは 3](#_Toc381565339)

[2.2 GPUとは 4](#_Toc381565340)

[3. GPUアーキテクチャ 5](#_Toc381565341)

[3.1 基本的な構造 5](#_Toc381565342)

[3.2 アーキテクチャの比較 7](#_Toc381565343)

[4. CUDAプログラミング 11](#_Toc381565344)

[4.1 ホストとデバイス 11](#_Toc381565345)

[4.2 並列計算 12](#_Toc381565346)

[4.3 WARP 14](#_Toc381565347)

[4.4 階層的メモリモデル 15](#_Toc381565348)

[4.5 性能の改善 17](#_Toc381565349)

[4.6 ランタイムAPI 20](#_Toc381565350)

[4.7 拡張機能 23](#_Toc381565351)

[4.7.1 dim3宣言 24](#_Toc381565352)

[4.7.2 カーネル関数の実行 25](#_Toc381565353)

[4.7.3 ビルトイン変数 26](#_Toc381565354)

[4.7.4 修飾子 27](#_Toc381565355)

[4.8 サンプルコード 28](#_Toc381565356)

[4.8.1 Hello, World! 28](#_Toc381565357)

[4.8.2 データの転送 29](#_Toc381565358)

[4.8.3 行列計算 31](#_Toc381565359)

[5. 高速化の手法 37](#_Toc381565360)

[5.1 はじめに 37](#_Toc381565361)

[5.2 シェアードメモリの利用 37](#_Toc381565362)

[練習問題 38](#_Toc381565363)

[5.3 テクスチャメモリの利用 38](#_Toc381565364)

[練習問題 39](#_Toc381565365)

[5.4 page-lockedなホストメモリの利用 40](#_Toc381565366)

[練習問題 40](#_Toc381565367)

[5.5 ゼロコピーホストメモリの利用 40](#_Toc381565368)

[練習問題 41](#_Toc381565369)

[5.6 その他 41](#_Toc381565370)

[6. CUDA Toolkit 43](#_Toc381565371)

[6.1 コンパイルと実行 43](#_Toc381565372)

[6.2 デバイスコードの確認 45](#_Toc381565373)

[6.3 デバッグ 47](#_Toc381565374)

[7. 事例 50](#_Toc381565375)

[7.1 HPC 50](#_Toc381565376)

[7.2 クラウドコンピューティング 52](#_Toc381565377)

[7.3 ビックデータ解析 52](#_Toc381565378)

[8. 参考文献 53](#_Toc381565379)

[9. 変更履歴 54](#_Toc381565380)

# はじめに

## 背景

今日では、プロセッサのマルチコアは当たり前になりました。シングルコアで性能向上を図るには動作周波数を上げるのが簡単ですが、それに伴い消費電力とプロセッサの温度が上がるので、シングルコアには限界があります。

プロセッサのコアは、ますます増え続けており、「メニーコア」の流れは明らかなものになっています。マルチコアとメニーコアの具体的な線引きはありませんが、従来のデュアルコアやクアッドコアといった構成よりも大量のプロセッサで構成されたものが、メニーコアと呼ばれています。

すでにHPC(High Performance Computing)の分野では、数百〜数十万ものコアで構成されたシステムが運用されており、その計算ノードにはメニーコアプロセッサが搭載されています。このようなシステムから得られる大規模な計算資源は、自動車の衝突シミュレーションや、遺伝子配列やタンパク質の解析など、様々な分野において利用され始めています。

昨今、上記のような大規模計算のプロセッサとして、GPUが使われるようになっています。GPUは汎用プロセッサと比較して、価格性能比が高く、消費電力が小さいという利点があるためです。

「CUDA」は、GPUを使った汎用計算のプログラミングを支援するための、統合開発環境です。CUDAはNVIDIA社が無償で公開しているので、PCにNVIDIA社のビデオカードを装着すれば、ユーザはいつでもCUDAを試すことができます。

## 本書について

本書は、CUDAを使用してGPUで汎用計算を行うための入門書です。GPUのアーキテクチャを理解すること、CUDAの基本的な使い方を習得することを目的にしています。

本書の対象となる学生は、C言語のプログラミング経験、プロセッサやメモリなどのコンピュータの基礎知識を修得していることを前提にしています。

2章では、CUDAとGPUの概要について説明します。3章では、NVIDIA社のGPUアーキテクチャの基本的な構造を説明し、実際の製品を例にアーキテクチャの性能を比較します。4章では、CUDAプログラミングで使われる用語や概念、ランタイムAPI、拡張機能について説明します。5章では、CUDA Toolkitの使い方について説明します。6章では、HPC、クラウドコンピューティング、ビックデータ解析を例に、GPUの事例を紹介します。

# CUDA

CUDAを利用するときに、その本来の目的を知れば、どのような分野に適しているか理解しやすいでしょう。

ここでは、CUDAとGPUの概要について説明します。

## CUDAとは

「CUDA」とは、Compute Unified Device Architectureの略で、GPUを使った汎用計算プラットフォームとプログラミングモデルです。現在有名なGPUベンダーにはNVIDIA社とAMD社がありますが、CUDAはNVIDIA社によって開発されています。

対応するハードウェアは、HPC向けのTeslaから、デスクトップワークステーション向けのQuadro、デスクトップ・ノートブック向けのGeForceまで、多数あります。詳しくは、下記のWebページを参照してください。

CUDA GPUs | NVIDIA Developer Zone

<https://developer.nvidia.com/cuda-gpus>

対応するOSは、下記の通りです。(CUDA 5.5の場合)

* Windows XP / Vista / 7 / 8.0 / 8.1 32bit/64bit
* Red Hat Enterprise Linux 5.5 / 6.0 64bit
* Fedora 18 64bit
* OpenSUSE 12.2 64bit
* SUSE Linux Enterprise Server 11 SP1/2 64bit
* Ubuntu 12.04 / 12.10 32bit/64bit
* Ubuntu 10.04 64bit
* Mac OS X 10.7 / 10.8 / 10.9

## GPUとは

「GPU」とは、Graphics Processing Unitの略で、画像処理に特化した演算装置です。身近な例では、PCに装着するビデオカードやチップセットなどに内蔵されています。

GPUは、コンピュータゲームの3Dグラフィックスなどで発展したハードウェアです。しかし、2001年にプログラマブルなGPUが発表されたことで、画像処理以外の、汎用的な目的にも利用できるのではないかという考えが現れ始めました。この考えは、GPGPU(General-Purpose computations on Graphics Processing Units:GPUによる汎用計算)と呼ばれています。

初期のGPGPUでは、GPUを制御するために、OpenGLやDirectXのようなグラフィックスのAPIを使用していました。このため、汎用計算をするためには、それらのAPIを勉強する必要がありました。また、それらのAPIが持つプログラミング上の制約に従わなければなりませんでした。

2006年に発表されたCUDAは、GPGPUを支援するために開発されました。CUDAは、汎用計算を無理なく実現できるように、これまでのGPUプログラミングの制限を、克服したものだったのです。

# GPUアーキテクチャ

CUDAを使ってGPUの性能を引き出すためには、GPUのアーキテクチャを理解した上で、適切にプログラミングしなければなりません。

ここでは、NVIDIA社のGPUアーキテクチャの基本的な構造を説明し、実際の製品を例にアーキテクチャの性能を比較します。

## 基本的な構造

NVIDIA社のGPUアーキテクチャの基本的な構造は、以下のようになっています。

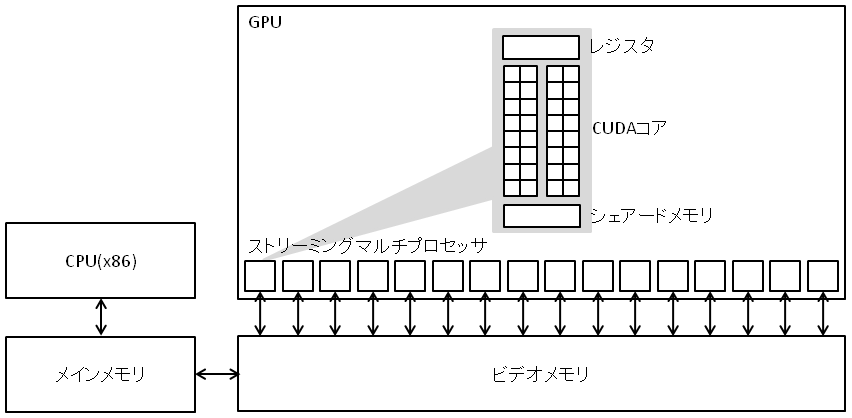


図 1 GPUアーキテクチャ

* ストリーミングマルチプロセッサ

GPUには、多数の「ストリーミングマルチプロセッサ」(SM)と呼ばれる演算装置が内蔵されています。SMの搭載数は、製品のグレードによって異なります。例えば、GeForce GTX 580では、16個のSMが搭載されています。

* CUDAコア

SMは、「CUDAコア」と呼ばれる単純な演算装置と、レジスタやシェアードメモリなどで構成されています。CUDAコアは、以前はストリーミングプロセッサ(SP)と呼ばれていました。1つのSMを構成しているCUDAコアの数は、製品の世代によって異なります。例えば、Fermiアーキテクチャでは、32個のCUDAコアが搭載されています。GeForce GTX 580はFermiのため、32 × 16 = 512個のCUDAコアが搭載されています。

* ビデオメモリ

GPU〜ビデオメモリ間は、メモリインタフェースで接続されています。CPU〜メインメモリ間と比較して、非常に大きなメモリバンド幅を持っており、高速にメモリにアクセスできます。例えば、GeForce GTX 580では、192GBytes/secでデータ転送できます。

GPUのアーキテクチャは、多数の演算装置を持ち、メモリと高速なデータ転送が可能なため、並列計算に適していると言えます。

* CPU(x86)

CPUは、CUDAのランタイムAPIや拡張機能を使って、GPUの制御やメインメモリ～ビデオメモリ間のデータ転送を行います。

* メインメモリ

メインメモリ〜ビデオメモリ間は、PCI Expressを介して、データがDMA転送されます。

## アーキテクチャの比較

NVIDIA社のGPUのアーキテクチャで、「Fermi」と「Kepler GK110」を採用した製品を例に、GPUの性能を比較します。

Fermiは、NVIDIA社の前世代のGPUアーキテクチャです。Fermiでは、1つのSMは32個のCUDAコアによって構成されます。GPUには、16個のSMが搭載され、合計512個のCUDAコアが搭載されます。また、64bitのメモリコントローラが6個搭載され、合計384bit幅でメモリアクセスが可能です。

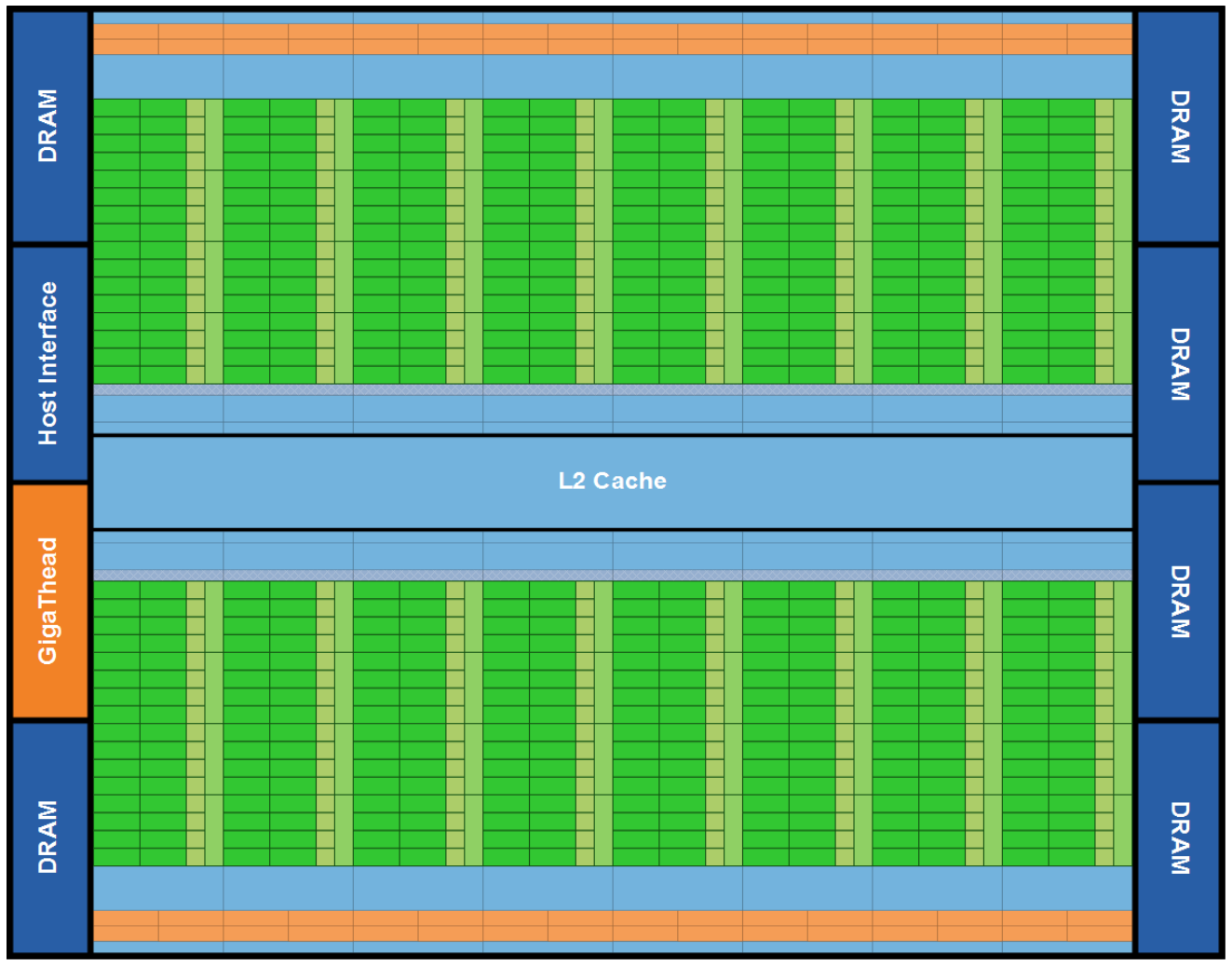


図 2 Fermiアーキテクチャ

Kepler GK110は、NVIDIA社の現世代のGPUアーキテクチャです。Keplerでは、SMはストリーミングマルチプロセッサエクストリーム(SMX)に置き換えられ、1つのSMXは192個のCUDAコアによって構成されます。Fermiと比較して、SMXを構成するCUDAコアが非常に増加したため、並列に命令を実行する機能が強化されています。GPUには、15個のSMXが搭載され、合計2880個のCUDAコアが搭載されます。また、Fermiと同様に、64bitのメモリコントローラが6個搭載され、合計384bit幅でメモリアクセスが可能です。

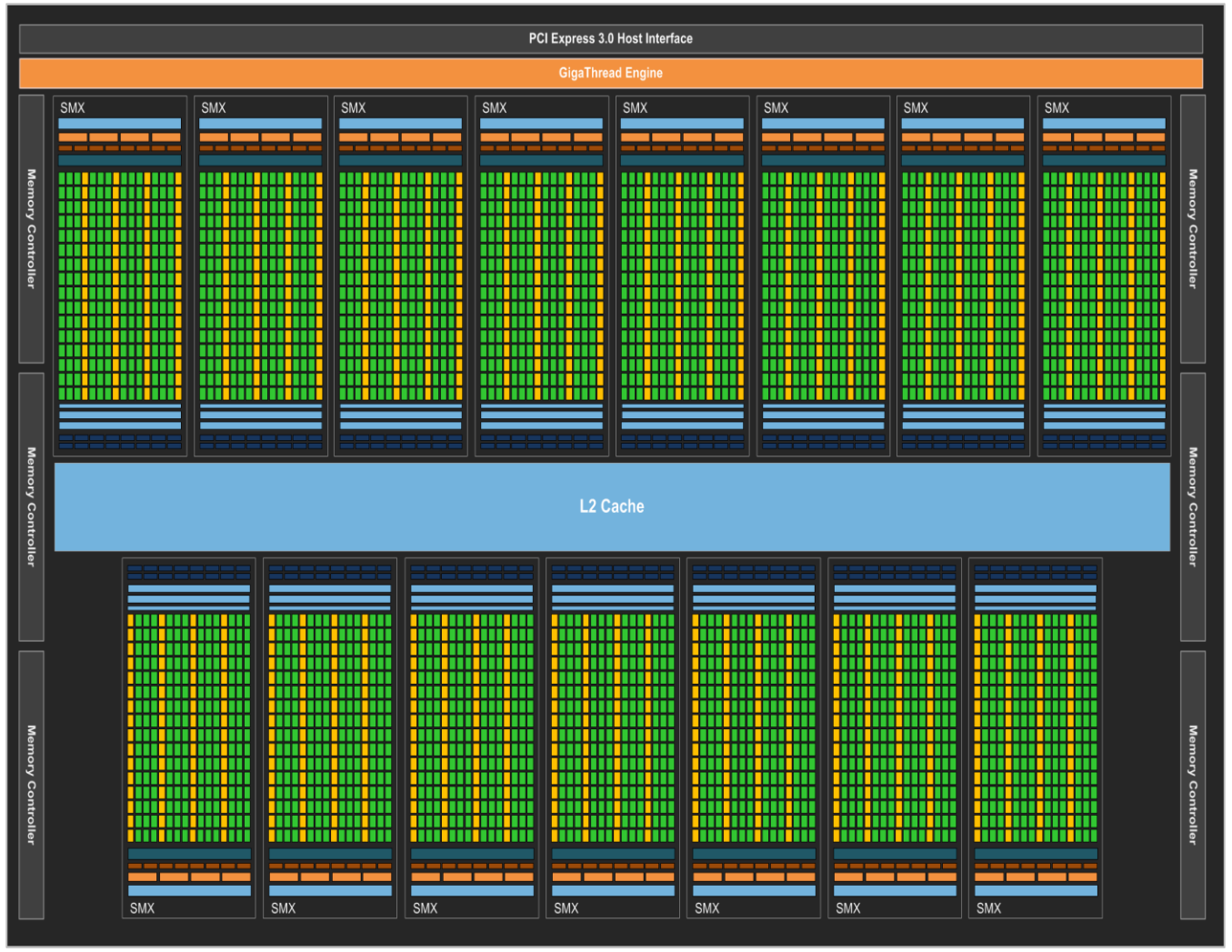


図 3 Kepler GK110アーキテクチャ

以下に、ハイエンドユーザ向けモデルのビデオカードGeForce GTX 580、GeForce GTX 780 Tiと、HPC向けモデルのTesla K20Xの仕様を挙げます。

表 1製品の仕様

|  |  |  |  |
| --- | --- | --- | --- |
| **項目** | **GeForce GTX 580** | **GeForce GTX 780 Ti** | **Tesla K20X** |
| GPUアーキテクチャ | Fermi | Kepler GK110 | Kepler GK110 |
| CUDAコア数 | 512 | 2880 | 2688 |
| 倍精度ユニット数 | 256 | 960 | 896 |
| CUDAコアクロック(MHz) | 1544 | 875 | 732 |
| 単精度演算 ピーク性能(GFLOPS)\* | 1581 | 5040 | 3935 |
| 倍精度演算 ピーク性能(GFLOPS)\* | 197 | 210 | 1311 |
| メモリデータレート(MHz) | 4008 | 7000 | 5200 |
| メモリインタフェース幅(bit) | 384 | 384 | 384 |
| メモリバンド幅(GBytes/sec) | 192 | 336 | 249 |

\*FLOPS(Floating-point Operations Per Second):1秒間に浮動小数点演算が何回できるか

理論的なピーク性能は、現状のNVIDIA社のGPUが、1クロックで実行できる最大の浮動小数点演算の回数が2であることから、以下の式で求められます。

* 単精度演算

CUDAコア数 × CUDAコアクロック(MHz) × 2

* 倍精度演算

GeForce GTX 580は、倍精度ユニットが、CUDAコアクロックの1/4で動作しています。

倍精度ユニット数 × CUDAコアクロック(MHz) × 2 ÷ 4

GeForce GTX 780 Tiは、倍精度ユニットが、CUDAコアクロックの1/8で動作しています。

倍精度ユニット数 × CUDAコアクロック(MHz) × 2 ÷ 8

Tesla K20Xは、倍精度ユニットが、CUDAコアクロックで動作しています。

倍精度ユニット数 × CUDAコアクロック(MHz) × 2

理論的なメモリバンド幅は、以下の式で求められます。

* メモリバンド幅

メモリデータレート(MHz) × メモリインタフェース幅(bit) ÷ 8(bit/Byte)

上記より、GeForce GTX 780 TiはGeForce GTX 580と比較して、単精度演算のピーク性能が約3.2倍に上がっています。これは、Keplerになって、GPUに内蔵されたCUDAコア数が増大したためです。しかし、倍精度演算のピーク性能は、目に見えて上がってはいません。これは、倍精度ユニットの動作クロックが低いためです。また、メモリバンド幅が約1.7倍に上がっています。メモリインタフェース幅に違いはありませんが、メモリデータレートが向上しているためです。

Tesla K20Xに注目すると、GeForce GTX 780 Tiと比較して倍精度演算のピーク性能が約6.2倍に上がっており、倍精度演算に重点を置いていることが分かります。

# CUDAプログラミング

CUDAでは、CPUで動作する処理については、CUDAを意識せずに、C言語やC++言語と同じようにプログラミングできます。GPUに対して何らかの操作をする場合は、CUDAのランタイムAPIや拡張機能を使う必要があります。

ここでは、CUDAプログラミングで使われる用語や概念、ランタイムAPI、拡張機能について説明します。

## ホストとデバイス

CUDAでは、CPUとGPUの両方が動作します。これらは特に、CPUやメインメモリを「ホスト」、GPUやビデオメモリを「デバイス」と呼び、区別しています。

CUDAのプログラムは、以下の2つの部分から構成されています。

* ホストコード

CPUで動作するコードを、「ホストコード」と呼びます。ホストコードの基本的な役割は、ホスト〜デバイス間のデータ転送や、デバイスコードの起動です。

* デバイスコード

GPUで動作するコードを、「デバイスコード」と呼びます。これは、「カーネル」や、特に関数を指して「カーネル関数」と、呼ばれることもあります。デバイスコードは、GPUで並列に実行されます。並列度の高い問題をデバイスコードで実行させることで、性能向上を図るのが、CUDAの基本的な戦略になります。

以下に、CUDAの単純なプログラムの流れを挙げます。

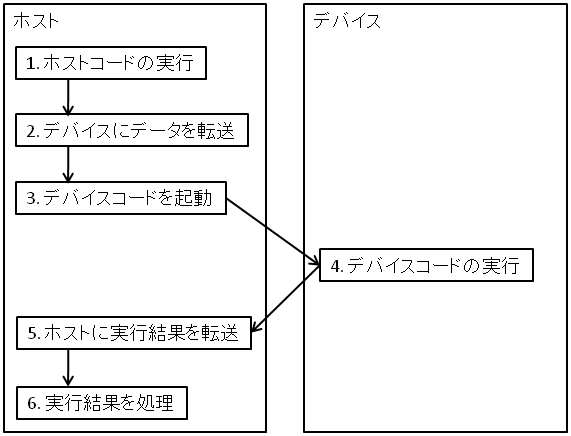


図 4 プログラムの流れ

## 並列計算

GPUでは、デバイスコードは「スレッド」と呼ばれる単位で、並列に実行されます。これは、UNIXのスレッドプログラミングにおける、スレッドに似た概念です。ただし、GPUはCPUと比較して、スレッドを実行できる演算装置が非常に多いです。このため、CUDAでは、「ブロック」と「グリッド」という概念を定義して、階層的にスレッドを管理するようにしています。

以下で、スレッド、ブロック、グリッドを説明します。

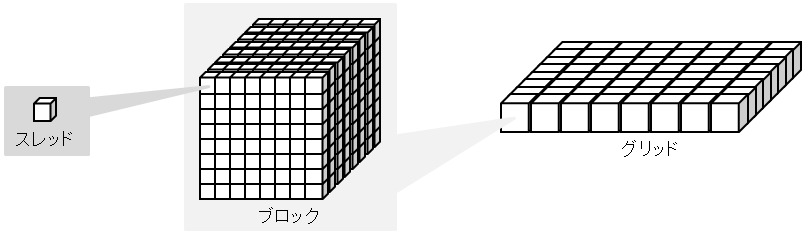


図 5 スレッド ブロック グリッド

* スレッド

スレッドは、デバイスコードを実行する最小単位です。スレッドは、CUDAコアと対応しています。それぞれのスレッドが、CUDAコアでデバイスコードを実行します。

* ブロック

ブロックは、スレッドをx方向、y方向、z方向の3次元に配置して、まとめたものです。ブロックは、SMと対応しています。1つのブロックは、割り当てられたSMから移動しません。

* グリッド

グリッドは、ブロックをx方向、y方向、z方向の3次元に配置して、まとめたものです。グリッドは、デバイスと対応しています。デバイスコードは、1つのグリッドとしてデバイスに渡され、そのグリッドに属するすべてのスレッドが、デバイスで実行されます。

デバイスコードを実行するためには、CUDAの拡張機能を使ってカーネル関数を呼び出します。このとき、スレッド、ブロック、グリッドの数を指定する必要があります。

また、デバイスコードを実行するすべてのスレッドは、同じカーネル関数を実行します。このような並列計算の方式は、「SPMD」(Single Program, Multiple Data)と呼ばれます。

## WARP

NVIDIA社のGPUは、内部では「WARP」(縦糸)という単位で、スレッドを管理しています。1WARPは、32スレッドのまとまりです。CUDAコアは命令を実行するとき、1WARP単位で同じ命令を実行します。

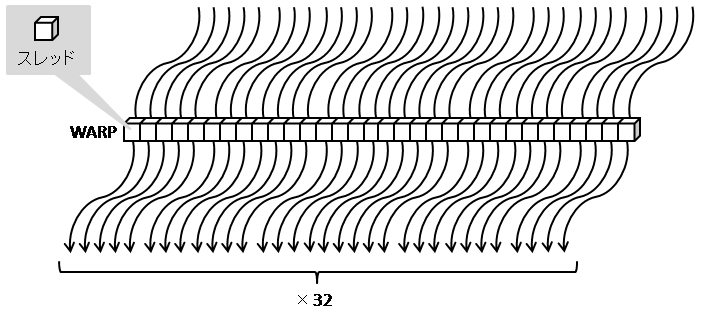


図 6 WARP

このように、WARP単位で見た場合、CUDAコアは、「SIMD」(Single Instruction, Multiple Data)と呼ばれる方式で動作しています。(NVIDIA社では、この方式を、CPUのSIMDと区別するために、「SIMT」(Single Instruction, Multiple Thread)と呼んでいます。)

## 階層的メモリモデル

NVIDIA社のビデオカードに搭載されているメモリは、以下のように分類されます。

表 2 メモリの分類

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **分類** | **場所** | **速度** | **アクセス** | **スコープ** | **サイズ\*** |
| レジスタ | SM | 高速 | 読み書き | スレッド | 65536個/  ブロック |
| ローカルメモリ | ビデオメモリ | 低速 | 読み書き | スレッド | コンパイル時に  決定 |
| シェアードメモリ | SM | 高速 | 読み書き | ブロック | 49152Bytes/  ブロック |
| グローバルメモリ | ビデオメモリ | 低速 | 読み書き | グリッド | 6144MBytes |
| コンスタントメモリ | ビデオメモリ | 低速 | 読み | グリッド | 65536Bytes |
| テクスチャメモリ | ビデオメモリ | 低速 | 読み | グリッド | 実行時に決定 |

\*サイズはGeForce GTX TITANの場合の値

メモリは、以下のような階層的なモデルになっています。

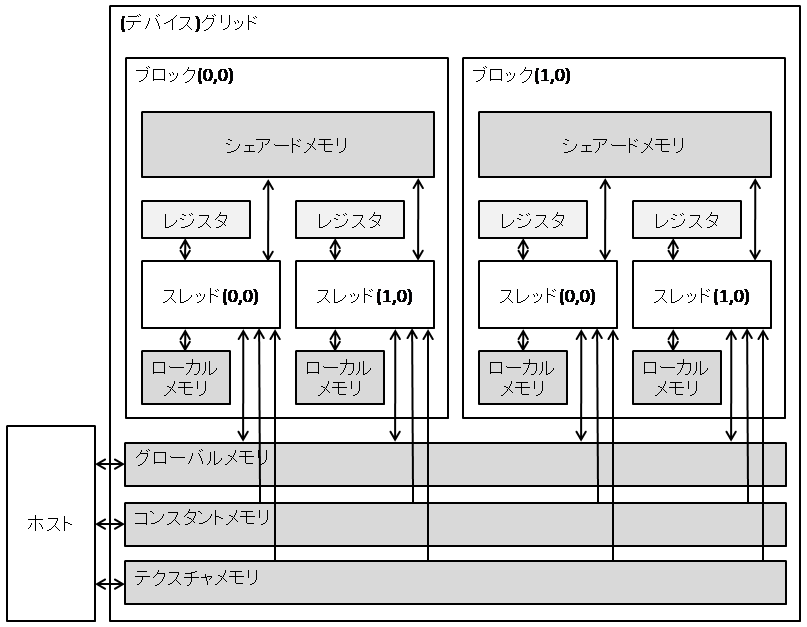


図 7 階層的メモリモデル

* レジスタ

カーネル関数が使用する変数の値は、「レジスタ」に格納されます。レジスタは、最も高速にアクセスできるメモリです。スレッド毎に独立しており、スレッド間で値は共有されません。例えば、Fermiアーキテクチャでは、SM毎に32768個のレジスタが使えます。

* ローカルメモリ

レジスタが不足した場合、レジスタの値は「ローカルメモリ」に退避されます。ローカルメモリは、レジスタと比較して低速です。高速な計算が必要な場合は、できる限り使用してはいけません。

* シェアードメモリ

カーネル関数の引数は、「シェアードメモリ」に格納されます。また、これは静的に確保することもできます。シェアードメモリは、レジスタと同等に高速です。SM毎に独立しており、同じブロックのスレッド間で値は共有されます。

* グローバルメモリ

「デバイスメモリ」(CUDAでは、ビデオメモリをデバイスメモリと呼びます。)の確保や、ホスト〜デバイス間のデータ転送は、「グローバルメモリ」が利用されます。グローバルメモリは、レジスタと比較して低速です。すべてのブロックのすべてのスレッドから、アクセスが可能です。

* コンスタントメモリ

「コンスタントメモリ」は、カーネル関数の定数を格納するために利用します。スレッドからは読み出しのみ可能で、専用のキャッシュを持っており、グローバルメモリよりも高速です。

* テクスチャメモリ

「テクスチャメモリ」は、テクスチャユニットと呼ばれる、3Dグラフィックスを高速化する演算装置が利用します。スレッドからは読み出しのみ可能で、専用のキャッシュを持っており、グローバルメモリよりも高速です。

レジスタやメモリの使用量は、コンパイル時に--ptxas-options=-vオプションを指定することで、表示することができます。

## 性能の改善

GPUの性能を引き出すためには、WARPやメモリモデルの内部構造を理解して、プログラミングする必要があります。

以下に、いくつかのポイントを簡単に説明します。

* WARPダイバージェント

「WARPダイバージェント」を減らすようにプログラミングしましょう。NVIDIA社のGPUは、WARP中のスレッドに異なる制御フローがある場合、その数だけの実行パスが必要となります。これによって、実行時間が増加する問題をWARPダイバージェントと呼びます。

例えば、以下のようなif文があった場合、奇数番目のスレッドと偶数番目のスレッドで、2つの実行パスが必要です。

|  |
| --- |
| if (threadIdx.x & 1)  a = a + 2;  else  a = a + 1; |

この場合、片方のパスを実行した後、もう片方を実行するため、実行時間が増加します。

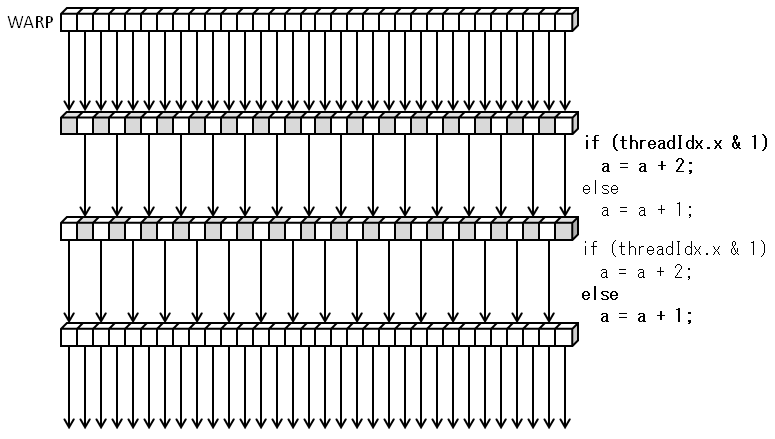


図 8 WARPダイバージェント

* スレッドの数

同時に動作するスレッドは、WARPを構成するスレッドの数(32)の倍数にしましょう。例えば、42スレッドで動作する場合、最初に32スレッドが動作した後、10スレッドが動作します。このとき、残りの22スレッドが割り当てられたCUDAコアは動作しないため、無駄になります。

* メモリアクセス

メモリアクセスを最適化するために、「コアレッシング」や「バンクコンフリクト」などを意識しましょう。

コアレッシングとは、複数のスレッドのグローバルメモリに対するアクセスを、まとめて実行する仕組みです。

例えば、以下のように隣接するスレッドが隣接するアドレスにアクセスする場合、メモリアクセスがまとめて実行され、高速に動作します。

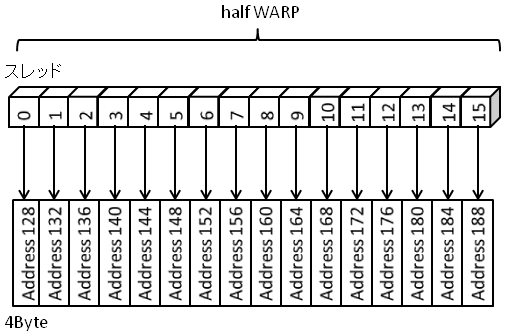


図 9 コアレッシング

バンクコンフリクトとは、シェアードメモリの同時アクセスできない範囲(バンク)にアクセスが集中することで、性能が低下する問題です。

例えば、以下のように偶数番目のバンクに対して2つのスレッドがアクセスする場合、バンクに対する同時アクセスはできないため、順番に実行され性能が低下します。

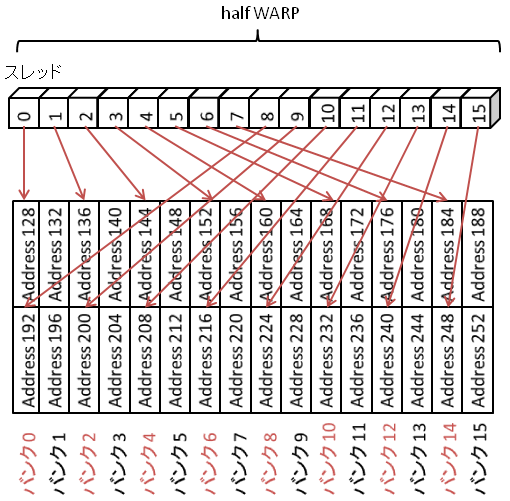


図 10 バンクコンフリクト

## ランタイムAPI

デバイスメモリの確保や、ホスト〜デバイス間のデータ転送などは、CUDAのランタイムAPIを使用して実現できます。

ランタイムAPIの詳細は、以下のURLを参照して下さい。

<http://docs.nvidia.com/cuda/cuda-runtime-api>

CUDAのランタイムAPIは、cuda〜から始まる関数名になっています。

以下に、よく使う関数を挙げます。

* cudaError\_t cudaSetDevice ( int device )

使用するデバイスの設定

引数

device デバイスコードを実行するデバイスの番号

戻り値

cudaSuccess 成功

cudaErrorInvalidDevice エラー

(無効なデバイスを指定したため)

cudaErrorDeviceAlreadyInUse エラー

(すでにデバイスが使用中であったため)

* cudaError\_t cudaMalloc ( void\*\* devPtr, size\_t size )

デバイスメモリの確保

引数

devPtr デバイスメモリのアドレスのポインタ

size 確保するサイズ(単位: Byte)

戻り値

cudaSuccess 成功

cudaErrorMemoryAllocation エラー

(十分なメモリを確保できなかったため)

* cudaError\_t cudaFree ( void\* devPtr )

デバイスメモリの解放

引数

devPtr 解放するデバイスメモリのアドレス

戻り値

cudaSuccess 成功

cudaErrorInvalidDevicePointer エラー

(無効なアドレスを指定したため)

cudaErrorInitializationError エラー

(ランタイムが初期化できなかったため)

* cudaError\_t cudaMallocHost ( void\*\* ptr, size\_t size )

page-lockedなホストメモリの確保

(ホストメモリは、malloc()やcalloc()でも確保できます。しかし、ランタイムAPIを使用することで、ホスト〜デバイス間を高速にデータ転送できる(page-locked)領域を、確保できます。)

引数

ptr ホストメモリのアドレスのポインタ

size 確保するサイズ(単位: Byte)

戻り値

cudaSuccess 成功

cudaErrorMemoryAllocation エラー

(十分なメモリを確保できなかったため)

* cudaError\_t cudaFreeHost ( void\* ptr )

page-lockedなホストメモリの解放

引数

ptr 解放するホストメモリのアドレス

戻り値

cudaSuccess 成功

cudaErrorInitializationError エラー

(ランタイムが初期化できなかったため)

* cudaError\_t cudaMemcpy ( void\* dst, const void\* src, size\_t count,   
   cudaMemcpyKind kind )

ホスト〜デバイス間のデータ転送

引数

dst 転送先のメモリアドレス

src 転送元のメモリアドレス

count 転送するサイズ(単位: Byte)

kind 転送のタイプ

表 3 転送のタイプ

|  |  |  |
| --- | --- | --- |
| **種類** | **転送元** | **転送先** |
| cudaMemcpyHostToHost | ホストメモリ | ホストメモリ |
| cudaMemcpyHostToDevice | ホストメモリ | デバイスメモリ |
| cudaMemcpyDeviceToHost | デバイスメモリ | ホストメモリ |
| cudaMemcpyDeviceToDevice | デバイスメモリ | デバイスメモリ |

戻り値

cudaSuccess 成功

cudaErrorInvalidValue エラー

(値が許可された範囲内になかったため)

cudaErrorInvalidDevicePointer エラー

(無効なアドレスを指定したため)

cudaErrorInvalidMemcpyDirection エラー

(転送のタイプが不正であったため)

## 拡張機能

CUDAは、GPGPUを支援するために、標準のC言語をいくつか拡張しています。

拡張機能の詳細は、以下のURLを参照して下さい。

<http://docs.nvidia.com/cuda/cuda-c-programming-guide>

以下に、よく使う機能を挙げます。

### dim3宣言

CUDAでは、3次元ベクトルの整数型を宣言するために、「dim3」を使います。これは、グリッドやブロックを宣言するときなどに、利用されます。

例えば、以下のように宣言できます。

|  |
| --- |
| dim3 a(2, 4, 8); |

aは、構造体として宣言されます。このとき、構造体のメンバには、以下のような値が代入されます。

|  |
| --- |
| a.x = 2; a.y = 4; a.z = 8; |

以下のように宣言することもできます。

|  |
| --- |
| dim3 a; |

このとき、構造体のメンバには、以下のような値が代入されます。

|  |
| --- |
| a.x = 1; a.y = 1; a.z = 1; |

メンバの値は、宣言後、以下のように変更することもできます。

|  |
| --- |
| dim3 a;  a.x = 2; a.y = 4; a.z = 8; |

また、以下のように省略することもできます。

|  |
| --- |
| dim3 a(2), b(4, 8); |

このとき、構造体のメンバには、以下のような値が代入されます。

|  |
| --- |
| a.x = 2; a.y = 1; a.z = 1;  b.x = 4; b.y = 8; b.z = 1; |

### カーネル関数の実行

CUDAでは、カーネル関数を実行するために、「<<<>>>」を使います。ここには、カーネル関数が実行時に使用する、グリッドやブロックのサイズなどを指定できます。

例えば、引数がa,b,cのカーネル関数は、以下のように実行できます。

|  |
| --- |
| func\_gpu<<< Dg, Db, Ns, S >>>(a, b, c); |

<<<>>>に指定できる値は、以下のようになっています。

* Dg

Dgは、グリッドのサイズをdim3型で指定します。Dgは省略できません。

* Db

Dbは、ブロックのサイズをdim3型で指定します。Dbは省略できません。

* Ns

Nsは、シェアードメモリのサイズ(単位: Byte)をsize\_t型で指定します。Nsを省略した場合、0が指定されます。

* S

Sは、ストリーム(0以上の整数)をcudaStream\_t型で指定します。Sを省略した場合、0が指定されます。ストリーム0は同時に1つしか流れません。

ストリームとは、カーネル関数やデータ転送など、GPUが実行する処理の一連の流れを表しています。ストリームを指定して関数を呼び出すことで、同じストリームの処理は呼び出した順番に実行されることが保証されます。

Dg,Dbは、以下のように省略することもできます。

|  |
| --- |
| func\_gpu<<< 2, 4 >>>(a, b, c); |

このとき、<<<>>>には、以下のようなDg,Dbが指定されます。

|  |
| --- |
| dim3 Dg(2, 1, 1), Db(4, 1, 1); |

Dg,Dbの上限は、Compute Capabilityによって異なります。Compute Capabilityとは、GPUが持つ機能を規定したものです。レベルが高いほど、より多くの機能を備えた新しい世代のGPUであることを示します。詳細は、以下のURLを参照して下さい。

<http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities>

### ビルトイン変数

CUDAでは、カーネル関数のスコープで、宣言なしに使うことができる変数があります。これらは、読み出しのみ可能です。

主な変数は以下の通りです。

* gridDim

gridDimは、dim3型の変数で、カーネル関数の実行時に指定した、グリッドのサイズを格納しています。

* blockDim

blockDimは、dim3型の変数で、カーネル関数の実行時に指定した、ブロックのサイズを格納しています。

* blockIdx

blockIdxは、uint3型の変数で、カーネル関数がグリッド中のどのブロックで実行されているかを表す、インデックスを格納しています。

* threadIdx

threadIdxは、uint3型の変数で、カーネル関数がブロック中のどのスレッドで実行されているかを表す、インデックスを格納しています。

* warpSize

warpSizeは、int型の変数で、WARPを構成するスレッドの数を格納しています。

### 修飾子

CUDAでは、プログラムはホストとデバイスで動作するため、どちらのメモリに確保するかを指定する「メモリ修飾子」と、どちらで実行する関数かを指定する「関数修飾子」があります。

主な修飾子は以下の通りです。

**メモリ修飾子**

* \_\_device\_\_

\_\_device\_\_で指定された領域は、グローバルメモリに確保されます。プログラムの実行が終了するまで確保されます。すべてのスレッドから読み書き可能で、ホストからは、ランタイムAPIを使って読み書き可能です。

* \_\_constant\_\_

\_\_constant\_\_で指定された領域は、コンスタントメモリに確保されます。プログラムの実行が終了するまで確保されます。すべてのスレッドから読み出しのみ可能で、ホストからは、ランタイムAPIを使って読み書き可能です。

* \_\_shared\_\_

\_\_shared\_\_で指定された領域は、シェアードメモリに確保されます。スレッドの実行中のみブロック単位で確保されます。ブロックに属するスレッドからのみ読み書き可能です。

**関数修飾子**

* \_\_global\_\_

\_\_global\_\_で指定された関数は、デバイスで実行されます。カーネル関数を修飾するために使用します。

* \_\_device\_\_

\_\_device\_\_で指定された関数は、デバイスで実行されます。デバイスからのみ呼び出せます。

* \_\_host\_\_

\_\_host\_\_で指定された関数は、ホストで実行されます。ホストからのみ呼び出せます。関数修飾子を省略した場合は、\_\_host\_\_が指定されます。

## サンプルコード

以下では、CUDAの簡単なソースコードの例を挙げ、いくつかのポイントを説明します。サンプルのため、戻り値を使ったエラーチェックは省略しています。

### Hello, World!

Hello, World!を表示するソースコードhelloworld.cuを、以下に挙げます。

|  |
| --- |
| #include <stdio.h>  \_\_global\_\_ void kernel(void)  {  }  int main(void)  {  kernel<<<1,1>>>();  printf("Hello, World!\n");  return 0;  } |

ポイントを、以下で説明します。

* \_\_global\_\_ void kernel(void)

関数kernel()は、\_\_global\_\_で修飾されているため、デバイスで実行されます。

* kernel<<< 1, 1 >>>();

関数kernel()を呼び出しています。グリッドとブロックのサイズは、ともに1を指定しています。上記では、kernel()は空の関数であるため、呼び出されたデバイスは何もせずに返ります。

実行結果を、以下に挙げます。

|  |
| --- |
| $ nvcc helloworld.cu  $ ./a.out  Hello, World! |

### データの転送

ホスト〜デバイス間でデータを転送するソースコードmemcpy.cuを、以下に挙げます。

|  |
| --- |
| #include <stdio.h>  \_\_global\_\_ void add(int a, int b, int \*c)  {  \*c = a + b;  }  int main(void)  {  int c;  int \*dev\_c;  cudaMalloc((void \*\*)&dev\_c, sizeof(int));  add<<<1,1>>>(2, 4, dev\_c);  cudaMemcpy(&c, dev\_c, sizeof(int), cudaMemcpyDeviceToHost);  printf("2 + 4 = %d\n", c);  cudaFree(dev\_c);  return 0;  } |

ポイントを、以下で説明します。

* cudaMalloc((void \*\*)&dev\_c, sizeof(int));

関数cudaMalloc()を使って、int型のサイズのデバイスメモリを確保しています。

* add<<<1,1>>>(2, 4, dev\_c);

関数add()を呼び出しています。呼び出されたデバイスは、第一引数と第二引数を足した結果を、第三引数が指すデバイスメモリに格納します。

* cudaMemcpy(&c, dev\_c, sizeof(int), cudaMemcpyDeviceToHost);

関数cudaMemcpy()を使って、デバイスメモリからホストメモリへ、関数add()の結果を転送しています。

* cudaFree(dev\_c);

関数cudaFree()を使って、確保したデバイスメモリを解放しています。

実行結果を、以下に挙げます。

|  |
| --- |
| $ nvcc memcpy.cu  $ ./a.out  2 + 4 = 6 |

### 行列計算

行列計算は、GPUが得意とする処理の1つです。以下のサンプルコードは、1024次正方行列の積をCPUとGPUのそれぞれで計算し、その計算時間を表示します。2つの計算時間にどの程度の差が現れるか、ぜひ試してみてください。

行列の積をCPUで計算するソースコードmatmul\_cpu.cuを、以下に挙げます。

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <sys/time.h>  /\* #define DEBUG \*/  #ifdef DEBUG  #define N 16  #else /\* !DEBUG \*/  #define N 1024  #endif /\* DEBUG \*/  #define MATRIX\_SIZE (sizeof(int) \* N \* N)  int main(void)  {  int i, j, k;  int \*a, \*b, \*c;  struct timeval start, end;  double time;  a = (int \*)malloc(MATRIX\_SIZE);  b = (int \*)malloc(MATRIX\_SIZE);  c = (int \*)malloc(MATRIX\_SIZE);  for (i = 0; i < N; i++) {  for (j = 0; j < N; j++) {  a[i \* N + j] = rand() % 10;  b[i \* N + j] = rand() % 10;  c[i \* N + j] = 0;  }  }  gettimeofday(&start, NULL);  for (i = 0; i < N; i++) {  for (j = 0; j < N; j++) {  for (k = 0; k < N; k++) {  c[i \* N + j] += a[i \* N + k] \* b[k \* N + j];  }  }  }  gettimeofday(&end, NULL);  #ifdef DEBUG  for (i = 0; i < N; i++) {  for (j = 0; j < N; j++) {  printf("%4d ", a[i \* N + j]);  }  printf("\n");  }  printf("\*\n");  for (i = 0; i < N; i++) {  for (j = 0; j < N; j++) {  printf("%4d ", b[i \* N + j]);  }  printf("\n");  }  printf("=\n");  for (i = 0; i < N; i++) {  for (j = 0; j < N; j++) {  printf("%4d ", c[i \* N + j]);  }  printf("\n");  }  #endif /\* DEBUG \*/  time = ((end.tv\_sec - start.tv\_sec) \* 1000 \* 1000) +  (end.tv\_usec - start.tv\_usec);  printf("Processing time: %.0lf (usec)\n", time);  free(a);  free(b);  free(c);  return 0;  } |

実行結果を、以下に挙げます。

|  |
| --- |
| $ nvcc matmul\_cpu.cu  $ ./a.out  Processing time: 52416689 (usec) |

行列の積をGPUで計算するソースコードmatmul\_gpu.cuを、以下に挙げます。

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <sys/time.h>  /\* #define DEBUG \*/  #ifdef DEBUG  #define N 16  #else /\* !DEBUG \*/  #define N 1024  #endif /\* DEBUG \*/  #define MATRIX\_SIZE (sizeof(int) \* N \* N)  #define BLOCK\_SIZE 16  \_\_global\_\_ void matmul(int \*dev\_a, int \*dev\_b, int \*dev\_c)  {  int i, j, k;  int sum = 0;  i = blockIdx.x \* blockDim.x + threadIdx.x;  j = blockIdx.y \* blockDim.y + threadIdx.y;  for (k = 0; k < N; k++) {  sum += dev\_a[i \* N + k] \* dev\_b[k \* N + j];  }  dev\_c[i \* N + j] = sum;  }  int main(void)  {  int i, j;  int \*a, \*b, \*c;  int \*dev\_a, \*dev\_b, \*dev\_c;  dim3 grid(N / BLOCK\_SIZE, N / BLOCK\_SIZE);  dim3 block(BLOCK\_SIZE, BLOCK\_SIZE);  struct timeval start, end;  double time;  a = (int \*)malloc(MATRIX\_SIZE);  b = (int \*)malloc(MATRIX\_SIZE);  c = (int \*)malloc(MATRIX\_SIZE);  cudaMalloc((void \*\*)&dev\_a, MATRIX\_SIZE);  cudaMalloc((void \*\*)&dev\_b, MATRIX\_SIZE);  cudaMalloc((void \*\*)&dev\_c, MATRIX\_SIZE);  for (i = 0; i < N; i++) {  for (j = 0; j < N; j++) {  a[i \* N + j] = rand() % 10;  b[i \* N + j] = rand() % 10;  }  }  gettimeofday(&start, NULL);  cudaMemcpy(dev\_a, a, MATRIX\_SIZE, cudaMemcpyHostToDevice);  cudaMemcpy(dev\_b, b, MATRIX\_SIZE, cudaMemcpyHostToDevice);  matmul<<<grid,block>>>(dev\_a, dev\_b, dev\_c);  cudaDeviceSynchronize();  cudaMemcpy(c, dev\_c, MATRIX\_SIZE, cudaMemcpyDeviceToHost);  gettimeofday(&end, NULL);  #ifdef DEBUG  for (i = 0; i < N; i++) {  for (j = 0; j < N; j++) {  printf("%4d ", a[i \* N + j]);  }  printf("\n");  }  printf("\*\n");  for (i = 0; i < N; i++) {  for (j = 0; j < N; j++) {  printf("%4d ", b[i \* N + j]);  }  printf("\n");  }  printf("=\n");  for (i = 0; i < N; i++) {  for (j = 0; j < N; j++) {  printf("%4d ", c[i \* N + j]);  }  printf("\n");  }  #endif /\* DEBUG \*/  time = ((end.tv\_sec - start.tv\_sec) \* 1000 \* 1000) +  (end.tv\_usec - start.tv\_usec);  printf("Processing time: %.0lf (usec)\n", time);  free(a);  free(b);  free(c);  cudaFree(dev\_a);  cudaFree(dev\_b);  cudaFree(dev\_c);  cudaDeviceReset();  return 0;  } |

実行結果を、以下に挙げます。

|  |
| --- |
| $ nvcc matmul\_gpu.cu  $ ./a.out  Processing time: 185489 (usec) |

# 高速化の手法

## はじめに

この章では、4.8.3で示した、行列の積をGPUで計算するソースコードmatmul\_gpu.cuをもとに、さらなる高速化を行います。

## シェアードメモリの利用

4.4節で示したように、高速なメモリを使用することと、パフォーマンスを上げることが可能です。その中でも、シェアードメモリは、ブロック内のスレッド間で共有でき、かつレジスタと同等に高速です。ただし、使用可能なシェアードメモリのサイズは大きくないため、行列のデータすべてをシェアードメモリに置くことはできません。

4.8.3のmatmul\_gpu.cuでは、16×16のブロック単位で処理を行っています(図 11 行列の積)。ブロック内の行列の積を並列処理する場合、カーネル関数matmul()では、ブロック単位でループさせると、いずれのスレッドもそのブロック内のデータを参照して計算を行います。そのため、ブロック内のデータをシェアードメモリにコピーした上で計算を行えば、グローバルメモリからの読み込み回数を減らすことができます。

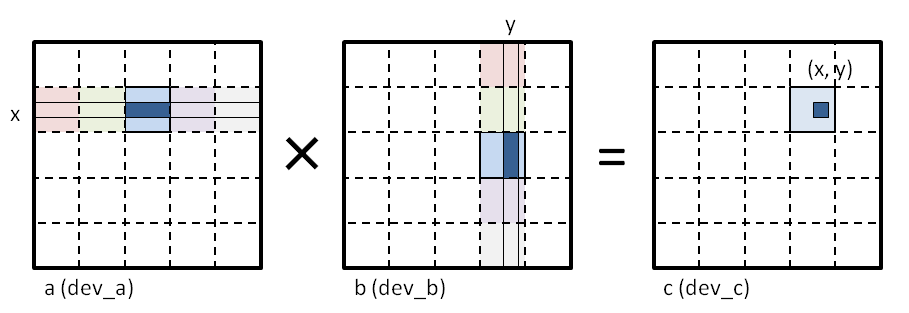


図 11 行列の積

シェアードメモリを使用するには、対象の変数に修飾子\_\_shared\_\_を指定します。

|  |
| --- |
| \_\_global\_\_ void matmul(int \*dev\_a, int \*dev\_b, int \*dev\_c)  {  \_\_shared\_\_ int dev\_as[BLOCK\_SIZE \* BLOCK\_SIZE];  \_\_shared\_\_ int dev\_bs[BLOCK\_SIZE \* BLOCK\_SIZE]; |

行列の積の計算部分を、ブロック単位に変更し、最初に各スレッドが自分の位置の値をコピーするようにします。

|  |
| --- |
| for (m = 0; m < N/BLOCK\_SIZE; m++) {  dev\_as[ty \* BLOCK\_SIZE + tx] =  dev\_a[i \* N + (m \* BLOCK\_SIZE + tx)];  dev\_bs[ty \* BLOCK\_SIZE + tx] =  dev\_b[(m \* BLOCK\_SIZE + ty) \* N + j];  \_\_syncthreads(); |

\_\_syncthreads()は、ブロック内のすべてのスレッドが、直前までの命令を完了していることを保証するためのCUDAのビルトイン関数です。計算を行う際、シェアードメモリにデータがすべて読み込まれている必要があるため、\_\_syncthreads()を呼び出して、すべてのスレッドの足並みをここで揃えます。

そして、シェアードメモリを利用して計算を行います。

|  |
| --- |
| for (k = 0; k < BLOCK\_SIZE; k++) {  sum += dev\_as[ty \* BLOCK\_SIZE + k] \*  dev\_bs[k \* BLOCK\_SIZE + tx];  }  }  dev\_c[i \* N + j] = sum; |

### 練習問題

上記の変更をmatmul\_gpu.cuに対して行い、 処理時間を計測しましょう。

## テクスチャメモリの利用

コンスタントメモリとテクスチャメモリは、読み込み専用という制約はありますが、グローバルメモリよりも高速です。計算の対象である行列をこれらのメモリに置くことで、高速になります。ただし、コンスタントメモリのサイズには制限があるため、ここではテクスチャメモリを使用して高速化を行います。

テクスチャメモリを使用するには、まずテクスチャメモリ用の変数を宣言します。

|  |
| --- |
| texture<int> tex\_a;  texture<int> tex\_b; |

ここでは、行列を2次元で扱いたいため、以下のように宣言します。

|  |
| --- |
| texture<int, 2> tex\_a;  texture<int, 2> tex\_b; |

グローバルメモリの確保は今まで通り行いますが、その後、cudaBindTexture2D()関数を使用して、確保したメモリに対してテクスチャをバインドします。

|  |
| --- |
| cudaChannelFormatDesc desc;  ...  desc = cudaCreateChannelDesc<int>();  cudaBindTexture2D(NULL, tex\_a, dev\_a, desc, N, N, sizeof(int)\*N);  cudaBindTexture2D(NULL, tex\_b, dev\_b, desc, N, N, sizeof(int)\*N); |

2次元の場合、cudaChannelFormatDescが必要になります。

ちなみに、1次元の場合は、cudaBindTexture()関数を使用します。

また、終了時には、cudaUnbindTexture()関数を使用して、バインドを解除します。

テクスチャメモリの値を参照するには、tex1Dfetch()やtex2D()などの関数を使用します。

|  |
| --- |
| \_\_global\_\_ void matmul(int \*dev\_c)  {  ...  for (k = 0; k < N; k++) {  sum += tex2D(tex\_a, k, i) \* tex2D(tex\_b, j, k);  } |

### 練習問題

上記の変更をmatmul\_gpu.cuに対して行い、処理時間を計測しましょう。

## page-lockedなホストメモリの利用

デバイス側だけでなく、ホスト側からも高速化を行う余地があります。4.6節で述べたように、page-lockedなホストメモリを使うと、そのメモリはOSによってスワップアウトされず、物理メモリ上にあることが保証されます。ただし、多用すると物理メモリの空きがなくなるため、注意が必要です。

page-lockedなホストメモリを使用するには、malloc()およびfree()の代わりに、それぞれcudaHostAlloc()およびcudaFreeHost()を呼び出します。

|  |
| --- |
| int main(void)  {  ...  cudaHostAlloc((void \*\*)&a, MATRIX\_SIZE, cudaHostAllocDefault);  cudaHostAlloc((void \*\*)&b, MATRIX\_SIZE, cudaHostAllocDefault);  cudaHostAlloc((void \*\*)&c, MATRIX\_SIZE, cudaHostAllocDefault);  ...  cudaFreeHost(a);  cudaFreeHost(b);  cudaFreeHost(c); |

### 練習問題

上記の変更をmatmul\_gpu.cuに対して行い、処理時間を計測しましょう。

## ゼロコピーホストメモリの利用

cudaHostAlloc()で割り当てられたメモリを、デバイスとホストの双方からアクセス可能にすることができます。同じメモリを参照するため、cudaMemcpy()でコピーする必要がなくなります。

ゼロコピーホストメモリを使用するには、cudaHostAlloc()の第3引数に、cudaHostAllocMappedおよびcudaHostAllocWriteCombinedフラグを指定します。前者はデバイスからメモリへのアクセスを可能にするためのフラグです。後者はデバイスからCPUキャッシュへのアクセスを可能にするためのフラグです。

|  |
| --- |
| cudaHostAlloc((void \*\*)&a, MATRIX\_SIZE,  cudaHostAllocMapped | cudaHostAllocWriteCombined);  cudaHostAlloc((void \*\*)&b, MATRIX\_SIZE,  cudaHostAllocMapped | cudaHostAllocWriteCombined);  cudaHostAlloc((void \*\*)&c, MATRIX\_SIZE,  cudaHostAllocMapped | cudaHostAllocWriteCombined);  cudaHostGetPointer(&dev\_a, a, 0);  cudaHostGetPointer(&dev\_b, b, 0);  cudaHostGetPointer(&dev\_c, c, 0); |

### 練習問題

上記の変更をmatmul\_gpu.cuに対して行い、処理時間を計測しましょう。

## その他

以上、一般的な高速化の手法について解説しました。これら以外にも、以下の手法があります。

* ローカル変数の利用

ローカル変数にはレジスタが用いられます。頻繁に使用する変数は、ローカル変数に代入して使用すると、速くなります。

* コアレッシングの利用

4.5節で示したように、グローバルメモリにアクセスする際、隣接するスレッドが隣接するアドレスにアクセスする方が、高速に動作します。

* WARPダイバージェントの削減

4.5節で示したように、カーネル関数に分岐があり、WARP内のスレッドが双方のパスに分岐する場合、それぞれを並列ではなく直列に実行されるため、時間がかかります。

* バンクコンフリクトの削減

4.5節で示したように、シェアードメモリは、バンクという単位に分けられており、別のバンクへのアクセスは並列に処理できます。ですが、複数のスレッドが同じバンク内のシェアードメモリにアクセスすると、速度が低下します。

* ループ展開

ループを展開することで、条件分岐がなくなり、速くなることがあります。ただし、コードサイズは増加します。

* 複数GPUの利用

cudaSetDevice()関数で、使用するデバイスを指定できます。デバイス数は、cudaGetDeviceCount()関数で確認できます。

ちなみに、使用可能なシェアードメモリやコンスタントメモリのサイズを確認するには、cudaGetDeviceProperties()関数を使用します。

|  |
| --- |
| int devcnt;  cudaGetDeviceCount(&devcnt);  for (i = 0; i < devcnt; i++) {  cudaDeviceProp prop;  cudaGetDeviceProperties(&prop, i);  printf("[%d] %s %d.%d\n",  i, prop.name, prop.major, prop.minor);  printf(" global=%ld const=%ld shared=%ld\n",  prop.totalGlobalMem, prop.totalConstMem,  prop.sharedMemPerBlock);  } |

他にも、グローバルメモリやWARPのサイズなどが確認できます。cudaDeviceProp構造体の詳細を確認するには、cuda\_runtime\_api.h を参照してください。

# CUDA Toolkit

CUDAを使ったGPGPUを始めるには、まず、CUDA Toolkitをインストールする必要があります。CUDA Toolkitは、以下のURLから入手できます。

<https://developer.nvidia.com/cuda-toolkit>

CUDA Toolkitには、NVIDIA社のGPU向けコンパイラ、数学ライブラリ、デバッガなどのツールが入っています。

ここでは、CUDA Toolkitの使い方について説明します。

## コンパイルと実行

CUDAでGPGPUを行うためには、ソースコードをコンパイルし、実行ファイルを生成しなければなりません。

CUDA Toolkitには、nvcc(NVIDIA CUDA Compiler)と呼ばれるコンパイラが入っています。nvccは、CUDA向けに拡張されたC言語風のソースコードをコンパイルして、実行ファイルを生成します。ソースコードの拡張子は、「.cu」とする決まりになっています。

nvccの詳細は、以下のURLを参照して下さい。

<http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc>

例えば、ソースコードfoo.cuは、以下のようにコンパイルできます。

|  |
| --- |
| $ nvcc foo.cu |

nvccは、ソースコードの「C言語やC++言語の部分」に対しては、gccやVisual Studioなどのホストのコンパイラを使用してコンパイルし、節で示した「CUDAの拡張部分」に対しては、nvccがコンパイルします。

ホストのコンパイラにgccを使ってコンパイルすると、オプションを指定しない場合、a.outという名前の実行ファイルが生成されます。

a.outは、以下のように実行できます。

|  |
| --- |
| $ ./a.out |

nvccのオプションとして、ホストのコンパイラのオプションを指定できます。例えば、gccであれば、-O3ような最適化オプションを指定することで、nvccはそれらのオプションをgccに渡すことができます。ただし、それらはデバイスコードに適用されるわけではありません。

以下に、デバイスコードに適用される、よく使うオプションを挙げます。

* --gpu-architecture <gpu architecture name> (-arch)

コンパイルすべきGPUの名前(GPUアーキテクチャ)を指定する。

以下に、例を挙げます。

|  |
| --- |
| $ nvcc -arch sm\_13 foo.cu |

これは、GPUアーキテクチャsm\_13向けの実行ファイルを生成します。

* --maxrregcount <N> (-maxrregcount)

1つのスレッドが使用するレジスタ数を、<N>に制限します。<N>より多くのレジスタが使用される場合、値はローカルメモリに退避されます。

* --use\_fast\_math (-use\_fast\_math)

精度は低いが高速な数学ライブラリを使用します。

* --device-debug (-G)

デバイスコードのためのデバッグ情報を生成します。

* --ptxas-options <options>,... (-Xptxas)

ptxasに<options>,...を指定します。(ptxasについては節を参照)

以下に、例を挙げます。

|  |
| --- |
| $ nvcc --ptxas-options=-v foo.cu |

これは、レジスタやメモリの使用量を、コンパイル時に表示します。

* --ptx (-ptx)

デバイスコードをPTXに翻訳してファイルに出力します。(PTXについては節を参照)

* --cubin (-cubin)

デバイスコードからCUBINを生成してファイルに出力します。(CUBINについては節を参照)

## デバイスコードの確認

nvccが、期待するデバイスコードを生成していることを、確認したい場合があります。

CUDA Toolkitには、ptxasと呼ばれるアセンブラが入っています。デバイスコードは、nvccの内部で、PTX(Parallel Thread Execution)と呼ばれるアセンブリ言語に翻訳されます。ptxasは、PTXをアセンブルして、CUBIN(CUDA Binary)と呼ばれるELF形式のオブジェクトファイルを生成します。

例えば、ソースコードfoo.cuは、以下のようにしてPTXに翻訳できます。

|  |
| --- |
| $ nvcc -ptx foo.cu |

実行後、foo.ptxという名前のファイルが生成されます。foo.ptxは、デバイスコードが、PTX ISA(PTX Instruction Set Architecture)で定義された命令に翻訳されたものです。

PTX ISAの詳細は、以下のURLを参照して下さい。

<http://docs.nvidia.com/cuda/parallel-thread-execution>

foo.ptxから、以下のようにCUBINを生成できます。

|  |
| --- |
| $ ptxas foo.ptx |

オプションを指定しない場合は、elf.oという名前のファイルが生成されます。elf.oは、GPUのネイティブコードを格納したELF形式のオブジェクトファイルです。

また、以下のようにソースコードfoo.cuから生成することもできます。

|  |
| --- |
| $ nvcc -cubin foo.cu |

実行後、foo.cubinという名前のファイルが生成されます。foo.cubinの内容は、elf.oと同じです。

CUDA Toolkitには、cuobjdumpと呼ばれる逆アセンブラが入っています。cuobjdumpは、実行ファイルやCUBINなどを、人間が理解しやすい形式で表示します。

例えば、foo.cubinは、以下のようにELF形式のセクションを表示できます。

|  |
| --- |
| $ cuobjdump -elf foo.cubin |

cuobjdumpは、実行ファイルやCUBINなどを、いくつかの形式で表示することができます。

以下に、形式を指定するための、よく使うオプションを挙げます。

* --dump-cubin (-cubin)

デバイス関数のCUBINの情報を表示します。

* --dump-elf (-elf)

ELF形式のセクションを表示します。

* --dump-elf-symbols (-symbols)

ELF形式のシンボル名を表示します。

* --dump-function-names (-fnam)

デバイス関数の名前を表示します。

* --dump-ptx (-ptx)

デバイス関数のPTXの情報を表示します。

* --dump-sass (-sass)

デバイス関数のSASSを表示します。(SASSとは、PTXよりも低水準な機械語に対応するアセンブリです。)

## デバッグ

nvccでコンパイルした実行ファイルが、予期しない動作をする場合は、デバッグをしなければなりません。

CUDA Toolkitには、cuda-gdbと呼ばれるデバッガが入っています。cuda-gdbを使えば、CUDAのプログラムの動作を解析できます。

cuda-gdbの詳細は、以下のURLを参照して下さい。

<http://docs.nvidia.com/cuda/cuda-gdb>

cuda-gdbを使ってデバッグをする場合、コンパイル時にnvccに対して-g,-Gを指定し、デバッグ情報を生成しておく必要があります。

|  |
| --- |
| $ nvcc -g -G foo.cu |

例えば、a.outをデバッグする場合は、以下のようにcuda-gdbを起動します。

|  |
| --- |
| $ cuda-gdb a.out |

起動後、通常のgdbと同様に、break,run,printなどのコマンドが使用できます。

cuda-gdbには、NVIDIA社のGPUのために、いくつかのコマンドが追加されています。

以下に、GPUに関係する、よく使うコマンドを挙げます。

* info cuda devices

システムの、すべてのGPUを表示します。

* info cuda sms

すべてのSMの情報を表示します。

* info cuda warps

フォーカスしているSMの、すべてのWARPの情報を表示します。

* info cuda lanes

フォーカスしているWARPの、すべてのlanes(スレッド)の情報を表示します。

* info cuda kernels

フォーカスしているGPUの、すべてのアクティブなカーネルの情報を表示します。

* info cuda blocks

フォーカスしているカーネルの、すべてのアクティブまたは走行中な、ブロックの情報を表示します。ブロックのフォーカスを切り替える場合は、cuda blockを使用します。

以下に、例を挙げます。

|  |
| --- |
| (cuda-gdb) cuda block 1 |

これは、ブロックのフォーカスを、ブロック番号1に切り替えます。

* info cuda threads

アクティブなブロックとスレッドの情報を表示します。スレッドのフォーカスを切り替える場合は、cuda threadを使用します。

以下に、例を挙げます。

|  |
| --- |
| (cuda-gdb) cuda thread 1 |

これは、スレッドのフォーカスを、スレッド番号1に切り替えます。

# 事例

最初は、天体計算の分野で始まったGPGPUの考えは、現在、様々な分野で利用されています。

ここでは、HPC、クラウドコンピューティング、ビックデータ解析を例に、GPUの事例を紹介します。

## HPC

GPUは、特にHPC(High Performance Computing)で利用されています。日本では、GPUを使ったHPCの先駆けは、長崎大学 先端計算研究センターの「DEGIMA」(DEstination for Gpu Intensive MAchine)です。DEGIMAは、760個のGPUによって構成されたスーパーコンピュータです。2009年には、DEGIMAは、HPCを活用した優れた業績に対してACM(Association for Computing Machinery)から授与される、ゴードン・ベル賞を受賞しました。ACMは、アメリカ合衆国を拠点とする計算機の国際学会です。また、2011年6月には、スーパーコンピュータの電力性能(性能÷電力)の世界ランキングGreen500において、3位を収めています。このGPUにはAMD社のATI Radeonが採用されています。

**表 4 2011年6月Green500 Top 10**

|  |  |  |  |
| --- | --- | --- | --- |
| **Rank** | **MFLOPS/W** | **Computer** | **Total Power (kW)** |
| 1 | 2,097.19 | NNSA/SC Blue Gene/Q Prototype 2 | 40.95 |
| 2 | 1,684.20 | NNSA/SC Blue Gene/Q Prototype 1 | 38.80 |
| 3 | 1,375.88 | DEGIMA Cluster, Intel i5, ATI Radeon GPU, Infiniband QDR | 34.24 |
| 4 | 958.35 | HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows | 1,243.80 |
| 5 | 891.88 | iDataPlex DX360M3, Xeon 2.4, nVidia GPU, Infiniband | 160.00 |
| 6 | 824.56 | K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect | 9,898.56 |
| 7 | 773.38 | QPACE SFB TR Cluster, PowerXCell 8i, 3.2 GHz, 3D-Torus | 57.54 |
| 8 | 773.38 | QPACE SFB TR Cluster, PowerXCell 8i, 3.2 GHz, 3D-Torus | 57.54 |
| 9 | 773.38 | QPACE SFB TR Cluster, PowerXCell 8i, 3.2 GHz, 3D-Torus | 57.54 |
| 10 | 718.13 | Supermicro Cluster, QC Opteron 2.1 GHz, ATI Radeon GPU, Infiniband | 416.78 |

他に有名なものは、東京工業大学 学術国際情報センターが運用している、クラスタ型スーパーコンピュータ「TSUBAME」があります。2013年11月には、次世代のTSUBAME3.0のテストシステムであるTSUBAME-KFC(TSUBAME Kepler Fluid Cooling)が、Green500とGreen Graph 500において、日本で初めて1位を収めています。Green Graph 500は、大規模グラフを処理する電力性能の世界ランキングです。TSUBAME-KFCは、2個のCPUと4個のGPUを搭載した計算ノードが、40台集まって構成されています。このGPUには、NVIDIA社のKeplerアーキテクチャのTesla K20X GPUが採用されています。

表 5 2013年11月Green500 Top 10

|  |  |  |  |
| --- | --- | --- | --- |
| **Rank** | **MFLOPS/W** | **Computer** | **Total Power (kW)** |
| 1 | 4,503.17 | TSUBAME-KFC - LX 1U-4GPU/104Re-1G Cluster, Intel Xeon E5-2620v2 6C 2.100GHz, Infiniband FDR, NVIDIA K20x | 27.78 |
| 2 | 3,631.86 | Wilkes - Dell T620 Cluster, Intel Xeon E5-2630v2 6C 2.600GHz, Infiniband FDR, NVIDIA K20 | 52.62 |
| 3 | 3,517.84 | HA-PACS TCA - Cray 3623G4-SM Cluster, Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband QDR, NVIDIA K20x | 78.77 |
| 4 | 3,185.91 | Piz Daint - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x  Level 3 measurement data available | 1,753.66 |
| 5 | 3,130.95 | romeo - Bull R421-E3 Cluster, Intel Xeon E5-2650v2 8C 2.600GHz, Infiniband FDR, NVIDIA K20x | 81.41 |
| 6 | 3,068.71 | TSUBAME 2.5 - Cluster Platform SL390s G7, Xeon X5670 6C 2.930GHz, Infiniband QDR, NVIDIA K20x | 922.54 |
| 7 | 2,702.16 | iDataPlex DX360M4, Intel Xeon E5-2650v2 8C 2.600GHz, Infiniband FDR14, NVIDIA K20x | 53.62 |
| 8 | 2,629.10 | iDataPlex DX360M4, Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband, NVIDIA K20x | 269.94 |
| 9 | 2,629.10 | iDataPlex DX360M4, Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband, NVIDIA K20x | 55.62 |
| 10 | 2,358.69 | CSIRO GPU Cluster - Nitro G16 3GPU, Xeon E5-2650 8C 2.000GHz, Infiniband FDR, Nvidia K20m | 71.01 |

このようなシステムから得られる大規模な計算資源は、自動車の衝突シミュレーションや、遺伝子配列やタンパク質の解析など、様々な分野において利用され始めています。

## クラウドコンピューティング

クラウドコンピューティングでも、GPUは利用され始めています。今日では、AWS(Amazon Web Services)のようなサービスを使って、誰でもクラウドコンピューティングを試すことができます。2013年11月、AWSはAmazon EC2(Amazon Elastic Compute Cloud)のインスタンスとして、GPUを搭載した「G2インスタンス」を発表しました。インスタンスとは、ユーザがアプリケーションを実行できる仮想的なサーバです。このGPUには、NVIDIA社のKeplerアーキテクチャのGRID GPUが採用されています。

## ビックデータ解析

GPUの高い演算性能を、ビックデータの解析に利用するといった考えも現れています。解析に使用されるMapReduceの仕組みは、並列計算に適した処理が多く、GPUを使って高速化できると考えられています。すでに、Hadoopのようなオープンソースの実装を改造して、MapReduceのいくつかのフェーズを、GPUで高速化するといった研究が現れています。

# 参考文献

* 青木 尊之, 額田 彰:「はじめてのCUDAプログラミング: 驚異の開発環境[GPU+CUDA]を使いこなす！」工学社, 2009.
* Jason Sanders, Edward Kandrot:「CUDA by Example: An Introduction to General-Purpose GPU Programming」Addison-Wesley Professional, 2010. クイープ:「CUDA by Example 汎用GPUプログラミング入門」インプレスジャパン, 2011.
* David B. Kirk, Wen-mei W. Hwu:「Programming Massively Parallel Processors: A Hands-on Approach」Morgan Kaufmann, 2010. Bスプラウト:「CUDAプログラミング実践講座: 超並列プロセッサにおけるプログラミング手法」ボーンデジタル, 2010.
* ゼロからはじめるGPUコンピューティング  
  <http://www.gdep.jp/page/view/248>
* CUDA Toolkit Documentation  
  <http://docs.nvidia.com/cuda>
* FERMI Compute Architecture White Paper  
  <http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf>
* Kepler Compute Architecture White Paper  
  <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>

# 変更履歴

|  |  |  |
| --- | --- | --- |
| **版数** | **変更内容** | **年月日** |
| 1.0 | 新規作成 | 2014/MAR/2 |