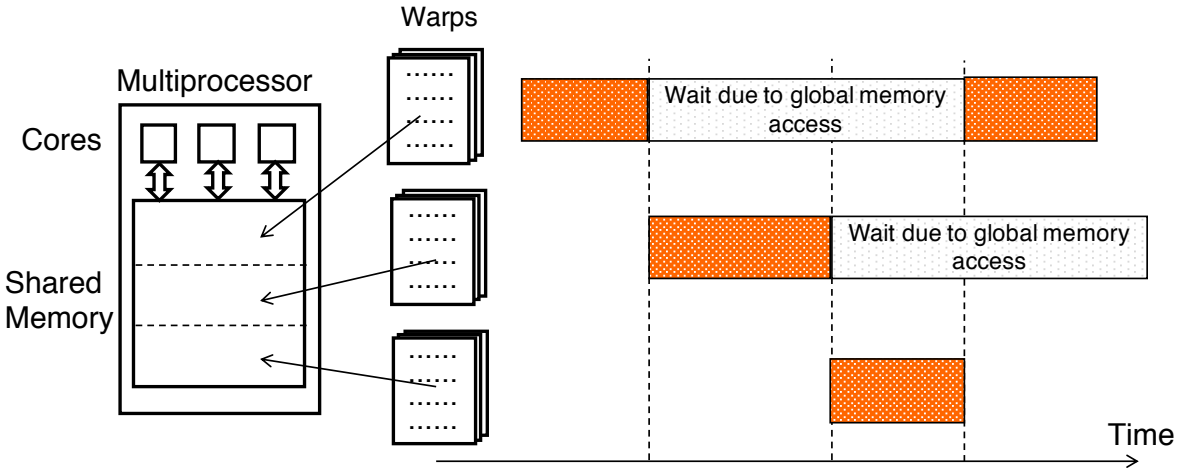
GPUプログラミング入門 講習会

GPUアーキテクチャとアルゴリズム設計

一関工業高等専門学校 未来創造工学科 情報・ソフトウェア系
小池 敦

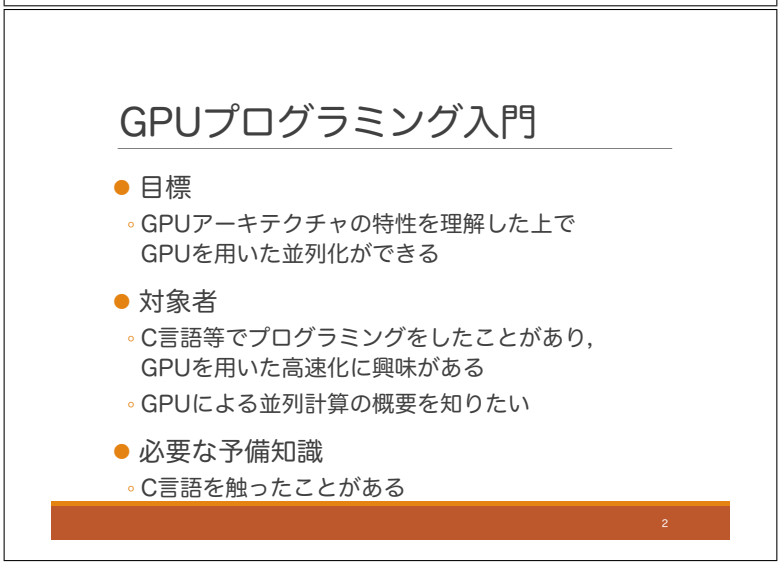


第1版

目 次

第1章　初日

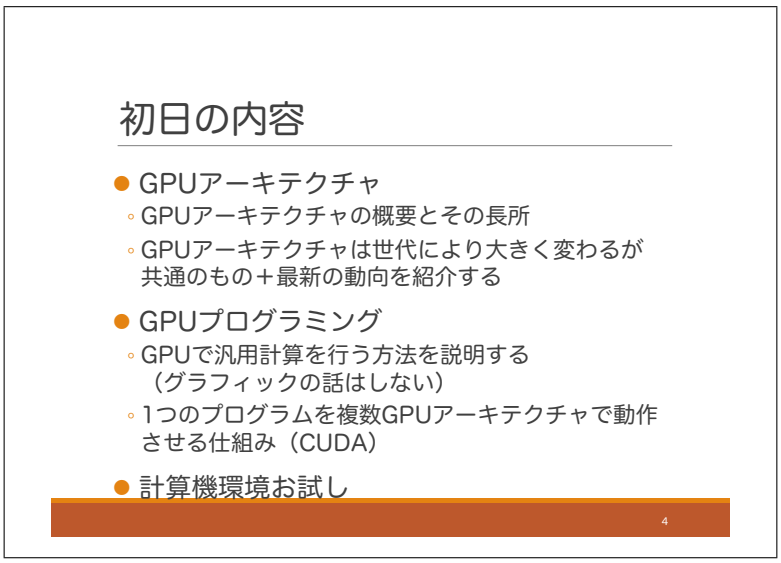
1.1　はじめに



本講習会は，GPU を使って高速な並列プログラムを書きたい人のための二日間の入門講座です．目標は，単にプログラミングの文法を知るだけでなく，GPU のアーキテクチャを理解した上で，高速に動作するプログラムが書けるようになることです．対象者は，C 言語でプログラミングをしたことがあり，GPU を用いた高速化に興味がある人です．C 言語については，簡単なコードが読めるくらいの知識があれば大丈夫です．また，単に GPU による並列計算の概要を知りたいという人も対象としています．

講習会の内容はスライドの通りです．初日は，GPU のアーキテクチャを中心に話します．その中で，GPU のアーキテクチャを活かすためには，どのようにプログラミングすべきかという話もします．初日は座学中心で，実習ついては最後にサンプルプログラムを動かしてみる程度です．二日目は実際に GPU のコードを書きながら，プログラミングの文法と高速化のノウハウを学習します．

本テキストは大阪大学サイバーメディアセンターでの講習会で使うことを想定して作られています．もし他の環境で使用している場合は，適宜読み替えてください．サイバーメディアセンターを利用する場合，GPU が使用できるシステムは OCTOPUS です．事前にアカウントを取得し，OCTOPUS にログインできるようにしておいてください．

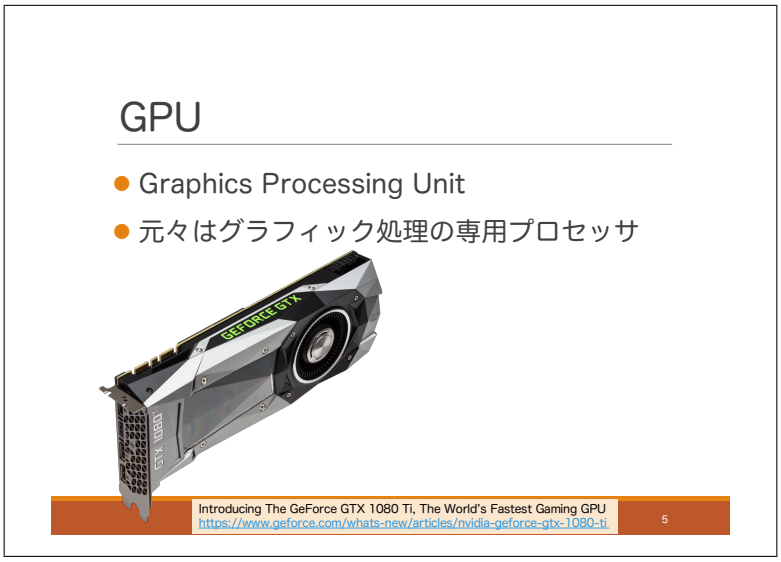


初日の内容の詳細はスライドのようになります．まず，GPU アーキテクチャの概要と長所について話すのですが，GPU アーキテクチャは世代によって，大きく変化しています．本講習会では主に世代で共通の部分を扱うこととし，最新の話については簡単に触れる程度とします．

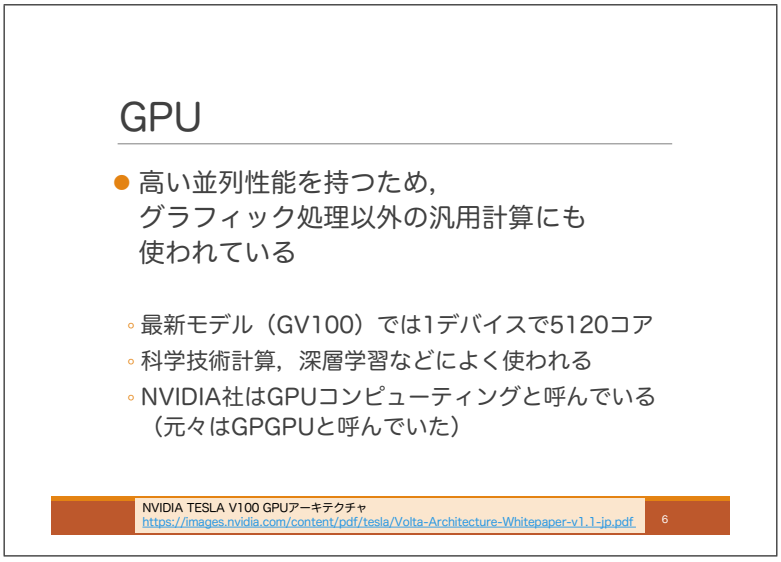
次に，GPU を使ったプログラミングの仕方について概要を話します．後でも話しますが，今回の講習会では，GPU を使った並列計算の話をします．GPU は本来グラフィック処理をするためのプロセッサですが，今回はグラフィックの話はしません．様々な GPU アーキテクチャ上で動作するプログラムを作る仕組みとして CUDA があります．CUDA にはプログラミング言語や開発環境が含まれており，それを説明します．

最後に，計算機環境のお試しとして，サンプルプログラムを実行してみます．もしうまくいかない人がいれば，二日目までに環境を確認しましょう．

1.2　GPU と GPU コンピューティング



それでは，本日の内容に入っていきましょう．まず，GPU について説明します．GPU は Graphics Processing Unit の略称で，その名の通り，元々はグラフィック処理専用のプロセッサでした．



GPU にはとても多くのコアが搭載されています．コアというのは，プログラムを実行する装置のことで，ざっくり言えば，例えば 4 つのコアが搭載されていれば 4 つのプログラムを同時に並列に実行できるということです．コアでなくプロセッサと呼んでも良いのですが，複数のコアを一つのデバイスに搭載した装置もプロセッサと呼ぶので，それと区別するために通常はコアと呼ばれています．NVIDIA の最新 GPU モデルである GV100 では，1 デバイスに 5120 個のコアが搭載されています．CPU は 4 コアとか 8 コアとかのレベルなので，GPU には圧倒的な数のコアが搭載されているということがわかると思います．この圧倒的な数のコアを活用して，グラフィック処理以外の処理を行わせるということが盛んに行われています．シミュレーションなどの科学技術計算においては，既に多くの分野で活用されていますし，近年では深層学習での活用が進んでいます．GPU をグラフィック以外に活用することは，General Purpose GPU (GPGPU) と呼ばれていましたが，この分野を引っ張ってきた NVIDIA 社は近年は GPGPU という用語を使用しなくなり，代わりに GPU コンピューティングと呼んでいるようです．この講習会ではこれのことを「GPU による汎用計算」もしくは「GPU コンピューティング」と呼ぶことにします．また，本講習会で GPU プログラミングと言ったら，それは GPU コンピューティングのためのプログラミングを意味します．

GPU

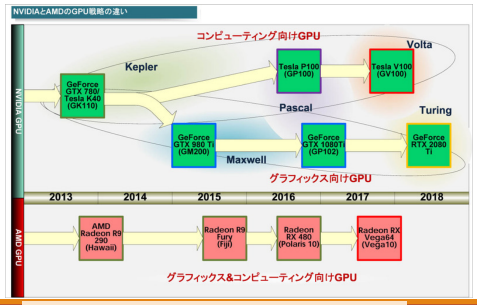
- 主な開発企業
  - NVIDIA
  - AMD
- 汎用計算においては，NVIDIAの1強
  - CUDAと呼ばれるNVIDIA社GPU向けのプログラミング環境（後述）が整備されていることが大きい
    - 高速化しやすい，そこそこプログラミングしやすい
  - 本講習会でもNVIDIA社GPUをターゲットとする

7

GPU は多くの半導体メーカーによって開発されていますが，代表的な企業として NVIDIA 社と AMD 社を挙げることができます．しかし，GPU を用いた汎用計算に関しては，NVIDIA 社の 1 強と言える状況です．これは NVIDIA 社 GPU 向けに CUDA と呼ばれるプログラミング環境を整備していることが大きな要因ではないかと思います．CUDA を使うと，それなりに簡単にプログラミングができ，頑張れば高速化のチューニングをすることも可能です．ということで，本講習会でも NVIDIA 社の GPU を用いて，CUDA 上でプログラミングを行います．

NVIDIA GPU

- 今回はTesla P100を使用(Pascalアーキテクチャ)

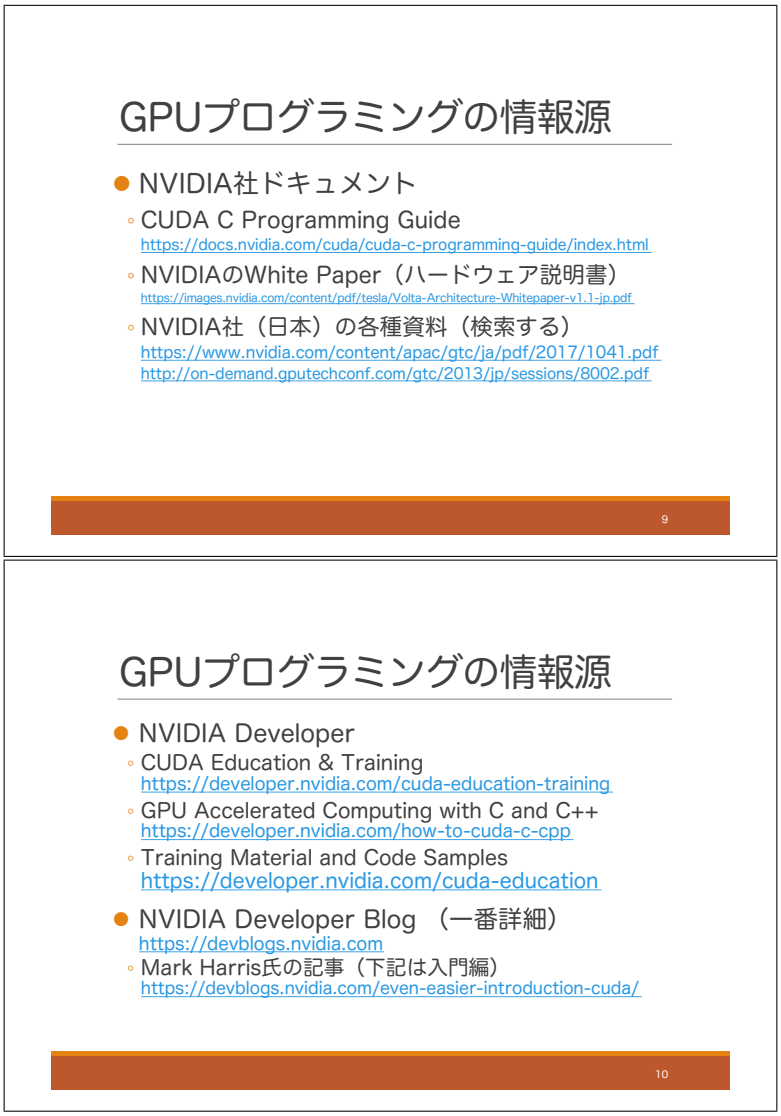


後藤弘茂のWeekly海外ニュース
https://pc.watch.impress.co.jp/docs/column/kaigai/1139009.html

8

この図は後藤弘茂氏による GPU モデルの変遷のまとめです．上の緑の行が NVIDIA 社 GPU で下の赤の行が AMD 社 GPU です．後藤氏のまとめによれば NVIDIA 社 GPU は 2014 年頃グラフィック処置をメインとする GPU と GPU コンピューティングをメインにする GPU の 2 系統に別れたようです．私の感覚的にも，その頃深層学習での GPU 利用が流行ってきて 2 系統に別れたという印象です．そして，今回の講習会で使用する GPU は最上段の右から二番目にある Tesla P100 です．これは最新モデルではありませんが，Pascal と呼ばれる高性能アーキテクチャを採用したモデルです．

1.3　GPU プログラミングの情報源と GPU 環境の取得

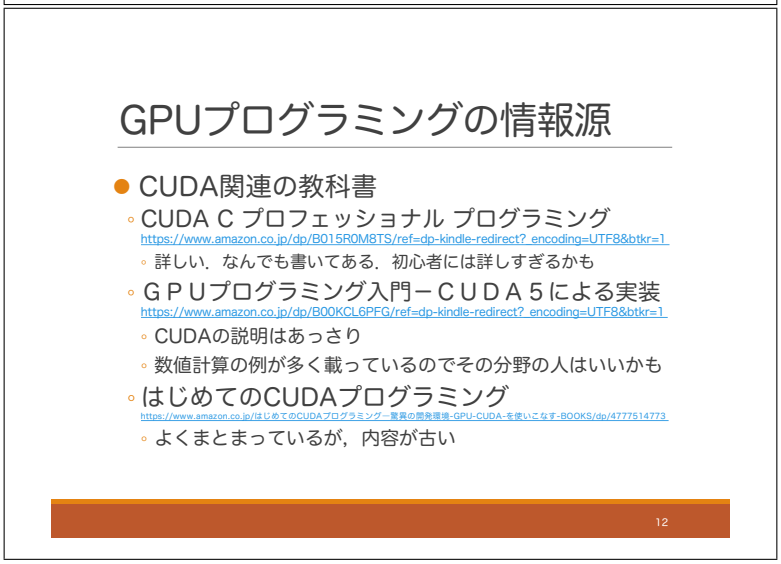


GPU プログラミングの詳細説明に入る前に，これらについて知るための情報源について紹介したいと思います．この講習会で話す内容も基本的にはこれらの情報源を基にしています．この講習会は入門編ですが，さらに詳細な知識を得たいと思った際には，これらの情報源を参照すると良いと思います．

まずは NVIDIA 社のドキュメントについて紹介します．最も基本となるドキュメントは CUDA C Programming Guide です．CUDA を使って GPU プログラミングする際に必要となる情報が網羅的に書かれています．プログラミングをしていて詳細な情報が欲しくなった時には，基本的にはこの資料を見るのが良いと思います．次が，NVIDIA 社のホワイトペーパーです．ホワイトペーパーという用語は，ソフトウェアの人には馴染みがないかもしれませんが，いわゆるハードウェアの詳細な説明書です．NVIDIA では，GPU アーキテクチャごとにホワイトペーパーを作っており，その GPU アーキテクチャの特徴や詳細な仕様を説明しています．なお，ホワイトペーパーについては，日本語版も作られているようです．また，日本の NVIDIA 社は講習会などのために日本語の資料を作っており，検索すると見つかる場合もあります．

NVIDIA Developer のサイトには，GPU プログラミングを勉強するための各種資料が掲載されています．動画やサンプルコードもあり，これから GPU プログラミングを始めたいと考えている人には役立つと思います．

別の情報源として，NVIDIA エンジニアによる Developer Blog があります．CUDA C Programming Guide などでは説明されていない詳細仕様が書かれていることもあり，GPU アーキテクチャや CUDA について詳細を知りたい時には重要な情報源です．特に Mark Harris 氏は多くの有益な記事を書いています．技術的な詳細の記事だけでなく，スライドに記載したような入門編の記事も書いています．

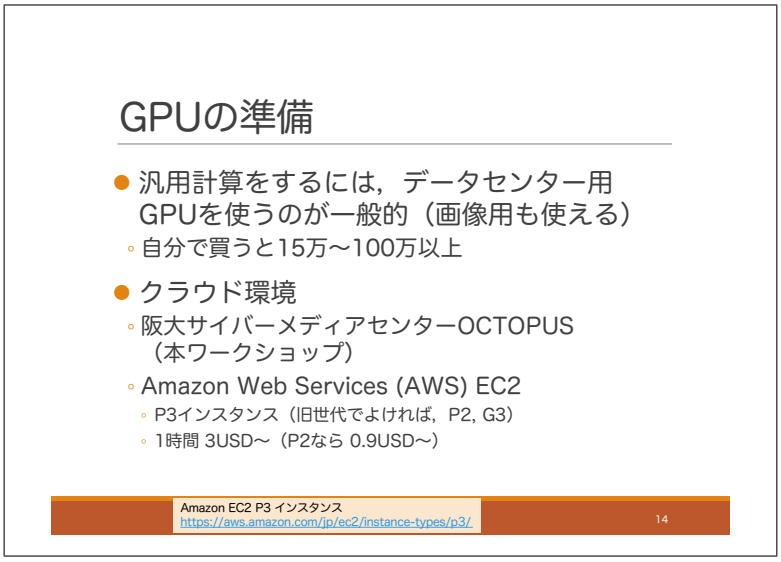


その他の情報源についてもいくつか挙げておきます．

まず，ニュース記事としては，後藤弘茂氏の記事が挙げられます．GPU プログラミング関連のニュース記事は NVIDIA 社の発表をそのまま記事にしたものが多いですが，後藤弘茂氏の記事は，幅広い知識に基づいて深い洞察が行われているものが多いです．スライドのリンクのように，企業戦略の話だったり，半導体製造技術の記事もあります．

次に CUDA の教科書について，私が所有しているものについてコメントしておきます．「CUDA C プロフェッショナル プログラミング」は CUDA についての詳細が網羅的にまとめられた本です．CUDA C Programming Guide が難しすぎるとか，日本語で読みたいとか思っている人には良いと思います．ただし，分厚い本なので，初心者が一から全部読むのは辛いかもしれません．「ＧＰＵプログラミング入門―ＣＵＤＡ５による実装」は，CUDA についての説明はあっさりしていて，これだけで理解するのは厳しいかもしれませんが，数値計算の例が多く載っているので，その分野の人にはいいかもしれません．「はじめての CUDA プログラミング」は CUDA について簡潔にまとめられており，初学者に向いています．しかし，2009 年に出版された本であり内容がかなり古いので，注意が必要です．最新の GPU や CUDA では異なっているという内容が多くあります．

GPU アーキテクチャについて記載されている教科書についても紹介しておきたいと思います．一冊目が通称「ヘネ＆パタ」と呼ばれているもので，正式名称はスライドの通りですが，この 4.4 節に GPU アーキテクチャの説明があります．GPU アーキテクチャの利点についてかなり詳しく説明されていると思います．二冊目は通称「パタ＆ヘネ」で，正式名称は「コンピュータの構成と設計」ですが，この付録に GPU アーキテクチャの説明があります．これは素晴らしいことに，このリンクから無料でダウンロードできるのですが，内容がかなり古いので注意が必要です．

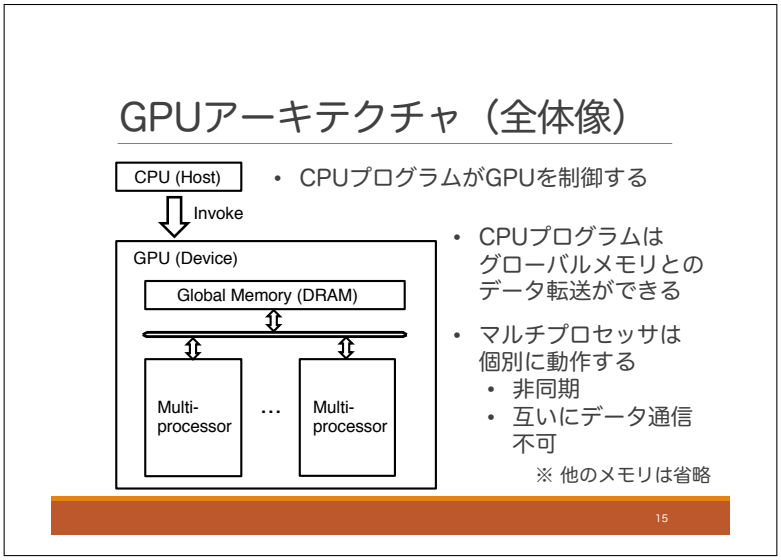


最後に GPU 環境を準備する方法について紹介します．NVIDIA の GPU であれば，CUDA を用いた GPU プログラミングを行うことができます．スライド 8 で紹介した通り，NVIDIA の GPU は GPU コンピューティング向けとグラフィックス向けに分かれています．どちらも CUDA が使えるのですが，汎用計算における性能等を考えると GPU コンピューティング向けを使う方がいいです．データセンター用 GPU と説明されているものが GPU コンピューティング向けの GPU です．自分で GPU 環境を構築しようとすると，安くて 15 万円くらい，高いと 100 万円以上します．

お手軽に GPU プログラミングを始めるにはクラウド環境がおすすめです．本講習会で使用する大阪大学サイバーメディアセンターの OCTOPUS には試用期間があり，お試しで使ってみることもできます．また，Amazon Web Services，略して AWS の EC2 というサービスでは，クラウド環境を時間単位の料金で使用することができます．P3 インスタンスというものを選択すると最新 GPU アーキテクチャの Volta が 1 時間 3US ドルで使用できますし，最新 GPU でなくても良いなら，P2 インスタンスを選択することで，1 時間 0.9US ドルで使用できます．

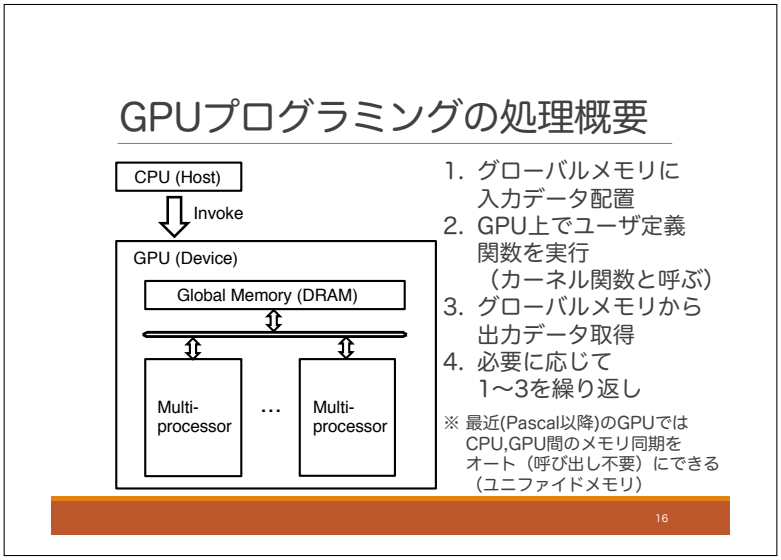
1.4 GPU アーキテクチャ

1.4.1 概要



それでは GPU アーキテクチャについて説明していきます．全体像はこのスライドのようになります．まず重要なこととして，GPU は単独では動作できず，常に CPU によって制御されます．CPU 上で動いているプログラムが GPU に対して処理を依頼すると GPU がそれを実行するという流れです．GPU プログラミングでは，CPU のことをホスト，GPU のことをデバイスと呼びます．例えば，ホストメモリと言った場合には CPU 上のメモリのことですし，デバイスメモリと言った場合は，GPU 上のメモリのことです．GPU は様々なモジュールで構成されますが，最も基本的なモジュールはグローバルメモリとマルチプロセッサです．本当はもっとたくさんの種類のメモリがあるのですが，ここでは省略しています．マルチプロセッサは，実際に処理を行う部分で，通常 1GPU デバイスの中に複数のマルチプロセッサが入っています．グローバルメモリはすべてのマルチプロセッサからアクセス可能なメモリです．CPU もグローバルメモリにアクセスできます．CPU プログラムは GPU に処理を依頼する前に，必要なデータをグローバルメモリに転送しておくことができます．少し細かいことを言うと，昔の GPU では CPU メモリ上にある必要なデータを必ず事前にグローバルメモリに転送しておかなくてはならなかったのですが，2016 年に発表された Pascal アーキテクチャ以降の GPU では CPU メモリ上のデータと GPU グローバルメモリ上のデータの自動同期の機能が搭載されたため，必ずしも事前に手動でデータ転送を行う必要はなくなりました．

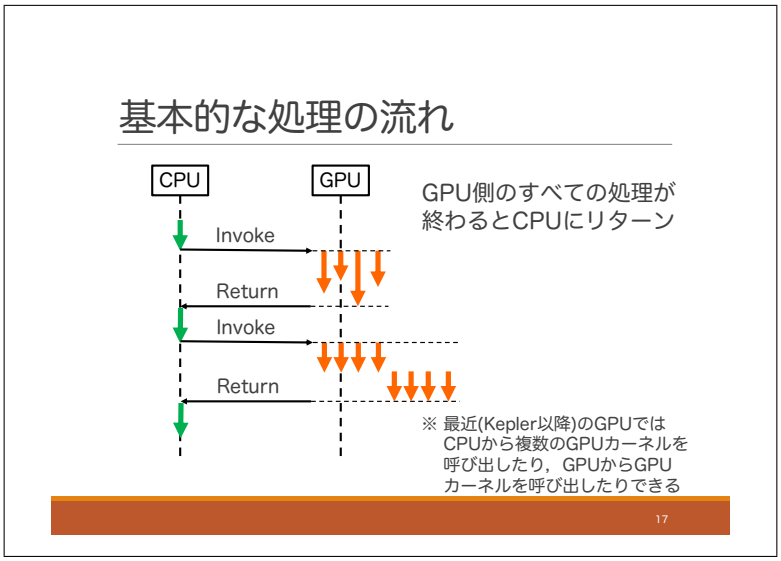
次にマルチプロセッサについてですが，大きな特徴は，マルチプロセッサのそれぞれが個別に動作するということです．もう少し具体的に言うと，各マルチプロセッサは，プログラムを独自のペースで実行しており，他のマルチプロセッサとタイミングを合わせるといった機能は持っていません．他のマルチプロセッサとタイミングを合わせる命令は同期命令と呼ばれますが，GPU ではマルチプロセッサ間の同期命令は存在せず，各マルチプロセッサは非同期に実行されるということになります．また，マルチプロセッサ間でデータ通信を行うこともできません．これも GPU プログラミングを行う際の大きな制約になります．なお，CPU において，すべてのマルチプロセッサが処理を完了するのを待つということはできます．あるマルチプロセッサでの処理結果を別のマルチプロセッサに渡したい時に，このことを活用することができます．



前のスライドと少し内容がかぶりますが，GPU プログラミングにおいて，プログラムがどのように処理を進めていくかということについて説明します．実際にはもっと様々な処理を行うことも可能なので，ここで説明するのは，基本的なパターンの一つです．

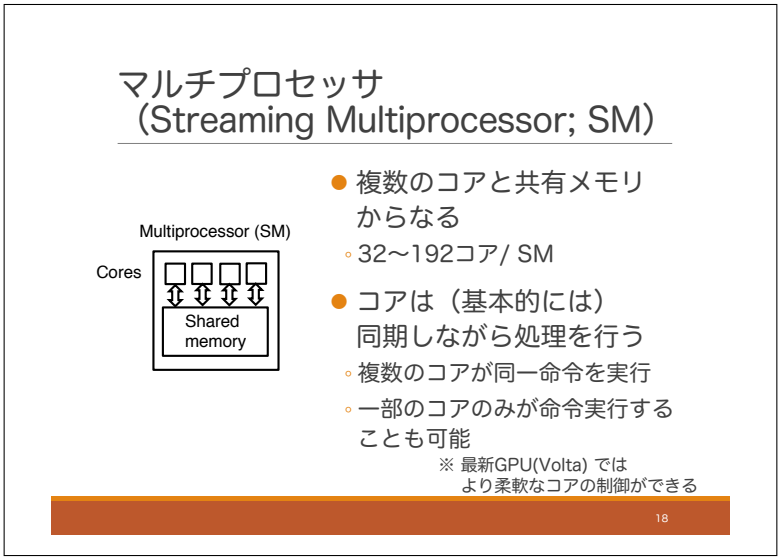
まず，手順 1 として CPU 上のプログラムは CPU 上のメモリにあるデータを GPU のグローバルメモリに転送します．手順 2 では，CPU 上のプログラムはユーザが作成した関数の実行を GPU に依頼します．すると，GPU はその関数を実行します．なお，GPU 上で実行される関数はカーネル関数と呼ばれます．手順 3 では，CPU 上のプログラムは，カーネル関数の実行終了後 GPU 上のグローバルメモリから必要なデータを取得します．その後，手順 4 として，必要に応じて手順 1〜3 を繰り返します．

なお，前のスライドでも話した通り，手順 1,3 については CPU メモリと GPU メモリの自動同期機能を使えば不要です．これはユニファイドメモリと呼ばれており，Pascal アーキテクチャ以降の GPU でサポートされています．



前のスライドの手順をシーケンスで表したのがこのスライドです．前のスライドの手順 1〜3 を 2 回繰り返す場合の例になっています．繰り返しになりますが，これも基本的なパターンの一つで，常にこのようなシーケンスになるというわけではありません．CPU プログラムがカーネル関数を呼び出すと，GPU 上の複数のスレッドが並列処理を行います．そしてすべての処理が終わると，処理が CPU に戻されます．

2013 年に発表された Kepler 以降の GPU では，複数のカーネル関数を同時に実行したり，カーネル関数の中から別のカーネル関数を呼び出したりできるようになっています．また，カーネル関数実行中に CPU が他の処理をすることもできます．なので，実際にはもっと複雑なシーケンスになることが多いです．

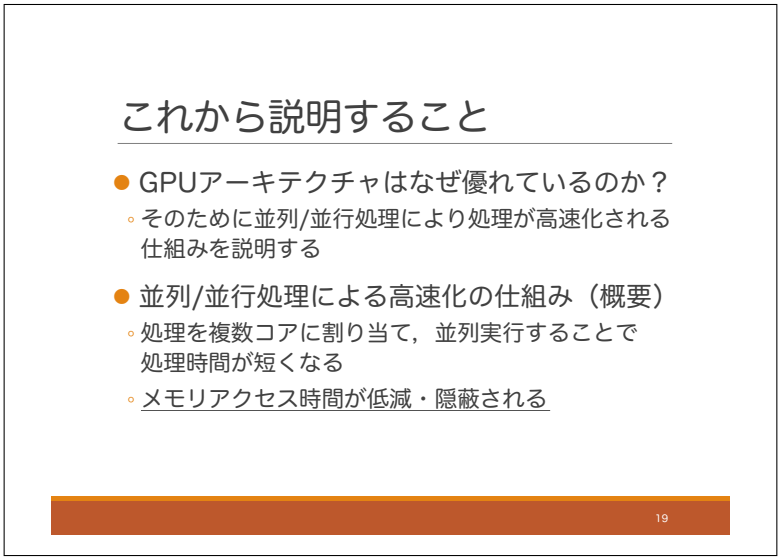


次に，マルチプロセッサの中身について見ていきます．NVIDIA はこれを Streaming Multiprocessor と呼んでおり SM と略しているので，本講習会でもこの略称を使います．

マルチプロセッサは，主に複数のコアと共有メモリと呼ばれるメモリからなります．これまでと同様に，マルチプロセッサにはもっと様々なモジュールが搭載されており，それらが高速化に役立っているのですが，ここでは省略します．1 マルチプロセッサに搭載されるコアの数は，アーキテクチャによって異なるのですが，32 コアから 192 コアの間くらいになっています．

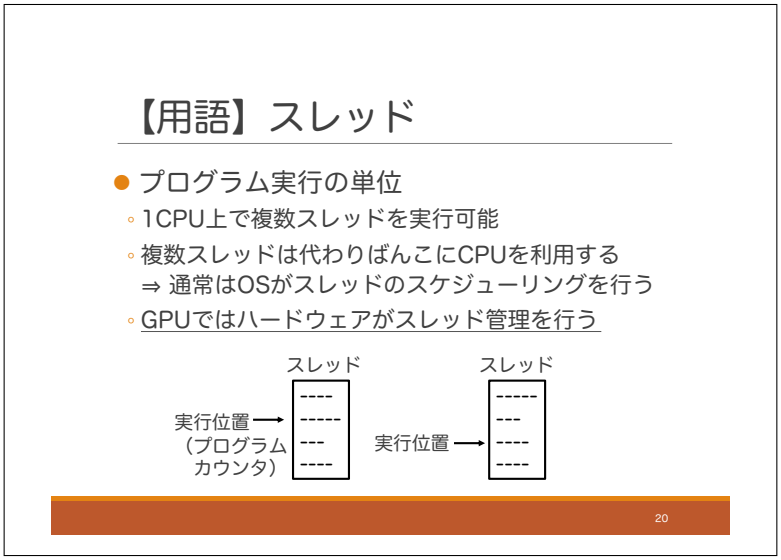
マルチプロセッサ内のすべてのコアは，基本的には同期して処理を行います．つまり，マルチプロセッサ内のコアは常に同じ命令を実行します．命令は同じですが，対象とするデータはコアごとに変えることができます．また，指定した条件を満たすコアのみ命令を実行するということもできます．複数のコアのそれぞれが異なる命令を実行したい場合には，1 つのコアのみが命令実行を行い，他のコアは休ませるという処理を繰り返すことになります．つまり，命令を並列に実行することはできず，一つのコアずつ順に命令を実行していくことになります．これでは効率の良い処理にならないので，プログラミングする際には極力このようなことがおきないようにアルゴリズムを設計する必要があります．

なお，2017 年に発表された Volta アーキテクチャでは良い柔軟なコアの制御が可能になっています．それ以前のアーキテクチャでは，すべてのコアは与えられたプログラムを同期実行していたのですが，Volta では条件分岐が発生した際にコアごとに異なる実行パスを辿れるようになりました．とはいえ，異なる命令を並列に実行できるようになったわけではないので，条件分岐が発生すると効率が悪くなることには変わりはありません．



GPU アーキテクチャの概要を話しましたので，これからは GPU アーキテクチャはなぜ優れているのかについて説明していきたいと思います．それを説明するために，まず，そもそも並列処理や並行処理によりなぜ処理が高速化されるのかということについて説明したいと思います．高速化される要因は主に二つあります．一つ目は，当たり前ですが，処理を複数のコアに割り当てて並列実行することで，処理時間が短縮できるということです．一人でやると時間がかかる仕事も 10 人でやれば早く終わるという感じで，これはわかりやすいと思います．もう一つは，並列/並行処理によりメモリアクセス時間が低減・隠蔽されるということです．隠蔽という言葉は後で説明するので，ここでは説明しないでおきます．ともかく，メモリアクセスに必要な時間コストが小さくなるんだと思ってもらえれば良いです．こちらはわかりづらいので，このあとは，メモリアクセスについて話していきたいと思います．

ちなみにこのスライドでは，並列/並行処理という言い方をしていますが，並列処理と並行処理には専門用語として使い分けがあります．並列処理と言った場合は，複数のコアを同時に動かすことで同時に複数の処理を行うことです．一方，並行処理は複数の処理を同時進行で実行することを意味し，この場合は，必ずしも複数のコアが同時に処理を実行する必要はありません．一つのコアが複数の処理を時分割で交互に処理する場合も並行処理となります．例えるなら，一人の人が複数の仕事をながら作業で同時こなしていくイメージです．10 人で 100 個の仕事をこなすような場合も並行処理と言えます．並列処置が高速化に繋がることはわかりやすいですが，並行処理自体が高速化に繋がることはわかりにくいかもしれません．しかし，並列処理を行わない並行処理，つまり一人で複数の処理をながら作業する場合でも一つ一つ個別に処理する場合に比べて処理が高速化されます．それは，メモリアクセス時間の隠蔽が大きな要因なのですが，それについて，これから説明していきたいと思います．

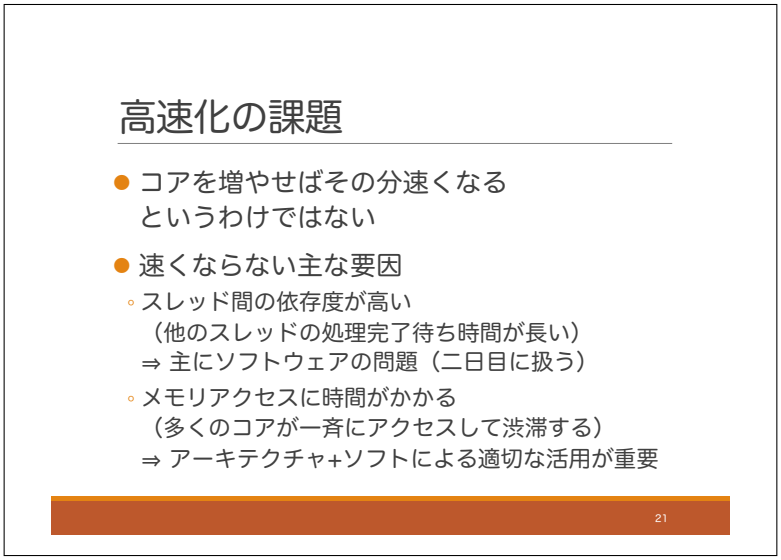


詳細な説明に入る前にスレッドという用語を説明しておきたいと思います．スレッドというのはプログラム実行の単位の一つです．コンピュータを使用してプログラムを実行する際，基本的には，コンパイルされたコードを上から下に 1 行ずつ実行していきます．ただし，ジャンプ命令というのがあると指定行にジャンプします．コンピュータは今実行している行を常に保持しており，それをプログラムカウンタと言います．

ここで，複数のコードを並行処理する場合，プログラムカウンタなどの内部状態をコードごとに保持しておく必要があります．コンピュータでは，複数のコードの 1 つ 1 つをスレッドという単位で管理しており，1 つのスレッドには一つのコードが割り当てられます．C 言語の初学者が習うような基本的なプログラムは，一つのスレッドからなる場合が多いです．しかし，グラフィカルユーザインタフェースつまり GUI により，マウスのクリックを検出するようなプログラムは基本的には複数のスレッドから構成されています．これは，マウスがクリックされた際，アプリを実行するためのコードとは別のコードが実行されるためです．

複数のスレッドの管理は通常は OS が CPU と連携して行うことが多いです．OS 自体もいくつかのスレッドの集まりですが，OS が他のスレッドに関して，いつどのスレッドを実行するのかを管理しています．

後で詳しく話しますが，GPU はこれらとは異なったスレッド管理の仕組みを持っています．GPU では OS を用いずハードウェアがスレッドの管理を行っています．これにより高速なスレッドの切り替えが可能です．

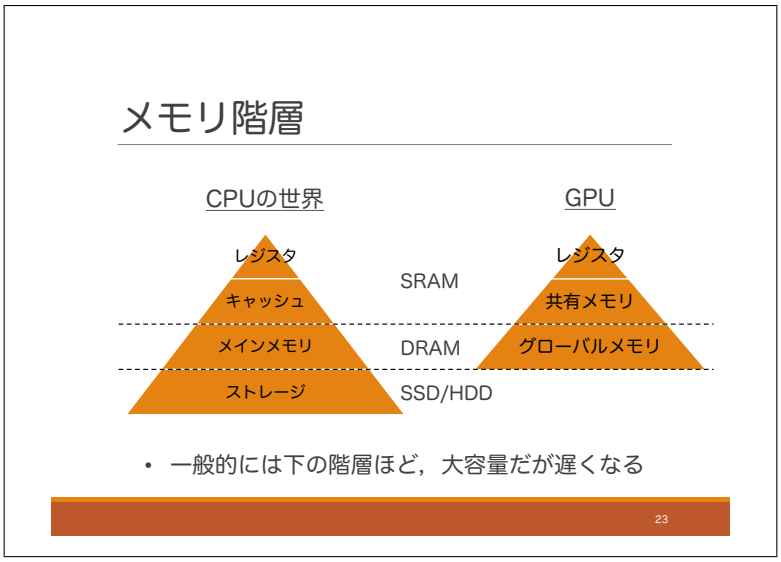


これまでの話の繰り返しになる部分もありますが，並列処理を用いて高速化を行う際の課題について見てみます．並列処理では，コア数を増やせばその分プログラムの実行が高速化されるというわけではありません．これは CPU のクロック周波数とは異なる状況です．クロック周波数については，周波数を大きくすれば，プログラムに何の変更を加えなくても実行速度が速くなりました．ところが並列処理については，そうではないということです．

コア数を大きくしても高速化されない原因の一つにスレッド間の依存度があります．例えば，100 コアによる並列処理を行うためには，少なくとも 100 個のスレッドを作る必要がありますが，もし無理矢理 100 個のスレッドを作ったとしても，あるスレッドが別のスレッドの処理完了を待たないといけないような状況が多いと処理は高速化されません．これはソフトウェアを工夫することで改善される場合もありますが，これについては二日目に扱います．別の原因として，メモリアクセスに時間がかかるということがあります．各コアはメモリからデータを読み込んで処理をすることが多いので，コア数を増やしていくと，一般にメモリアクセスが渋滞します．これは GPU のように膨大な数のコアを搭載したデバイスでは特に深刻であり，GPU にはアーキテクチャとソフトウェアが協調して効率の良いメモリアクセスを実現するための仕組みがたくさん搭載されています．

1.4.2 メモリ

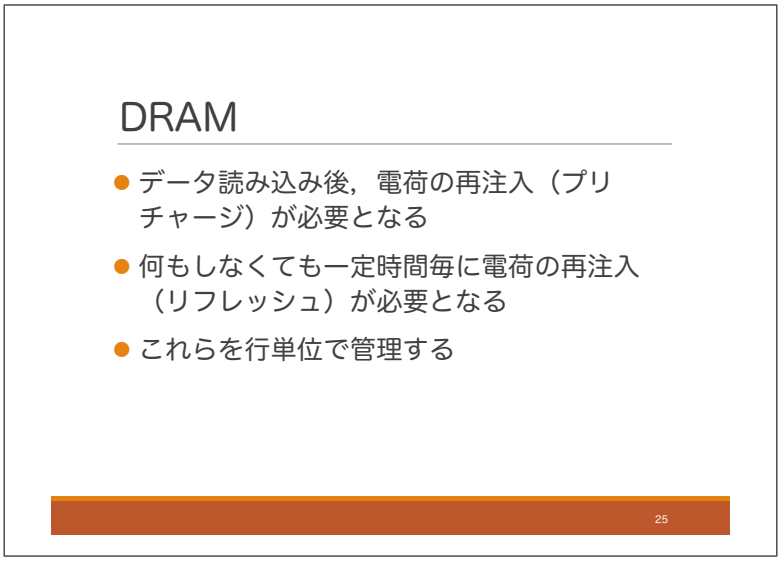
これからしばらくの間 GPU が備えている効率の良いメモリアクセスの仕組みについて説明したいと思います．まずメモリのハードウェア的な特性について説明したのち，それを踏まえてメモリへの効率的なアクセス方法を説明します．GPU のグローバルメモリには DRAM というメモリが使われており，共有メモリには SRAM というメモリが使われていますので，DRAM と SRAM について説明します．



まず全体像の説明としてメモリ階層について見てみましょう．コンピューターアーキテクチャのメモリシステムでは，大容量化と高速化の両方の需要に対応するため，複数のメモリを組み合わせて使用します．まず，CPU においては，レジスタとキャッシュのために SRAM が使われています．これは高速アクセスができますが，低容量です．プログラム実行時により大きいデータを扱えるようにするため，別途メインメモリが搭載されますが，これには DRAM が使われます．DRAM は SRAM よりもアクセスに時間がかかります．さらに大容量なデータをコンピューターに保持しておくためにストレージが搭載されますが，これには SSD や HDD と呼ばれるデバイスが使われます．これは大容量ですが，DRAM よりもさらにアクセスに時間がかかります．

一方 GPU ですが，レジスタや共有メモリには高速アクセス可能な SRAM が使用されます．グローバルメモリにはメインメモリと同じく DRAM が使われています．

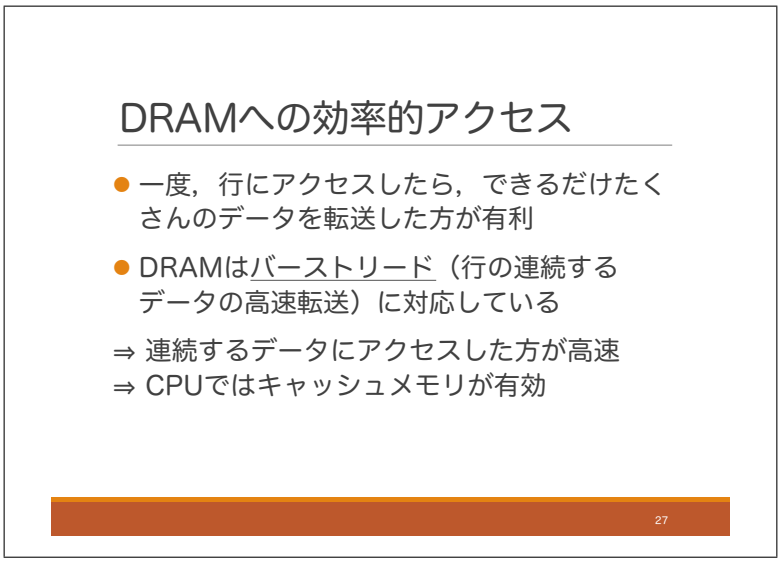
まず，DRAM についてですが，DRAM はキャパシタを用いて情報を記録するメモリです．キャパシタというのは電荷を蓄えることができる素子で，つまりはコンデンサのことです．コンデンサに電荷が溜まっているかどうかで 1 ビットを記録できます．電荷が溜まっていれば 1，溜まっていなければ 0 という感じです．DRAM ではこのキャパシタが格子状に配置されています．アドレスが 1 行目から順に配置され，行番号が大きくなるに従ってアドレスも大きくなっていきます．近年の DRAM ではこの格子が複数枚配置されていて，それぞれをバンクと呼んでいます．その際のアドレス配置について色々な方法が考えられますが，ここでは触れないでおきます．



DRAM のハードウェアとしての特徴はいくつかあり，一つ目はデータ読み込み処理の後，電荷の再注入が必要となることです．これはプリチャージと呼ばれています．読み込みの際，コンデンサに溜まっている電荷を吸い取ってしまうのでそれを元に戻してあげるのがプリチャージです．二つ目の特徴として，何もしない場合でも一定時間ごとに電荷の再注入が必要になります．これはリフレッシュと呼ばれています．コンデンサに溜まった電荷は何もしなくても少しずつ放電してしまうので，定期的に再チャージをするということです．DRAM ではプリチャージやリフレッシュを行単位で行います．DRAM のキャパシタが格子状に配置されていることにはこのようなメリットがあります．

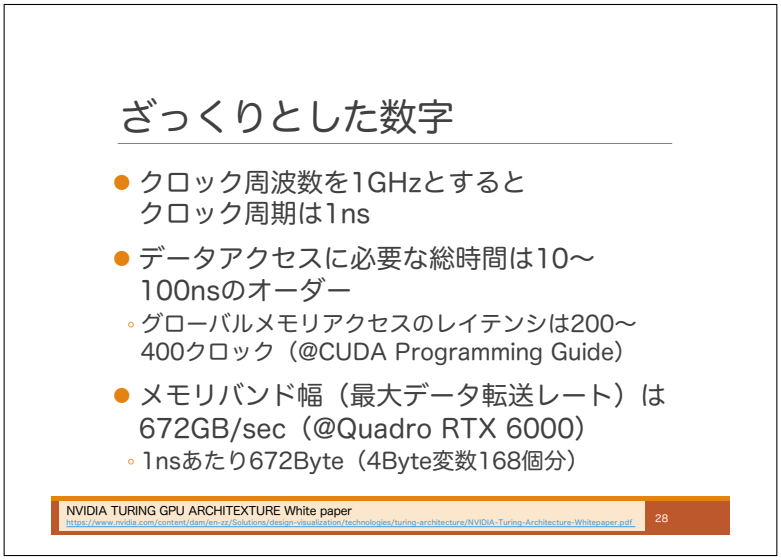
次に DRAM のデータ読み込み手順について見てみます．手順 1 が行のアクティベートで，信号線を DRAM の指定行に接続する処理です．手順 2 が指定列の転送で，アクティベートした行の中から，必要な列のデータを転送する処理です．手順 3 がプリチャージで，行の電荷を再チャージする処理です．これは他の行を読む前に行う必要があり，ディアクティベートとも呼ばれます．

ここで重要なことは，DRAM のデータを読む際，実際のデータ転送以外にも時間がかかるということです．



DRAM から高速にデータを転送するためには，一度行にアクセスしたら，できるだけたくさんのデータを転送した方が有利です．DRAM にはバーストリードという機能があり，行の中の連続するデータを高速に転送することができます．通常は読み込む列を指定するための制御信号を DRAM に送信するとそのデータが送られてくるのですが，バーストリードでは，制御信号をやり取りすることなく連続するデータを送信できるので高速なデータ転送が可能になります．

ということで，DRAM では同じ行の連続するデータにアクセスすると効率的だということがわかります．CPU のキャッシュメモリは DRAM に対しこのようなアクセスをしており，とても効率的だと言えます．



ここで時間の数値に関する感覚を持ってもらうために時間に関するざっくりとした数字を見てみたいと思います．まずプロセッサのクロック周波数ですが，だいたい 1GHz くらいのオーダーですので，クロック周期は 1ns くらいということになります．ざっくりいうと，プロセッサは 1ns に 1 回命令を実行できる能力を持っているということです．DRAM 中のデータへのアクセスに必要な時間はざっくり 10ns から 100ns のオーダーです．CUDA C Programming Guide には，DRAM が使われているグローバルメモリへのアクセスの待ち時間（レイテンシ）は 200〜400 クロックであると書かれています．ざっくり言うと，メモリアクセスには他の単純命令の 200 命令分くらいの時間が必要になると言うことです．DRAM からデータを読み込む際の最大データ転送レートを見てみると例えば，Quadro RTX 6000 と呼ばれるモデルでは，672GB/sec で，1ns あたりに直すと 672Byte です．これは 4Byte の変数，168 個分に相当します．これは結構速いのではないかと思います．つまりデータの転送自体は速いが，DRAM のアクセスには色々な手順があるので，トータルの待ち時間としては遅くなるということです．

DRAMへの効率的なアクセス

1. キャッシュメモリ
   - 多くのプログラムにおいて，一度アクセスしたアドレスとその周辺は近いうちにアクセスされる傾向がある（時間的局所性，空間的局所性）
   - ある1ワードにアクセスした時，そのワードが含まれるブロック（行の一部）の全データをキャッシュにバースト転送する
   - GPUもキャッシュメモリを持っている

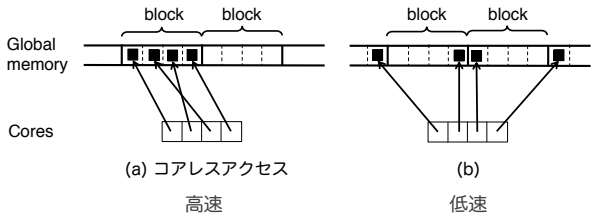
29

それでは，DRAM に効率的にアクセスするための具体的な方法についてみていきます．3 つ紹介しますが，まず最初は，キャッシュメモリです．多くのプログラムでは一度アクセスしたアドレスやその周辺のアドレスには近いうちにアクセスされやすいということが経験的に知られています．これは時間的局所性，空間的局所性と呼ばれるものです．そこで，アドレスを複数のブロックに分割しておき，DRAM のあるアドレスにアクセスした際，そのアドレスを含むブロックをキャッシュに転送します．転送は DRAM のバースト転送を使って高速に行えます．また，キャッシュには SRAM が使われるため，高速アクセスが可能です．

強調されることが少ないように思いますが，GPU もキャッシュを持っています．あまり強調されないのは，GPU は DRAM に効率的にアクセスするための別の手段を持っていてそれが強調されるためだと思います．それについては，後で説明します．

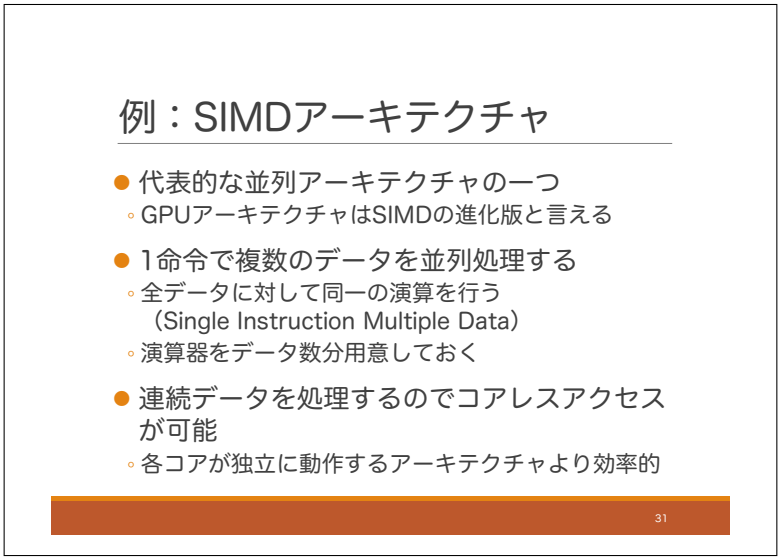
DRAMへの効率的なアクセス

2. コアレスアクセス
   - 複数のコアが連続するデータにアクセスする時，DRAMのバーストリードにより高速読込できる

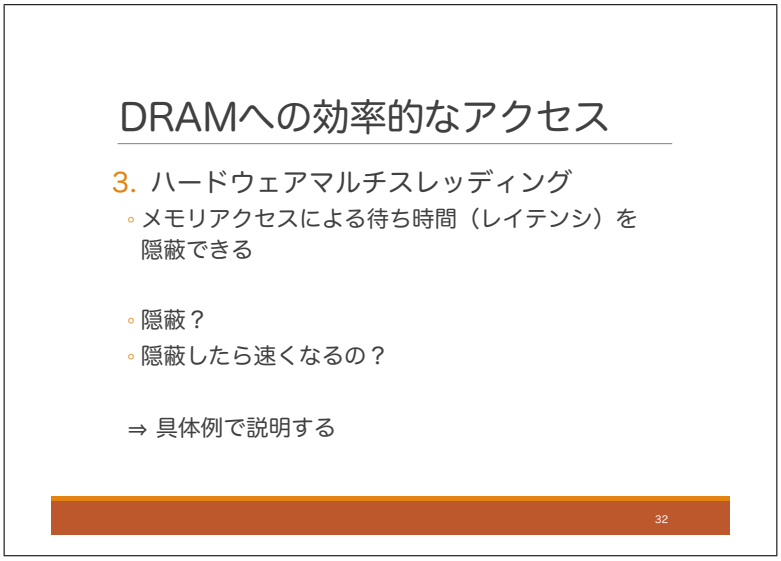


30

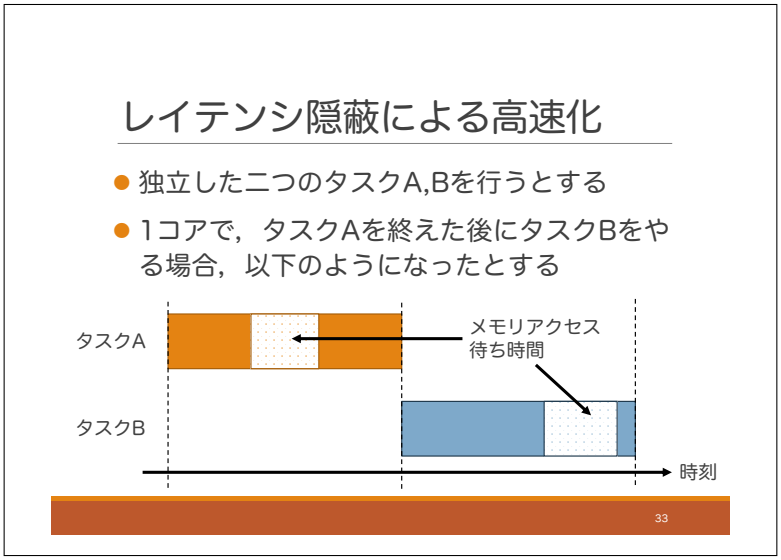
2 つ目は，並列アーキテクチャ限定の話になりますが，コアレスアクセスです．DRAM は複数のブロックに分かれていますが，(a) のように複数のコアが DRAM の同一ブロックにアクセスする時，バーストリードが使えるため，高速にアクセスできます．これをコアレスアクセスと呼んでいます．一方 (b) のようにコアが別々のブロックにアクセスする時は，ブロックごとに別々にアクセスしないといけなくなるため低速になります．



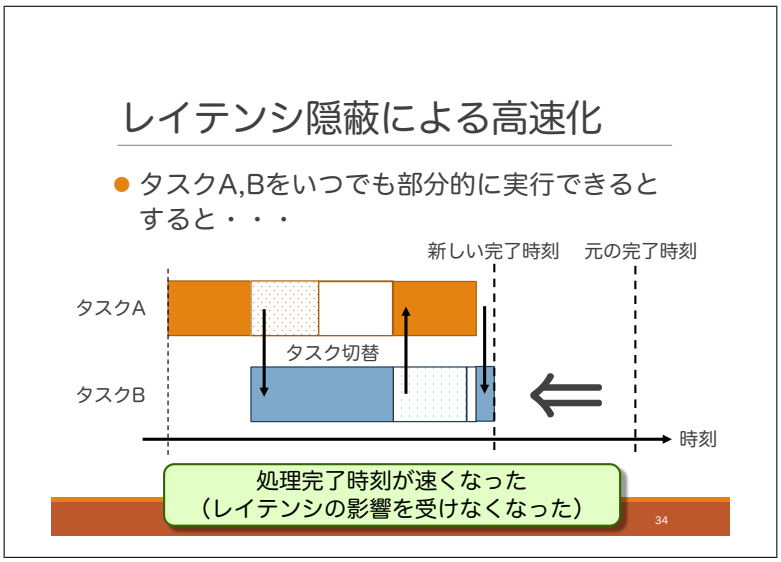
コアレスアクセスを活かせるアーキテクチャとして SIMD アーキテクチャがあります．SIMD アーキテクチャは代表的な並列アーキテクチャの一つです．GPU アーキテクチャは SIMD アーキテクチャの進化版であり，SIMD アーキテクチャの多くの特徴を引き継いでいます．SIMD アーキテクチャは複数のデータに対し同一命令の並列実行が可能なアーキテクチャです．そのために複数の演算器が搭載されています．SIMD とは Single Instruction Multiple Data の略です．SIMD アーキテクチャでは連続する領域のデータに対して，同一命令を実行するため，コアレスアクセスが可能です．各コアが独立に動作する並列アーキテクチャでは，アクセスするデータのアドレスはコアによりバラバラになるため，このようなコアレスアクセスはできません．SIMD アーキテクチャには様々な長所がありますが，効率の良いメモリアクセスも長所の一つと言えます．



3 つめは，ハードウェアマルチスレッディングです．メモリアクセスによる待ち時間（レイテンシ）を隠蔽する技術で，これが GPU アーキテクチャの大きな特徴なのですが，「隠蔽」とは何かがわかりづらく，また，隠蔽したからといって高速になるのかという疑問も湧いてくるところなので，レイテンシ隠蔽による高速化について，具体例を使って説明したいと思います．

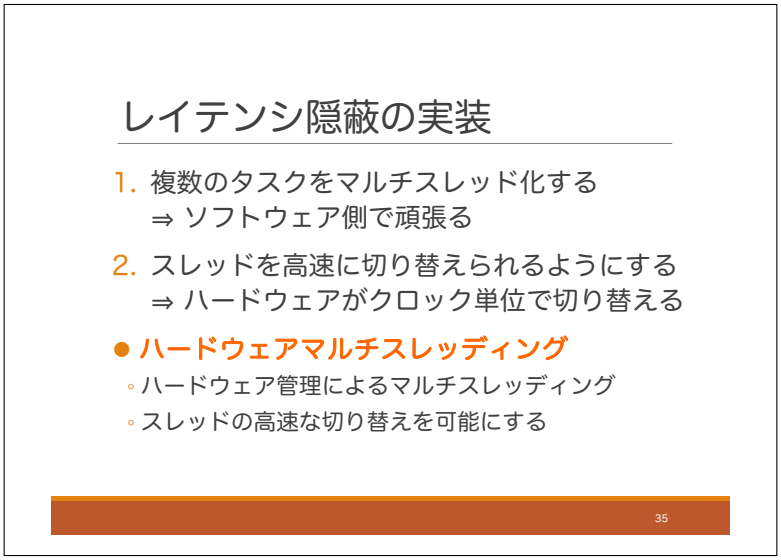


今，独立した二つのタスク A,B を 1 コアで行うとします．まず一つ目のパターンとして，タスク A を終えた後にタスク B をやることにするとスライドのようになったとします．タスク A,B はともに途中でメモリアクセスが必要で，その際，データを要求してからデータが届くまで長時間待たされるものとします．水玉になっている領域は何もできず待たされている時間帯です．



この時，二つ目のパターンとして，コアはいつでもタスク A,B を部分的に実行できるものとします．すると，タスク A のメモリアクセス待ち時間にタスク B を実行できます．また，タスク B のメモリアクセス待ち時間にタスク A を実行できます．このようにすると，タスクの完了時間がメモリアクセス待ち時間の影響を受けなくなり，結果として，タスク A,B の順に実行する場合よりも早く処理が完了します．これがレイテンシの隠蔽と呼ばれるものです．

少し脱線しますが，日常的な例として，無駄な会議を考えてみましょう．無駄な会議はなくすのが一番ですが，様々な理由でなかなか無くせない場合もあります．そこで，別のアプローチとして，会議の前に雑用を溜め込んでおき，会議中に雑用を内職するという方法が考えられます．こうすれば無駄な会議であっても，時間を無駄にせずにすみ，結果的に労働時間を減らすことができます．レイテンシの隠蔽とはそのような技術です．



レイテンシ隠蔽を行うために必要なことを見てみます．まず，ソフトウェア側で必要な対応として，複数のタスクをマルチスレッド化する必要があります．先ほどの無駄な会議の例では，雑用を溜め込んでおくということに相当します．次にハードウェア側で必要なこととして，処理するスレッドを高速に切り替えられるようにする必要があります．スレッドの切り替えに時間がかかってしまうと，その度に待ち時間が発生してしまうので，クロックごとに実行するスレッドを切り替えられることが望ましいです．

それを実現する技術がハードウェアマルチスレッディングです．ハードウェア管理によるマルチスレッディングであり，実行スレッドの高速な切り替えを可能にします．CPU が搭載されたコンピューターでは，一般的には OS がスレッドの切り替えを行います．しかしそのようにすると OS ソフトウェアがスレッドを管理するための処理が待ち時間となり，頻繁にスレッドを切り替えると待ち時間が増えていきます．GPU では，複数スレッドが実行可能状態で待機できるようにし，クロック毎にハードウェアが実行レッドを選択できるようにすることで，スレッド切り替えの待ち時間が少なくなるようにしています．

SRAM

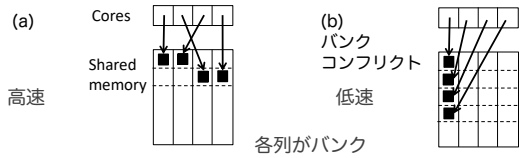
- 半導体（フリップフロップ回路）により情報を記憶する
- 高速アクセスが可能（1〜数クロックで1ワード(通常64bit)を取得可能）
- 通常、複数のバンクで構成される
- マルチプロセッサの共有メモリがSRAM

36

それでは，次に SRAM に移りたいと思います．SRAM は DRAM よりも高速なので，計算時間に対するインパクトは少ないですが，それでも高速アクセスのためのコツがありますので，それを話したいと思います．まず，SRAM とは半導体のフリップフロップ回路を用いて情報を記録するメモリのことです．高速アクセスが可能であり，だいたい 1 クロックから数クロックで 1 ワードを取得できます．1 ワードというのは 64bit アーキテクチャなら 64bit のことです．並列アーキテクチャでは，SRAM は通常複数のバンクから構成されます．つまり，SRAM は物理的に分けられた複数のメモリブロックから構成され，そのそれぞれをバンクと呼びます．GPU では，マルチプロセッサ内の共有メモリに SRAM が使われています．

SRAM

- 各バンクに同時にアクセスできる
  - 複数のコアが別々のバンクにアクセスする時，同時に処理が可能
  - 逆に複数のコアが同一バンクにアクセスすると、待ちが発生する→バンクコンフリクト

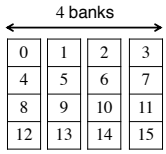


37

バンクのそれぞれは各コアと別々のバス（つまりデータ送信用の信号線）で繋がれており，各コアは異なるバンクに並列にアクセスすることができます．例としてコア数と共有メモリバンク数が共に 4 の場合で考えてみます．共有メモリの各列が 1 つのバンクだとします．(a) のように 4 つのコアがすべて別々のバンクにアクセスする時，並列にデータを取得できるため高速です．しかし，並列にアクセスできない場合もあります．複数のコアが同じバンクにアクセスすると，並列処理できません．(b) のように 4 つのコアがすべて同じバンクにアクセスすると並列処理は行われず，各コアにシーケンシャルにデータが転送されることになります．このように複数のコアが同じバンクにアクセスすることをバンクコンフリクトと呼びます．つまり，共有メモリに高速にアクセスするには極力バンクコンフリクトを避ける必要があるということです．

SRAM上のアドレス配置

- 連続するアドレスが異なるバンクになるように配置される
  ⇒ 各コアが連続するワードにアクセスすることでバンクコンフリクトを回避できる



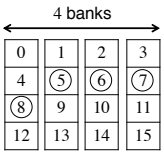
4バンクSRAMでのアドレス配置

38

バンクコンフリクトを避ける方法について考えるためにまず，SRAM 上のアドレスの配置について説明します．SRAM 上のアドレスは連続するアドレスが異なるバンクになるように配置されます．例えばバンク数が 4 つの場合はスライドのようになります．なので，例えば 4 つのコアが連続するアドレスにアクセスするような場合にはバンクコンフリクトは起きません．

例：SIMDとバンクコンフリクト

- 処理するデータを連続データにすることでバンクコンフリクトを回避できる



4並列SIMDでのバンクコンフリクト回避例
（○がアクセスするワード）

39

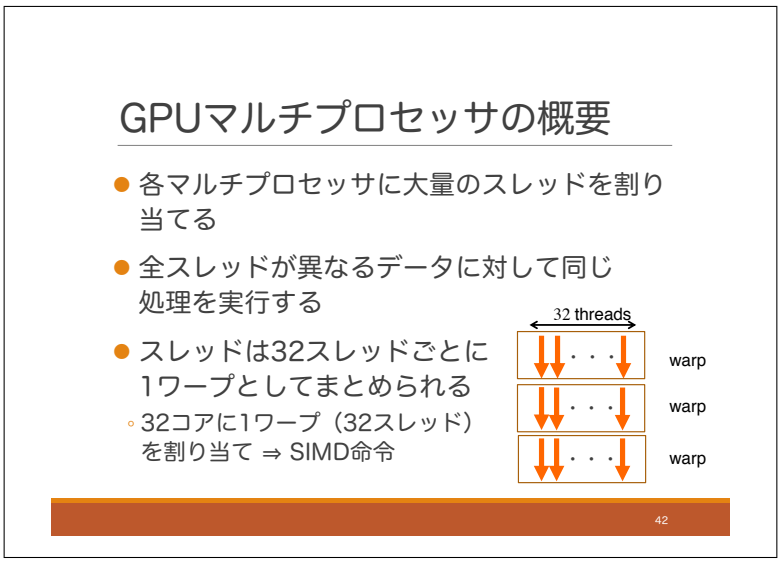
再び SIMD アーキテクチャの例を話すと，処理するデータを連続データにしている限りはバンクコンフリクトを回避できます．例えば，4 並列の SIMD でアドレス 5,6,7,8 にアクセスする場合にはバンクコンフリクトは起きません．この意味でも SIMD アーキテクチャは効率的なメモリアクセスが可能であると言えます．

効率的なメモリアクセスの方法についてまとめてみます．まず，DRAM も SRAM も連続するアドレスに同時アクセスすることで，高速にデータ転送が行えます．SIMD アーキテクチャでは，これを活かして効率の良いメモリアクセスを実現しています．

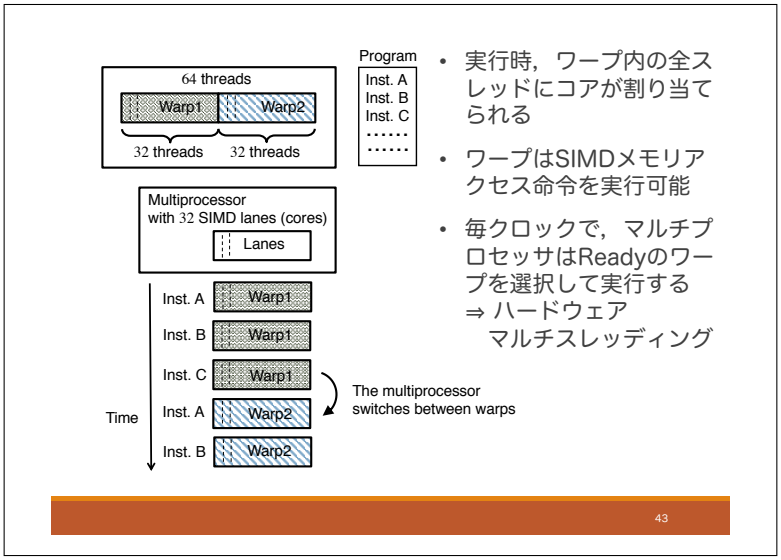
DRAM に効率よくアクセスするための別の手段として，ハードウェアマルチスレッディングがあります．これはデータ転送自体を高速化するのではなく，レイテンシを隠蔽するというものです．繰り返しになりますが，レイテンシの隠蔽とは，待ち時間に他の仕事をすることで待ち時間を無駄にしないという技術です．

1.4.3　マルチプロセッサ

さて，それでは GPU のマルチプロセッサの説明に移りたいと思います．GPU マルチプロセッサの長所を理解するということが今日の目的であり，今日のメインパートになります．GPU マルチプロセッサの方式ですが，NVIDIA 社は SIMT と呼んでいます．SIMT とは Single Instruction Multiple Threads の略です．その特徴ですが，SIMD アーキテクチャと同様の効率的なメモリアクセスとハードウェアマルチスレッディングによるレイテンシ隠蔽の機能の両方を備えていることです．コンピュータアーキテクチャの標準的な教科書の一つである「ヘネ＆パタ」ではこれをマルチスレッド SIMD プロセッサと呼んでいます．つまりハードウェアによるマルチスレッディングが可能な SIMD プロセッサということです．

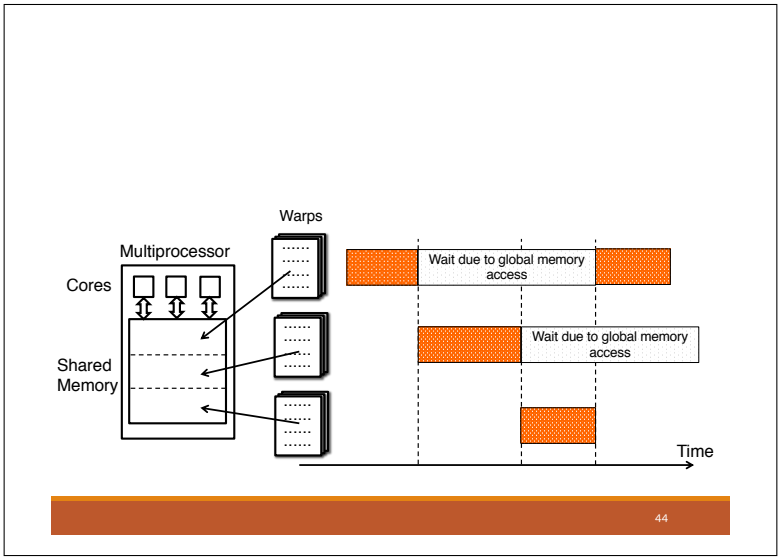


それでは，それをどのように実現するのかをみていきましょう．まず，各マルチプロセッサに大量にスレッドを割り当てます．もちろんソフトウェア側でたくさんのスレッドを生成しておかないと大量のスレッドを割り当てることはできませんが，そのようにソフトウェアを設計したものとします．異なるデータに対して，これらの全スレッドが同じ処理を行います．SIMD ではすべてのコアが同じ処理を行いますが，今回の SIMT ではコアからスレッドに置き換わっています．マルチプロセッサに割り当てられたスレッドは 32 スレッドごとにまとめられます．この 32 スレッドのまとまりのことをワープと呼びます．そして基本的にはマルチプロセッサ内の 32 個のコアに 1 ワープ（32 スレッド）を割り当てます．ワープ内の全スレッドは常に同じ処理を行うので，マルチプロセッサはワープに対して，SIMD 命令を実行することになります．そして，この SIMD 命令をすべてのワープに対して実行することで，全スレッドに対して同じ処理を行うことができます．

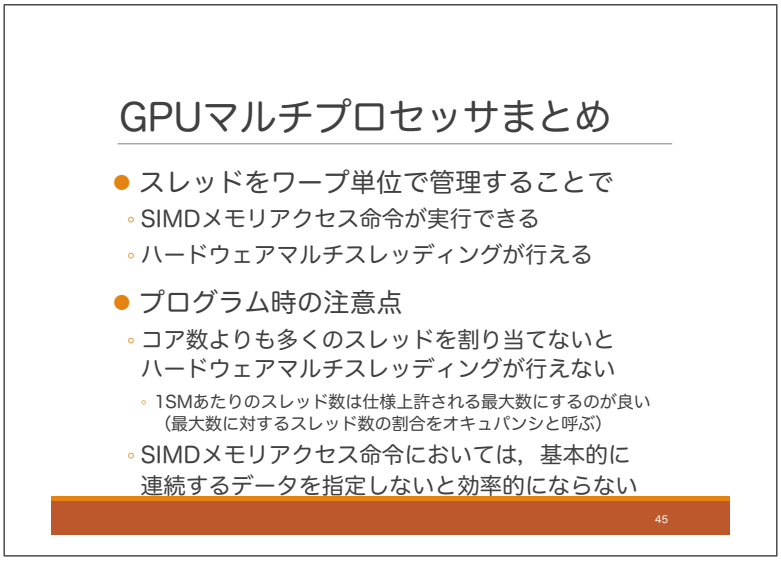


次にワープのコアへの割り当てについて具体例を使って説明します．やや複雑ですが，大事な部分なのでじっくりみていきましょう．あるマルチプロセッサに 64 個のスレッドが割り当てられており，すべてのスレッドは命令 A，命令 B，命令 C … という同じ処理を行うものとします．本当は 1024 個とか 2048 個とかくらい割り当てた方がいいのですが，ここではわかりやすくするため，64 個とします．前のスライドで話した通り，マルチプロセッサへの割り当てスレッドは 32 スレッドずつワープとしてまとめられます．なので，この例では 2 つのワープがあることになります．これらのスレッドをマルチプロセッサ内の 32 個のコアで実行することを考えます．

この時，各ワープを交互に実行する必要はありません．ハードウェアが毎クロックにおいて，実行可能なワープの中からどのワープを実行するかを決定します．スライドの例では一つ目のワープの命令を 3 つ続けて実行したのち，二つ目のワープの命令を 2 つ続けて実行しています．これは SIMD マシンに対するハードウェアマルチスレッディングと言えます．通常のハードウェアマルチスレッディングではクロックごとに実行するスレッドを 1 つ選択していましたが，GPU ではクロックごとに実行する 32 スレッドを選択します．このように 32 スレッド単位でガバッと入れ替えられるようにすることで，SIMD とハードウェアマルチスレッディングを両立しています．



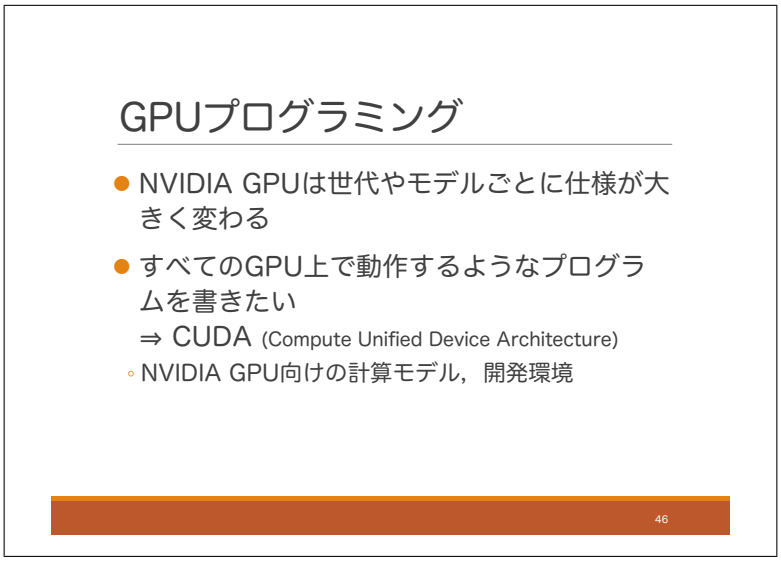
GPU マルチプロセッサがハードウェアマルチスレッディングによりグローバルメモリアクセスのレイテンシを隠蔽する例をみてみましょう．一番上のワープが SIMD 命令によりグローバルメモリアクセスを行なったことで待ち時間に入ったとします．するとマルチプロセッサは他の実行可能なワープ，例えば真ん中のワープを実行します．そのワープもグローバルメモリへの SIMD アクセスにより待ち時間に入ったとするとマルチプロセッサは 3 つ目のワープを実行します．そして，一つ目のワープが待ちから復帰すると，再び一つ目のワープを実行します．このような感じで，グローバルメモリに SIMD 命令で効率よくアクセスしながらもレイテンシを隠蔽することができます．



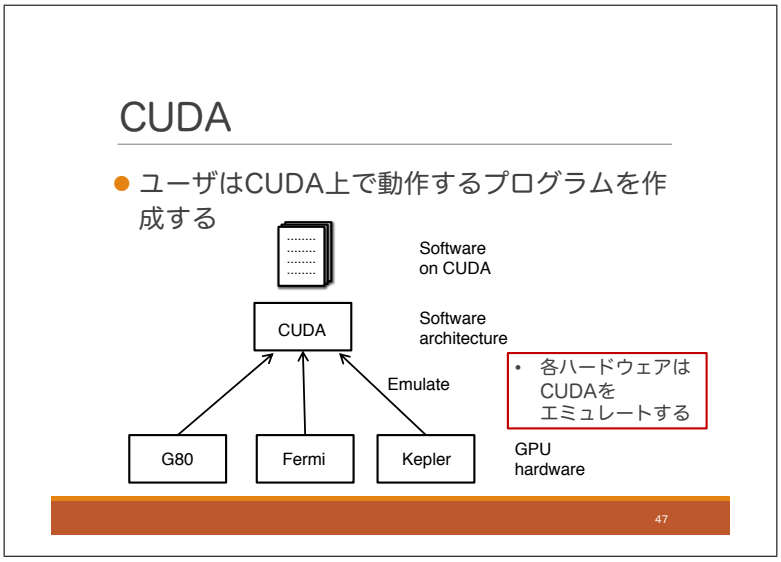
GPU マルチプロセッサについてまとめると，スレッドをワープ単位で管理することが特徴です．これにより，SIMD メモリアクセス命令に対応しつつも，ハードウェアマルチスレッディングも行えます．

これを踏まえてプログラミングする際の注意点は以下の通りです．まず，ハードウェアマルチスレッディングを行うためにコア数よりも多くのスレッドを生成しないといけないということです．アーキテクチャごとにマルチプロセッサに割り当て可能なスレッドの最大数が決まっていますが，レイテンシ隠蔽の効果を大きくするためには，最大数のスレッドを割り当てるようにするのが良いです．最大数に対する実際の割り当てスレッド数の割合をオキュパンシと呼んでいます．なので，オキュパンシを 100%にするのが良いということです．オキュパンシを大きくする際の障壁については二日目に話します．次にワープ内のスレッドは基本的にはメモリ内の連続する領域にアクセスするのが良いです．これはメモリのところで説明した通りです．

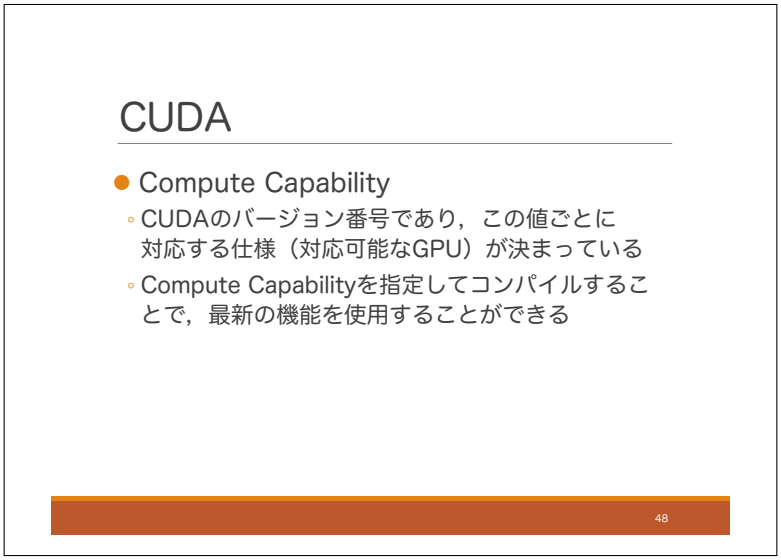
1.5 GPU プログラミング



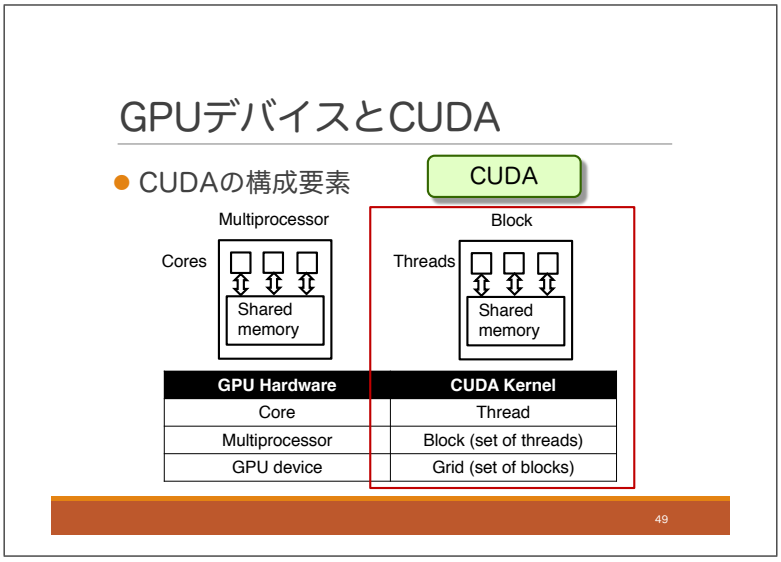
さて，ここからは GPU プログラミングの説明をしていきます．まずは全体的な話をしますが，NVIDIA 社の GPU は世代やモデルごとに仕様が大きく変わります．その際，異なる GPU を使うたびにプログラムを書き換えないといけないのでは大変ですので，すべての GPU で動作するようなプログラムが書けることが望ましいです．そのようなことが可能になる仕組みが CUDA です．CUDA は Compute Unified Device Architecture の略です．アーキテクチャとなっていますが，それだけでなく，GPU プログラムの開発環境などもすべて含めて CUDA と呼ばれています．なお，CUDA は NVIDIA 社 GPU のためのものであり，他社 GPU はサポートされていません．



CUDA のアーキテクチャの概要はこのようになります．まず CUDA はソフトウェアアーキテクチャとなっており，ソフトウェアは CUDA 上で動作するように作成します．そして，GPU ハードウェアはそれぞれ CUDA をエミュレートします．このようにすることで，GPU ハードウェアの仕様が異なっていても同一のプログラムを実行できるようになっています．

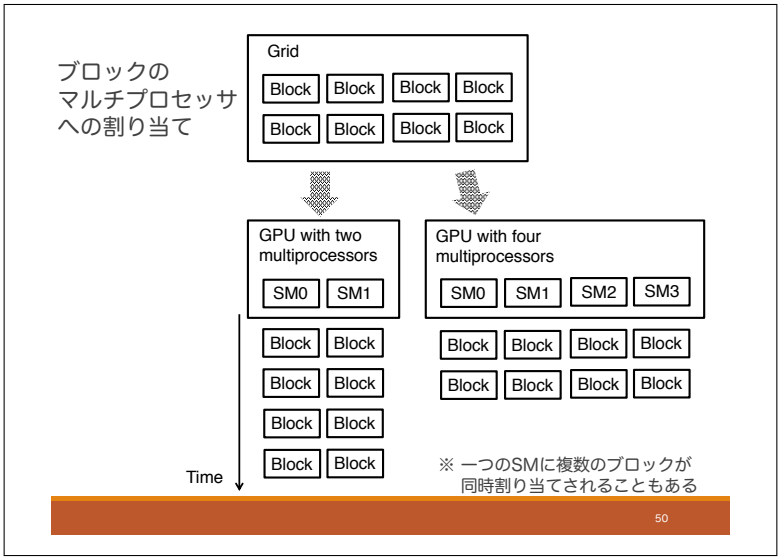


どの GPU でも動かせるプログラムを書きたいという話をしましたが，実際にはそうでない場合もあり，例えば，最新 GPU の新しい機能を使って特別なことをやりたいということもあります．このような場合に対応するため，CUDA では対応する機能や仕様のセットに対してバージョン番号をつけており，それを Compute Capability と呼びます．各 GPU モデルに対し，そのモデルに付けられた Compute Capability を確認することで対応する機能や仕様を知ることができます．また，Compute Capability を指定してコンパイルすることで，一部の GPU でしか対応していない機能を使うことができます．



CUDA アーキテクチャの中身について見てみましょう．CUDA では様々なスレッドの集合が定められています．GPU ハードウェアとの対応を見ながら見ていきましょう．

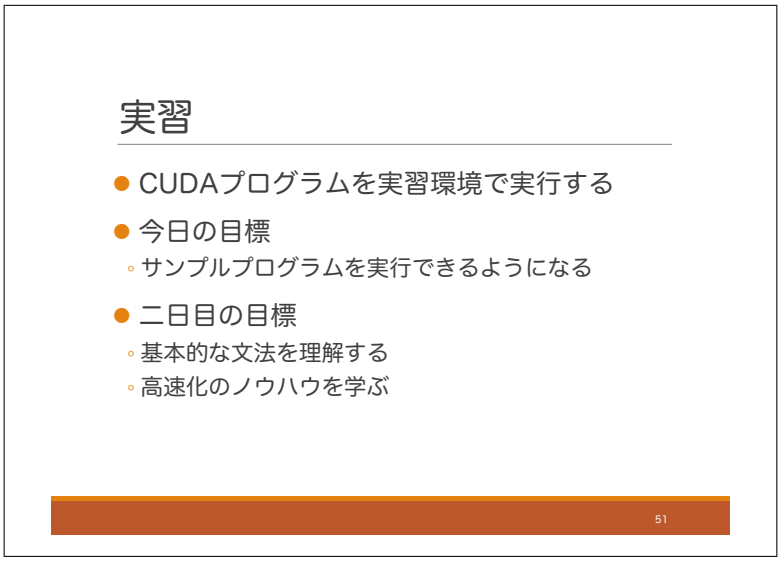
まず，コアに対応するのはスレッドです．実際には，一つのコア上で複数のスレッドが実行されます．次にマルチプロセッサに対応するのはブロックです．ブロックとは同一のマルチプロセッサで実行されるスレッドの集合です．実際には，一つのマルチプロセッサ上で複数のブロックが実行されます．しかし，一つのブロックが複数のマルチプロセッサで実行されることはありません．最後に GPU デバイスに対応するのが，グリッドです．グリッドはブロックの集合です．



CUDA 上で開発されたプログラムが GPU 上でどのように実行されるかについてみてみましょう．例として，1 グリッド，8 ブロックからなる CUDA プログラムを考えます．まず，このプログラムを 2 つのマルチプロセッサからなる GPU で実行する場合には，左の矢印のように一つのマルチプロセッサが 4 つのブロックを担当することになります．もし 4 つのマルチプロセッサからなる GPU で実行する場合は，右の矢印のように一つのマルチプロセッサが 2 つのブロックを担当することになります．このように GPU の構成が変わっても同じプログラムを実行できます．

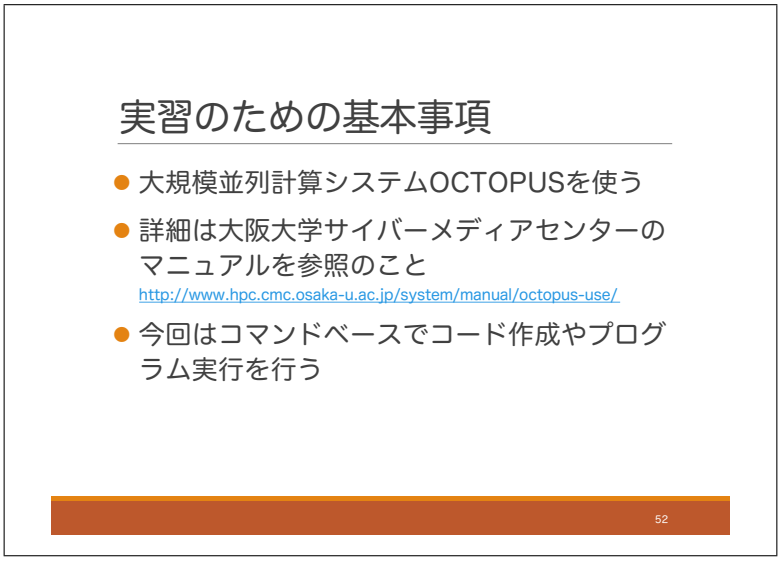
なお，このスライドでは各マルチプロセッサはブロックを一つずつ計算していますが，実際は一度に複数のブロックが割り当てられることが多いです．

1.6 実習



それでは実習に移りましょう．実習では CUDA 上で開発したプログラムを実習環境で実行します．

今日の目標は，サンプルプログラムを実行できるようになることです．二日目は CUDA の基本的な文法を理解し，高速化のノウハウを身に着けることを目標とします．



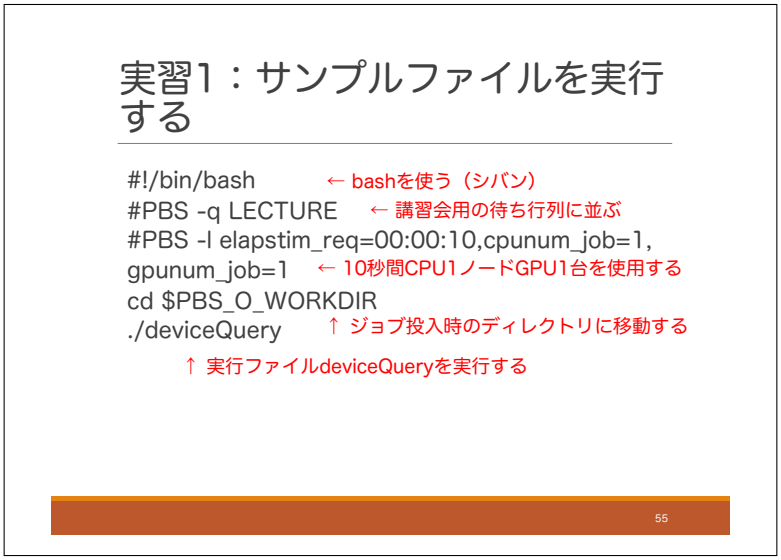
実習のための基本事項をいくつか述べます．まず実習環境としては，大阪大学サーバーメディアセンターの大規模並列計算機システム OCTOPUS を使います．詳細はサイバーメディアセンター作成のマニュアルをご覧ください．また，今回はコマンドベースでコード作成やプログラム実行を行います．GUI が使える場合には Nsight と呼ばれる開発プラットフォームを使うのが便利ですが，今回は使用しません．

それでは，本日の実習としてサンプルファイルを実行してみます．ここで，注意事項ですが，本計算機環境ではファイル実行の仕方がパソコンで行う場合とは異なります．通常はコマンドラインから実行ファイル名を打ち込むだけですが，今回はそうではありません．本環境は複数のユーザが共同利用しているので，ジョブスケジューラと呼ばれる各ユーザのジョブを管理するソフトウェアが導入されています．プログラムを実行するにはジョブスケジューラにジョブを投げる必要があります．

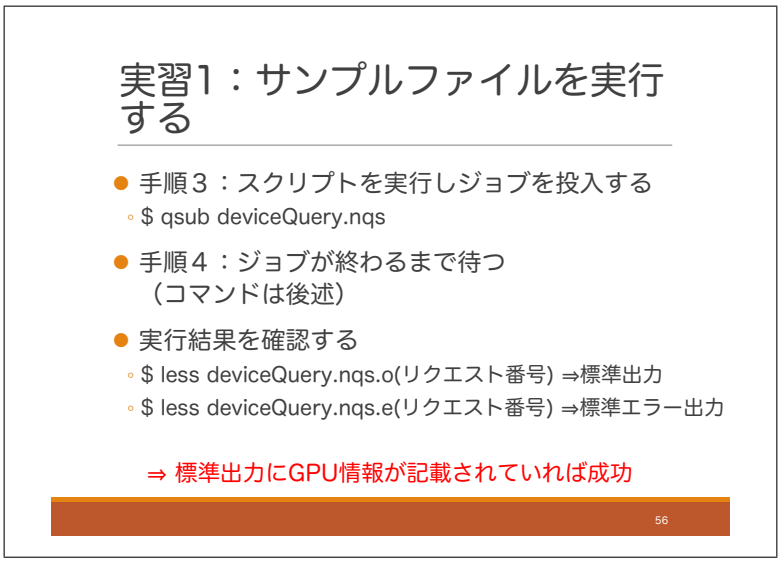
この実習ではビルド済みの実行ファイルを実行してみます．実行するのは deviceQuery と呼ばれるプログラムで，これは CUDA において最初に実行されることが多いサンプルの一つです．使用する GPU の仕様を出力するプログラムになっており，この後のプログラミングにも役立ちます．

それでは実際にやってみましょう．計算機システムにログインしたのち，スライドの手順 1,2 をやってみてください．手順 1 はサンプルファイルを自分のホームディレクトリにコピーする処理です．手順 2 ではジョブスケジューラにジョブを投げるためのスクリプトを作ります．vi を立ち上げたのち，その下 5 行を打ち込んで保存してください．

なお，Linux の操作に不安のある方は，付録の Linux 入門の説明をチェックしてみてください．cd, cp コマンドや vi については後半の「基本的な Linux コマンド」のところに説明があります．

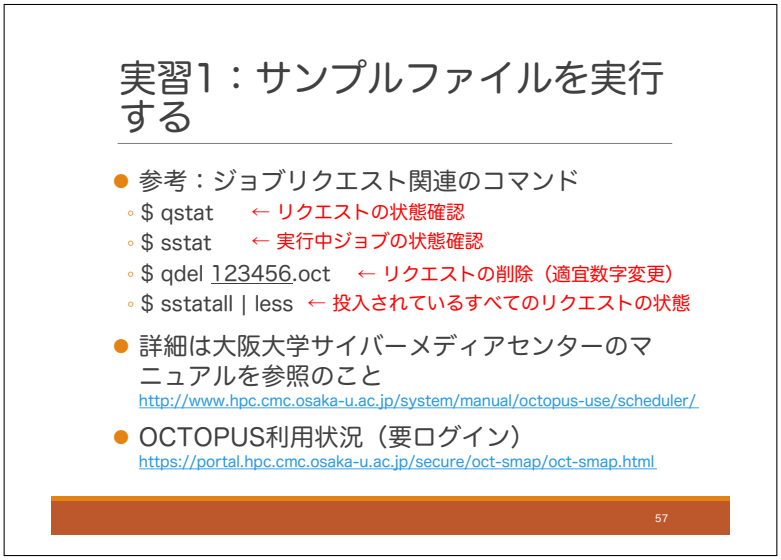


手順 2 で作成したスクリプトについて，少し中身を見てみましょう．1 行目は bash のパスを指定しています．いわゆるシバンと呼ばれるもので，おまじないだと思って書いておきましょう．2 行目はジョブの順番待ちのために並ぶ待ち行列を指定しています．この講習会では LECTURE という特別な待ち行列を作っていますので，そこに並びます．3 行目ではジョブ実行について設定しています．ここでは CPU1 ノードと GPU1 台を 10 秒間使う設定になっています．下 2 行は，ファイルを実行するフォルダに移動して，実際にファイル実行する処理です．



それでは実際にジョブを投入してみましょう．手順 3 によりジョブが投入されます．その後手順 4 として実行が終わるまで待ちましょう．ジョブの状態を確認するコマンドは次のスライドに書いてあります．最後に実行結果を確認します．実行結果ですが，端末に表示されるのではなく，ファイルに出力されます．プログラムの出力には，標準出力と標準エラー出力がありますが，スクリプト名の後ろに，ドット＋ｏ＋ リクエスト番号が書かれたものが標準出力です．また，スクリプト名の後ろに，ドット＋ｅ＋リクエスト番号が書かれたものが標準エラー出力です．

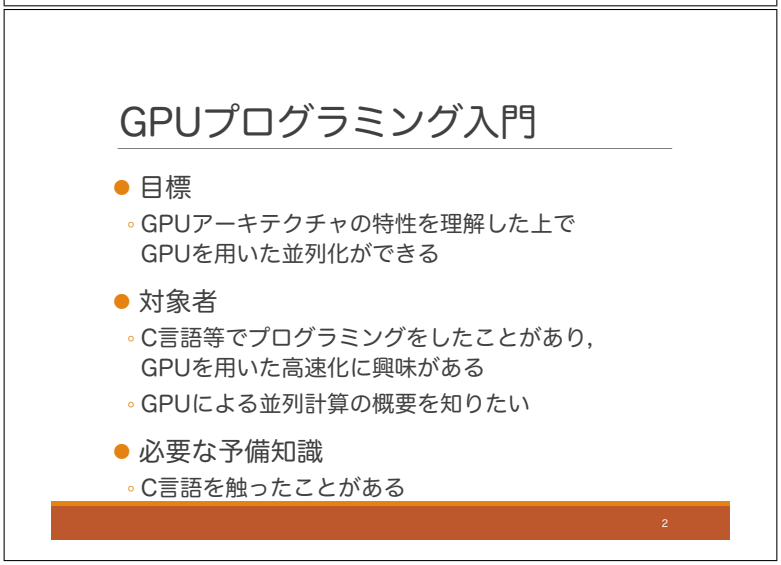
標準出力に GPU の仕様の情報が書かれていれば成功です．



最後に，リクエストしたジョブの状態を確認するコマンドを紹介します．まず，qstat は投入したジョブの状態を確認するコマンドです．sstat は実行中のジョブの詳細情報を確認するコマンドです．投入したジョブを削除するには qdel を使います．自分が投入した以外のジョブも含めてすべてのジョブのスケジューリング状態を調べるには sstatall を使います．詳細についてはサイバーメディアセンター作成のマニュアルをご覧ください．また，OCTOPUS の利用状況について，Web ページで確認できます．混雑状況の確認などにご利用ください．

第2章　二日目

2.1　はじめに



二日目の講義を始めます．初日の繰り返しになりますが，本講習会は，GPU を使って高速な並列プログラムを書きたい人のための二日間の入門講座です．目標は，単にプログラミングの文法を知るだけでなく，GPU のアーキテクチャを理解した上で，高速に動作するプログラムが書けるようになることです．対象者は，C 言語でプログラミングをしたことがあり，GPU を用いた高速化に興味がある人です．C 言語については，簡単なコードが読めるくらいの知識があれば大丈夫です．また，単に GPU による並列計算の概要を知りたいという人も対象としています．

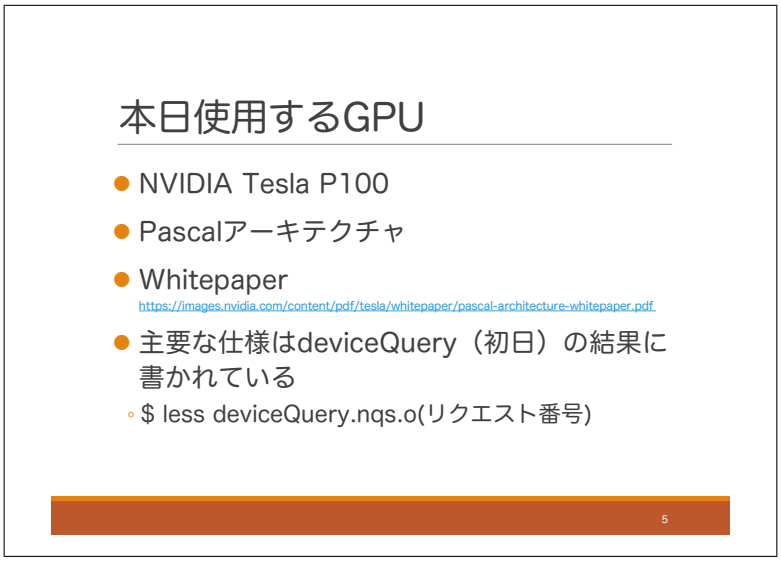
講習会の内容はスライドの通りです．初日は，GPU のアーキテクチャを中心に話しました．初日は座学中心でしたが，最後にサンプルプログラムを動かしてみました．二日目は実際に GPU のコードを書きながら，プログラミングの文法と高速化のノウハウを学習します．

初日の繰り返しになりますが，本テキストは大阪大学サイバーメディアセンターでの講習会で使うことを想定して作られています．もし他の環境で使用している場合は，適宜読み替えてください．サイバーメディアセンターを利用する場合，GPU が使用できるシステムは OCTOPUS です．事前にアカウントを取得し，OCTOPUS にログインできるようにしておいてください．

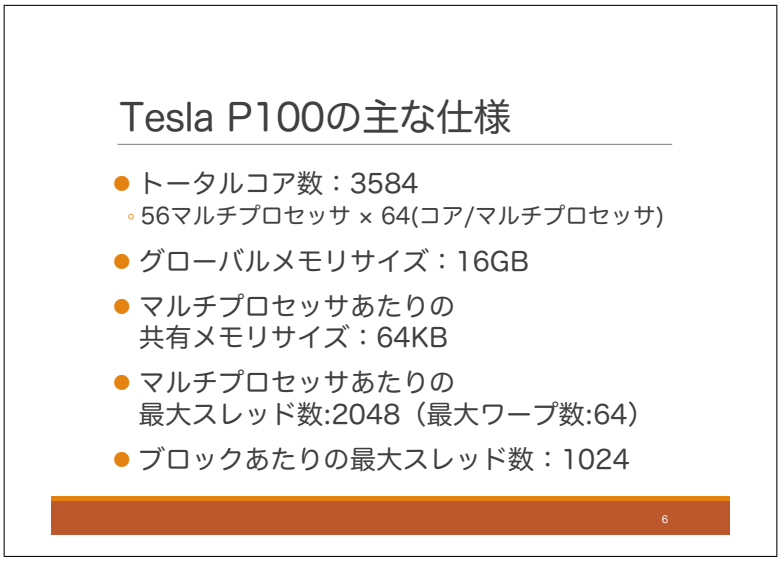
二日目の内容の詳細はスライドのようになります．

まず，ハローワールドということで，環境設定をしたのち，簡単なプログラムを書きながら CUDA の基礎的な文法を学習します．CUDA のプログラミング言語は既存の言語を GPU が使えるように拡張する形で作られています．C++，Fortran, Python などから拡張されていますが，本日は C++を GPU 向けに拡張した言語である CUDA C++を扱います．その後，高速なアルゴリズムを設計するための考え方をリダクションと呼ばれる問題を例にして説明します．最後に様々な高速化関連トピックを扱います．

2.2　使用する GPU の仕様



プログラミングを始める前に，本日使用する GPU の仕様をみておきましょう．大型計算機システム OCTOPUS には NVIDIA Tesla P100 という GPU が搭載されています．2016 年に発表された Pascal アーキテクチャを採用したモデルです．詳細はスライドに記載のホワイトペーパーをご覧ください．仕様については，初日に実行した deviceQuery と呼ばれるサンプルファイルを実行することでも取得することができます．初日にこのファイルを実行済みの場合には，その結果を見てみましょう．



主な仕様について，スライドに書いておきます．まず，1 デバイスあたりのコア数は 3584 です．内訳としては，56 個のマルチプロセッサが，それぞれ 64 個のコアを持っています．メモリについては，グローバルメモリのサイズが 16GB，マルチプロセッサあたりの共有メモリのサイズが 64KB です．

また，1 マルチプロセッサに最大 2048 個のスレッドを割り当てることができます．これをワープ数に直すと 64 個になります．また，1CUDA ブロックあたりの最大スレッド数は 1024 です．これについては後でででてきます．

2.3 環境設定

- CUDAファイルのコンパイルはnvccコマンドで行うため，そのコマンドへのパスを通す

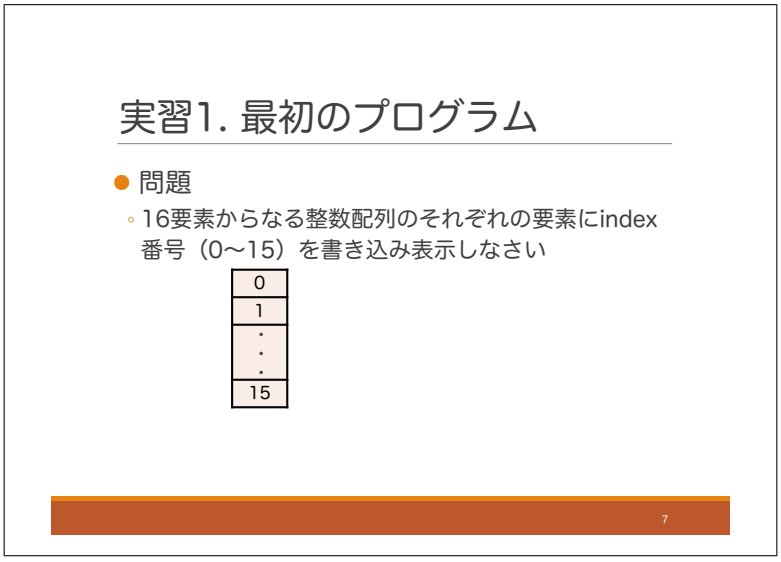
```
$ echo 'export PATH=/octfs/apl/CUDA/cuda-9.0/bin:$PATH' >> ~/.bash_profile
$ source ~/.bash_profile
```

- 大阪大学サイバーメディアセンター 大規模計算機システムの「OCTOPUSの利用方法」を参照のこと
  http://www.hpc.cmc.osaka-u.ac.jp/system/manual/octopus-use/
  - フロントエンド利用方法：
    http://www.hpc.cmc.osaka-u.ac.jp/system/manual/octopus-use/frontend/

それでは，実習を始めましょう．まず環境設定をします．CUDA プログラムをコンパイルする際，nvcc というコマンドを使いますが，そのコマンドへのパスを通します．OCTOPUS 環境であれば，スライドのようにコマンドを打ってください．「.bash_profile」という設定ファイルにパス情報を追記して，その設定ファイルの再読み込みを行っています．

環境設定は以上ですが，OCTOPUS を使用する際はその前に関連ドキュメントを確認しておきましょう．そのURL を書いておきます．

2.4 実習 1：最初のプログラム



まず最初の実習として，スライドのような問題を考えます．16 要素の配列のそれぞれに index の値を書き込むという問題ですが，これを並列化しない場合とする場合の 2 通りでプログラミングしようと思います．

2.4.1 並列化しないプログラム

✏️ C++プログラム（並列化なし）による計算

ソースコード（ex1-c.cpp）

```cpp
#include <stdio.h>

void WriteIndex(int* a, int n)
{
    for (int i = 0; i < n; i++) {
        a[i] = i;
    }
}

int main()
{
    // Init
    int N = 16;
    int* h_a = new int[N]; // on host (CPU)

    // Main
    WriteIndex(h_a, N);

    // Display results
    for (int i = 0; i < N; i++) {
        printf("%d\n",h_a[i]);
    }

    // Free memory
    delete [] h_a;

    return 0;
}
```

まずは，並列化しない場合のソースコードを書いてみましょう．ファイル名を ex1-c.cpp とします．この後 CUDA プログラムを作ることを考えて若干変なコードになっていますが，普通に読めると思います．

次に，このプログラムを実習環境で実行してみます．初日に話したように実習環境でプログラムを実行するには，ジョブを投入して待ち行列に並ぶ必要があります．

ジョブスクリプト（cuda.nqs）

- 10秒制限
- 実行ファイル名をa.outにする
- 講習会用の待ち行列（LECTURE）に投入する
- 今回GPUは使用しないが気にしない💦

```
#!/bin/bash
#PBS -q LECTURE
#PBS -l elapstim_req=00:00:10,cpunum_job=1,gpunum_job=1
cd $PBS_O_WORKDIR
./a.out
```

まず，ジョブスクリプトを書きましょう．vi 等のエディタで上記の内容を書いて保存しましょう．ファイル名はなんでもいいですが，ここでは cuda.nqs とします．制限時間を 10 秒とし，講習会用専用の待ち行列である LECTURE にジョブを投入することにします．

コンパイルとジョブ投入

- コンパイル(g++)により実行ファイルa.outが生成される

```
$ g++ ex1-c.cpp
$ qsub cuda.nqs
```

それではコンパイルして，ジョブを投入しましょう．今回は GPU を使用しない普通の c++プログラムなので，g++コマンドを使ってコンパイルすることにします．上記の g++コマンドを端末で実行すると a.out というファイルが出来上がります．次に qsub コマンドを用いて，ジョブを投入します．初日に反した通り，ジョブ投入後は qstat コマンドでリクエストの状態を確認できます．また，実行中のジョブに関しては，sstat で状態を確認できます．

2.4.2 CUDA による GPU プログラム（その 1：1 ブロックのみ使用）

ソースコード（ex1-cuda.cu）

```
#include <stdio.h>

__global__ void WriteIndex(int* a, int n)
{
    int i = threadIdx.x;
    if (i < n) {
        a[i] = i;
    }
}

int main()
{
    // Init
    int N = 16;
    int* h_a = new int[N]; // on host (CPU)
    int* d_a;              // on device (GPU)
    cudaMalloc(&d_a, N * sizeof(int));

    // Copy data from host to device
    // Do nothing

    // Main
    int blocks = 1;
    int threadsPerBlock = N;
    WriteIndex<<<blocks, threadsPerBlock>>>(d_a, N);

    // Copy data from device to host
    cudaMemcpy(h_a, d_a, N * sizeof(int), cudaMemcpyDeviceToHost);

    // Display results
    for (int i = 0; i < N; i++) {
        printf("%d\n",h_a[i]);
    }

    // Free memory
    delete [] h_a;
    cudaFree(d_a);

    return 0;
}
```

✏️ CUDAプログラムによる計算（1ブロックのみ使用）

- 本プログラムでは同一ブロック内の16スレッドが，それぞれ1要素の書き込みを行う
- 同一ブロック内のスレッドは必ず同一マルチプロセッサ上で動作することに注意する

それでは CUDA による GPU プログラムを書きましょう．ファイル名を ex1-cuda.cu とします．CUDA プログラムでは拡張子を cu にすることが多いです．エディタで前ページのソースコードを入力してください．

ソースコード解説

- 3-9行目：デバイス(GPU)上で実行される関数
  - デバイス上で実行される関数のうち，ホスト(CPU)から呼ばれるものには `__global__` をつける
  - 参考：デバイス内で呼び出される関数には `__device__` をつける
  - ブロック内のスレッドには `0` から `blockDim.x - 1` (ブロック内のスレッド最大値 - 1)までのIDが割り当てられ，スレッドIDは `threadIdx.x` で参照できる．
- 16,17行目：デバイス上のグローバルメモリにメモリ確保するには `cudaMalloc` 関数を使用する
- 19,20行目：通常，GPUでの処理の前にホストのデータをデバイスに転送する処理を行うが，今回は不要
- 25行目，GPU関数（カーネル関数）を呼び出すには，関数名の後に三重ブラケットをつける． `<<<blocks, threadsPerBlock>>>` と書いた時，ブロックの数が `blocks` になり，1ブロックあたりのスレッド数が `threadsPerBlock` になる．今回は1ブロック16スレッドにする．
- 27,28行目：GPU処理が完了したら，デバイスのデータをホストに転送する．そのために `cudaMemcpy` 関数を使用する．
  - 第二引数から第一引数へデータ転送される
  - 参考：ホストからデバイスにデータ転送するには `cudaMemcpyDeviceToHost` を `cudaMemcpyHostToDevice` にする
- 37行目：デバイス上のメモリを解放するためには， `cudaFree` 関数を用いる

ソースコードを解説します．まず，3 行目ですが，関数の戻り値指定の前に「__global__」がついています．これは CPU（ホスト）から GPU（デバイス）に対して呼び出される関数，つまりカーネル関数であることを示しています．ちなみに GPU（デバイス）の内部で使用される関数の場合は，関数の戻り値指定の前に「__device__」をつけます．GPU 内の各スレッドがこの関数を個別に処理します．スレッドには 0 始まりの ID がついていて，今回はスレッド 0 からスレッド 15 までの 16 個のスレッドが作られます．これの指定方法についてはすぐ後に説明します．このスレッド ID は threadIdx.x で参照できます．本関数ではスレッド i（i は 0 から 15）が，個別に $a[i]$ に i を書き込みます．

次に main 関数を見てみます．こちらは CPU で実行される処理です．まず，16, 17 行目について，デバイスのグローバルメモリにメモリ領域を確保する処理が「cudaMalloc」です．C 言語のメモリ確保命令である malloc と違い，GPU デバイスの中にメモリを確保します．同様に，37 行目の cudaFree は C 言語の free と違い，GPU デバイス中のメモリを解放します．

初日に GPU プログラムの基本的なステップは，(1) 入力データの CPU から GPU へのコピー，(2) GPU 上での関数実行，(3) 出力データの GPU から CPU へのコピー の 3 つであると述べました．今回は (1) はありません．25 行目が (2) に相当します．GPU に関数を実行させるには，スレッドの数とブロック数を指定する必要があります．関数名の後に三重ブラケットを使って<<<blocks, threadsPerBlock>>> と書くと，ブロックの数が blocks になり，1 ブロックあたりのスレッド数が threadsPerBlock になります．28 行目は (3) の GPU から CPU へのデータコピーです．cudaMemcpy 関数を使うことで，CPU-GPU 間でデータをコピーすることができます．

コンパイルとジョブ投入

- コンパイル(nvcc)により実行ファイルa.outが生成される

```
$ nvcc ex1-cuda.cu
$ qsub cuda.nqs
```

それでは，このソースコードをコンパイルして実行しましょう．ジョブスクリプトは「並列化なし」の時に作成した cuda.nqs を使用します．CUDA C++で書かれたコードをコンパイルするコマンドは nvcc です．コマンドオプションについては g++と同様のものを持っていますが，それ以外にも GPU 向け専用のものもあります．後でいくつか出てきます．ジョブ投入コマンド qsub については並列化なしの時と同様です．ジョブを投入したら qstat, sstat コマンドで状態を確認しましょう．

プログラムの性能の考察

- 全スレッドがグローバルメモリにコアレスアクセスしている（連続する要素にアクセスしている）点については効率が良いと言える
- スレッド数が少ないため，ハードウェアマルチスレッディングは活用できない（もっと要素数が多くないと活用できない）

補足

- ユニファイドメモリを使うとデバイス-ホスト間の手動でのデータ転送処理が不要になるが，今回は扱わない
- カーネル関数でブロック数，スレッド数を指定したが，それぞれ代わりに3次元座標で指定することもできる（例えば，1000と指定する代わりに(10, 10, 10)（$10 \times 10 \times 10 = 1000$である）と指定できる）．今回は省略する．

プログラムの性能について考察してみましょう．初日に GPU アーキテクチャの特性について話しましたが，アーキテクチャの特性を活かしたプログラムになっているでしょうか．7 行目の配列 a へのアクセスに関して，この配列は cudaMalloc によって確保された d_a であるため，グローバルメモリ中に存在します．そして，CUDA においては，スレッドは 32 スレッドずつまとめられて 1 つのワープを形成しますが，スレッド 0 から順に 32 スレッドずつワープが作られます．よって，このプログラムで作成されるスレッド 0 から 15 は同じワープに属します．同一ワープに属するスレッドが $a[0]$ から $a[15]$ という連続領域にアクセスするので，このアクセスはコアレスアクセスであることがわかります．

また，今回はマルチプロセッサに割り当てられるワープは 1 つだけなので，ハードウェアマルチスレッディングは活用できません．もう少し要素数が多い場合でないと，活用することはできません．

最後にこのプログラムについて，いくつか補足しておきます．CUDA では CPU-GPU 間のデータ転送を自動化する仕組みとしてユニファイドメモリという機能を提供しています．しかし，この講習会では GPU の内部処理について理解したいため，この機能は使用しません．

ソースコード 25 行目のカーネル関数の呼び出しに関して，今回のプログラムではブロック数やブロックごとのスレッド数を 1 つの数値で指定しましたが，代わりに 3 次元座標形式で指定することもできます．例えば，1000 と指定する代わりに (10,10,10) と指定することができます．こうするとブロックやスレッドは x 方向に 10 個，y 方向に 10 個，z 方向に 10 個の 3 次元配列として生成されるため，合計は 1000 個となります．この方式は空間のシミュレーションを行う場合などに便利です．

2.4.3 CUDA による GPU プログラム（その 2：2 ブロック使用）

✏️ CUDAプログラムによる計算（複数ブロック使用）

- 本プログラムでは8スレッドからなるブロック2つを用いて計算を行う
  - 2つのブロックは同一マルチプロセッサ上で動作することもあれば，そうでないこともある（デバイスが自動でスケジューリングする）
  - スレッドIDはブロック毎に別々に割り当てられるということに注意する
- 計算時間を計測する処理も追加する

CUDA の文法を学ぶため，いま作成したプログラムを少し変更してみます．まず，ブロック数を 2 つにしてブロックあたりのスレッド数を 8 にします．これは勉強のためだけにやっていることに注意してください．こんなことをしてもまったく効率的になりません．また，各ブロックがどのマルチプロセッサで実行されるかについてはデバイスが自動でスケジューリングするため，ユーザは指定できないことにも注意しましょう．複数のブロックが同一のマルチプロセッサ上で動作することもあれば，そうでないこともあります．スレッド ID はブロックごとに個別に割り当てられますので，そのことにも注意しましょう．

ブロック数の変更と同時に，計算時間を計測する処理もプログラムに追加してみます．

それでは，ソースコードを作成しましょう．レイアウトの関係上 2 つに分割しています．

ソースコード（ex1-cuda2.cu）（2分割のうち1つ目）

```
#include <stdio.h>

__global__ void WriteIndex(int* a, int n)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) {
        a[i] = i;
    }
}

int main()
{
    // Init
    int N = 16;
    int* h_a = new int[N]; // on host (CPU)
    int* d_a;              // on device (GPU)
    cudaMalloc(&d_a, N * sizeof(int));

    // Measure the elapsed time
    cudaEvent_t start, stop;
    float time_ms;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);

    // Copy data from host to device
    // Do nothing
```

ソースコード（ex1-cuda2.cu）（2分割のうち2つ目）

```
    // Main
    int threadsPerBlock = 8;
    int blocks = (N + threadsPerBlock - 1) / threadsPerBlock; // round up
    WriteIndex<<<blocks, threadsPerBlock>>>(d_a, N);

    // Copy data from device to host
    cudaMemcpy(h_a, d_a, N * sizeof(int), cudaMemcpyDeviceToHost);

    // elaspsed time
    cudaEventRecord(stop, 0);
    // Wait until cudaEventRecord is completed.
    // (Note: cudaEventRecord is an asynchronous function)
    cudaEventSynchronize(stop);

    // Display results
    for (int i = 0; i < N; i++) {
        printf("%d\n",h_a[i]);
    }
    cudaEventElapsedTime(&time_ms, start, stop);
    printf("exe time: %f ms\n", time_ms);

    // Free memory
    delete [] h_a;
    cudaFree(d_a);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    return 0;
}
```

ソースコード解説

- 1つ目のファイルの5行目：全ブロックのスレッドに一意の通し番号を振る処理.
  - `blockIdx.x` はブロックIDで，各ブロックに対し0から「ブロック数-1」までの番号が振られる.
  - `blockDim.x` はブロックあたりのスレッド数，`threadIdx.x` はブロック内でのスレッド番号.
- 2つ目のファイルの3,4行目：ブロック数とブロックあたりのスレッド数を決める処理.
  - 先に「ブロックあたりのスレッド数」を決めてから，その後必要なブロック数を計算している.
  - 「配列サイズ/1ブロックあたりのスレッド数」（今回は$16/8 = 2$）がブロック数になるが，この値が整数にならない時は切り上げる（そのため式が複雑になっている）.
- 計算時間計測処理
  - 1つ目のファイルの19-24行目：計測用のイベント作成とstartイベントの時刻記録
  - 2つ目のファイルの10-14行目：stopイベントの時刻記録
  - 2つ目のファイルの20,21行目：startイベントからstopイベントの経過時間算出．時間はミリ秒の単位で得られ，0.5マイクロ秒の精度を持つ（ベストプラクティスガイド）.
  - 2つ目のファイルの26,27行目：イベント削除

ソースファイルの解説をします.

まず，1 つ目のファイルの 5 行目は全ブロックのスレッドに一意の通し番号を振る処理です．スレッド ID はブロック毎に個別に割り当てられる ID ですが，全スレッドの通し番号があったほうが便利なことが多いため，これは定番の処理です.「blockIdx.x」はブロック ID で，各ブロックに対し 0 から「ブロック数-1」までの番号が振られます．また「blockDim.x」はブロックあたりのスレッド数,「threadIdx.x」はブロック内でのスレッド番号です.

次に，2 つ目のファイルの 3,4 行目はブロック数とブロックあたりのスレッド数を決める処理です．ここでは「ブロックあたりのスレッド数」を先に決めてから，その後に必要なブロック数を決めています．ブロックあたりのスレッド数が 8 の時，必要なブロック数は明らかに $16/8 = 2$ なのですが，ここでは少し複雑な書き方をしています．基本的には「配列サイズ/1 ブロックあたりのスレッド数」がブロック数になりますが，この値が整数にならない時は整数値に切り上げる必要があります．今回は切り上げ処理は不要ですが，定番処理なのであえて切り上げ処理を含んだ式を記載しています.

残りの変更箇所は計算時間計測に関連する処理です．計算時間を計測するには処理開始時刻と処理終了時刻の差分をとれば良いですが，CUDA では時刻を取得するための cudaEventRecord という API と時刻を保持するための cudaEvent_t という構造体が用意されています．まず，1 つ目のファイルの 19-24 行目では計測用のイベントを作成し start イベントの時刻記録を行なっています．2 つ目のファイルの 10-14 行目では stop イベントの時刻記録を行なっています．注意事項として，時刻を記録する API である cudaEventRecord は非同期関数です．つまり，stop イベントへの時刻の書き込みが終わっていなくてもリターンします．そこで，cudaEventSynchronize という API を呼ぶことで書き込みが完了するまで待つ必要があります．2 つ目のファイルの 20,21 行目は start イベントから stop イベントの経過時間を算出して表示する処理です．時間はミリ秒の単位で得られます．CUDA のベストプラクティスガイドによれば，精度は 0.5 マイクロ秒とのことです．最後に 2 つ目のファイルの 26,27 行目はイベントの削除処理です.

コンパイルとジョブ投入

- コンパイル(nvcc)により実行ファイルa.outが生成される
- 最適化O3オプションとアーキテクチャを指定することで効率的なコードを生成する
  - アーキテクチャ指定についてはdeviceQueryの結果およびnvccのドキュメントを参照のこと
    https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#options-for-steering-gpu-code-generation

```
$ nvcc -O3 -arch=sm_60 ex1-cuda2.cu
$ qsub cuda.nqs
```

先ほど話した通りコンパイルには nvcc コマンドを使用しますが，今回はオプションをつけてみます．まず，O3 オプションをつけると最適化が行われます．これは g++ と同様です．次に arch オプションをつけると，アーキテクチャを考慮した最適化が行われます．今回しようしている P100 では sm_60 を指定します．この値については，サンプルファイルの deviceQuery の実行結果や nvcc のドキュメントを参照してください．

ジョブの投げ方はこれまでと同様です．

プロファイリング（パフォーマンスの解析）

- CUDAプログラムのためのコマンドベースのプロファイリングツールとしてnvprofがある
- 詳細はNVIDIAのProfiler User's Guideを参照のこと
  https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview
  - オプションを指定することでより詳細な情報を取得できる
- GUIが使える時は総合開発環境であるNsightを使ってプロファイリングするのが便利
- nvprofを利用するためにはジョブスクリプト(prof.nqs)を以下のようにする

```
#!/bin/bash
#PBS -q LECTURE
#PBS -l elapstim_req=00:00:10,cpunum_job=1,gpunum_job=1
cd $PBS_O_WORKDIR
nvprof ./a.out
```

この実習の最後に，プロファイリング，すなわちパフォーマンス等の詳細解析のやり方について話しておきたいと思います．計算時間が想定より遅い場合に，プロファイリングをすることにより，どの処理が遅いのかを解析することができます．コマンドベースのプロファイリングツールとして，nvprof があります．詳細は NVIDIA の Profiler User's Guide をご覧ください．オプションを指定することで様々な情報を取得することができます．一部については今日の後半でやります．

なお，GUI が使える場合には，Nsight という総合開発環境を使ってプロファイリングするのが便利ですが，今日は端末上で実習しているため，nvprof を使います．

nvprof を使うには，nvprof コマンドの後に実行ファイル名を指定します．本演習環境では，ジョブスクリプトをこのように書き換えて，prof.nqs という名前で保存します．

- ジョブの投入

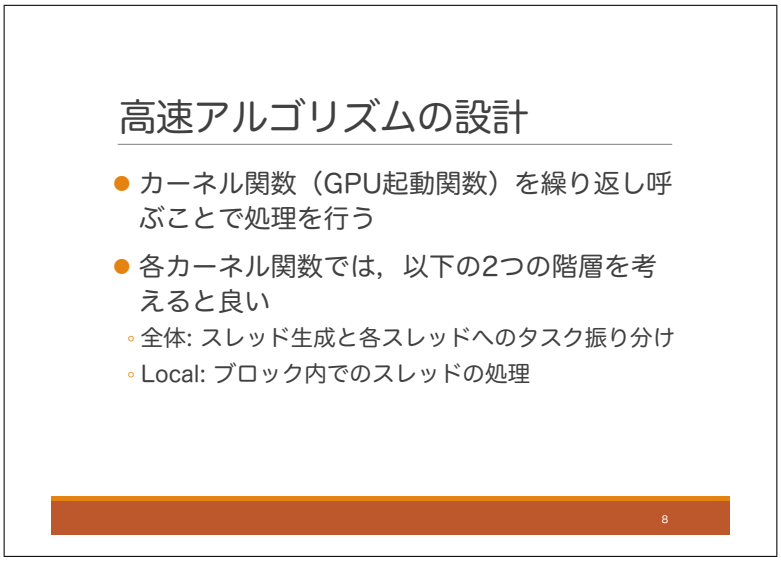
```
$ qsub prof.nqs
```

- 結果は標準エラー出力に出力される（以下は具体例）

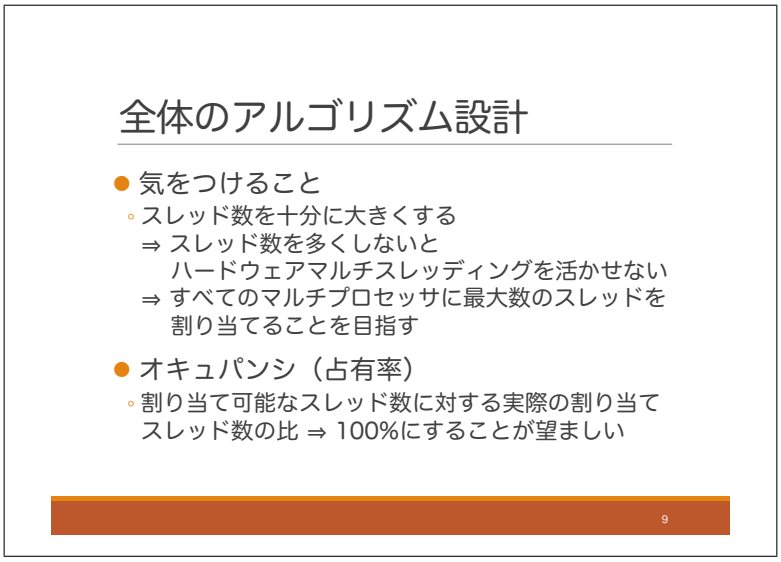
```
-bash-4.2$ cat prof.nqs.e389728
==92462== NVPROF is profiling process 92462, command: ./a.out
==92462== Profiling application: ./a.out
==92462== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   60.75%  2.0800us         1  2.0800us  2.0800us  2.0800us  WriteIndex(int*, int)
                   39.25%  1.3440us         1  1.3440us  1.3440us  1.3440us  [CUDA memcpy DtoH]
      API calls:   99.75%  520.14ms         1  520.14ms  520.14ms  520.14ms  cudaMalloc
                    0.13%  682.87us        94  7.2640us     114ns  282.80us  cuDeviceGetAttribute
                    0.04%  214.33us         1  214.33us  214.33us  214.33us  cudaFree
                    0.04%  209.93us         1  209.93us  209.93us  209.93us  cuDeviceTotalMem
                    0.01%  56.766us         1  56.766us  56.766us  56.766us  cuDeviceGetName
                    0.01%  45.203us         1  45.203us  45.203us  45.203us  cudaLaunch
                    0.01%  30.483us         1  30.483us  30.483us  30.483us  cudaMemcpy
                    0.00%  11.443us         2  5.7210us  2.2800us  9.1630us  cudaEventRecord
                    0.00%  8.3070us         2  4.1530us     682ns  7.6250us  cudaEventCreate
                    0.00%  6.3850us         1  6.3850us  6.3850us  6.3850us  cudaEventSynchronize
                    0.00%  6.1000us         2  3.0500us     193ns  5.9070us  cudaSetupArgument
                    0.00%  5.0610us         1  5.0610us  5.0610us  5.0610us  cudaEventElapsedTime
                    0.00%  3.0570us         3  1.0190us     185ns  2.6390us  cuDeviceGetCount
                    0.00%  1.9980us         2     999ns     517ns  1.4810us  cudaEventDestroy
                    0.00%  1.2720us         1  1.2720us  1.2720us  1.2720us  cudaConfigureCall
                    0.00%     595ns         2     297ns     146ns     449ns  cuDeviceGet
```

ジョブを投入すると，処理完了後，標準エラー出力にプロファイリング結果が出力されます．これはその一例です．

2.5　高速アルゴリズムの設計

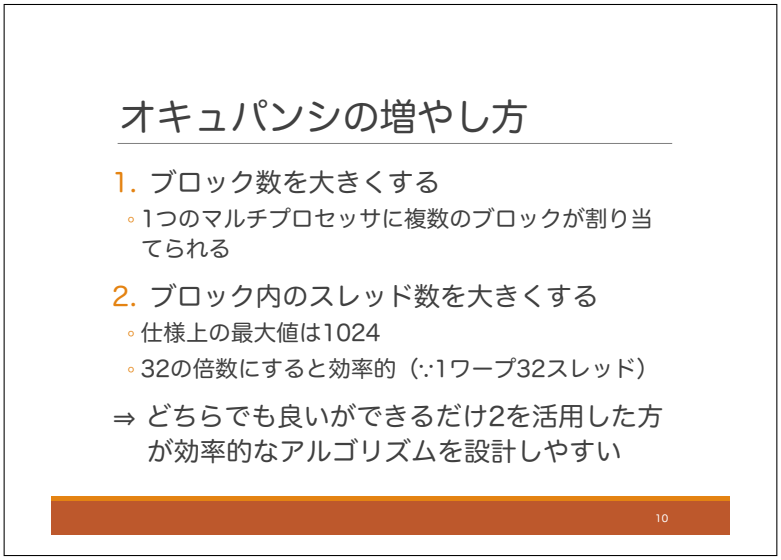


ここからは高速アルゴリズムを設計する方法について解説します．まず，繰り返しになりますが，GPU プログラムは，CPU から GPU に対し繰り返しカーネル関数を呼び出すことで処理を行います．各カーネル関数を設計する際は，2 つの階層を考慮すると良いと思います．一つはスレッドをいくつ生成し，タスク全体をどのようにスレッドに割り振るかという点で，もう一つはブロック内のスレッドがどのように効率よく処理を行うかという点です．ここでは一つ目の階層を全体，二つ目の階層をローカルと呼ぶことにします．これらは独立して考えられるものではないですが，ここでは一つずつ見ていくことにします．



まず，アルゴリズムの全体設計において考慮すべき点について説明します．一つ目は，スレッド数を十分に大きくするということです．スレッド数をコア数より大きくする必要があるということはすぐに分かると思いますが，初日に話した通り，GPU のハードウェアマルチスレッディングを活用するためにスレッド数をコア数よりも大幅に大きくする必要があります．マルチプロセッサには同時に割り当て可能なスレッド数の最大値が決まっていますが，各マルチプロセッサに最大数のスレッドを割り当てられるようにする必要があります．

初日の復習になりますが，GPU デバイスへの最大割り当て可能スレッド数に対する実際の割り当てスレッド数をオキュパンシと呼びます．GPU デバイスの特性を活用するためには，オキュパンシが 100%になることが望ましいです．

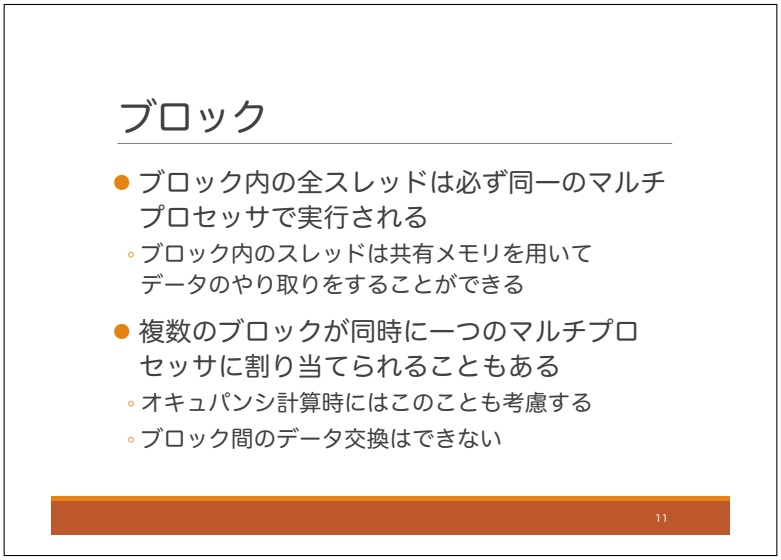


それではオキュパンシを大きくするための方法を見ていきます．単にスレッド数を多くすれば，オキュパンシが大きくなるという訳ではありません．それ以外にもメモリ使用量を考慮する必要があります．ですが，まずはスレッド数を大きくする方法についてみていきます．

総スレッド数を大きくするには，ブロック数を大きくする方法とブロックごとのスレッド数を大きくする方法が考えられます．まず一つ目ですが，一つのマルチプロセッサに複数のブロックを同時割り当て可能ですので，ブロック数を大きくすることはスレッド数を増やすのに有効です．ただし，1 マルチプロセッサに同時割り当て可能なブロック数の上限がアーキテクチャごとに決まっていますので注意が必要です．今回使用する P100 では 32 個が上限となっています．他のアーキテクチャについては CUDA C Programming Guide の Compute Capability の章に記載されています．

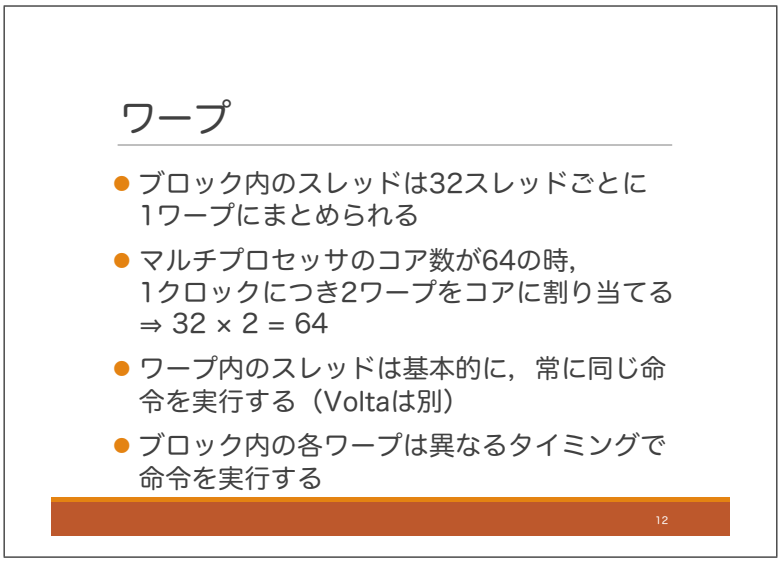
スレッド数を増やす二つ目の方法はブロック内のスレッド数を大きくすることです．ただし，ブロック内のスレッド数についても上限が決まっており，今の所はすべてのアーキテクチャで 1024 です．なお，1 ワープあたりのスレッド数は 32 なので，ブロック内のスレッド数を 32 の倍数にしておくと効率の良いアルゴリズムを設計しやすいです．

これら 2 つのどちらの方法でもスレッド数を増やせるのですが，出来るだけ後者を活用した方がアルゴリズムを設計しやすいことが多いです．これを説明するためにブロックについて復習と補足説明をします．

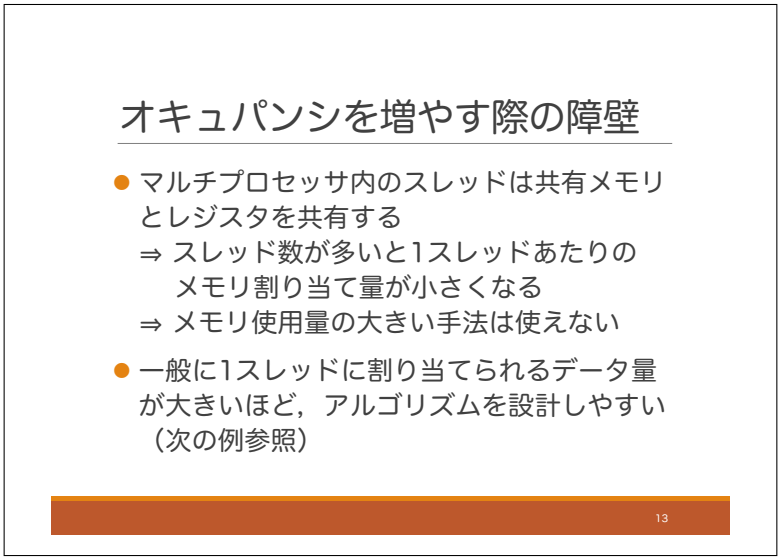


ブロック内の全スレッドは必ず同一のマルチプロセッサを用いて実行されます．そして，同じブロック内のスレッドとは共有メモリなどを介してデータのやり取りができます．一方で，異なるブロックのスレッドとはデータのやり取りができません．なので，出来るだけ同じブロックに多くのスレッドがいた方がアルゴリズムを設計しやすくなります．

また，複数のブロックが同時に 1 つのマルチプロセッサに割り当てられることもあります．ただしその場合であってもブロック間のデータやり取りはできません．ちなみに P100 では 1 マルチプロセッサに同時割り当て可能な最大スレッド数は 2048 です．1 ブロックあたりの最大スレッド数は 1024 なので，マルチプロセッサに最大数のスレッドを割り当てる時，少なくとも 2 つのブロックがマルチプロセッサに割り当てられていることになります．



ワープについても復習と補足説明をしておきましょう．初日に話した通り，ブロック内のスレッドは 32 スレッドごとにワープとしてまとめられます．そしてワープ内の全スレッドは基本的には常に同じ命令を実行するのでした．2017 年に発表された Volta アーキテクチャでは少し異なりますがここでは置いておきます．P100 ではマルチプロセッサあたりのコア数は 64 です．この時，各クロックにおいて 2 つのワープがマルチプロセッサに割り当てられます．どのワープが割り当てられるかはハードウェアにより計算されます．なので，ブロック内の各ワープは異なるタイミングで命令列を実行していくことになります．このタイミングを合わせる命令もあるのですが，それについては後で説明します．

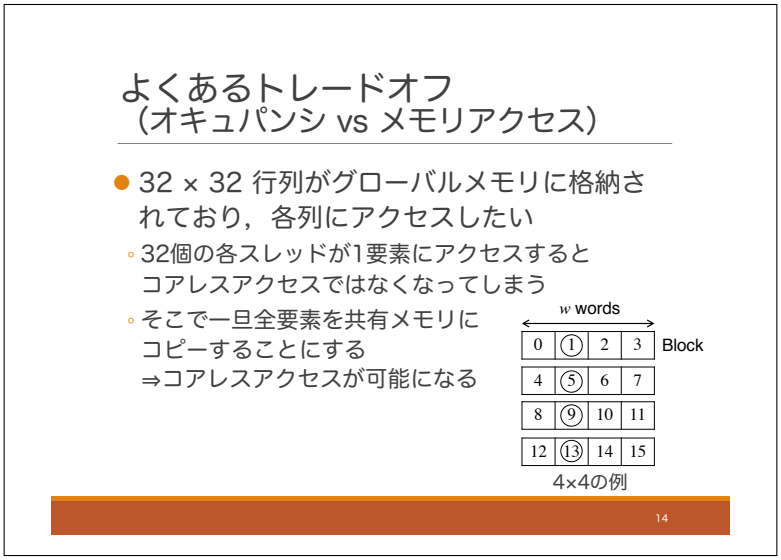


それでは本線に戻って，オキュパンシの増やし方について説明します．単にスレッド数を多くすればオキュパンシが大きくなるという訳ではありません．オキュパンシを大きくする際の障壁としてメモリ使用量があります．それについて説明します．

マルチプロセッサに複数のスレッドを割り当てる時，それらのスレッドはハードウェア資源を共有します．特に共有メモリについては，決められた容量をみんなで使うことになります．CUDA ではブロックごとに共有メモリを確保するのですが，マルチプロセッサにたくさんのブロックを割り当てようとすると，1 ブロックあたりで使えるメモリ量は少なくなります．逆にブロックのメモリ使用量が大きい場合，1 つのマルチプロセッサに割り当て可能なブロック数は小さくなります．

GPU ではレジスタもスレッドごとに個別に割り当てられます．もし複数のスレッドが同じレジスタを利用する場合，スレッドを切り替えるたびにレジスタ内容の切り替え処理が必要になり時間がかかりますが，GPU ではそのような処理が不要です．これはハードウェアマルチスレッディングでクロックごとに実行スレッドを高速切り替えする際に有利です．しかし，このような仕様によりマルチプロセッサ内のスレッド数を増やすと，1 スレッドあたりが使用できるレジスタ数は少なくなります．逆にスレッドのレジスタ使用量が多い場合，1 つのマルチプロセッサに割り当て可能なスレッド数は小さくなります．

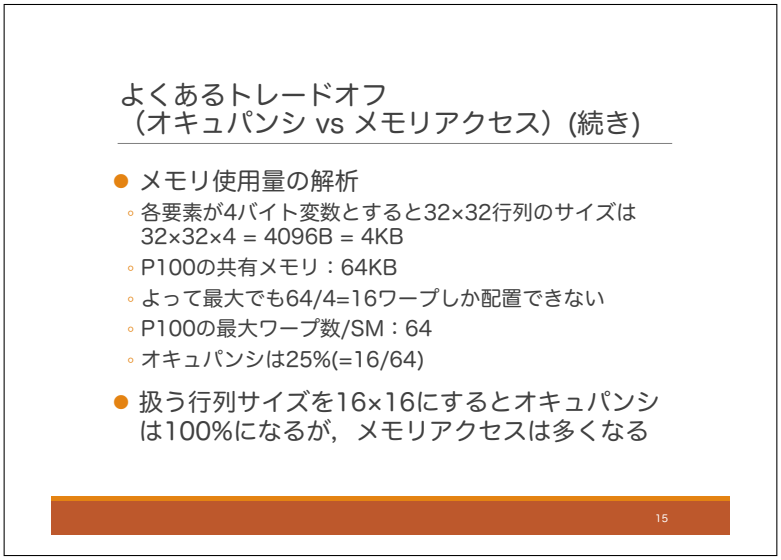
ということで，オキュパンシのことを考えると共有メモリやレジスタの使用量は十分に小さくする必要があります．しかし一般的にはメモリ使用量が多い方が効率的なアルゴリズムを設計しやすくなるので，これらはトレードオフの関係になっています．このトレードオフについて一つ例を挙げてみます．



よく遭遇するトレードオフのシチュエーションとして，行列計算があります．行列計算では，行列の各列にアクセスしたいことがよくあります．行列積を計算する場合がその典型例です．グローバルメモリに格納された行列の各列にアクセスすることを考えます．この例では 4 コアで 4×4 行列の各列にアクセスすることを考えます．つまり最初のアクセスで 4 つのコアは 0,4,8,12 にアクセスし，二回目のアクセスで 1,5,9,13 にアクセスします．以下同様です．この場合，各コアはグローバルメモリの不連続なアドレスにアクセスすることになりコアレスアクセスではなくなります．

この時，メモリ使用量を大きくすることで，コアレスアクセスを可能にする方法があります．それは，まず最初に行列の全データをグローバルメモリから共有メモリにコピーするという方法です．その際，1 回目で 0,1,2,3 をコピーし，2 回目で 4,5,6,7 をコピーするという感じで行ごとにデータをコピーすることで全てのデータコピーをコアレスアクセスにより行うことができます．1 回データを共有メモリにコピーしてしまえば，もはやコアレスアクセスについて気にする必要はありません．ここでは，行列の各列にアクセスしたいわけですので，コピーした全てのデータが使われることになり，無駄がありません．また，共有メモリのバンクコンフリクトについて気になる人がいるかもしれませんが，うまく回避する方法があります．これは後でやります．

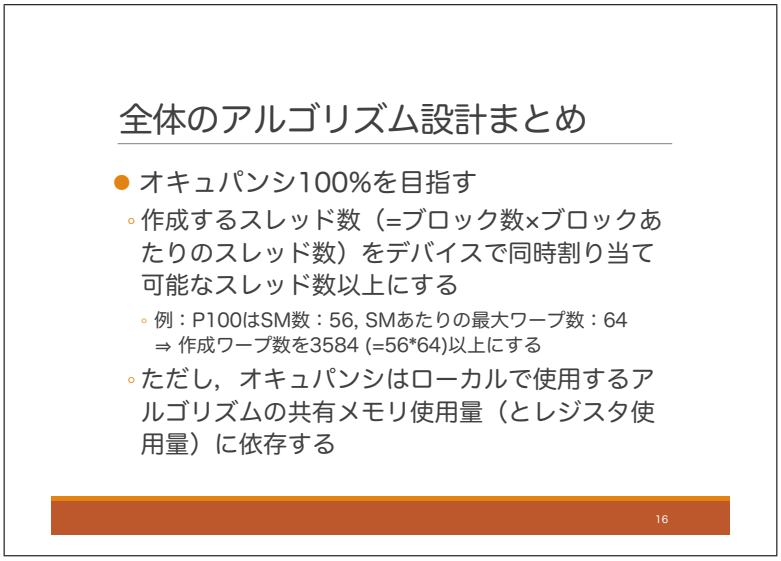
ということで，この方法はすごくいいような気がするのですが，問題はメモリ使用量が大きいということです．これによりオキュパンシが小さくなってしまいます．



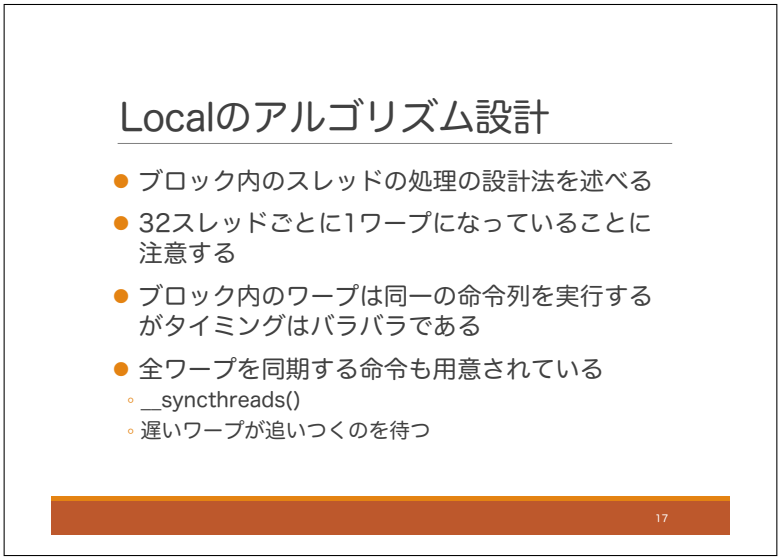
今回の実習で使用する P100 を例として，具体的な値を計算してみます．グローバルメモリ内に 4 バイト変数を要素とする 32×32 行列があり，ワープ内のスレッドが各列にアクセスするものとします．この時，行列の全要素を共有メモリにコピーすることを考えると，そのサイズは $32 \times 32 \times 4\mathrm{B} = 4\mathrm{KB}$ になります．一方で P100 マルチプロセッサの共有メモリ容量は 64KB です．1 つのワープが一つの行列を扱うとすると，1 マルチプロセッサに配置可能なワープ数は $64/4 = 16$ となります．P100 では 1 マルチプロセッサあたり 64 ワープを配置可能という仕様でした．最大 64 ワープに対して，16 ワープしか配置できないので，オキュパンシは 16/64 で 25%となります．

ちなみにこの全データコピーのアイデアを活かしたまま，オキュパンシを大きくする方法があります．それは扱う行列サイズを 16×16 にすることです．すると行列サイズは 1KB になるので，オキュパンシは 100%となります．しかし，このようにするとメモリアクセス回数は大きくなります．

以上のように，行列の列アクセスではオキュパンシとメモリアクセス回数のトレードオフが発生します．

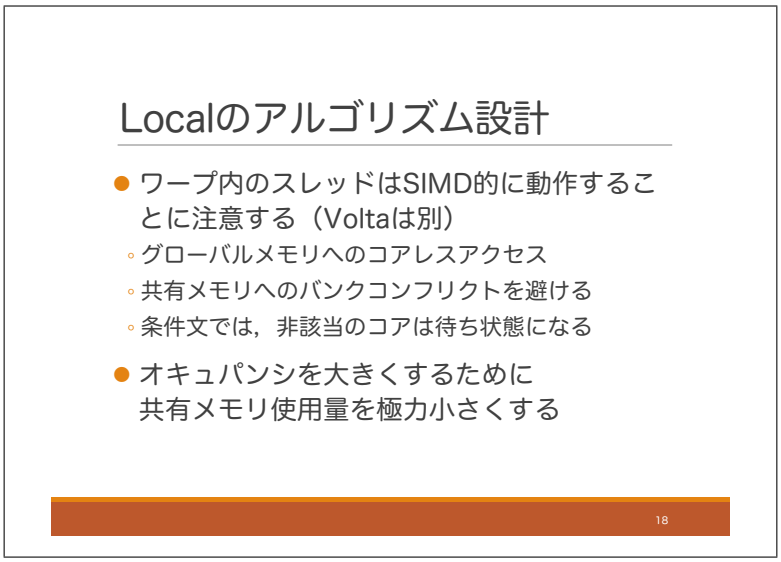


全体のアルゴリズム設計についてまとめておきます．目指すのはオキュパンシを 100%にすることです．そのためには作成するスレッド数を大きくすることと，使用するメモリ，レジスタ量が小さくなるアルゴリズムを設計することが必要です．



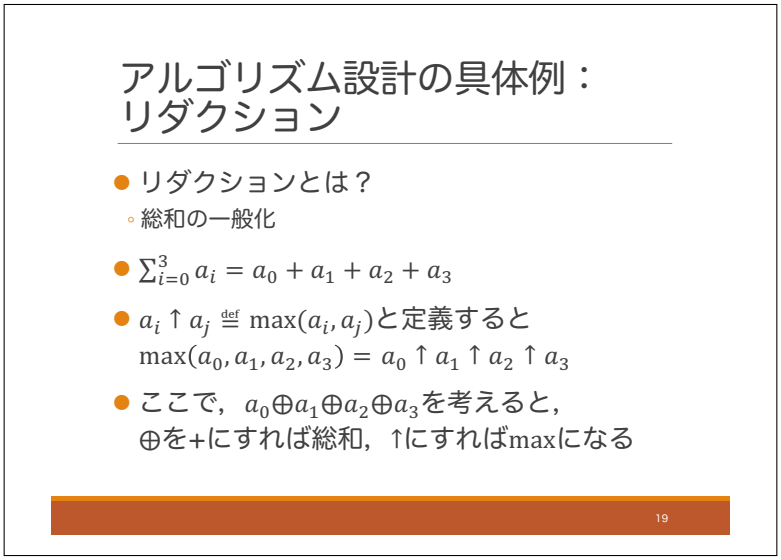
続いてローカルのアルゴリズム設計について説明します．つまりブロックの中の各スレッドに対するアルゴリズムの設計ということです．

まず繰り返しになりますが，ブロックの中のスレッドは 32 ワープずつ 1 つのワープを構成していることに注意します．また，ブロック内のすべてのワープが同じ命令列を実行しますが，ワープにより実行されるタイミングはバラバラです．あるワープは非常に速く計算が進み，別のワープは非常に遅く計算が進むかもしれません．すべてのワープを強制的に同じタイミングにするコマンドとして__syncthreads 関数があります．これは，この関数が書かれた地点で一番遅いワープが追いつくのを待つという命令です．共有メモリを用いてワープ間でデータのやり取りをしたい場合などに便利です．



次に，ワープ内のスレッドは SIMD 的に動作することに注意が必要です．Volta アーキテクチャはちょっと異なりますがここでは置いておきましょう．このことを活用するとグローバルメモリにコアレスアクセスできたり，共有メモリのバンクコンフリクトを避けたりすることができます．また条件文において，条件に非該当のスレッドは待ち状態になります．そのため条件文を大量に使うとコアの稼働率を下げることになるので注意が必要です．

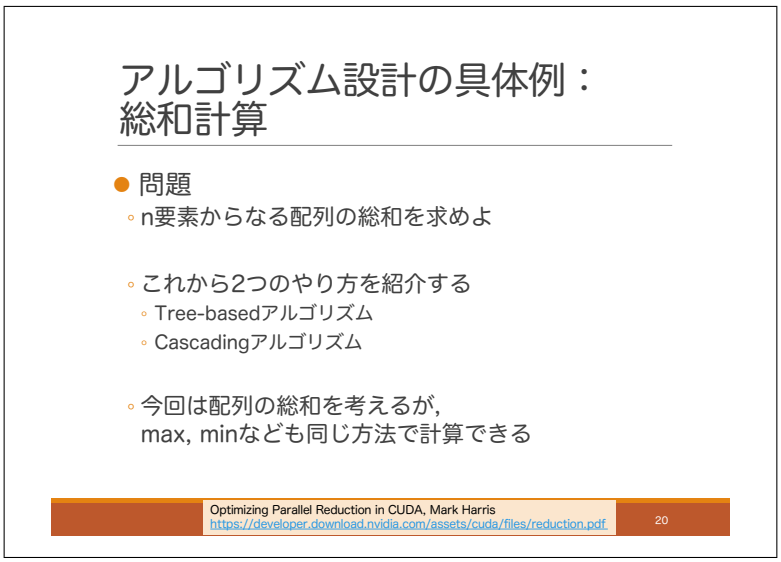
また，全体アルゴリズムでも述べましたが，オキュパンシを大きくするために共有メモリ使用量とレジスタの使用量を極力小さくする必要があります．



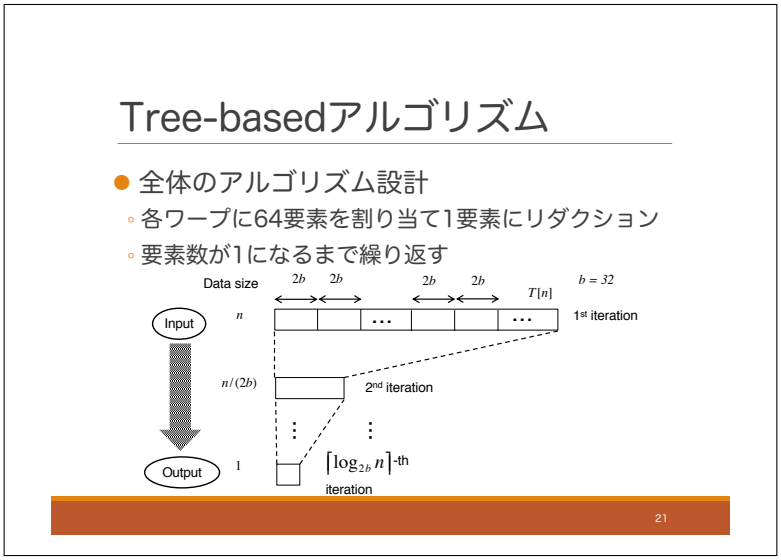
アルゴリズム設計をする時に全般的に考慮しないといけないことは，これまで話した通りです．個々の高速化テクニックについては後半で簡単に紹介しようと思います．ここからは，これまで話してきた考慮すべき項目についてどのようにアルゴリズム設計に適用するのかということについて説明したいと思います．実際にアルゴリズムを設計してみるのが良いと思いますので，ここでは具体例としてリダクションと呼ばれる操作のためのアルゴリズムを設計してみます．

リダクションというのは配列の総和を求める計算を一般化したものです．総和というのは配列の全ての要素に対し，加算という二項演算を繰り返し適用し，1 つの値を返す処理ですが，リダクションとは二項演算子について加算だけでなく任意の演算子を取れるようにしたものです．やや細かい話をしておくと，演算子は結合則を満たすものとします．また，本日は可換である場合を扱います．

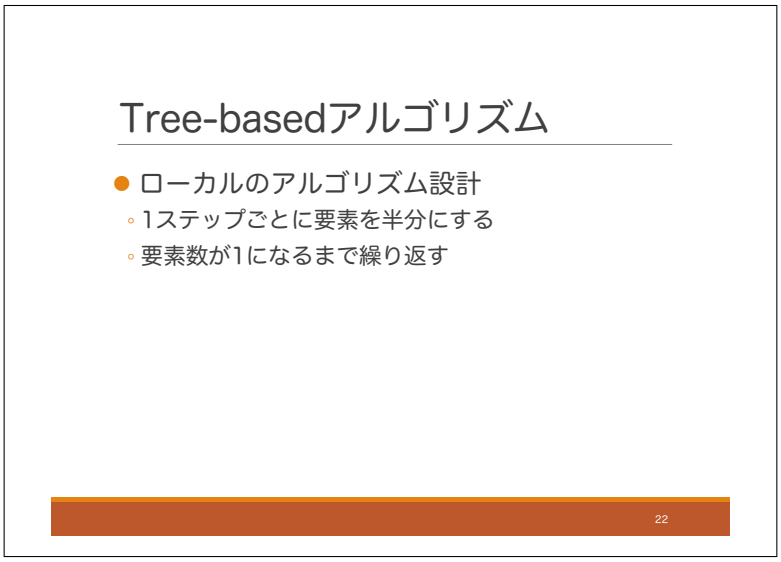
色々細かい話をしていますが，本日の内容に関しては，配列の総和を高速計算するアルゴリズムを設計する思っていただいて問題ありません．ここで言いたいのは，そのアルゴリズムが他の様々な計算にもそのまま使えますよということだけです．他の計算の例として，配列の最大値計算が挙げられます．今，2 要素のうち大きい方を返す演算を上矢印で書くことにすると，配列の要素の最大値は，配列に関してこの上矢印でリダクションをすることにより得られます．



ということで配列が与えられた際，その総和を求める問題を考えます．配列は GPU のグローバルメモリに格納されているものとします．この問題に対して，二つのアルゴリズムが知られています．Tree-based アルゴリズムと Cascading アルゴリズムです．後者の方が高速なのですが，これが良いアルゴリズムだということを理解するのが今日の目標です．



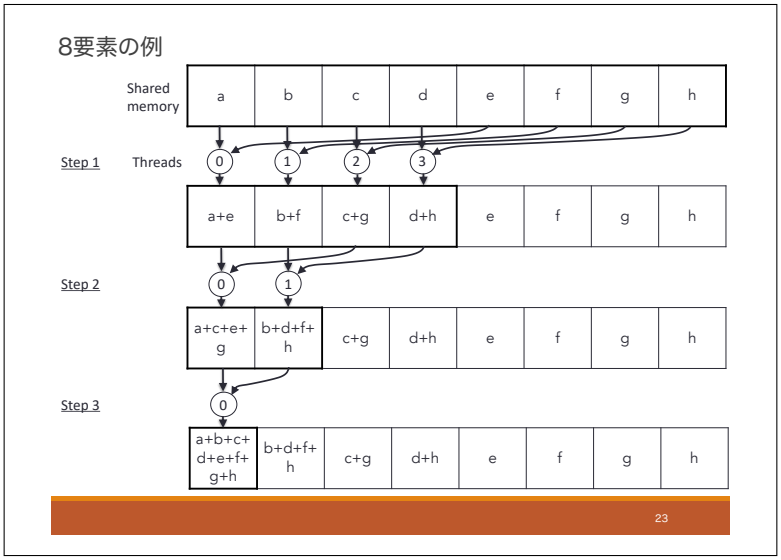
まず，Tree-based アルゴリズムを見てみましょう．まず全体のアルゴリズム設計についてです．このアルゴリズムでは，まず入力データを 64 要素からなる小ブロックに分割します．そして，1 つの小ブロックを一つの CUDA ブロックに割り当てることにします．各 CUDA ブロックでは 64 要素の和を計算して，グローバルメモリの所定の場所に出力します．入力データが n 個だとすると，この処理の終了後，データサイズはだいたい $n/64$ 個ぐらいになります．この処理をデータサイズが 1 になるまで繰り返すと最終的な答えを得ることができます．



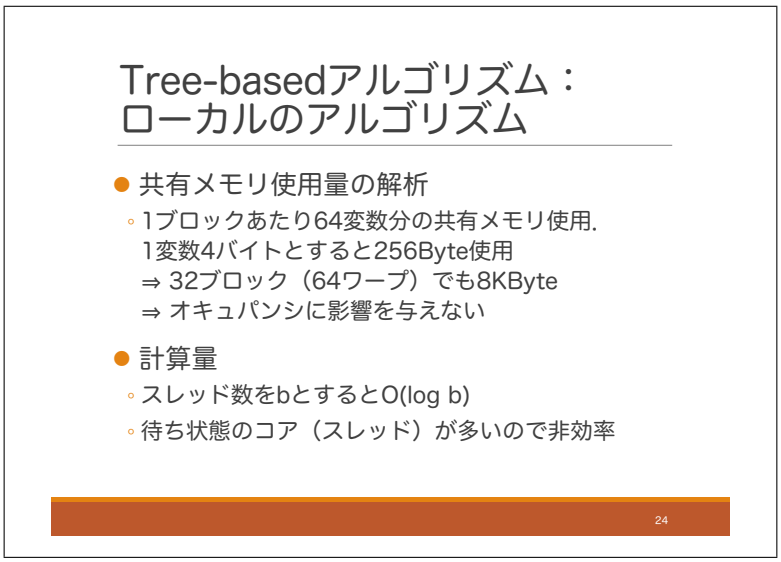
次にローカルのアルゴリズム設計について説明します．ここでは各ブロックは 64 要素の和を計算します．まずブロックの構成についてですが，各ブロックが 64 個のスレッドで構成されるようにします．32 スレッドごとに 1 つのワープとなることに注意してください．

まず，ブロック内の 64 スレッドは，入力データをグローバルメモリから共有メモリにコピーします．次に 32 スレッドの各スレッドが 2 要素ごとの和を求めて共有メモリに書き込むと，32 個の出力が得られます．次に 16 スレッドが 2 要素の和を求めると 16 個の和が得られ，これを繰り返すと 64 要素全ての和が求まります．

なお，ブロックあたりのスレッド数をもっと大きくすると，もう少し効率的なアルゴリズムができるのですが，ここでは深入りしないことにします．ブロックあたりのスレッド数を増やすと小ブロックのサイズを 64 より大きくできて，全体アルゴリズムにおける繰り返し処理の回数が少なくなります．



これは 8 要素の和を求める例です．四角が共有メモリで，丸がスレッドです．8 スレッドが 8 要素を共有メモリにコピーした後に，共有メモリの内容がどのように書き換わるかを時系列で示しています．各ステップでデータサイズが半分になる様子を確認してください．この例では，1 ステップ後に有効データ数が 4 になり，2 ステップ後に 2，3 ステップ後に 1 となっています．ローカルアルゴリズムにおける和計算のステップ数は，要素数の対数をとったものになります．データサイズが 64 要素の場合は $\log_2 64 = 6$ ステップです．



ローカルアルゴリズムのメモリ使用量や計算量を解析してみましょう．

まず，共有メモリの使用量ですが，1 ブロックあたり 64 変数分の共有メモリを使用します．もし 1 変数のサイズが 4 バイトなら，ブロック全体では，$4 \times 64 = 256$ バイトです．1 ブロックは 2 ワープで構成されており，1 マルチプロセッサあたりの最大ワープ割り当て数は 64 です．よって，最大で 32 ブロックをマルチプロセッサに割り当て可能です．もし 32 ブロックを割り当てたとしてもメモリ使用量は $32 \times 256\text{Byte} = 8\text{KB}$ です．P100 マルチプロセッサの共有メモリサイズは 64KB ですので，メモリサイズが原因でオキュパンシが低下することはなさそうです．

次にローカルアルゴリズムの計算量ですが，ブロックのスレッド数を b とすると，$\mathcal{O}(\log b)$ となります．このアルゴリズムの欠点として，コアの稼働率が低くなるということが挙げられます．ワープ内の 32 スレッドのそれぞれにコアが割り当てられますが，和計算の 2 ステップ以降は 16 個以下のスレッドしか使われません．この時，未使用スレッドに割り当てられたコアは待ち状態となります．Tree-based アルゴリズムでは，後半のステップで待ち状態のコアが多くなります．

Cascadingアルゴリズム

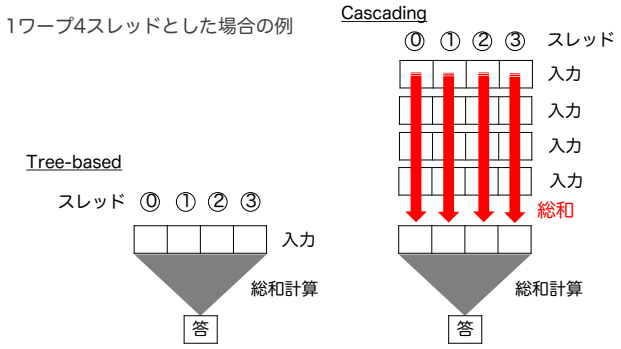
- Tree-basedアルゴリズムでは，
  ローカルリダクションが非効率だった
  ⇒ 待ち状態のコアが多くもったいない
- 各ワープへの割り当てデータ量を増やすこと
  で効率的にすることができる
  ⇒ Cascadingアルゴリズム

25

それでは次に Cascading アルゴリズムを見てみましょう．前のスライドで見たように，Tree-based アルゴリズムでは，ローカルリダクションが非効率的でした．待ち状態のコアが多いため，それらをもっと活用できれば，高速化できそうです．

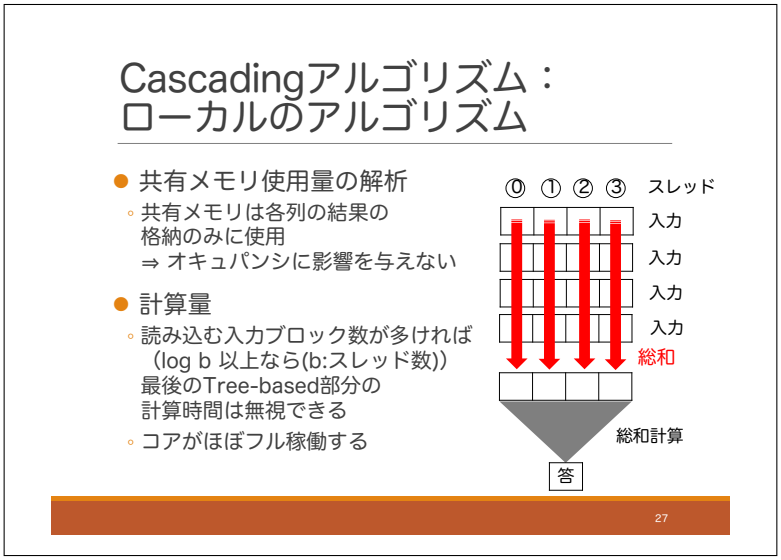
実は各ワープへの割り当てデータ量を増やすことで，効率的にすることができます．これを Cascading アルゴリズムと呼びます．

Cascadingアルゴリズム：ローカルのアルゴリズム



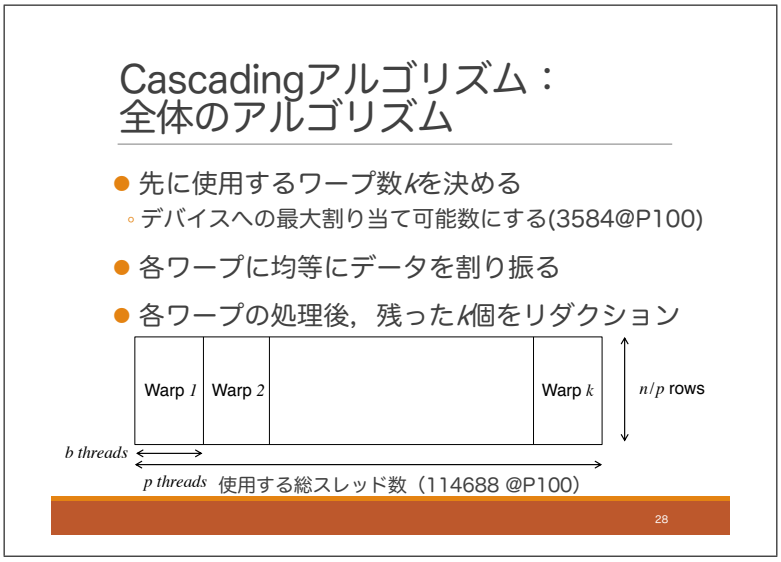
26

Cascading アルゴリズムの概要を Tree-based アルゴリズムと対比しながら説明します．ここでは，入力データは各ワープに分割して割り当てられるものとし，各ワープが割り当てデータの総和を求めるためのアルゴリズムを説明します．図を見やすくするため，図では 1 ワープ 4 スレッドとして描かれています．本当は 1 ワープは 32 スレッドです．Tree-based アルゴリズムでは，スライドの左側のように読み込んだ入力データをスレッドが協調しながら計算します．一方，Cascading アルゴリズムでは，その処理の前に各スレッドが個別に総和を計算するフェーズが入ります．ワープ内のスレッドはコアレスアクセスによりグローバルメモリから入力を複数回読み込み，各スレッドは決まった位置 (index) に対する総和を求めます．その後その結果を共有メモリに書き込み，後は Tree-based アルゴリズムと同様の総和計算を行います．



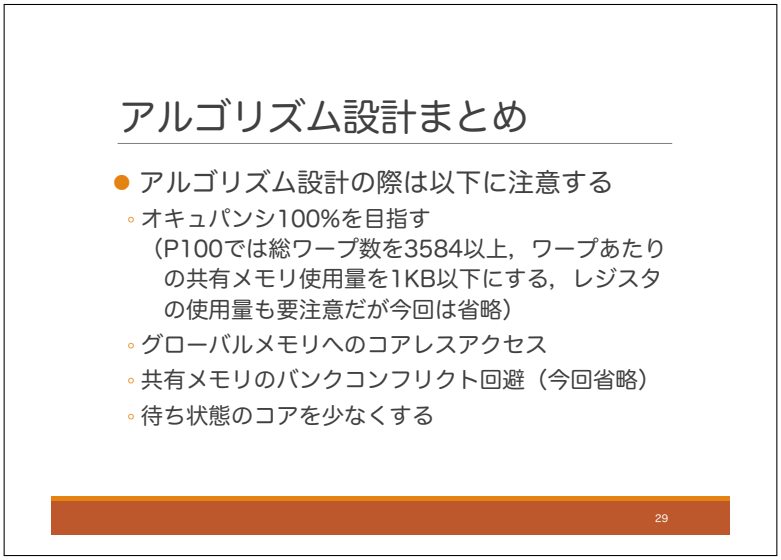
ローカルのアルゴリズムについて，メモリ使用量と計算量の解析をしてみましょう．まず共有メモリの使用量ですが，各入力ブロックについては，スレッドが有するレジスタに格納すればよく共有メモリを使う必要はありませんので，メモリ使用量は Tree-based アルゴリズムと同様になります．すなわち，メモリ使用量が原因でオキュパンシが低下することはありません．

次に計算量ですが，読み込む入力ブロック数が十分に多ければ，最後の Tree-based 部分の計算量は漸近解析としては無視できます．各ワープに十分に多い n_w 個の入力データが割り当てられる時，1 ワープのスレッド数を b とすると計算量は $\mathcal{O}(n_w/b)$ となります．このアルゴリズムはコアがほぼフル稼働するので，効率が良いと言えます．



次に全体のアルゴリズムについて，説明します．Cascading アルゴリズムでは，各ワープにできるだけたくさんの入力データを割り当てたほうが効率が良くなります．しかし，それを大きくしようとすると生成するワープの数が小さくなり，オキュパンシが低下してしまいます．そこで，オキュパンシを 100%に保てるためのワープ数を先に計算して，各ワープに均等に入力データを割り振ることにします．P100 では 56 個のマルチプロセッサが搭載されており，各マルチプロセッサでは最大 64 個のワープを同時割り当て可能です．よって，ワープを $56 \times 64 = 3584$ 個生成すればオキュパンシを 100%にできることがわかります．これはスレッド数に直すと $3584 \times 32 = 114688$ 個です．

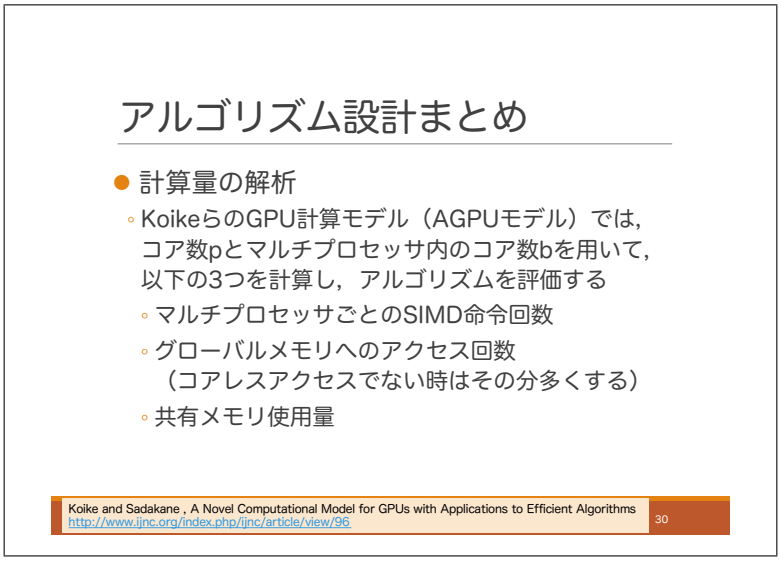
ということで，入力データを 3584 個に分割し，各ワープがそれらを並列に総和計算します．最後に 3584 個のデータを一つのワープを用いて総和計算します．ここもさらに高速化できますが，そもそもそれほど時間のかかる処理ではないため，ここでは扱いません．



最後に繰り返しになりますが，アルゴリズム設計時の注意点をまとめておきます．まず，オキュパンシを 100% にすることを目指します．具体的に言うと，P100 では総ワープ数を 3584 個以上にし，ワープあたりの共有メモリ使用量を 1KB 以下にすることが必要です．レジスタ使用量が多すぎる場合もオキュパンシが低下しますが，ここでは省略します．

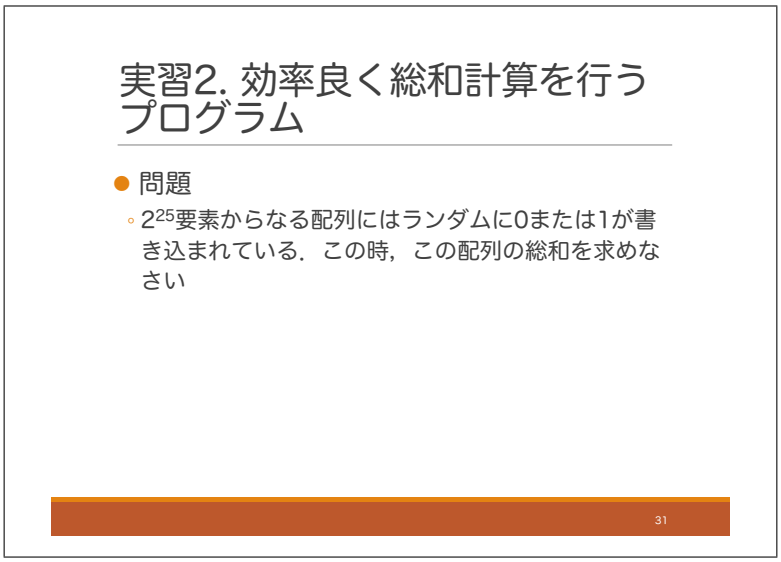
そしてメモリアクセスの仕方として，グローバルメモリへのコアレスアクセスと共有メモリへのバンクコンフリクト回避が有効です．後者についてはこの例では出てきませんでしたが，本日後半の高速化トピックの中でバンクコンフリクト回避のためのテクニックが少しだけ出てきます．

あとは，待ち状態のコアを少なくすることが重要です．マルチプロセッサ内のコアは SIMD 命令を実行しているという感覚を持ちながらアルゴリズムを設計すると良いと思います．



最後にやや宣伝になりますが，発表者らは GPU アルゴリズムを評価するための計算モデルを提案しています．ここでは，コア数 p とマルチプロセッサ内のコア数 b を用いて，マルチプロセッサごとの SIMD 命令回数，グローバルメモリへのアクセス回数，共有メモリ使用量を算出し，アルゴリズムを評価します．このモデルでは 1 マルチプロセッサでは 1 ワープのみが実行されると仮定しますが，オキュパンシについては共有メモリ使用量から別途評価します．

2.6　実習 2：効率良く総和計算を行うプログラム



これまで解説してきた総和のアルゴリズムを実際に動かしてみましょう．スライドに書いてある通り，2 の 25 乗個の要素からなる配列の総和を求めましょう．各要素は 0 または 1 です．これは桁あふれを考慮しなくていいようにするための仕様です．

2.6.1　Cascading アルゴリズムによる総和計算

> ✏️ **Cascadingアルゴリズムによる総和計算**
>
> - 各ブロックの処理の流れ
>   1. ワープ内のスレッドが個別に担当データの総和を求める
>   2. ワープ内の各スレッドが求めた結果に対する総和計算を行う
>   3. ブロック内の各ワープが求めた結果に対する総和計算を行う

各ブロックの処理の流れは以下の 3 つです．まず．ワープ内の各スレッドが個別に担当データの総和を計算します．これはこれまで説明したように，ワープ内のスレッドがコアレスアクセスにより 32 個のデータを取得したのち個々のスレッドが 1 要素分の和の計算を行うのでした．この処理がすべて終わったら次にワープ内の各スレッドが持っている結果について，総和計算を行います．これはこれまで話してきた通りです．最後に，これはこれまでの説明で話してないですが，ちょっとした工夫として，ブロック内の各ワープの結果について，さらに総和計算を行います．これで，あるブロックで担当したすべてのデータの総和が計算できます．

それではソースコードを見ていきましょう．長いので 4 分割して表示します．最初の 2 つがカーネル関数，つまり GPU で動作する関数で，残りの 2 つが CPU 側の処理です．

ソースコード（ex2.cu）（4分割のうち1つ目）

```
#include <stdio.h>
#include <math.h>
// #include <thrust/reduce.h>
// #include <thrust/device_ptr.h>

// Device parameter
const int NUM_SM = 56;
const int WARPS_PER_SM = 64;
const int WARPS_PER_BLOCK = 32;
const int NUM_BLOCKS = NUM_SM * WARPS_PER_SM / WARPS_PER_BLOCK;
const int THREADS_PER_WARP = 32;
const int THREADS_PER_BLOCK = WARPS_PER_BLOCK * THREADS_PER_WARP;

__global__ void ReductionCascading(
    int* inArray, unsigned int numElements, int* halfwayResult)
{
    // thread id in each block
    const unsigned int tid  = threadIdx.x;
    // warp id in each block
    const unsigned int wid  = tid / THREADS_PER_WARP;
    // lane id = thread id in each warp
    const unsigned int lid  = tid % THREADS_PER_WARP;
    // sequential thread id in the kernel
    const unsigned int sid  = blockIdx.x * blockDim.x + threadIdx.x;
    const unsigned int numTotalThreads = gridDim.x * blockDim.x;

    int item1 = 0;
    __shared__ int x[WARPS_PER_BLOCK * THREADS_PER_WARP];
    __shared__ int y[WARPS_PER_BLOCK];

    const int numIter = 1 + (numElements - 1) / numTotalThreads; //round up
    // Main
    for (int i = 0; i < numIter - 1; i++) {
        item1 += inArray[numTotalThreads * i + sid];
    }
    // Last
    int idx = numTotalThreads * (numIter - 1) + sid;
    if (idx < numElements) {
        item1 += inArray[idx];
    }
    x[tid] = item1;
    __syncwarp();
```

ソースコード（ex2.cu）（4分割のうち2つ目）

```
    // Reduction for a warp
    x[tid] += x[tid + 16];
    __syncwarp();
    x[tid] += x[tid + 8];
    __syncwarp();
    x[tid] += x[tid + 4];
    __syncwarp();
    x[tid] += x[tid + 2];
    __syncwarp();
    x[tid] += x[tid + 1];
    if( 0 == lid ){
        y[wid] = x[tid];
    }
    __syncthreads();

    // Reduction for a block
    if (tid < 32) {
        y[tid] += y[tid + 16];
        __syncwarp();
        y[tid] += y[tid + 8];
        __syncwarp();
        y[tid] += y[tid + 4];
        __syncwarp();
        y[tid] += y[tid + 2];
        __syncwarp();
        y[tid] += y[tid + 1];
    }
    if( 0 == tid ){
        halfwayResult[blockIdx.x] = y[tid];
    }

    return;
}

```

ソースコード（ex2.cu）（4分割のうち3つ目）

```
int main()
{
    int logNumElements = 25; // input_size = 2^25
    //int logNumElements = 10; // input_size = 2^25
    int numElements = 1 << logNumElements;
    int* h_input = new int[numElements]; // on host (CPU)
    int* h_halfwayResult = new int[NUM_BLOCKS];
    int  h_output = 0;
    int* d_input;                        // on device (GPU)
    int* d_halfwayResult;
    cudaMalloc(&d_input, numElements * sizeof(int));
    cudaMalloc(&d_halfwayResult, NUM_BLOCKS * sizeof(int));
    srand(0);
    for( unsigned int i = 0; i < numElements; ++i) {
        h_input[i] = (int)(rand() % 2);
    }
    int answer = 0;
    for (int i = 0; i < numElements; i++) {
        answer += h_input[i];
    }

    // Measure the elapsed time
    cudaEvent_t start, stop;
    float time_ms;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    // Copy data from host to device
    cudaMemcpy(d_input, h_input, numElements * sizeof(int),
        cudaMemcpyHostToDevice);

    // Main
    printf("numElements = %d (2^%d)\n", numElements, logNumElements);
    cudaEventRecord(start, 0);
    ReductionCascading<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(
        d_input, numElements, d_halfwayResult);

    // Copy data from device to host
    cudaMemcpy(h_halfwayResult, d_halfwayResult, NUM_BLOCKS * sizeof(int),
        cudaMemcpyDeviceToHost);
```

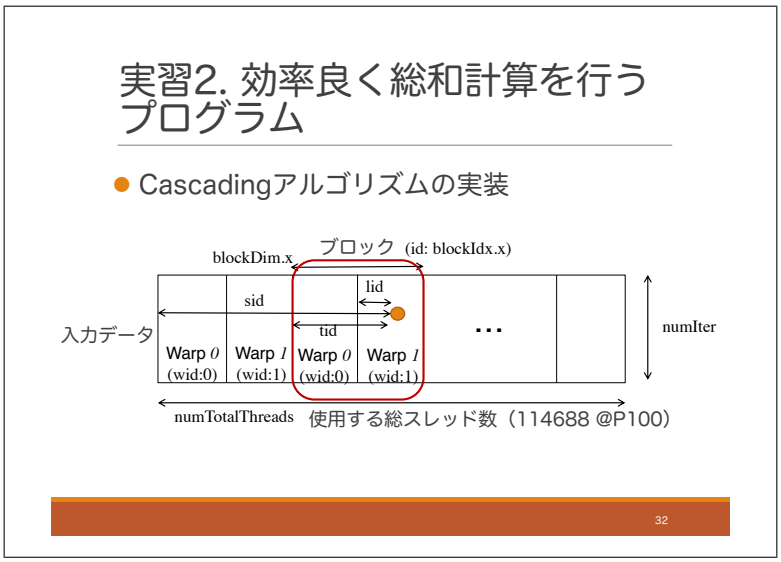
ソースコード（ex2.cu）（4分割のうち4つ目）

```
    h_output = 0;
    for (int i = 0; i < NUM_BLOCKS; i++) {
        h_output += h_halfwayResult[i];
    }

    // elaspsed time
    cudaEventRecord(stop, 0);
    // Wait until cudaEventRecord is completed.
    // (Note: cudaEventRecord is an asynchronous function)
    cudaEventSynchronize(stop);

    // Display results
    printf("Our result = %d, answer = %d\n", h_output, answer);
    cudaEventElapsedTime(&time_ms, start, stop);
    printf("  exe time: %f ms\n", time_ms);

    // Thrust Library
    // int h_outputThrust;
    // thrust::device_ptr<int> d_input_ptr = thrust::device_pointer_cast(d_inpu
    // cudaEventRecord(start, 0);
    // h_outputThrust = thrust::reduce(d_input_ptr, d_input_ptr + numElements);
    // cudaEventRecord(stop, 0);
    // // Wait until cudaEventRecord is completed.
    // // (Note: cudaEventRecord is an asynchronous function)
    // cudaEventSynchronize(stop);
    // printf("Thrust result = %d, answer = %d\n", h_outputThrust, answer);
    // cudaEventElapsedTime(&time_ms, start, stop);
    // printf("  exe time: %f ms\n", time_ms);

    // Free memory
    delete [] h_input;
    delete [] h_halfwayResult;
    cudaFree(d_input);
    cudaFree(d_halfwayResult);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    return 0;
}

```

ソースコード解説（4分割のうち1つ目）

- 6-10行目は各種パラメータの設定．内容は変数名の通り
- 14行目から2つ目のソースコードの最後まではカーネル関数
- 17行目から25行目は各種ID等の定義．詳細は後述する
- 27行目の変数 `item1` に各スレッドの総和の途中経過を保存する．このように書くと通常はレジスタに変数が確保される
- 28,29行目：共有メモリ上に変数を定義する． `__shared__` とつけると共有メモリ上にメモリが確保される． `x` はワープ内の各スレッドの総和計算結果を格納する配列． `y` は各ワープの総和計算結果を格納する配列．
- 31-42行目は各スレッドが担当データの総和を計算する処理．33-35行目がメイン処理．各スレッドに割り当てる要素数について，入力データ数によっては一つ多く割り当てられるスレッドとそうでないスレッドが出てくるので，イテレーションの最後の1回にはその判定処理を入れる．
- 42行目の `__syncwarp()` は，ワープ内のスレッドで同期をとる処理．P100では不要だが，Voltaアーキテクチャではスレッド内のスレッドが同期処理されないことがあるので，入れてある

1 つ目のソースコードの説明は上にまとめた通りです．変数の意味については以下のスライドにまとめます．



この図の見方ですが，入力データを 2 次元に配置し，各スレッドが 1 列の総和を計算します．その際の自スレッドの担当列の index が sid です．この図では各ブロックは 2 つのワープを持つものとして書かれていますが，実際には 32 個のワープを持っています．あるスレッドが所属するブロックの通し番号は，blockId.x で取得でき，ブロック内でのスレッドの通し番号は threadIdx.x で取得できます．また，各ブロックに割り当てられているスレッドの数は blockDim.x で取得できます．

スレッドに対して，様々な ID のつけ方が考えられるので，先に計算しておくことにします．ワープ内でのスレッド ID(lid: Lane ID)，ブロック内でのスレッド ID(tid: Thread ID)，カーネル関数全体でのスレッド ID(sid: Sequential Thread ID) などはよく使われるので先に計算しておくとアルゴリズムを書く際に便利です．ブロック内でのワープの ID（wid: Warp ID）もよく使います．ちなみにこれらの変数名は必ずしも標準的とは言えませんが，Lane ID という名称については，比較的広く使われている印象です．

ソースコード解説（4分割のうち2つ目）

- 2-15行目はワープ内の各スレッドの計算結果に対して総和をとる処理．`__syncwarp()`はVoltaアーキテクチャ用に入れてある．3-11行目では処理を行う必要のないスレッドも処理を行うことになるが，余計な条件文を入れたくないので気にしないことにする．
- 15行目の`__syncthread()`はブロック内の全スレッドの同期をとる処理．すべてのスレッドが共有メモリに結果を書き込み終わった後に次の処理に進む必要があるため，この処理が必要．
- 17-28行目は各ワープの計算結果に対して総和をとる処理．18行目で条件を指定することにより1ワープのみを使って計算を行なっている．
- 29-31行目はブロックでの計算結果をグローバルメモリに書き込む処理．これによりグローバルメモリには各ブロックの結果が書き込まれる．

ソースコード解説（4分割のうち3つ目）

- これより先はCPUで処理されるmain関数
- 3-20行目は初期化処理で，必要なメモリ領域の確保や，入力の準備，正しい答えの準備などを行なっている
- 22行目から40行目はGPUを使用する時の定番処理

ソースコード解説（4分割のうち4つ目）

- 2-5行目は各ブロックの総和計算結果を足し合わせて最終的な出力を得る処理．GPUで行なっても良いが，あまり時間がかからないと思われるので，今回はCPUで処理する．

2 つ目から 4 つ目のソースコードの説明は上にまとめた通りです．
それでは，このプログラムを実習環境で実行しましょう．

ジョブスクリプト（cuda.nqs）

- 10秒制限
- 実行ファイル名をa.outにする
- 講習会用の待ち行列（LECTURE）に投入する
- 実習1と同内容

```
#!/bin/bash
#PBS -q LECTURE
#PBS -l elapstim_req=00:00:10,cpunum_job=1,gpunum_job=1
cd $PBS_O_WORKDIR
./a.out
```

まず，上記のように実習 1 と同様のジョブスクリプトを書きます．

コンパイルとジョブ投入

- コンパイル(nvcc)により実行ファイルa.outが生成される

```
$ nvcc -O3 -arch=sm_60 ex2.cu
$ qsub cuda.nqs
```

上記のようにコンパイルして，ジョブを投入しましょう．ジョブを投入したら qstat や sstat コマンドで状況確認しながらジョブが終了するのを待ちましょう．ジョブが完了したら標準出力を確認し，正しく実行できたことを確認しましょう．

オキュパンシのプロファイリング

- オキュパンシをプロファイリングするためのジョブスクリプト（occupancy.nqs）を作成する

```
#!/bin/bash
#PBS -q LECTURE
#PBS -l elapstim_req=00:00:10,cpunum_job=1,gpunum_job=1
cd $PBS_O_WORKDIR
nvprof --metrics achieved_occupancy ./a.out
```

これまでオキュパンシにこだわってアルゴリズムを設計してきましたが，実際にオキュパンシが高くなっていることを確認しましょう．実習 1 とは異なるプロファイリングとして，実際に達成したオキュパンシの値を計測してみます．そのためには，nvprof コマンドのオプションとして，–metrics achieved_occupancy とつけます．詳細については，NVIDIA の Achieved Occupancy の説明ページを参照してください．
(https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm)

ジョブ投入

```
$ qsub occupancy.nqs
```

それでは，ジョブを投入し，処理が終わるのを待って標準エラー出力を確認しましょう．高いオキュパンシが実現できているのを確認できるはずです．

2.6.2 Thrust ライブラリとの比較

✏️ Thrustライブラリとの比較

ソースコード（ex2.cuを変更）

- ex2.cuのThrustに関する行（1つ目のソースコードの3,4行，4つ目のソースコードの19-29行）のコメントアウトを取り除く

ジョブスクリプト（cuda.nqs）

- 前節と同じものを使う

コンパイルとジョブ投入

- コンパイル(nvcc)により実行ファイルa.outが生成される

```
$ nvcc -O3 -arch=sm_60 ex2.cu
$ qsub cuda.nqs
```

- ジョブ完了後，標準出力ファイルを開き，自作とライブラリの処理時間をチェックする

最後に CUDA のライブラリである Thrust を使った場合と処理時間を比較してみましょう．ソースコードの Thrust に関する部分のコメントアウトを取り除いてコンパイルした後，再びジョブを投入してみましょう．そしてジョブ終了後，出力を確認し，自作のアルゴリズムとライブラリで処理時間を比較してみましょう．自作アルゴリズムの方が処理時間が短くなっているはずです．

処理時間プロファイリング用のジョブスクリプト（prof.nqs）

- 10秒制限
- 実行ファイル名をa.outにする
- 講習会用の待ち行列（LECTURE）に投入する
- 実習1と同内容

```
#!/bin/bash
#PBS -q LECTURE
#PBS -l elapstim_req=00:00:10,cpunum_job=1,gpunum_job=1
cd $PBS_O_WORKDIR
nvprof ./a.out
```

処理時間の詳細を調べるためにプロファイリングをしてみましょう．まずそのためのジョブスクリプトを作成します．これは実習 1 と同じものです．

処理時間のプロファイリング

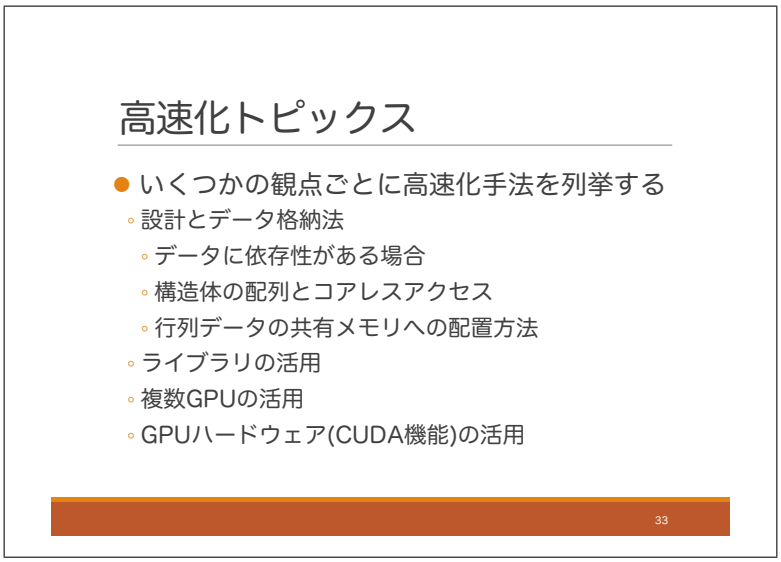
- ジョブ投入

```
$ qsub prof.nqs
```

- 自作とライブラリでメイン処理の処理時間がほとんど変わらないことを確認する

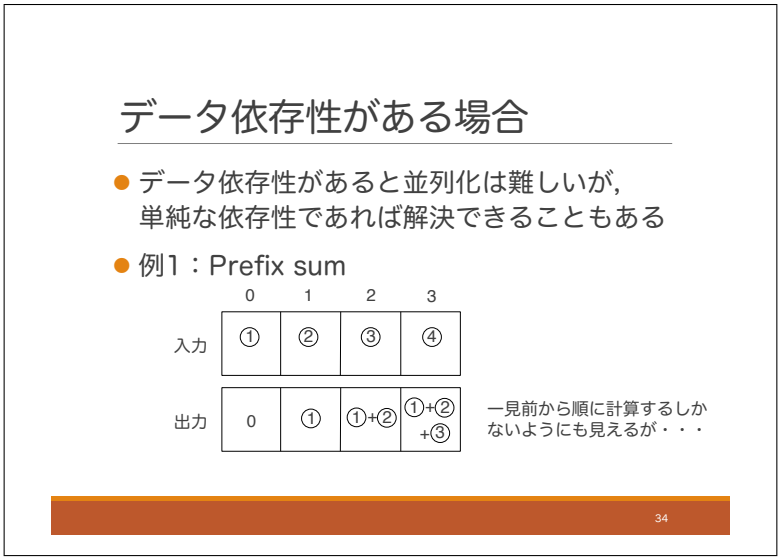
それではジョブを投入し，処理終了後，標準エラー出力を確認してみましょう．総和の計算部分では，自作アルゴリズムとライブラリはほぼ同じ計算時間になっているはずです．

2.7　高速化トピックス



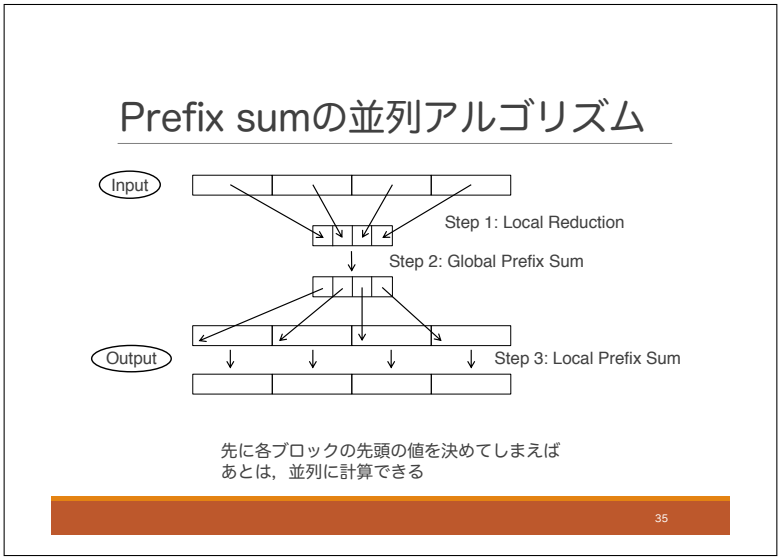
ここからは高速化トピックスということで，高速化のための技法を列挙していきます．まずはアルゴリズムの設計とデータ格納法に関していくつかのトピックを紹介します．その後，ライブラリの活用，複数 GPU の活用，GPU ハードウェアの活用などについてトピックを紹介します．

2.7.1　データに依存性がある場合



アルゴリズムの設計とデータ格納法に関する最初のトピックはデータに依存性がある場合の処理についてです．データに依存性があるというのは，ここでは，ある出力データを計算するために別の出力データが計算されるのを待たなければならないという場合を指しています．このような依存性があると，並列化は難しくなるのですが，簡単な依存性であれば解決できることもあります．

その例を 2 つ挙げたいと思いますが，一つ目はプレフィックスサムです．プレフィックスサムというのは図のような出力を計算する処理です．ざっくりいうと各 index において自分より左にいる入力配列の和を求める計算です．例えば出力配列の index 2 は入力配列の index0 と 1 の和となります．これを計算する場合，どの index も一つ前の index に依存性があるため，一見，前から順に計算していくしかないようにも見えます．しかし，このような場合に並列化を行うための定番のやり方があります．

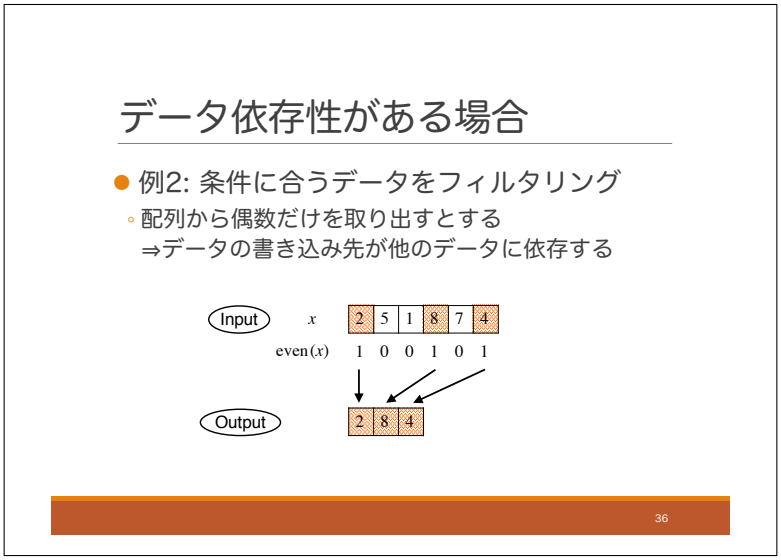


プレフィックスサムを複数のマルチプロセッサを用いて並列に計算することを考えます．ここでは，4つのマルチプロセッサを使うものとして説明します．その場合，入力配列を4分割して，各ブロックをマルチプロセッサに一つずつ分配する方法が考えられます．しかし，前のスライドで説明したように，このままでは1つ目のブロックから順に計算していくしかありません．各ブロックのが並列に計算を行えるようにするにはどうすればよいでしょうか．少し考えてみると，実は，各ブロックの先頭の出力値さえ分かればあとは各ブロックで並列に計算が進められるということがわかります．

そこで，先に各ブロックの先頭の出力値を求めることを考えます．そのためには各ブロックの総和の値が必要になるので，まずそれを並列で計算します．それが図のステップ1です．

次にステップ1の結果に対してプレフィックスサムを計算します．それが図のステップ2です．実はこれが各ブロックの先頭の出力値になります．例えば，2つ目のブロックの先頭の出力値は，1つ目のブロックの総和になりますし，3つ目のブロックの先頭の出力値は，1つ目のブロックと2つ目のブロックの総和になります．このプレフィックスサムの計算は並列計算せずシーケンシャルに計算することにします．入力要素数に比べてブロック数が十分に小さい場合は，このように計算したとしてもステップ2の計算量は漸近的には無視できます．

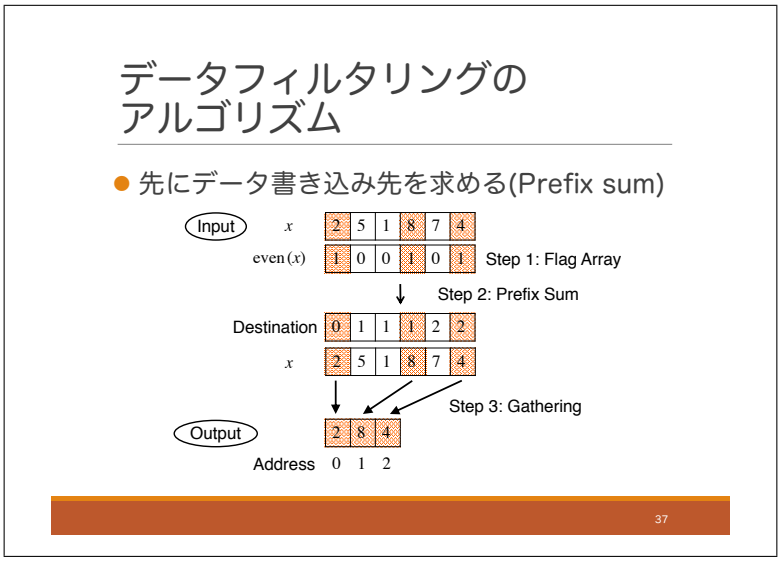
ということでめでたく各ブロックの先頭の出力値が計算できたので，あとは，各ブロックにおいて並列にプレフィックスサムを計算します．これがステップ3です．各ブロックのプレフィックスサムの計算についても色々工夫できそうですが，ここでは深入りしないでおきます．



データに依存性がある場合の処理についての二つ目の例として，データのフィルタリングを考えてみます．膨大な入力データの中から条件を満たす要素のみを取り出したいという状況です．ここでは，入力配列の中から偶数のみを取り出して，配列としてまとめて出力することを考えましょう．

各要素が偶数かどうかを判定することは並列に行えるのですが，問題は出力の配列を作る部分です．偶数が見つかったとして，それを出力配列のどこに書き込めばよいのかがわかりません．それは他の出力に依存します．

このような場合への対処についても定番のやり方があります．



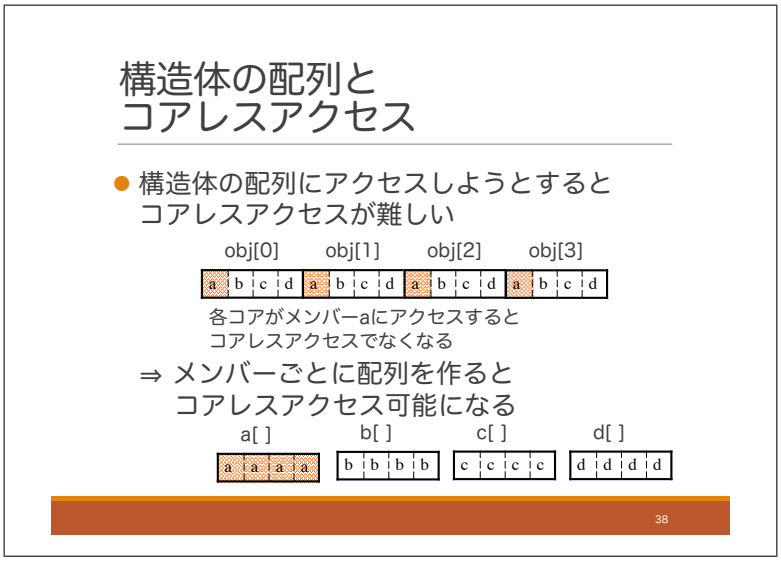
このような場合への対処法ですが，データの出力先がわからないことが問題なので，先にそれを求めることにします．実はその計算にはプレフィックスサムを活用できます．

まず，ステップ 1 として，入力の各要素が偶数かどうかを判定し，結果をフラグ配列に書き込みます．偶数なら 1，そうでなければ 0 を書くことにします．

そして次がポイントなのですが，ステップ 2 として，このフラグ配列のプレフィックスサムを計算します．そうすると実はプレフィックスサムの値が出力の書き込み先 index になっています．プレフィックスサムにより，自分より左に条件を満たす要素が何個あるかを計算できていることを確認してください．

最後に条件を満たす要素について，並列に出力配列への書き込みを行います．書き込み先についてはステップ 2 の結果を使います．これがステップ 3 です．これを GPU で実行する場合，少し工夫する余地がありそうですが，ここでは深入りしないでおきます．

2.7.2 構造体の配列とコアレスアクセス

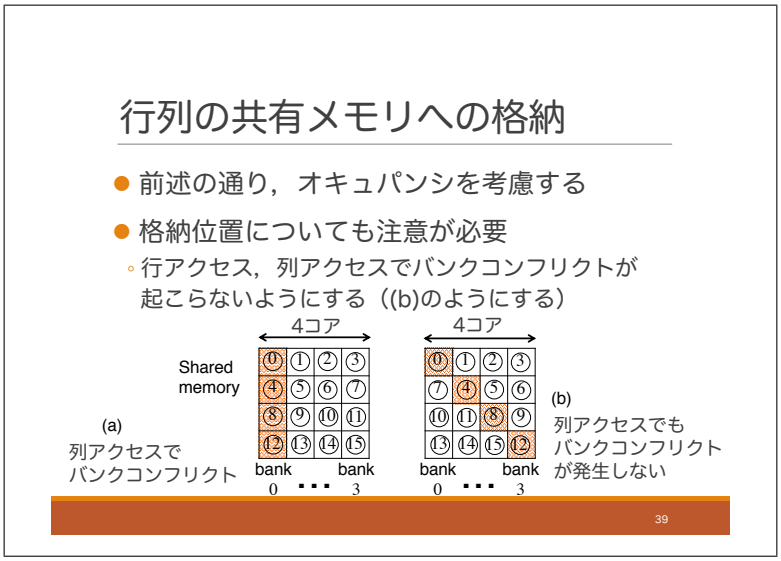


それでは次のトピックに移ります．次は構造体の配列を作る際の注意事項です．

グローバルメモリにデータを保持する際，構造体の配列を作ると，コアレスアクセスが難しくなることがあります．スライドの上の図では各コアが構造体のメンバー a にアクセスしようとしていますが，飛び飛びのアドレスにアクセスしているため，コアレスアクセスではなくなっています．これへの対処法として，下の図のようにメンバーごとに配列を作ることにすればコアレスアクセスが可能になります．

上の図のような構成は AoS (Array of Structures)，下の図のような構成は SoA (Structures of Array) と呼ばれています．GPU プログラミングでは，SoA を活用することで効率的なメモリアクセスができるようになります．

2.7.3　行列の共有メモリへの格納



次は行列などの 2 次元データを扱う際の注意事項です．本日の前半で話した通り，2 次元データの列に対して並列アクセスする場合は，工夫が必要です．まず，前半に話した通り，オキュパンシを考慮して，あまり大きいサイズのデータを一気に共有メモリにコピーしないほうがいいです．

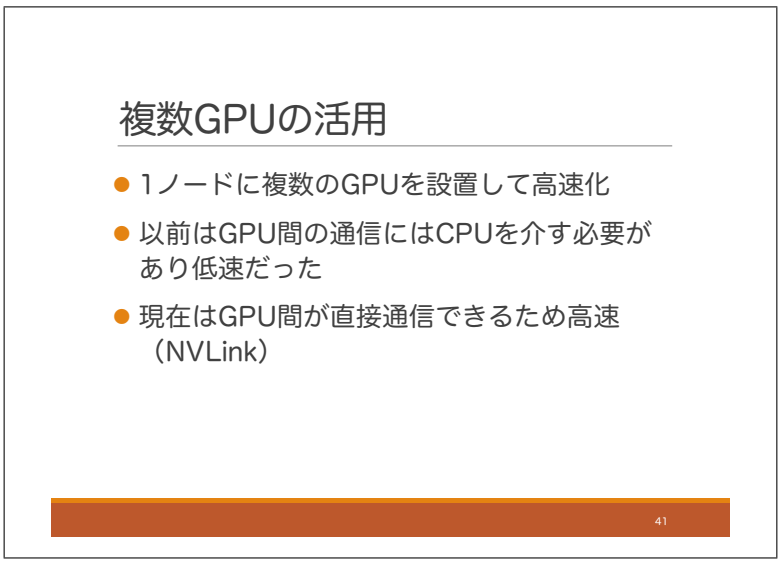
さらにコピーする際のデータの配置についても工夫が必要です．バンク数 4 の共有メモリにマルチプロセッサの 4 コアが並列アクセスする場合を考えます．図の各列が 1 つのバンクになっているとします．まず 4×4 の 16 データをグローバルメモリから共有メモリにコピーし，左のような順番で並べたとします．⓪から⑮はデータの順番です．最初の 4 要素をコアレスアクセスによりグローバルメモリから取得し，1 行目にコピーし，次の 4 要素は 2 行目にコピーという感じで，16 要素を 4 要素ずつ共有メモリにコピーします．その後，列の各要素に並列アクセスしようとすると，全てのコアが同じバンクにアクセスすることになり，バンクコンフリクトが発生します．

これに対し，データの格納方法を工夫することで，バンクコンフリクトを回避することができます．グローバルメモリからコピーした 4 要素を共有メモリに貼り付ける際，貼り付け先のアドレスを変更し，右の図のようにします．このようにすると，列に並列アクセスする際も各コアが別々のバンクにアクセスすることになり，バンクコンフリクトが発生しません．

2.7.4 ライブラリの活用

これまで話してきた通り，GPU では総和計算ですら効率的なコードを書くのは一苦労です．様々な問題に対し，GPU 上で動くプログラムを書きたいという場合には，極力ライブラリに頼るのが良いと思います．リダクションやソートなどの基本的な処理については，Thrust というライブラリがありますし，FFT についても cuFFT があります．他にもたくさんのライブラリがありますので，例えばスライドの URL を参考にしてみてください．

2.7.5 複数 GPU の活用



一つのマシンに複数の GPU を取り付けて高速化することができます．以前は GPU 間の通信は一旦 CPU を介す必要があり低速でしたが，現在は GPU 間で直接通信ができるため高速です．初日に紹介した CUDA C プロフェッショナルプログラミングという本には複数 GPU を使用したプログラミングについての詳細な解説があります．興味のある方はチェックしてみてください．

2.7.6　GPU ハードウェアの活用

最後に GPU のハードウェアを活用した高速化について簡単に紹介したいと思います．

まずはコンスタントメモリとテクスチャメモリです．どちらもリードオンリーのデータのためのメモリです．コンスタントメモリは GPU デバイス内に存在する定数格納用の小容量メモリですが，各マルチプロセッサ内のコンスタントキャッシュと呼ばれる部分に値がキャッシュされるため，高速アクセスが可能です．メモリサイズは今の所どのモデルでも 64KB になっています．全スレッドが同じ値を読み込む必要があるような場合に高速にデータ読み込みを行えます．テクスチャメモリとはグラフィック処理におけるテクスチャ，つまりオブジェクト表面に貼り付けられる画像を格納するためのメモリで，2 次元データを扱う際，アクセスした場所の近傍のデータへのアクセスが高速化されるように設計されています．アプリケーションによっては，グローバルメモリにアクセスするよりも高速なアクセスが可能です．

次はユニファイドメモリです．これは，CPU と GPU 間のデータ同期を自動化する技術です．ユニファイドメモリを使わない場合，CPU と GPU の間のデータ転送はすべてプログラムで記述しなければいけませんが，これを使う場合は，プログラムにデータコピーを記述しなくても，必要に応じて自動で同期が行われます．不要なデータコピーがなくなることで，プログラムが高速化されることもあるようです．

テンソルコアは 4×4 の行列演算をハードウェアで行うモジュールで，行列計算を高速化できます．近年 GPU が深層学習に多く活用されていますが，そこでは行列計算を高速化することが重要です．テンソルコアを活用することでその処理が高速化されます．

ここからはハードウェアというよりは CUDA の機能と言った方がいいかもしれませんが，それをいくつか紹介します．

まず，マルチストリームですが，複数のタスクを GPU に同時に行わせたり，GPU の処理中に CPU で別の処理を行ったりできる機能です．

次に，ダイナミックパラレリズムですが，GPU の処理中に別の GPU カーネル関数を呼ぶことができる機能です．これにより，CPU を介さずに動的なタスクの追加などができるようになります．

次がワープシャッフルです．これはスレッドがワープ内の他のスレッドのレジスタを読めるようにする機能です．これによりワープ内のスレッドとのデータのやり取りが高速化されます．これまではワープ内のスレッド間でのデータのやり取りは，共有メモリを介して行う必要がありましたが，ワープシャッフルにより，他のスレッドに直接アクセスできるようになりました．

最後が独立型スレッドスケジューリングです．これは 2017 年に発表された Volta アーキテクチャに搭載された機能なので，今回使用している P100 では使用できません．これはワープ内のスレッドごとに異なるタイミングで命令を実行できるようにする機能です．この講習会では，ワープ内のスレッドは常に同じ命令を同じタイミングで実行すると説明してきましたが，Volta アーキテクチャではこれは当てはまりません．この機能によりアプリケーションによってはプログラミングがしやすくなりますが，ワープ内スレッドの同期実行を前提に作成されたプログラムは正常に動作しなくなることもあります．

付録：Linux入門

Linux入門

一関高専 未来創造工学科 情報・ソフトウェア系
小池　敦

1

今日の内容

- Linuxの基本
  - 端末（ターミナル）の基本操作
  - ユーザーとグループ、スーパーユーザー
  - パーミッション
  - パイプラインとリダイレクト
  - 環境変数と設定ファイル
  - Linuxディストリビューション
- 基本的なLinuxコマンド（別資料）
  - ネットワーク関連コマンド

2

端末（ターミナル）の基本操作

- $ (プロンプト)の後ろにコマンドを打つ
  - $ ifconfig
- 基本的な操作
  - 矢印左右：カーソルの移動
  - 矢印上下：履歴の表示
  - Tabキー：入力の補完（$ ifc[Tabキー] ⇒ifconfig）
  - Ctrl-c：ジョブ（コマンド実行）の強制終了
  - Ctrl-a,e：カーソルを先頭（最後）に移動する
  - Ctrl-d: カーソル位置の文字（前方文字）を消去

3

練習

- ping www.google.co.jp をした後、中断して、traceroute www.google.co.jp を実行してください

実行例

$ ping www.google.co.jp

Ctrl-c

上矢印 Ctrl-a Ctrl-d(4回) tracer[Tab]

$ traceroute www.google.co.jp

4

端末（ターミナル）の基本操作

- 端末はどこかのディレクトリ（フォルダ）で作業している
  - 端末起動時はホームディレクトリ(/home/ユーザー名)にいる
- 現在の作業ディレクトリの表示
  - $ pwd
- 特別なディレクトリ名
  - ~ : ホームディレクトリ(/home/ユーザー名)
  - . : 作業ディレクトリ
  - .. : 作業ディレクトリの一つ上のディレクトリ

5

端末（ターミナル）の基本操作

- 作業ディレクトリの変更
  - $ cd /etc ⇒ /etc ディレクトリに移動
  - $ cd .. ⇒ 一つ上のディレクトリに移動
  - $ cd ⇒ ホームディレクトリに移動
- ディレクトリ内のファイル一覧表示
  - $ ls
  - $ ls -l ⇒ 各ファイルの詳細情報を表示
  - $ ls –a ⇒ 隠しファイルを含む全ファイルを表示
  - $ ls –la ⇒ 全ファイルを詳細表示

6

ユーザーとグループ

- ユーザーは必ずどこかのグループに所属
  - 複数のグループに属することも可能
- 各ファイルにも所有者とグループが設定される
  - $ touch test.txt ⇒ test.txtファイルを作成
    $ ls –l ⇒ 所有者とグループを表示
- ファイルの所有者情報を変更するには chownコマンドを使う

7

スーパーユーザー

- スーパーユーザー(rootユーザー)
  - 管理者権限を持つ
  - 強力な権限を持つのでスーパーユーザーで作業する時は作業ミスに注意する
- 一般ユーザーの一部はスーパーユーザー権限でのコマンド実行を許可されている

8

スーパーユーザー権限での コマンド実行

- コマンドの前にsudo をつける
  - 例： $ sudo less /var/log/secure
    ⇒ 管理者権限でセキュリティ関連ログをチェック
- スーパーユーザーに変身
  - $ su –
    ⇒ スーパーユーザーに変身（戻るにはexit）

9

ユーザーとグループ、スーパーユーザー

- sudoを実行可能なユーザー
  - /etc/sudoersに記載されている
    （本ファイルを編集する時はvisudoを使うこと）
- 一部の慣習として、wheelグループに属するユーザーにsudo権限を与えることがある
  - ユーザーにsudo権限を与えるには、
    そのユーザーをwheelグループに所属させればよい

10

パーミッション

- すべてのファイルにアクセス制限（パーミッション）が設定されている
- 誰に？
  - ファイル所有者
  - グループ
  - それ以外
- 制限内容（これ以外もあるが省略）
  - 読み込み
  - 書き込み
  - 実行

2進数で指定する
例：1 1 0 (=6_{10})⇒ 読み込み書き込み可、実行不可
読み込み　書き込み　実行

11

パーミッション

- パーミッションの指定
  - ファイル所有者、グループ、それ以外の順に制限内容を10進数で書く
  - 例： ６４４
    - 所有者　：6 (=110_2)
    - グループ：4 (=100_2)
    - それ以外：4 (=100_2)
- パーミッションを変更するにはchmodを使う
  - $ chmod 600 test.txt
    ⇒ test.txtのパーミッションを600に変更する

12

パイプライン

- あるコマンドの出力を次のコマンドの入力にする
  - 例 $ ps aux ⇒ 実行中の全プロセスを表示する
    $ ps aux | grep ssh
    ⇒ 実行中の全プロセスから
    sshが含まれる行を検索する
    （パイプライン利用してps auxの出力を
    grep ssh の入力にしている）

13

リダイレクト

- あるコマンドの出力をファイルに保存する
  - 例 $ ps aux > test.txt
    ⇒ ps auxの実行結果をtest.txtに保存する
    $ ifconfig >> test.txt
    ⇒ ifconfigの実行結果をtest.txtに追記する
    （既存の内容の後ろに追記する）
    $ cat text.txt ⇒ test.txtを表示する
- エラーメッセージのみを別ファイルに保存可能
  - 例 $ aaa 2> error.txt （追記したければ 2>> とする）
    $ cat error.txt

14

環境変数と設定ファイル

- 環境変数：設定情報を保持する変数
  - 例：PATH
    ⇒ パスなしでファイルを指定した際に探すディレクトリ
- 環境変数一覧表示：　$ printenv
- 環境変数の編集
  - ~/.bash_profileもしくは~/.bashrcに設定を書く
  - どちらに書いてもそんなに変わらない
    （.bash_profileに書く人が多い？）
  - 設定後は再読み込みを行う $ source ~/.bash_profile

15

Linuxのディストリビューション

- Linux: UNIXを参考にヘルシンキ大学の学生だったリーナス・トーバルズ氏が開発
- オープンソースラインセンスのため誰でも自由に改変して配布できる（ディストリビューション）
  - Debian系とRed Hat系が主流
- Linuxディストリビューションの例
  - CentOS: Red Hat系。企業サーバー等でよく使われる。
    本演習環境はCentOS7。
  - Ubuntu：Debian系。デスクトップOSとして使われる。
  - Amazon Linux: Red Hat系。
    Amazonのクラウドサービス（AWS）で利用できる。

16

基本的なLinuxコマンド

基本	ファイル操作	ファイル閲覧,編集等	管理者	ネットワーク	その他
man	cp	less	sudo	ifconfig	ps
pwd	mv	grep	su	ping	kill
ls	rm	find	shutdown	traceroute	df
cd	mkdir	cat	useradd	netstat	tar
	touch	diff	passwd	nslookup	
	chmod			dig	
	chown	vi		arp	
		emacs		ip	
				ss	
				curl	
				ssh	

基本

man	コマンドのマニュアル表示
コマンドや設定ファイル等のマニュアル表示。 表示後の操作はlessを参照。	
[オプション]	
[使用例]	
$ man ls ⇒ lsコマンドのマニュアルを表示	
$ man sshd_config ⇒ SSH設定ファイルsshd_configの設定マニュアルを表示	

このページのtopへ

pwd	作業ディレクトリの表示
現在作業しているディレクトリ（フォルダ）の絶対パスを表示する。	
[オプション]	
[使用例]	
$ pwd ⇒ 作業ディレクトリを表示	

このページのtopへ

ls	ディレクトリ内のファイル一覧表示
作業ディレクトリ（フォルダ）に存在するファイル、ディレクトリの一覧を表示する。	
[オプション]	
-l : 詳細(サイズ,所有者,最終更新,パーミッション)表示。	
-a : すべてのファイル,ディレクトリを表示(隠しファイル(ドットで始まるファイル)も表示)	

ls	ディレクトリ内のファイル一覧表示

[使用例]

```
$ ls    ⇒ 作業ディレクトリ内のファイルを表示
$ ls -a    ⇒ 作業ディレクトリ内のすべてのファイルを表示
$ ls -la    ⇒ 作業ディレクトリ内のすべてのファイルの詳細を表示
$ ls -la | less    ⇒ 作業ディレクトリ内のすべてのファイルの詳細をlessを使って表示
```

このページのtopへ

cd	ディレクトリの移動

作業ディレクトリ（フォルダ）を変更する。

[オプション]

[使用例]

```
$ cd /etc/    ⇒ /etc ディレクトリに移動
$ cd    ⇒ ホームディレクトリ ~ (/home/ユーザの名前/)に移動
$ cd ~/html/    ⇒ ~(ホームディレクトリ)の下の html ディレクトリに移動
$ cd tmp/    ⇒ 作業ディレクトリ(.)の下の tmp ディレクトリに移動
$ cd ..    ⇒ 作業ディレクトリ(.)の一つ親のディレクトリ(..)に移動
```

このページのtopへ

ファイル操作

cp	ファイルのコピー

ファイルやディレクトリをコピーする。

[オプション]

-R : サブディレクトリも含めてすべてコピーする

[使用例]

```
$ cp readme.txt readme.backup    ⇒ readme.txtと同内容のファイルreadme.backupを作る
$ cp ~/Downloads/Anaconda3-2.3.0-Linux-x86_64.sh .    ⇒ Anaconda3-2.3.0-Linux-x86_64.shを作業ディレクトリ（.）にコピー
$ cp -R src_dir target_dir    ⇒ src_dirを丸ごとtarget_dirにコピーする
$ cp -R src_dir/ target_dir    ⇒ src_dirの中身を丸ごとtarget_dirにコピーする（src_dir自身はコピーされない）
```

このページのtopへ

mv	ファイル移動、名称変更

ファイルやディレクトリの移動や名称変更を行う。

[オプション]

[使用例]

```
$ mv main.c src/    ⇒ main.cを作業ディレクトリ直下のsrcディレクトリに移動する
$ mv old_name.txt new_name.txt    ⇒ ファイルold_name.txtの名称をnew_name.txtに変更する
```

このページのtopへ

rm	ファイル削除

ファイルやディレクトリを削除する。

[オプション]

-r : サブディレクトリも含めてすべて削除する（-Rも同じ）

-f : 確認メッセージを表示せずに即座に削除する

[使用例]

rm	ファイル削除
$ rm index2.html	⇒ ファイルindex2.htmlを消去する
$ rm -rf tmp	⇒ tmpフォルダを消去確認なしに完全に削除する

このページのtopへ

mkdir	ディレクトリの作成
ディレクトリを作成する。	
[オプション]	
[使用例]	
$ mkdir src	⇒ srcディレクトリを作成する

このページのtopへ

touch	ファイルの新規作成、タイムスタンプ変更
ファイルの新規作成やファイルのタイムスタンプ（最終アクセス時刻、最終変更時刻）変更を行う。	
[オプション]	
-t : ファイルのアクセス時刻、変更時刻を修正する	
[使用例]	
$ touch readme.txt	⇒ ファイルreadme.txtを作成する（中身は空）
$ touch -t 201805311300 readme.txt	⇒ readme.txtのタイムスタンプを2018/5/31/ 13:00に変更する

このページのtopへ

chmod	ファイルのパーミッションの変更
ファイルのパーミッションを変更する。 "所有者(u)", "所属グループ(g)", "そのほか(o)"のそれぞれに対し "読み出し(r)", "書き込み(w)", "実行(x)" の許可(1)不許可(0)を設定。 rwx とも許可なら111で2進数から10進数に直し7が設定値となる。	
[オプション]	
-R : サブディレクトリも含めてすべてファイルのパーミッションを変更する	
[使用例]	
$ chmod 755 a.out	⇒ a.outのパーミッションを755に設定する（所有者のみ全操作が可能で、それ以外は読み出し、実行が可能）
$ chmod 644 readme.txt	⇒ readme.txtのパーミッションを644に設定する（所有者のみ読み込み、書き込みが可で、それ以外は読み出しのみ可）
$ chmod 600 id_rsa	⇒ id_rsa（SSH用の秘密鍵）のパーミッションを600に設定する（所有者のみ読み込み、書き込みが可で、それ以外は読み出し書き込み不可）

このページのtopへ

chown	ファイル所有者の変更
ファイルの所有者やグループを変更する。	
[オプション]	
-R : サブディレクトリも含めてすべてファイルのパーミッションを変更する	
[使用例]	
$ chown alice index.html	⇒ index.htmlの所有者をaliceに変更する
$ chown alice:group1 index.html	⇒ index.htmlの所有者をaliceにグループをgroup1に変更する

このページのtopへ

ファイル閲覧・編集等

less	テキストファイルの内容表示
テキストファイルの内容を表示する。文書内の移動や単語の検索が可能である。	
[less起動中の操作]	
q : less を終了	
h : help を表示	
矢印キー上下 : 上下に1行分スクロール	
u,d : 上下に半ページ分スクロール	
スペースキー : 下に1ページ分スクロール	
b : 上に1ページ分スクロール	
g,G : ファイルの先頭,最後に移動	
/hoge : hoge を検索。n,Nで前後のhogeにジャンプ	
[使用例]	
$ less readme.txt ⇒ readme.txtをlessを起動して閲覧する	
$ ls -l /etc/ | less ⇒ /etc/ ディレクトリの中身をlessを使って閲覧する	

このページのtopへ

grep	文字列の検索
指定した入力ファイルの中から条件を満たす行を検索する。正規表現を利用可能。	
[オプション]	
-i : 大文字と小文字の区別をしない	
[使用例]	
$ grep DirectoryIndex /etc/httpd/conf/httpd.conf ⇒ httpd.confの中からDirectoryIndexを含む行を検索する	
$ ps aux | grep ssh ⇒ ps（プロセス状態表示）の実行結果からsshという文字を含む行を検索する	

このページのtopへ

find	ファイル検索
条件を満たすファイルを検索する。	
[オプション]	
-name : 検索するファイル名の指定（部分一致条件を指定するには、"*hoge*"のようにワイルドカードを使う）	
-iname : 検索するファイル名の指定（大文字小文字を区別しない）	
-type : ファイルの種類を指定する（-type f とすると通常ファイルのみを検索する（ディレクトリ等は排除される））	
[使用例]	
$ find ~ -name "*bash*" ⇒ ホームディレクトリの中のすべてのディレクトリの中からbashという名前を含むファイルを検索する	
$ find / -name network -type f 2>/dev/null ⇒ コンピュータ全体からnetworkという名前のファイルを検索する（2>/dev/null はエラーメッセージを捨てる処理）	

このページのtopへ

cat	ファイル内容の表示
ファイルの中身を表示する。lessのように文書内を移動するような機能はなく、ただ出力するだけである。	
[オプション]	
[使用例]	
$ cat ~/.bashrc ⇒ .bashrcの中身を表示する	

このページのtopへ

diff	2ファイルの差分表示

diff	2ファイルの差分表示

2つのファイルの差分を探して表示する。

[オプション]

-u : unified形式（一つのコンテンツの中に両ファイルの内容を表示）で差分を表示する

-y : 2つのファイルを左右に分けて表示する

[使用例]

$ diff test.backup test.txt ⇒ test.backupとtest.txtの差分を表示する

このページのtopへ

vi	テキストエディタviの起動

テキストエディタviを起動する。

[vi起動中の操作]

viには2つのモードがあるので、状況に応じて使い分ける

- コマンドモード：検索、移動、セーブ、終了などのコマンドが使えるが文字入力はできない。iを入力すると編集モードに移行する
- 編集モード：文字を入力するためのモード。Escキーを入力するとコマンドモードに戻る。

コマンドモードでの主なコマンド

- i ⇒ 編集モードに移行し、現在のカーソル位置に文字入力を行えるようにする（Escキーでコマンドモードに戻る）
- :wq ⇒ セーブしてviを終了する
- :q! ⇒ セーブせずにviを終了する
- :w ⇒ セーブする
- dd ⇒ 1行削除（切り取り）
- u ⇒ 直前の操作の取り消し（アンドゥー）
- /hoge ⇒ 文字列hogeを検索する。n,Nで前後のhogeに移動する。
- Ctrl-d ⇒ 半ページ下に移動する
- Ctrl-u ⇒ 半ページ上に移動する
- G ⇒ 最終行に移動する
- 行数G ⇒ 「行数」で指定した行に移動する。先頭行なら1Gと入力する。

[使用例]

$ vi ~/.bash_profile ⇒ .bash_profileをviで編集する

このページのtopへ

emacs	テキストエディタemacsの起動

テキストエディタemacsを起動する。環境によってはインストールされていないので注意。

[emacs起動中の操作]

C-x C-c ⇒ 終了する(CはCtrlキー；Ctrlを押しながらxとcを押す)

C-x C-s ⇒ 保存する

C-x C-w ⇒ 別名保存する

C-x C-f ⇒ ファイルを開くまたは新規作成

C-g ⇒ キャンセル（コマンド中断）

C-@ ⇒ 範囲指定開始

C-w ⇒ 指定した範囲をカット(キル)

M-w ⇒ 指定した範囲をコピー(Mは通常はEscキー；Escを押した後wを押す)

C-k ⇒ カーソルから行末までをカット

C-y ⇒ 張り付け(ヤンク)

C-x 1 ⇒ ウィンドウの分割をやめる（C-x 2 とするとウィンドウを縦に分割）

C-s ⇒ 前向き検索

C-r ⇒ 後ろ向き検索

M-% ⇒ 問い合わせ置換(Mは通常はEscキー；Escを押した後%を押す)

C-/ ⇒ もとに戻す(Undo)

[使用例]

emacs	テキストエディタemacsの起動
$ emacs ~/.bash_profile　⇒ .bash_profileをemacsで編集する	

このページのtopへ

管理者

sudo	管理者としてコマンドを実行する
スーパーユーザー（管理者）としてコマンドを実行する。sudoを許可されたユーザーしか本コマンドを実行できない。一部のLinuxではwheelグループに属しているユーザーに権限を与えるという慣習もある。	
[オプション]	
[使用例]	
$ sudo less /var/log/secure　⇒ 管理者権限でセキュリティ関連ログを見る	

このページのtopへ

su	他ユーザーに切り替わる
他のユーザーに切り替わる。元のユーザーに戻るにはexitと入力する。	
[オプション]	
- : 環境変数を新ユーザーに引き継がない（新ユーザーの環境を利用する）	
[使用例]	
$ su - alice　⇒ ユーザーaliceに切り替わる	
$ su -　⇒ スーパーユーザー（管理者）に切り替わる	

このページのtopへ

shutdown	シャットダウン、再起動
マシンのシャットダウンや再起動を行う。管理者権限が必要。	
[オプション]	
-h : シャットダウンする	
-r : 再起動する	
[使用例]	
$ sudo shutdown -h now　⇒ すぐにマシンをシャットダウンする	
$ sudo shutdown -r now　⇒ すぐにマシンを再起動する	

このページのtopへ

useradd	ユーザーの追加
システムにユーザーを追加する。	
[オプション]	
-G : ユーザが所属する補助グループを指定する	
[使用例]	
$ sudo useradd alice　⇒ システムにユーザーaliceを追加する	
$ sudo useradd -G wheel alice　⇒ システムにユーザーaliceを追加し、wheelグループに加える（sudo実行権限を与える）	

このページのtopへ

passwd	パスワード変更
ユーザーのパスワードを変更する	
[オプション]	
[使用例]	

passwd	パスワード変更
$ sudo passwd alice	⇒ ユーザーaliceのパスワードを変更する

このページのtopへ

ネットワーク

ifconfig	ネットワーク設定情報出力
ネットワーク設定情報の表示や変更を行う。	
[オプション]	
-a : 停止しているものも含めすべてのネットワークインターフェースの状態を表示する	
[使用例]	
$ ifconfig	⇒ ネットワーク設定情報を表示する

このページのtopへ

ping	ネットワーク疎通確認
指定ノードへのネットワークの到達性を確認する。ネットワークに問題がなくてもpingに応答しないサーバーもいるので注意。Ctrl-c で中断する。	
[オプション]	
-c : 指定回数のみパケットを送付する	
[使用例]	
$ ping www.google.co.jp	⇒ www.google.co.jpとの疎通確認を行う
$ ping -c 5 www.google.co.jp	⇒ www.google.co.jpとの疎通確認を5回行う

このページのtopへ

traceroute	ネットワークの経路表示
指定ノードへのネットワーク経路を表示する。ネットワーク上のどこで通信が失敗しているかやボトルネックがどこかなどの解析に利用できる。 ネットワークに問題がなくても応答しないサーバーもいるので注意。	
[オプション]	
[使用例]	
$ traceroute www.google.co.jp	⇒ www.google.co.jpへの経路を表示する
$ traceroute www.tohoku.ac.jp	⇒ www.tohoku.ac.jpへの経路を表示する（サーバが応答しない例）

このページのtopへ

netstat	ソケット一覧表示
ノードに設定されているソケット（他マシンとの通信路設定）の一覧を表示する。	
[オプション]	
-t : tcpソケットのみを表示する（--tcpとしても同じ）	
-r : OSが保持するルーティングテーブルを表示する	
-i : ネットワークインターフェースの統計状態を表示する	
[使用例]	
$ netstat -t	⇒ TCPソケット一覧を表示する
$ netstat -r	⇒ ルーティングテーブルを表示する
$ netstat -i	⇒ インターフェースごとの統計情報を表示する

このページのtopへ

nslookup	ドメイン名からIPアドレスを調べる
ドメイン名から対応するIPアドレスを調べる。	

nslookup	ドメイン名からIPアドレスを調べる
[オプション]	
-type : レコード（サーバーの種類等）の指定（AAAA:IP6アドレス、MX:メール、NS:ネームサーバー）	
[使用例]	
$ nslookup www.ichinoseki.ac.jp ⇒ 一関高専Webサーバーの IPアドレスを調べる	
$ nslookup -type=AAAA www.google.co.jp ⇒ Google検索サイトのIPv6アドレスを調べる	

このページのtopへ

arp	ARPテーブルの表示
OSで管理しているARPテーブル（IPアドレスとMACアドレスの対応表）の表示や設定を行う。	
[オプション]	
-a : ARPテーブルを表示する。	
-d : 指定したホストのARPエントリーを廃棄する（ネットワーク設定が変更になった場合等に用いることができる）	
[使用例]	
$ arp -a ⇒ ARPテーブルのすべてのエントリーを表示する	
$ sudo arp -d 192.168.0.1 ⇒ ARPテーブルから192.168.0.1に関するエントリーを削除する	

このページのtopへ

dig	ネームサーバーへのドメイン名問い合わせ
ネームサーバーにドメイン名を問い合わせて結果を表示する。nslookupに比べ問い合わせ結果があまり加工されずに表示されるので、ネットワーク管理者にとっては扱いやすい（らしい）。	
[オプション]	
[使用例]	
$ dig www.ichinoseki.ac.jp ⇒ 一関高専Webサーバーの IPアドレスを調べる	
$ dig gmail.com mx ⇒ gmail.comのメールサーバーの IPアドレスを調べる	

このページのtopへ

ip	ネットワーク設定の表示
ネットワーク設定情報の表示や変更を行う。現在ifconfig等のコマンドはメンテナンスがされていないため非推奨となっており、それらの代替となるコマンドがipである。 機能が多岐にわたるため、「ip addr」や「ip link」のようにサブコマンドを後ろにつけて使用する。情報表示だけでなく変更もできるがこの表では省略する。	
[サブコマンド]	
addr : 自身のIPアドレス等のネットワーク情報を表示する（a としても同じ）	
route : ルーティングテーブルの情報を表示する（r としても同じ）	
neighbor : ARPテーブルの情報を表示する（neigh や n としても同じ）	
help : ヘルプを表示する（n としても同じ）	
[使用例]	
$ ip a ⇒ 自身のIPアドレス情報を表示する（レガシーコマンドの ifconfig に相当；a はaddrの省略形）	
$ ip r ⇒ ルーティングテーブルの情報を表示する（レガシーコマンドの netstat -r に相当；r はrouteの省略形）	
$ ip n ⇒ ARPテーブル情報を表示する（レガシーコマンドの arp -a に相当；n はneighborの省略形）	
$ ip -s link ⇒ インターフェースごとの統計情報を表示する（レガシーコマンドの netstat -i に相当）	

このページのtopへ

ss	ソケット情報表示
ノードに設定されているソケット（他マシンとの通信路設定）の情報を表示する。 現在netstatコマンドはメンテナンスがされていないため非推奨となっており、その代替となるコマンドがssである。	
[オプション]	
-t : tcpソケットのみを表示する（--tcpとしても同じ）	
[使用例]	
$ ss -t ⇒ TCPソケット一覧を表示する（レガシーコマンドの netstat -t に相当）	

このページのtopへ

curl	URLからのデータ取得
指定したURLからデータを取得する	
[オプション]	
-L : リダイレクトされた場合でもデータを取得する	
-O : 取得したデータをファイル保存する（このオプションを指定しないときは標準出力に出力される）	
[使用例]	
$ curl -L "http://zip.cgis.biz/xml/zip.php?zn=0218511" ⇒ 郵便番号検索APIで021-8511(一関高専)を検索し、結果を表示する	
$ curl -OL "http://weather.livedoor.com/forecast/rss/area/030010.xml" ⇒ Livedoorから盛岡の天気情報をダウンロードし、ファイルに保存する	

このページのtopへ

ssh	SSHクライアントプログラムの起動
SSHクライアントプログラムを起動する	
[オプション]	
-p : ポート番号を指定する（指定しないときは22）	
[使用例]	
$ ssh -p 10022 alice@198.51.100.14 ⇒ ユーザーaliceで192.51.100.14のポート10022にSSHログインする	

このページのtopへ

その他

ps	実行中プロセス一覧表示
実行中のプロセスの一覧を表示する。	
[オプション]	
a : 自ユーザー以外のプロセスも表示する	
u : 読みやすさを重視したフォーマットで出力する	
x : 端末を持たないプロセス（OSが利用するプロセス等）も表示する	
f : 階層構造（呼び出し関係）が分かるように出力する	
[使用例]	
$ ps uf ⇒ 自ユーザーが起動したプロセスの一覧をみる	
$ ps auxf | less ⇒ すべてのプロセスの一覧をlessを使って閲覧する	

このページのtopへ

kill	プロセスの強制終了
指定したプロセスID（PID）のプロセスを終了させる。PIDはpsコマンドを用いて調べられる。	

kill	プロセスの強制終了
[オプション]	
[使用例]	
$ kill 24567　⇒ PIDが24567のプロセスを強制終了する	

このページのtopへ

df	ディスクの空き容量表示
ディスクの空き容量を表示する	
[オプション]	
-h : 人間に読みやすいフォーマットで表示する（容量をG,Mなどのサイズ文字を用いて表示する）	
[使用例]	
$ df -h　⇒ ディスクの空き容量を表示する	

このページのtopへ

tar	ファイルの解凍・圧縮
ファイルの解凍・圧縮を行う。	
[オプション]	
-x : 解凍	
-c : 圧縮	
-v : 状況表示	
-f : 圧縮ファイル名	
[使用例]	
$ tar -xvf hoge.tar　⇒ hoge.tarを解凍する(gzファイルやbz2ファイルの解凍も同じコマンド)	
$ zip -r hoge.zip foo　⇒ fooフォルダをhoge.zipという名前でzip圧縮する（参考）	
$ unzip hoge.zip　⇒ hoge.zipを解凍する（参考）	

このページのtopへ