CUDA テクニカル トレーニング

Vol I: CUDA プログラミング入門

制作および提供: NVIDIA

Q2 2008

目次

セクション スライド

GPUコンピューティングの概要

並列コンピューティングの黄金期

- 1980年代から1990年前半にかけて: 並列コンピューティングの黄金期
  - 特にデータ並列コンピューティング
- マシン
  - Connection Machine、MasPar、Cray
  - 一般のPCとは別ものともいえる、強力で高価な正真正銘のスーパーコンピュータ
- アルゴリズム、言語、プログラミングのモデル
  - 広範囲にわたる問題を解決
  - さまざまな並列アルゴリズムモデルの開発
  - P-RAM、V-RAM、回路、ハイパーキューブなど

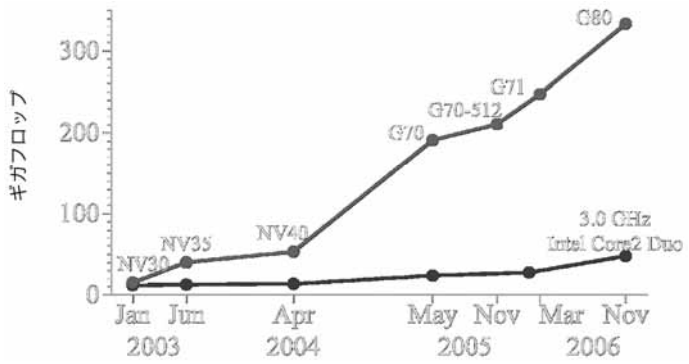
並列コンピューティングの暗黒期

- データ並列コンピューティングによる影響の限界
  - 7 CM-1マシン販売の低迷（合計数百システムの販売実績）
  - MasParは200システム以下の販売実績
- 業務およびリサーチ活動の沈静化
  - 大規模並列マシンは、より強力になった、普及品のマイクロプロセッサのクラスタに移行
  - Beowulf、Legion、グリッドコンピューティングなど

大規模並列コンピューティングは、一般普及品の技術の強力な進歩により勢いを失う

GPUの登場

- GPUは大規模マルチスレッドのメニーコアチップ
  - 何百のコア、何千の並行スレッド
  - 非常に負荷の高い、複雑な計算をPCで実行可能に
  - 高い量産効果



CUDAの登場

- CUDA (Compute Unified Device Architecture)
- GPUコンピューティングのためのNVIDIA GPUの真の計算能力を発揮するために共同デザインされた、ハードウェアとソフトウェア
- ソフトウェア
  - C言語の小さい拡張セット
  - 習得が容易
- ハードウェア
  - 共有メモリ - 拡張性可能なスレッドの協調動作

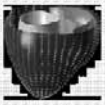
CUDAプログラミングモデル: 高度なマルチスレッド対応コプロセッサ

- GPUは計算デバイスである
  - ホストCPUのコプロセッサとして機能
  - カード上に自身のデバイスメモリを持つ
  - 平行して多数のスレッドを実行
- 多数のスレッドで1本のプログラムを実行する、並列カーネル
- GPUスレッドは非常に軽量
  - スレッドの作成とコンテキストの切り替えは事実上なし
- GPUをフル活用すれば、数千のスレッドを実行可能

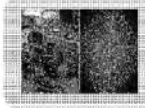
GPUコンピューティングが活用される分野の例


計算
地球科学

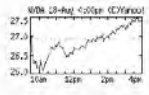

計算化学

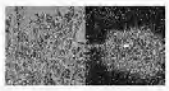

計算医学


計算
モデリング


計算科学


計算
生物学


計算
ファイナンス

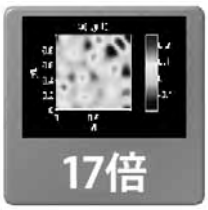

画像処理

GPUコンピューティングのスイートスポット

- 適用分野:
  - 高い計算集約度
    密な線形代数、PDE、N体、差分化
  - 広い帯域幅:
    順序付け (ウィルススキャン、ゲノム解析) 、ソート、データベースなど
  - バーチャルコンピューティング:
    グラフィックス、イメージ処理、トモグラフィ、マシンビジョンなど

アプリケーションの高速化

容積測定時の白質連結の
インタラクティブな視覚化



Matlabでの等方性乱流
のシミュレーション

天体物理学における
N体計算

高度に最適化されたオブ
ジェクト指向分子動力学

LIBORモデルのスワップ
付き金融シミュレーション

類似タンパク質および
遺伝子配列検索用の
厳密なCmatch文字列照合

CUDAプログラミングモデルの概要

設計目標

- 数百コア、数千の並列スレッドに拡張する
- プログラマを並列アルゴリズムに集中させる
  - 並列プログラミング言語の仕組みは気にしなくてよい
- 異種混在のシステム（すなわち、CPU+GPU）を可能にする
  - CPUとGPUは別のDRAMを持つ、個別のデバイス

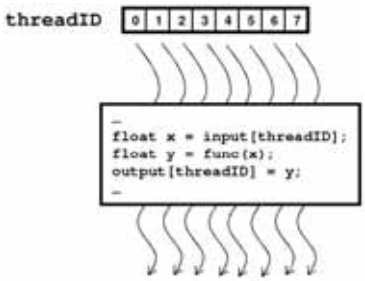
CUDAカーネルとスレッド

- アプリケーションの並列部分はカーネルとしてデバイス上で実行される
  - 1度に実行されるカーネルは1つ
  - 多数のスレッドが各カーネルを実行する
- CUDAスレッドとCPUスレッドの違い
  - CUDAスレッドは非常に軽量
    - 作成のオーバーヘッドがほとんどない
    - 瞬時に切り替えが可能
  - CUDAは数千ものスレッドによって高い効率性を実現
    - マルチコアCPUで使用できるスレッドは、数個のみ

定義:
デバイス＝GPU、**ホスト**＝CPU
カーネル＝ホストから呼び出され、デバイス上で実行される機能

並列スレッドの配列

- CUDAカーネルはスレッドの配列で実行される
  - すべてのスレッドは同じコードを実行する
  - 各スレッドはメモリアドレスを計算し、制御を決定するためのIDを持つ



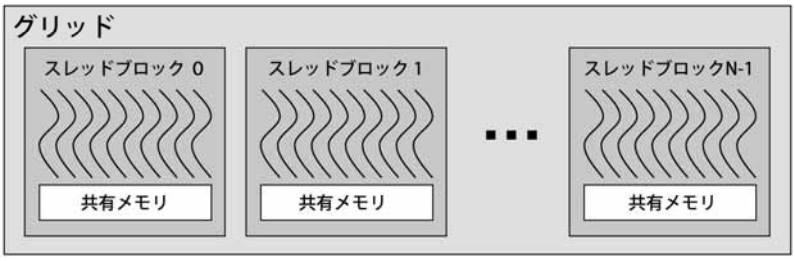
スレッドの協調

- 並列スレッドの不足部分: スレッドの協調が必要
- スレッドの協調は重要
  - 結果を共有して冗長な計算を避ける
  - メモリアクセスを共有する
    - 帯域幅が大幅に低減される
- スレッドの協調はCUDAの強力な機能

- モノリシックなスレッド配列間の協調は、拡張性がない
  - スレッドの小さい集合内での協調は拡張性がある

スレッドのバッチ処理

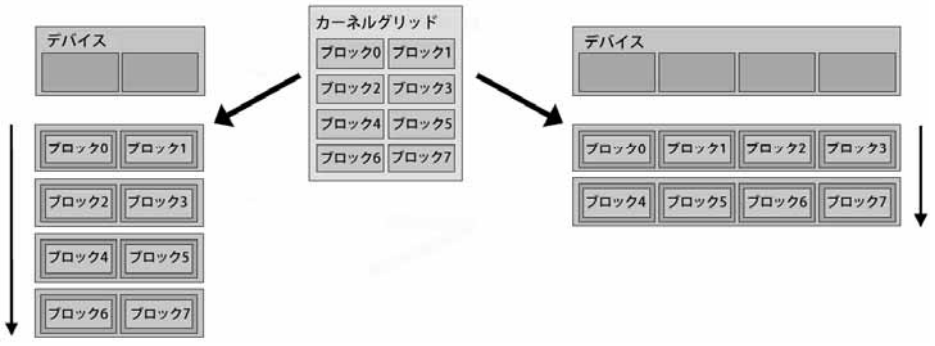
- カーネルはスレッドブロックのグリッドを起動する



  - ブロック内のスレッドは共有メモリを介して協調する
  - 別のブロックのスレッドは協調できない
- プログラムは別のGPUに**透過的に拡張する**ことができる

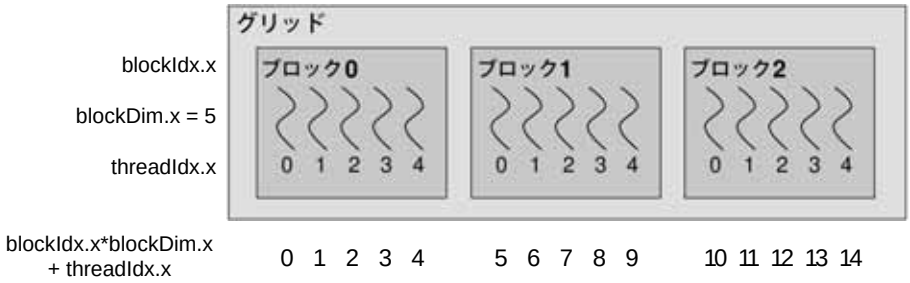
透過的な拡張性

- ハードウェアは任意のプロセッサのスレッドブロックを自由にスケジュールできる
  - カーネルは並列マルチプロセッサにわたって拡張可能



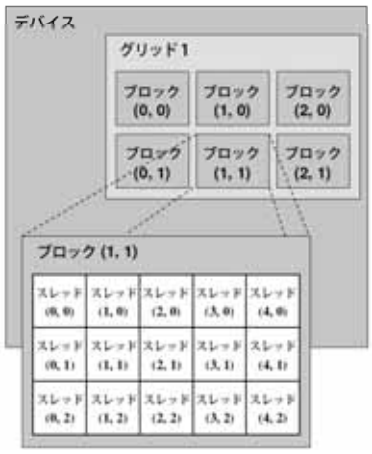
データの分解

- カーネル内の各スレッドから配列の別の要素にアクセスしたい場合がある
- 各スレッドは以下にアクセス
  - threadIdx.x – ブロック内のスレッドID
  - blockIdx.x – グリッド内のブロックID
  - blockDim.x – ブロックあたりのスレッドの数



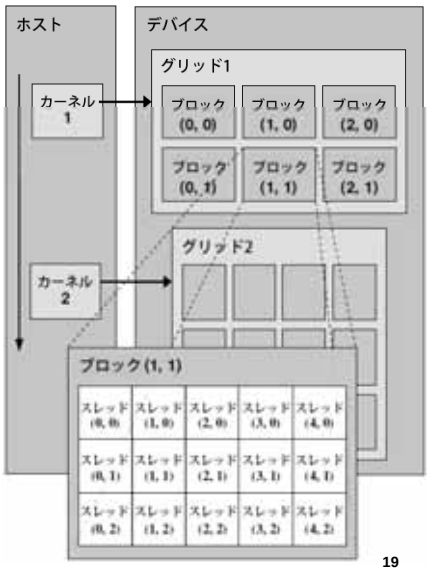
多次元のID

- ブロックID: 1次元または2次元
- スレッドID: 1次元、2次元、3次元
- 多次元データを処理する場合のメモリアドレス指定を簡略化
  - 画像処理
  - ボリューム上のPDEの解決



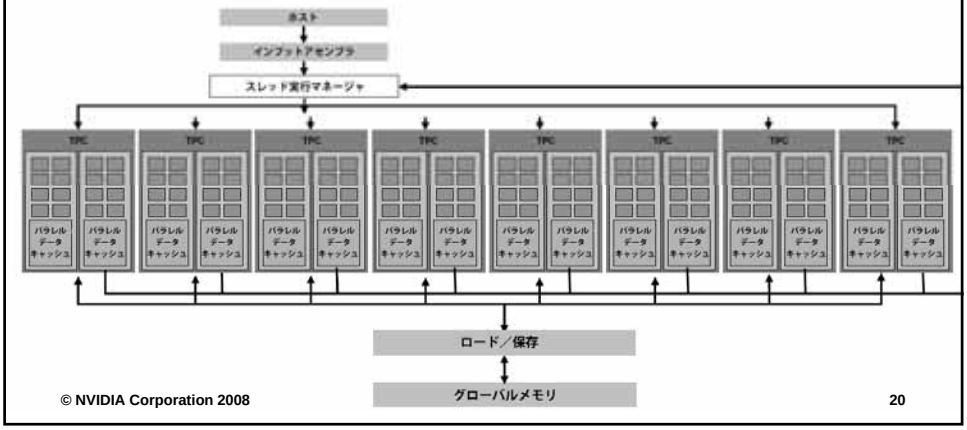
CUDAプログラミングモデル

- カーネルはスレッドブロックのグリッド単位で実行される
- スレッドブロックとは、以下によって互いに協調できるスレッドの集合
  - 共有メモリを介したデータの共有
  - 実行の同期
- 別のブロック内のスレッドとは協調できない

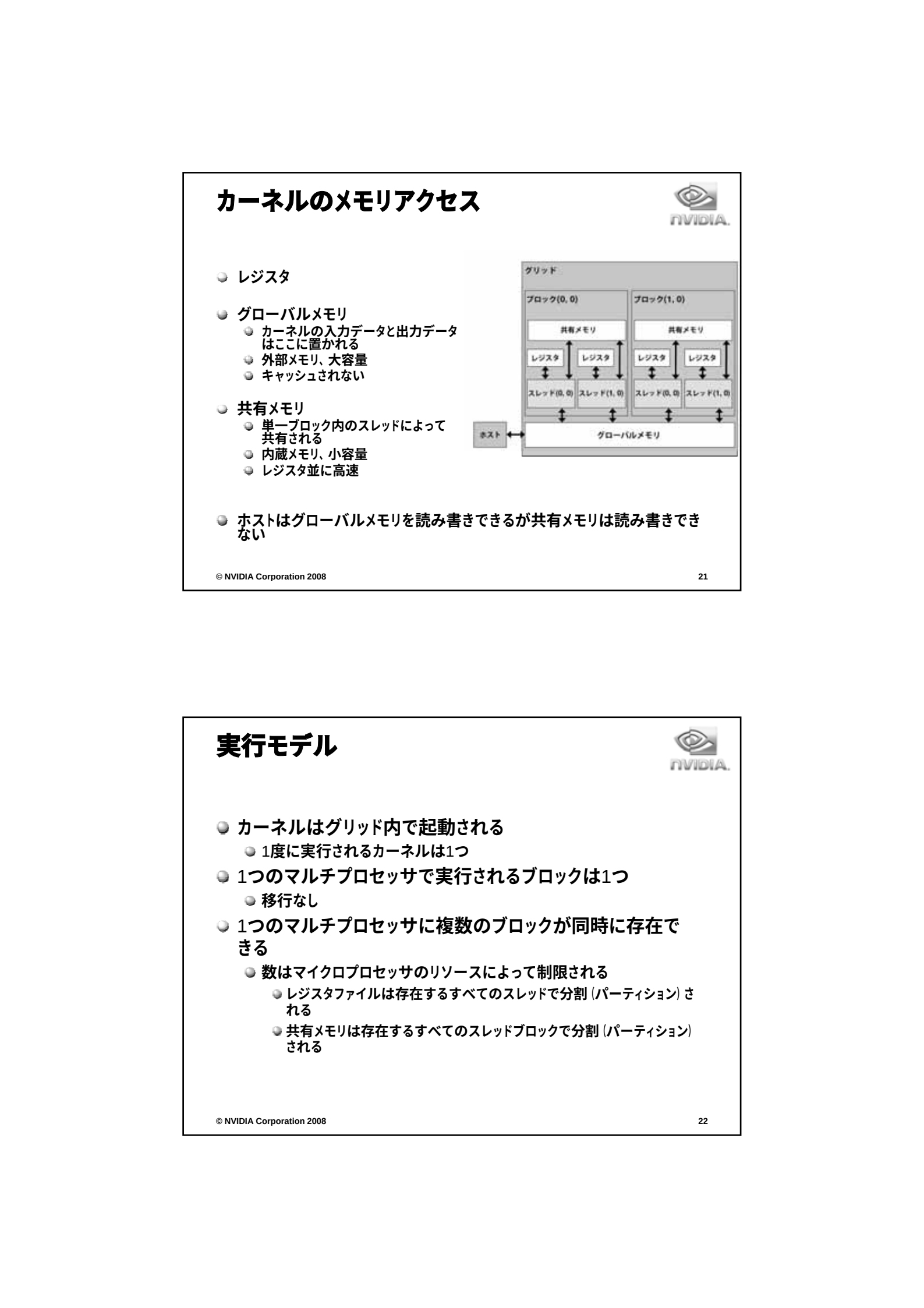


G80デバイス

- プロセッサがスレッドの計算を実行する
- スレッド実行マネージャがスレッドを発行する
- 128のスレッドプロセッサが16のマルチプロセッサ (SM) にグループ化される
- パラレルデータキャッシュがスレッドを協調させる



カーネルのメモリアクセス



- レジスタ
- グローバルメモリ
  - カーネルの入力データと出力データはここに置かれる
  - 外部メモリ、大容量
  - キャッシュされない
- 共有メモリ
  - 単一ブロック内のスレッドによって共有される
  - 内蔵メモリ、小容量
  - レジスタ並に高速

- ホストはグローバルメモリを読み書きできるが共有メモリは読み書きできない

実行モデル

- カーネルはグリッド内で起動される
  - 1度に実行されるカーネルは1つ
- 1つのマルチプロセッサで実行されるブロックは1つ
  - 移行なし
- 1つのマルチプロセッサに複数のブロックが同時に存在できる
  - 数はマイクロプロセッサのリソースによって制限される
    - レジスタファイルは存在するすべてのスレッドで分割（パーティション）される
    - 共有メモリは存在するすべてのスレッドブロックで分割（パーティション）される

従来のGPGPUと比較した場合のCUDAの利点

- バイト単位でアドレス指定可能なランダムアクセスメモリ
  - スレッドから任意の位置にアクセスできる
- 制限なしのメモリへのアクセス
  - スレッドは必要なだけ、いくつもの位置を読み書きできる
- 共有メモリ（ブロックごと）およびスレッドの同期
  - スレッドは協調して共有メモリにデータをロードできる
  - 任意のスレッドから共有メモリの任意の位置にアクセスできる
- 習得が容易
  - C言語の簡単な拡張
  - グラフィックスの知識は不要

CUDAモデルのまとめ

- 何千もの軽量の並行スレッド
  - 切り替えのオーバーヘッドなし
  - 命令とメモリのレイテンシを隠す
- 共有メモリ
  - ユーザー管理のデータキャッシュ
  - ブロック内でのスレッドの通信／協調
- グローバルメモリへのランダムアクセス
  - 任意のスレッドで任意の位置を読み書き可能

メモリ	位置	キャッシュ	アクセス	スコープ（「讖?」）
共有	内臓	N/A	読み取り／書き込み	ブロック内のすべてのスレッド
グローバル	外部	なし	読み取り／書き込み	すべてのスレッド＋ホスト

CUDAプログラミング

基本

CUBAの基本の概要

- GPUコードのセットアップと実行の基本
  - GPUのメモリ管理
  - GPUカーネルの起動
  - GPUコードの特性
- その他の機能
  - ベクトル型
  - 同期
  - CUDAエラーのチェック
- 注: 基本的な機能のみを説明します
  - API関数の詳細については『プログラミングガイド』を参照してください
  - 「最適化」セクションにも詳細があります

メモリ管理

- CPUとGPUは別のメモリ空間を持つ
- ホスト(CPU)コードがデバイス(GPU)メモリを管理する
  - 割り当て／解放
  - デバイス間のデータのコピー
  - グローバルデバイスメモリ(DRAM)でも同様

CPUメモリの割り当て／解放

- cudaMalloc(void ** pointer, size_t nbytes)
- cudaMemset(void * pointer, int value, size_t count)
- cudaFree(void* pointer)

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int *d_a = 0;
cudaMalloc( (void**)&d_a, nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

データのコピー

- cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);
  - `direction`は、(ホストまたはデバイスの) `src`と`dst`の位置を指定
  - CPUスレッドをブロック: コピーが完了したら戻す
  - 前のCUDA呼び出しが完了するまでコピーは開始されない
- enum cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice

CUDAエクササイズ

- CUDA実践エクササイズ用に、スケルトンと解答をご用意しました。
- 各エクササイズで、コードの欠けている部分を補ってください。
  - コンパイルし、プログラムを実行して「Correct!」と出力されたら、完了です。
- 解答は、各エクササイズの「solution」フォルダにあります。

コードのコンパイル: Windows

- Microsoft Visual Studioで<プロジェクト>.slnファイルを開く
  - プロジェクトをビルドする
  - 4つの構成オプション
    - Release、Debug、EmuRelease、EmuDebug
- コードをデバッグする場合はEmuDebug構成でビルド
  - カーネル内にブレイクポイントを設定可能（`__global__`または`__device__`関数）
  - 通常どおりに`printf`でもコードをデバッグ可能
  - 1つのGPUスレッドに対して1つのCPUスレッド
  - 実際には、GPU上ではスレッドは並列でない

コードのコンパイル: Linux

`nvcc <filename>.cu [-o <executable>]`
- リリースモードでビルド

`nvcc –g <filename>.cu`
- デバッグ（デバイス）モードでビルド
- ホストコードはデバックできるが、デバイスコード（GPUで実行）はデバッグできない

`nvcc –deviceemu <filename>.cu`
- デバイスエミュレーションモードでビルド
- デバッグシンボルなし、CPUですべてのコードを実行

`nvcc –deviceemu –g <filename>.cu`
- デバッグデバイスエミュレーションモードでビルド
- デバッグシンボル付き、CPUですべてのコードを実行
- gdbまたはその他のlinuxデバッガを使用してデバッグ

エクササイズ1: ホストとデバイス間でのコピー

- 「gcudaMallocAndMemcpy」テンプレートから開始
- Part1: デバイス上にポインタd_aとd_bのメモリを割り当てる
- Part2: ホスト上のh_aをデバイス上のd_aにコピーする
- Part3: d_aからd_bへデバイス間のコピーを実行する
- Part4: デバイス上のd_bをホスト上のh_aにコピーバックする
- Part5: ホスト上のd_aとd_bを解放する

GPUでのコードの実行

- カーネルは何らかの制限のあるC関数
  - GPUメモリのみにアクセスできる
  - 戻り型voidが必要
  - 可変個引数 (varargs) なし
  - 再帰的にはできない
  - static変数なし
- 関数の引数はCPUからGPUメモリに自動的にコピーされる

関数の修飾子

- __global__ : ホスト (CPU) コード内で呼び出し。デバイス (GPU) コードからは呼び出せない。戻り型voidが必要
- __device__ : 別のGPU関数から呼び出される。ホスト (CPU) コードからは呼び出せない
- __host__ : CPUでのみ実行可能。ホストから呼び出される
- __host__と__device__の修飾子は組み合わせられる
  - 使用例: 演算子のオーバーロード
  - コンパイラはCPUコードとGPUコードの両方を生成する

カーネルの起動

- C関数呼び出し構文の変形

```
kernel<<<dim3 grid, dim3 block>>>(…)
```

- 実行の構成 (“<<< >>>”)
  - グリッドの次元: xとy
  - スレッドブロックの次元: x、y、z

```
dim3 grid(16, 16);
dim3 block(16,16);
kernel<<<grid, block>>>(...);
kernel<<<32, 512>>>(...);
```

CUDA組み込みデバイス変数

- __global__と__device__のすべての関数は、これらの自動的に定義された変数にアクセスできる
  - dim3 gridDim;
    - ブロックのグリッドの次元（最大で2次元）
  - dim3 blockDim;
    - スレッドのブロックの次元
  - dim3 blockIdx;
    - グリッド内のブロックのインデックス
  - dim3 threadIdx;
    - ブロック内のスレッドのインデックス

最小カーネル

```
__global__ void minimal( int* d_a)
{
    *d_a = 13;
}

__global__ void assign( int* d_a, int value)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    d_a[idx] = value;
}
```

共通パターン

例: 配列要素の増分

N要素のベクトルをスカラbで増分

N=16、blockDim=4、4ブロックと仮定

blockIdx.x=0 blockDim.x=4 threadIdx.x=0,1,2,3 idx=0,1,2,3	blockIdx.x=1 blockDim.x=4 threadIdx.x=0,1,2,3 idx=4,5,6,7	blockIdx.x=2 blockDim.x=4 threadIdx.x=0,1,2,3 idx=8,9,10,11	blockIdx.x=3 blockDim.x=4 threadIdx.x=0,1,2,3 idx=12,13,14,15

int idx = blockDim.x * blockId.x + threadIdx.x;

ローカルのインデックスthreadIdxからグローバルのインデックスにマップ

注意: 実際のコードではblockDimは32以上。これは例にすぎません

例: 配列要素の増分

CPUプログラム

```
void increment_cpu(float *a, float b, int N)
{

     for (int idx = 0; idx<N; idx++)
       a[idx] = a[idx] + b;
}


void main()
{
  .....
    increment_cpu(a, b, N);
}
```

CUDAプログラム

```
__global__ void increment_gpu(float *a, float b, int N)
{
     int idx = blockIdx.x * blockDim.x + threadIdx.x;
     if (idx < N)
       a[idx] = a[idx] + b;
}


void main()
{
 …
   dim3 dimBlock (blocksize);
   dim3 dimGrid( ceil( N / (float)blocksize) );
   increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

2Dデータの最小カーネル

```
__global__ void assign2D(int* d_a, int w, int h, int value)
{
    int iy = blockDim.y * blockIdx.y + threadIdx.y;
    int ix = blockDim.x * blockIdx.x + threadIdx.x;
    int idx = iy * w + ix;

    d_a[idx] = value;
}
...
assign2D<<<dim3(64, 64), dim3(16, 16)>>>(...);
```


ホストの同期

- すべてのカーネルは非同期で起動される
  - 即座にCPUに制御が戻る
  - カーネルは前のCUDA呼び出しがすべて完了してから実行される
- cudaMemcpy()は同期
  - コピーの完了後CPUに制御が戻る
  - 前のCUDA呼び出しがすべて完了してからコピーが開始される
- cudaThreadSynchronize()
  - 前のCUDA呼び出しがすべて完了するまでブロック

例: ホストコード

```
// ホストメモリを割り当て
int numBytes = N * sizeof(float)
float* h_A = (float*) malloc(numBytes);

// デバイスメモリを割り当て
float* d_A = 0;
cudaMalloc((void**)&d_A, numbytes);

// ホストからデバイスにデータをコピー
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);

// カーネルを実行
increment_gpu<<< N/blockSize, blockSize>>>(d_A, b);

// デバイスからホストにデータをコピーバック
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);

// デバイスメモリを解放
cudaFree(d_A);
```


エクササイズ2: カーネルの起動

- 「myFirstKernel」テンプレートから開始
- Part1: ポインタd_aを使用して、カーネルの結果に対応するデバイスメモリを割り当てる
- Part2: 1-Dスレッドブロックの1-Dグリッドを使用してカーネルを構成して起動する
- Part3: 各スレッドで以下のようにd_aの要素を設定する

```
idx = blockIdx.x*blockDim.x + threadIdx.x
d_a[idx] = 1000*blockIdx.x + threadIdx.x
```

- Part4: d_aの結果をホストのポインタh_aにコピーバックする
- Part5: 結果が正しいことを検証する

さまざまな修飾子 (GPUコード)

- __device__
  - デバイスメモリに保存する (大容量、高いレイテンシ、キャッシュなし)
  - cudaMallocで割り当てられる (__device__修飾子は暗黙)
  - すべてのスレッドからアクセス可能
  - 生存期間: アプリケーション
- __shared__
  - 内蔵の共有メモリに保存する (レイテンシは非常に低い)
  - 実行の構成またはコンパイル時に割り当てられる
  - 同じブロック内のすべてのスレッドからアクセス可能
  - 生存期間: カーネルの実行中
- 非修飾の変数
  - スカラと組み込みのベクトル型はレジスタに保存される
  - 要素が5以上ある配列はデバイスメモリに保存される

共有メモリの使用

コンパイル時にサイズを決定

```
__global__ void kernel(…)
{
  …
  __shared__ float sData[256];
  …
}
int main(void)
{
  …
  kernel<<<nBlocks,blockSize>>>(…);
  …
}
```

カーネルの起動でサイズを決定

```
__global__ void kernel(…)
{
  …
  extern __shared__ float sData[];
  …
}
int main(void)
{
  …
  smBytes =
    blockSize*sizeof(float);
  kernel<<<nBlocks, blockSize,
    smBytes>>>(…);
  …
}
```

組み込みのベクトル型

CPUコードとGPUコードで使用可能

- [u]char[1..4], [u]short[1..4], [u]int[1..4], [u]long[1..4], float[1..4]
  - x、y、z、wフィールドでアクセスされる構造体:

```
uint4 param;
int y = param.y;
```

- dim3
  - uint3が基本
  - 次元の指定に使用
  - デフォルト値(1,1,1)

GPUスレッドの同期

- void __syncthreads();
- ブロック内のすべてのスレッドを同期する
  - 境界同期命令を生成する
  - ブロック内のすべてのスレッドに達するまで、どのスレッドもこの境界を越えることはできない
  - 共有メモリへのアクセスでRAW ／ WAR ／ WAWハザードを避けるために使用
- 条件がスレッドブロック全体で一律の場合に限り、条件コードに記述できる

GPUでの整数のアトミック演算

- Compute capability 1.1に対応するハードウェアが必要
  - G80 = Compute capability 1.0
  - G84/G86/G92 = Compute capability 1.1
- グローバルメモリでの整数のアトミック演算
  - 符号付／符号なし整数の連想演算
  - 加算、減算、最小、最大...
  - ANDまたはXOR
  - 増分、減分
  - 交換、比較、スワップ

CPUへのCUDAのエラーレポート

- すべてのCUDA呼び出しはエラーコードを返す
  - カーネルの起動は除く
  - cudaError_t型
- cudaError_t cudaGetLastError(void)
  - 最後のエラーのコードを返す（ノーエラーにもコードがある）
  - カーネルの実行でのエラーの取得にも使用できる
- char* cudaGetErrorString(cudaError_t code)
  - エラーを表すヌル終端キャラクタ文字列を返す

printf(“%s¥n”, cudaGetErrorString(cudaGetLastError()));

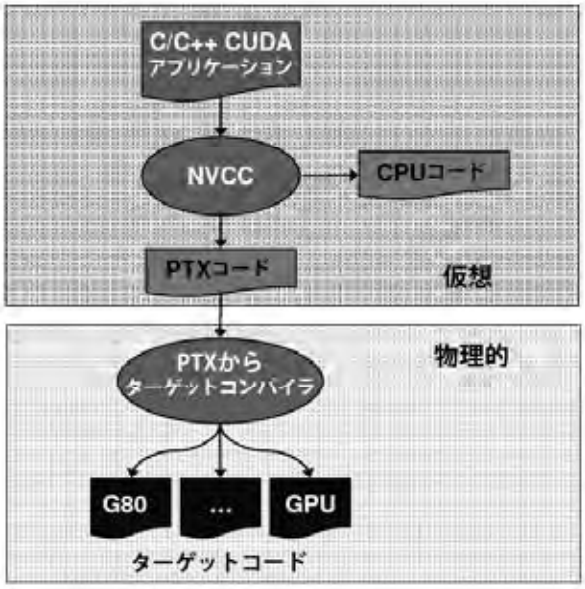
エクササイズ3: 配列の反転 (単一ブロック)

- ポインタd_aに入力配列{a_0, a_1, …, a_{n-1}}がある場合に、ポインタd_bに反転した配列{a_{n-1}, a_{n-2}, …, a_0}を保存する
- 「greverseArray_singleblock」テンプレートから開始
- スレッドブロックを1つだけ起動し、配列のサイズを反転させる
  N = numThreads = 256要素
- Part1 : カーネル「greverseArrayBlock()」の本体を実装するのみ
- 各スレッドで単一の要素を反転の位置に移動する
  - d_aポインタから入力を読み込む
  - d_bポインタに位置を反転した出力を保存する

エクササイズ4: 配列の反転 (複数ブロック)

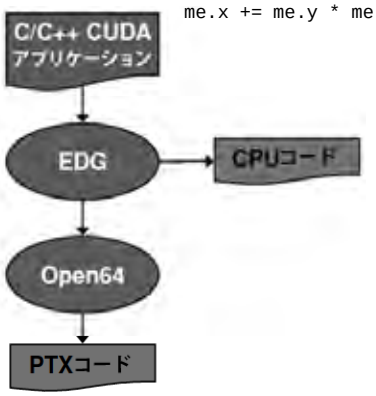
- ポインタd_aに入力配列{a_0, a_1, …, a_{n-1}}がある場合に、ポインタd_bに反転した配列{a_{n-1}, a_{n-2}, …, a_0}を保存する
- 「greverseArray_multiblock」テンプレートから開始
- 256スレッドのブロックを複数起動する
  - サイズNの配列を反転するには、N/256ブロック
- Part1: 起動するブロック数を計算する
- Part2: カーネルのreverseArrayBlock()を実装する
- 以下の両方を計算する必要がある
  - ブロック内で反転した位置
  - ブロックの先頭に対する、反転されたオフセット

CUDAのコンパイル



nvccとPTX仮想マシン

```
float4 me = gx[gtid];
me.x += me.y * me.z;
```



- EDG
  - GPUコードとCPUコードを分離
- Open64
  - GPU PTXアセンブリを生成
- Parallel Thread eXecution (PTX)
  - 仮想マシンとISA
  - プログラミングモデル
  - 実行リソースと状態

```
ld.global.v4.f32 {$f1,$f3,$f5,$f7}, [$r9+0];
mad.f32          $f1, $f5, $f3, $f1;
```

コンパイル

- CUDA言語拡張が記述されているソースファイルは、nvccでコンパイルする必要がある
- nvccはコンパイラドライバ
  - すべての必要なツールとcudacc、g++、clなどのコンパイラを起動させることで動作する
- nvccは以下のいずれかを出力できる
  - Cコード (CPUコード)
    - 別のツールを使用して、アプリケーションの残りといっしょにコンパイルする必要がある
  - PTXオブジェクトコードに直接
- CUDAコードの実行可能プログラムで必要なもの
  - CUDAコアライブラリ (cuda)
  - CUDAランタイムライブラリ (cudart)
    - ランタイムAPIを使用する場合
    - CUDAライブラリをロードする

パフォーマンスの最適化

概略

- 概要
- G8xハードウェア
- メモリの最適化
- 実行構成の最適化
- 命令の最適化
- まとめ

GPUでのアルゴリズムの最適化

- 独立した並列性を最大化する
- 計算集約度を最大にする (計算／帯域幅)
- キャッシュするよりも再計算するほうが適している場合もある
  - GPUはメモリでなく論理演算装置でトランジスタを消費
- GPUでの計算を多くして、負荷の高いデータ転送を避ける
  - 並列性が低い計算でもホストとの転送を行うよりも速い場合がある

メモリアクセスの最適化

- 結合 vs. 非結合=桁違いの差
  - グローバル／ローカルデバイスのメモリ
- キャッシュされているテクスチャメモリの空間的局所性に最適化
- 共有メモリでは過度なバンク競合を避ける

共有メモリの活用

- グローバルメモリよりも数百倍高速
- スレッドは共有メモリを介して協調できる
- 1つまたは少数のスレッドを使用して、すべてのスレッドに共有されるデータのロードや計算を実行する
- 非結合アクセスを避けるために使用する
  - 共有メモリでロードと保存を実行し、非結合アドレス指定を再度順序付ける
  - SDKサンプルの「Matrix Transpose」

効率のよい並列性の使用

- 計算を分割してGPUマルチプロセッサを均一に稼動させる
  - スレッドの数が増えるとスレッドブロックの数も増える
- マルチプロセッサごとに複数のアクティブなスレッドブロックをサポートするように、リソースの使用を抑える
  - レジスタ、共有メモリ

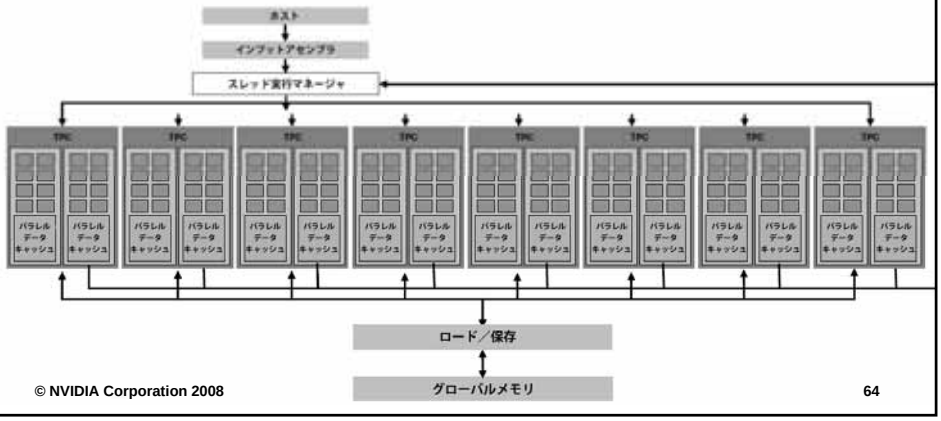
G8xハードウェア

用語

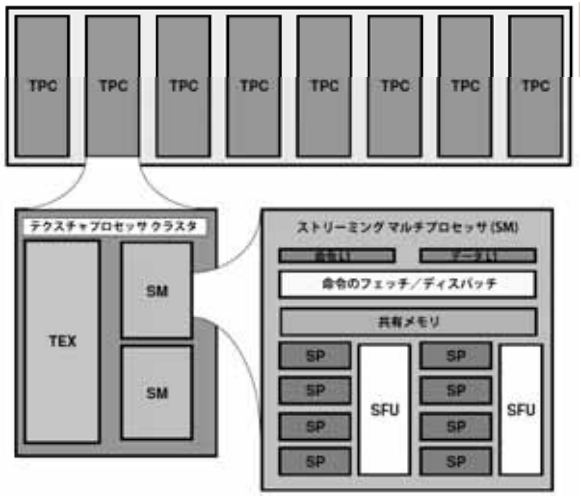
- スレッド: CUDAデバイスで（他のスレッドと並列に）実行される並行コードと関連付けられている状態
  - CUDAの並列処理の単位
  - CPUスレッドとの違いに注意: GPUスレッドの作成コスト、リソースの使用量、切り替えコストはずっと少ない
- ワープ: 物理的に並列で実行されるスレッドのグループ（SIMD）
  - ハーフワープ: スレッドのワープの最初または2番目の半分
- スレッドブロック: 一緒に実行され、1つのマルチプロセッサのメモリを共有できるスレッドのグループ
- グリッド: 単一のGPU上で1つのCUDAカーネルを論理上並列で実行する、スレッドブロックのグループ

G80デバイス

- スレッドの計算を実行するプロセッサ
- スレッド実行マネージャがスレッドを発行
- 128のスレッドプロセッサ
- パラレルデータキャッシュが処理を高速化

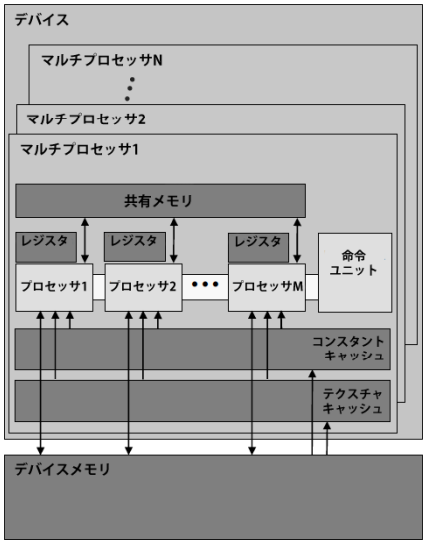


TPC (テクスチャプロセッサ クラスタ)



メモリアーキテクチャ

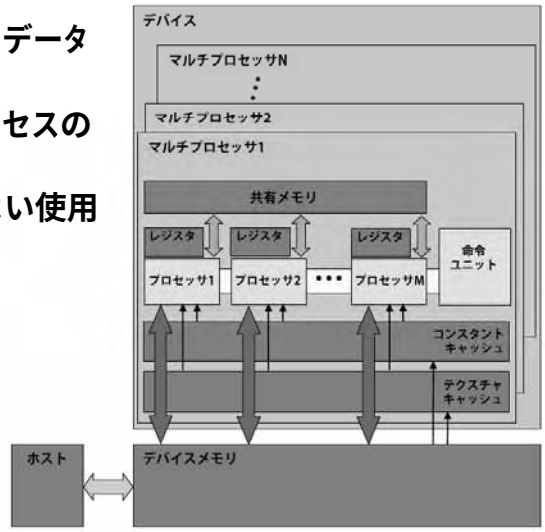
- グローバル、コンスタント、テクスチャの空間はデバイスメモリの領域
- 各マルチプロセッサの構成
  - プロセッサごとに32ビットのレジスタのセット
  - オンチップ共有メモリ
    - 共有メモリ空間が置かれる
  - 読み取り専用のコンスタントキャッシュ
    - コンスタントメモリ空間へのアクセスを高速化する
  - 読み取り専用のテクスチャキャッシュ
    - テクスチャメモリ空間へのアクセスを高速化する



メモリの最適化

メモリの最適化

- ホストーデバイス間のデータ転送の最適化
- グローバルデータアクセスの結合
- 共有メモリの効率のよい使用



ホストーデバイス間のデータ転送

- デバイスメモリからホストメモリの帯域幅はデバイスメモリからデバイスの帯域幅に比べて非常に狭い
  - ピーク時4GB/s（PCIe x16 1.0）vs.ピーク時76 GB/s（Tesla C870）
- 転送の最小化
  - ホストメモリにコピーすることなく、中間のデータ構造で割り当て、演算、解放できる
- 転送のグループ化
  - 細かい多数の転送よりも、一度の大きい転送のほうがよい

ページロックのデータ転送

- cudaMallocHost()では、ページロック（ピン固定）してホストメモリを割り当てることができる
- 最も高いcudaMemcpyパフォーマンスが可能
  - 3.2 GB/s（PCIe x16 1.0）
  - 5.2 GB/s（PCIe x16 2.0）
- CUDA SDKサンプルの「bandwidthTest」を参照
- 使用時の注意！！
  - ページロックのメモリを多く割り当てすぎると、システム全体のパフォーマンスが低下する場合がある
  - 限界を把握するために、システムとアプリケーションをテストする必要がある

非同期のメモリコピー

- ホストーデバイス間でのピン固定されたメモリ（Cの「cudaMallocHost」で割り当て）の非同期のメモリコピーは、すべてのCUDA対応デバイスのCPUを解放する
- ストリームを使用してオーバーラップを実装する
- ストリーム=順番に実行する、一連の演算
- ストリームAPI:
  - 0 = デフォルトのストリーム
  - `cudaMemcpyAsync(dst, src, size, direction, 0);`

カーネルとメモリコピーのオーバーラップ

- カーネルとピン固定メモリのホストーデバイス間のメモリコピーを同時実行
  - Compute capability 1.1以上に対応するデバイス（G84以上）
  - CUDAツールキットv1.1のプレビュー機能として入手可能
  - あるストリームのカーネルの実行を別のストリームのメモリコピーでオーバーラップ
- ストリームAPI
  - `cudaStreamCreate(&stream1);`
  - `cudaStreamCreate(&stream2);`
  - `cudaMemcpyAsync(dst, src, size, dir, stream1);`
  - `kernel<<<grid, block, 0, stream2>>>(…);`
  - `cudaStreamQuery(stream2);`

オーバーラップされる

グローバルメモリと共有メモリ

- G8x GPUではグローバルメモリはキャッシュされない
  - レイテンシは長いが、多くのスレッドを起動してレイテンシを隠す
  - アクセスを最小限に抑えることが重要
  - グローバルメモリアクセスの結合（後述）
- 共有メモリはオンチップで帯域幅が非常に広い
  - 低いレイテンシ
  - ユーザー管理のマルチプロセッサごとのキャッシュと同様
  - バンク競合を最低限に抑え、できるだけ避ける（後述）

テクスチャメモリとコンスタントメモリ

- テクスチャのパーティションはキャッシュされる
  - テクスチャキャッシュはグラフィックスでも使用される
  - 2次元の空間的局所性に最適化
  - ワープのスレッドが2次元上で集約的な位置を読み込む場合に、優れたパフォーマンスを発揮
- コンスタントメモリはキャッシュされる
  - 1つのワープで、アドレスごとに4サイクルの読み込み
    - ワープ内のすべてのスレッドが同じアドレスを読み込む場合は、合計4サイクル
    - すべてのスレッドが別のアドレスを読み込む場合は合計64サイクル

グローバルメモリの読み取り／書き込み

- G8xではグローバルメモリはキャッシュされない
- 最も高い命令レイテンシ: 400〜600クロックサイクル
- パフォーマンスのボトルネックを発生させやすい
- 最適化するとパフォーマンスが大幅に向上する

グローバルメモリのロードと保存

- nvccのptxフラグを使用して命令を検査:

4バイトのロードと保存

```
ld.global.f32    $f1, [$rd4+0];   // id:74
…
st.global.f32    [$rd4+0], $f2;   // id:75
```

8バイトのロードと保存

```
…
ld.global.v2.f32 {$f3,$f5}, [$rd7+0];      //
…
st.global.v2.f32 [$rd7+0], {$f4,$f6};      //
```

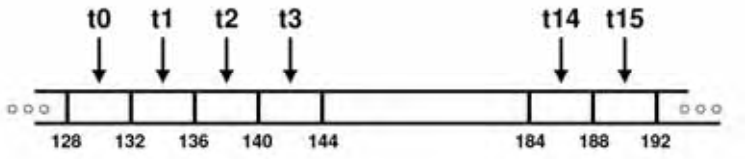
16バイトのロードと保存

```
…
ld.global.v4.f32 {$f7,$f9,$f11,$f13}, [$rd10+0];    //
…
st.global.v4.f32 [$rd10+0], {$f8,$f10,$f12,$f14};   //
```

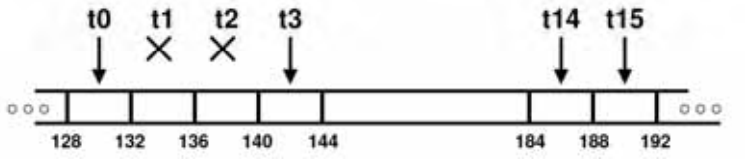
結合

- ハーフワープ（16スレッド）で読み込みを協調
- グローバルメモリの連続した領域:
  - 64バイト - 各スレッドはシングルワード（int、floatなど）を読み込む
  - 128バイト - 各スレッドはダブルワード（int2、float2など）を読み込む
  - 256バイト - 各スレッドはクワッドワード（int4、float4など）を読み込む
- その他の制限
  - 領域の開始アドレスは領域サイズの倍数でなくてはならない
  - ハーフワープのk番目のスレッドは読み込まれるスレッドのk番目の要素にアクセスしなくてはならない
- 例外: 適用されないスレッドもある
  - 述語アクセス、ハーフワープ内の分岐

結合アクセス: floatの読み込み

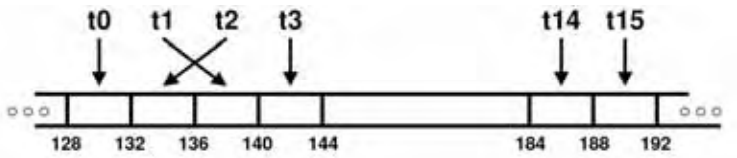


すべてのスレッドが参加

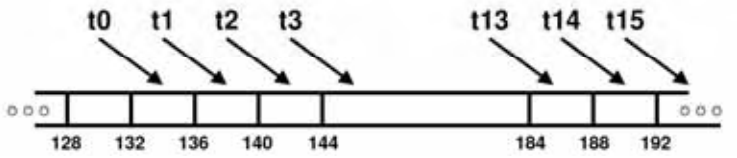


いくつかのスレッドは参加しない

非結合アクセス: floatの読み込み



スレッドにより順序変更されたアクセス



開始アドレスの位置ずれ (64の倍数でない)

結合: 時間測定の結果

- 実験
  - カーネル: floatを読み込んで増分し、ライトバックする
  - 3Mの浮動小数点数 (12MB)
  - 10,000超の実行で時間を平均
- 12,000ブロック×256スレッド
  - 356µs – 結合
  - 357µs – 結合されるが一部のスレッドは参加しない
  - 3,494µs – 順序変更や位置ずれのあるスレッドアクセス

実践: 配列の反転

- メモリの結合の限界を考察し、実装でのデータアクセスパターンを分析する
- データアクセスパターンを向上させるためにできることは何か？

非結合のfloat3コード

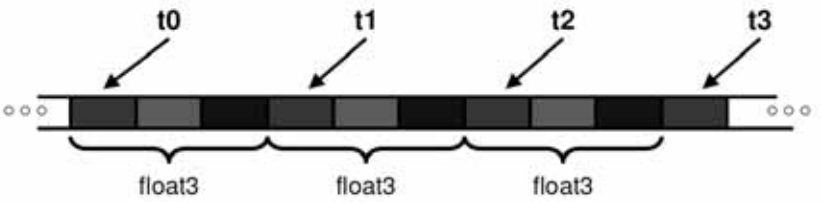
```
__global__ void accessFloat3(float3 *d_in, float3 d_out)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    float3 a = d_in[index];

    a.x += 2;
    a.y += 2;
    a.z += 2;

    d_out[index] = a;
}
```

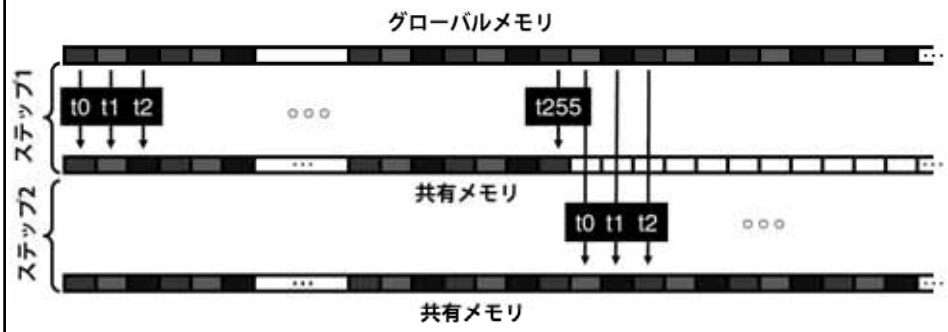
非結合アクセス: float3の場合

- float3は12バイト
- 各スレッドは3つの読み込みを実行する
  - sizeof(float3)は4、8、16ではない
  - ハーフワープは3つの64Bの非連続領域を読み込む



最初の読み込み

Float3アクセスの結合



同様に、ステップ3では512のオフセットから開始

結合アクセス: float3の場合

- 共有メモリを使用して結合を可能にする
  - sizeof(float3)*(スレッド数/ブロック)バイトの共有メモリが必要
  - 各スレッドはスカラの浮動小数点数を3つ読み込む
    - オフセット: 0、(スレッド数/ブロック)、2*(スレッド数/ブロック)
    - 他のスレッドで処理される可能性が高いため、同期
- 処理
  - 各スレッドは共有メモリ配列からfloat3を取得
    - 共有メモリのポインタを(float3*)にキャスト
    - スレッドIDをインデックスとして使用
  - 残りの計算コードは変更なし！

結合されたfloat3コード

```
__global__ void accessInt3Shared(float *g_in, float *g_out)
{
  int index = 3 * blockIdx.x * blockDim.x + threadIdx.x;
  __shared__ float s_data[256*3];
  s_data[threadIdx.x] = g_in[index];
  s_data[threadIdx.x+256] = g_in[index+256];
  s_data[threadIdx.x+512] = g_in[index+512];
  __syncthreads();
  float3 a = ((float3*)s_data)[threadIdx.x];

  a.x += 2;
  a.y += 2;
  a.z += 2;

  ((float3*)s_data)[threadIdx.x] = a;
  __syncthreads();
  g_out[index] = s_data[threadIdx.x];
  g_out[index+256] = s_data[threadIdx.x+256];
  g_out[index+512] = s_data[threadIdx.x+512];
}
```

共有メモリから入力を読み込む

計算コードは変更なし

共有メモリから結果を書き込む

結合: 時間測定の結果

- 実験
  - カーネル: floatを読み込んで増分し、ライトバックする
  - 3Mの浮動小数点数(12MB)
  - 10,000超の実行で時間を平均
- 12,000ブロック×256スレッドでfloatを読み込む
  - 356µs – 結合
  - 357µs – 結合されるが一部のスレッドは参加しない
  - 3,494µs – 順序変更や位置のずれのあるスレッドアクセス
- 4,000ブロック×256スレッドでfloat3を読み込む
  - 3,302µs – float3は結合されない
  - 359µs – float3は共有メモリを介して結合される

結合:
構造体のサイズが4、8、16バイト以外

- AoS (Array of Structure: 構造体の配列) ではなくSoA (Structure of Array: 配列の構造体) を使用する
- SoAを使用できない場合
  - 強制的に構造体を位置合わせする: __align(X)、ただしX = 4、8、16とする
  - 共有メモリを使用して結合を実現する

結合: まとめ

- 結合により、スループットは大幅に向上する
- メモリの制限があるカーネルでは重要
- サイズが4、8、16バイト以外の構造体を読み込むと結合が行われる
  - AoSでなくSoAを使用する
  - SoAを使用できない場合は共有メモリから読み書きする
- その他の情報:
  - SDKサンプルの「Aligned Types」

プロファイラの信号

- イベントはチップ上の信号でハードウェアカウンタによって追跡される
  - timestamp
  - gld_incoherent
  - gld_coherent
  - gst_incoherent
  - gst_coherent

    グローバルメモリのロード／保存は結合（コヒーレント）されるまたは結合されない（インコヒーレント）
  - local_load
  - local_store

    ローカルでロード／保存
  - branch
  - divergent_branch

    合計のブランチと分岐のブランチはスレッドで取得される
  - instructions – 命令のカウント
  - warp_serialize – 共有メモリまたはコンスタントメモリと競合するアドレスでシリアル化するスレッドワープ
  - cta_launched – 実行されたスレッドブロック

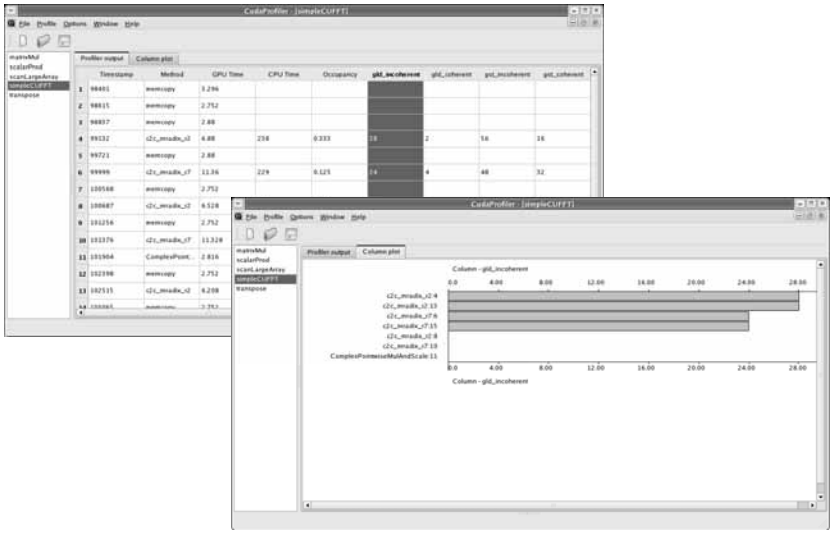
プロファイラの制御

- CUDA_PROFILE : 1または0に設定してプロファイラの有効／無効を切り替える
- CUDA_PROFILE_LOG : ログファイルの名前を設定する（デフォルトは./cuda_profile.log）
- CUDA_PROFILE_CSV : 1または0に設定して、ログのカンマ区切りバージョンの有効／無効を切り替える
- CUDA_PROFILE_CONFIG : 最大4つの信号でconfigファイルを指定する

プロファイラカウンタの解釈

- 値はスレッドワープ内のイベントを表す
- 1つのマルチプロセッサのみをターゲットにする
  - 値は特定のカーネルで起動されたワープの合計数とは一致しない
  - ターゲットのマルチプロセッサが合計の作業に対して一貫した割合を割り振られるよう、十分な数のスレッドブロックを起動する
- 最適化されていないコードと最適化されたコードの相対的なパフォーマンスの違いを識別するためには、値を使用するのが一番
  - 例: 非結合のロードの数を0以外の値から0にするなど

Visual Profiler



実践: プロファイラの使用

- プロファイラ (コマンドラインファイル、テキスト構成ファイル、ビジュアルインターフェースのいずれか) を使用して、インプレースの配列反転の実装でのデータアクセスパターンの分析を確認する

共有メモリ

- グローバルメモリよりも数百倍高速
- データをキャッシュしてグローバルメモリへのアクセスを低減する
- スレッドは共有メモリを介して協調できる
- 非結合アクセスを避けるために使用する
  - 共有メモリでロードと保存を実行し、非結合アドレス指定を再度順序付ける
  - SDKサンプルの「Matrix Transpose」を参照

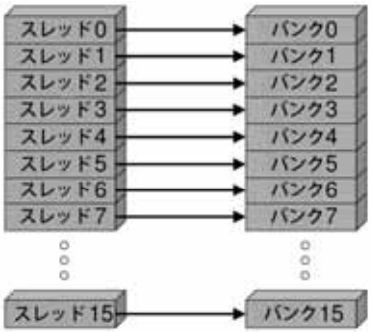
並列メモリアーキテクチャ

- 多数のスレッドによるメモリのアクセス
  - そのためメモリはバンクに分割される
  - 広い帯域幅を獲得するために必須
- 各バンクは1サイクルあたり1つのアドレスを提供
  - メモリはバンクと同じ数の同時アクセスが可能
- バンクへの複数の同時アクセスではバンクの競合が発生
  - 競合するアクセスはシリアル化される

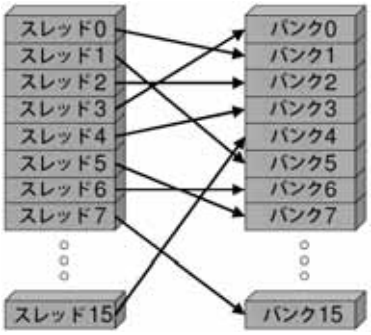
バンクアドレス指定の例

- バンクの競合なし
  - 線形アドレス指定

    stride == 1



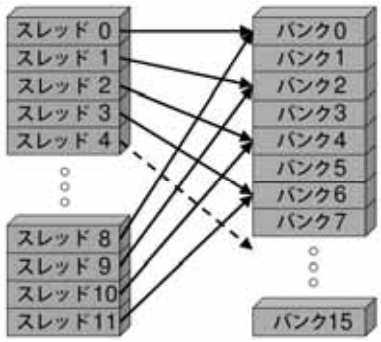
- バンクの競合なし
  - ランダムな1:1順列



バンクアドレス指定の例

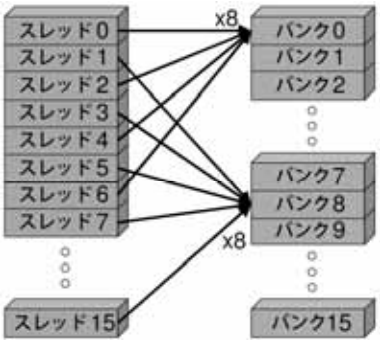
- 2wayのバンク競合
  - 線形アドレス指定

    stride == 2



- 8wayのバンク競合
  - 線形アドレス指定

    stride == 8



共有メモリのバンク競合

- バンクの競合がなければ、共有メモリは最も高速なレジスタ
- SDKのバンクチェックマクロを使用して競合をチェックする
  - 通常、競合のチェックにはwarp_serialize信号を使用
- 速い場合
  - ハーフワープのすべてのスレッドが別のバンクにアクセスしている場合はバンクの競合はない
  - ハーフワープのすべてのスレッドが異なるアドレスを読み込む場合はバンクの競合はない (ブロードキャスト)
- 遅い場合
  - バンクの競合: 同じハーフワープの複数のスレッドが同じバングにアクセスしている
  - アクセスをシリアル化する必要がある
  - コスト=単一のバンクに同時アクセスする最大数

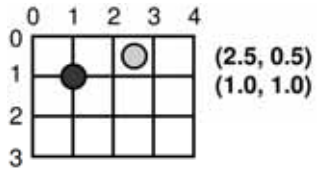
実践: 配列の反転パフォーマンス

- 配列の反転コードを改良し、共有メモリの使用によって結合されたロードとストアで、グローバルメモリにアクセスする

CUDAのテクスチャ

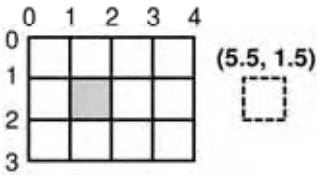
- テクスチャはデータを**読み込む**ためのオブジェクト
- 利点
  - データがキャッシュされる（2次元の局所性に最適）
    - 結合が問題になるときに役立つ
  - フィルタリング
    - 線形／双線形／三線形
    - 専用ハードウェア
  - ラップモード（「境界外部」のアドレス）
    - エッジにクランプまたは繰り返し
  - 1次元、2次元、3次元でアドレス指定可能
    - 整数または正規化されていない座標を使用
- 使用法
  - CPUコードでデータをテクスチャオブジェクトにバインドする
  - カーネルでfetch関数を呼び出してデータを読み込む

テクスチャのアドレス指定



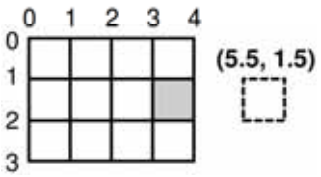
ラップ

- 境界外部の座標は折り返される（モジュロ演算）



クランプ

- 境界外部の座標は最も近い境界に置き換えられる



2種類のCUDAテクスチャ

- 線形メモリにバインド
  - グローバルメモリアドレスがテクスチャにバインドされる
  - 1次元のみ
  - 整数のアドレス指定
  - フィルタリングとアドレス指定のモードはなし
- CUDA配列にバインド
  - CUDA配列がテクスチャにバインドされる
  - 1次元、2次元、3次元
  - floatのアドレス指定（サイズベースまたは標準化）
  - フィルタリング
  - アドレス指定モード（クランプ／繰り返し）
- 両方
  - 要素型または標準化されたfloatを返す

CUDAにおけるテクスチャリングのステップ

- ホスト（CPU）コード
  - メモリの割り当て／確保を実行する（グローバルの線形またはCUDA配列）
  - テクスチャ参照オブジェクトを作成する
    - 現在はフルスコープでなくてはならない
  - テクスチャの参照をメモリ／配列にバインドする
  - 終了後
    - テクスチャの参照をアンバインドしてリソースを解放する
- デバイス（カーネル）コード
  - テクスチャ参照を使用してフェッチする
  - 線形のメモリテクスチャ
    - tex1Dfetch()
  - 配列のテクスチャ
    - tex1D()、tex2D()、tex3D()

実行構成の最適化

占有率

- スレッド命令は逐次実行されるので、レイテンシを隠してハードウェアを使用中にしておくための唯一の方法は、他のワープを実行すること
- 占有率=マルチプロセッサで同時に実行されているのワープの数を同時に実行できるワープの最大数で除算したもの
- 以下のリソースの使用率で制限される
  - レジスタ
  - 共有メモリ

グリッド/ブロックサイズの経験則

- ブロックの数 > マルチプロセッサの数
  - そのためすべてのマルチプロセッサは1つ以上のブロックを実行できる
- ブロックの数 / マルチプロセッサの数> 2
  - 1つのマルチプロセッサで複数のブロックを同時に実行できる
  - __syncthreads()で待機しないブロックはハードウェアをビジーにする
  - レジスタ、共有メモリなどのリソースの可用性の影響を受ける
- ブロックの数 > 100から将来のデバイスに応じて拡張
  - ブロックはパイプライン形式で実行される
  - 数世代で1グリッドあたり1,000ブロックに拡張

レジスタの従属性

- RAW (Read-After-Write) レジスタの従属関係
  - 命令の結果は11サイクル以内に読み込まれる
  - シナリオ

CUDA:

```
x = y + 5;
z = x + 3;
```

PTX:

```
add.f32   $f3, $f1, $f2
add.f32   $f5, $f3, $f4
```

```
s_data[0] += 3;
```

```
ld.shared.f32   $f3, [$r31+0]
add.f32         $f3, $f3, $f4
```

- レイテンシを完全に隠すためには
  - マルチプロセッサあたり、最低192スレッド (6ワープ) を実行する
    - 25%以上の占有率
  - スレッドは同じスレッドブロックに属す必要はない

レジスタの圧力

- SMあたりのスレッドを多くしてレイテンシを隠す
- 制限要因
  - カーネルあたりのレジスタ数
    - SMあたり8,192個。平行スレッドに分割される
  - 共有メモリの量
    - SMあたり16KB。平行スレッドブロックに分割される
- 「–ptxas-options=-v」フラグでコンパイル
- ncvvで「–maxrregcount=N」フラグを使用する
  - N = カーネルあたりに必要とされる最大レジスタ数
  - ある地点でLMEMへの「流出」が発生する可能性がある
    - パフォーマンスの低下- LMEMは低速

リソースの使用量の測定

- カーネルコードを「-cubin」フラグでコンパイルしてレジスタの使用量を測定する
- .cubinファイルを開いて「code」セクションを探す

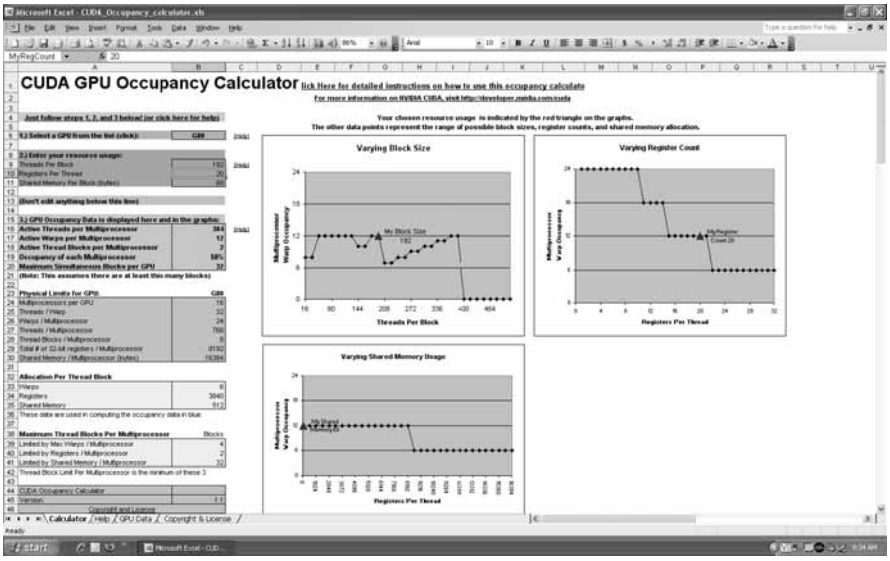
```
architecture {sm_10}
abiversion {0}
modname {cubin}
code {
        name = BlackScholesGPU
        lmem = 0
        smem = 68
        reg = 20
        bar = 0
        bincode {
                0xa0004205 0x04200780 0x40024c09 0x00200780
                …
```

スレッドあたりのローカルメモリ

スレッドブロックあたりの共有メモリ

スレッドあたりのレジスタ

CUDA Occupancy Calculator



ブロックあたりのスレッドの最適化

- ブロックあたりのスレッドをワープサイズの倍数で選択する
  - 最適でないワープを扱うことによる無駄な計算を避ける
- 1ブロックあたりのスレッド数の増加==メモリのレイテンシがよりよく隠れる
- 反面、1ブロックあたりのスレッド数の増加==1スレッドあたりのレジスタが減少する
  - 多くのレジスタが使用されているとカーネルの呼び出しに失敗する可能性がある
- 経験則
  - 最小: 1ブロックあたり64スレッド
    - 複数の平行ブロックがある場合のみ
  - 192または256スレッドが最適
    - 通常は、これでもコンパイルと呼び出しに十分なレジスタが使用可
  - 計算によって異なるため、実験が必要

占有率 != パフォーマンス

- 占有率を上げてもパフォーマンスが向上するとは限らない

しかし

- 占有率の低いマルチプロセッサは、メモリにバインドされたカーネルのレイテンシをうまく隠せない
  - (すべてが計算集約度と並列の可用性に関係します)

アプリケーションのパラメータ化

- パラメータ化はさまざまなGPUへの適用に役立つ
- GPUは多くの点で異なる
  - マルチプロセッサの数
  - メモリの帯域幅
  - 共有メモリのサイズ
  - レジスタファイルのサイズ
  - 1ブロックあたりの最大スレッド数
- 自己調整アプリケーションを作成することもできる (例: FFTW、ATLASなど)
  - 「Experiment」モードではオプションの構成を発見して保存する

命令の最適化

CUDA命令のパフォーマンス

- 命令サイクル（1ワープあたり）は、以下の合計
  - オペランドの読み込みサイクル
  - 命令実行サイクル
  - 結果の更新サイクル
- そのため、命令のスループットは以下の要因で決定される
  - 名目上の命令のスループット
  - メモリレイテンシ
  - メモリの帯域幅
- 「サイクル」はマルチプロセッサのクロック速度
  - 例: Tesla C870では1.35 GHz

命令のスループットの最大化

- 帯域幅の広いメモリを最大限に使用する
  - 共有メモリを最大限に使用する
  - グローバルメモリへのアクセスを最小限に抑える
  - グローバルメモリへアクセスを最大限に結合する
- ハードウェア計算でメモリアクセスをオーバーラップしてパフォーマンスを最適化する
  - 計算集約度の高いプログラム
    - すなわち、メモリトランザクションに対して計算の割合が高い
  - 多くの平行スレッド

演算命令のスループット

- intとfloatのadd、shift、min、maxおよびfloatのmul、mad: 1ワープあたり4サイクル
  - int multiply (*)はデフォルトで32ビット
    - 1ワープあたり複数のサイクルが必要
  - 4サイクル、24ビットのint multiplyでは__mul24() / __umul24()命令を使用
- 整数の除算とモジュロはさらに多くのメモリが必要
  - コンパイラは定数の2の累乗の除算をシフトに変換する
    - しかし見落とされる場合もある
  - コンパイラによって除数が2の乗数であることを通知できない場合は、明示で指定する
  - 便利なテクニック: nが2の累乗の場合 foo % n == foo & (n-1)が成立する

演算命令のスループット

- 「__」のプレフィックス付きの組み込みの逆数、逆数平方根、sin／cos、対数、指数は、1ワープあたり16サイクル
  - 例: __rcp()、__sin()、__exp()
- その他の関数は上の組み合わせ
  - y / x == rcp(x) * yは1ワープあたり20サイクル
  - sqrt(x) == x * rsqrt(x)は1ワープあたり20サイクル

ランタイム計算ライブラリ

- 実行時の計算操作には2種類ある
  - __func(): ハードウェアISAに直接マッピング
    - 速度は速いが精度は下がる（詳細については『プログラミングガイド』を参照）
    - 例: __sin(x), __exp(x), __pow(x,y)
  - func() : 複数の命令にコンパイル
    - 速度は低いが精度は高い（5ulp以下）
    - 例: sin(x), exp(x), pow(x,y)
- -use_fast_mathコンパイラオプションを使用すると、すべてのfunc()が__func()にコンパイルされる

GPUの結果はCPUと一致しない場合もある

- 多くの可変要素: ハードウェア、コンパイラ、最適化の設定
- CPU演算は厳密には0.5ulpに制限されない
  - 80ビットの拡張精度の論理演算装置により、演算の順序はより正確
- 浮動小数点数の演算は結合しない！

浮動小数点数の演算は結合しない

- 象徴的な演算(x+y)+z == x+(y+z)
- 浮動小数点数の加算では必ずしもあてはまらない
  - $x = 10^{30}$、$y = -10^{30}$、$z = 1$で上の式を実行
- 並列計算では、内部的に操作の順序が変更される可能性がある
- 並列の結果は逐次演算の結果と一致しない場合がある
  - これはGPUまたはCUDAに固有なのではなく並行の実行の特性である

浮動小数点数の特性

	G8x	SSE	IBMAltivec	Cell SPE
Format	IEEE 754	IEEE 754	IEEE 754	IEEE 754
FADDとFMULの丸めモード	近い値と0に丸める	近似、0、inf、-infに丸める	近似に丸めるのみ	0に丸めると切り捨てのみ
非正規化処理	0にクリア	サポート。1,000サイクル以上	サポート。1,000サイクル以上	0にクリア
NaNサポート	あり	あり	あり	なし
オーバーフローと無限のサポート	あり。最大ノルムにクラプ	あり	あり	なし、無限
フラグ	なし	あり	あり	一部
平方根	ソフトウェアのみ	ハードウェア	ソフトウェアのみ	ソフトウェアのみ
除算	ソフトウェアのみ	ハードウェア	ソフトウェアのみ	ソフトウェアのみ
逆数の推定精度	24ビット	12ビット	12ビット	12ビット
逆数の推定精度	23ビット	12ビット	12ビット	12ビット
log2(x)と2^xの推定精度	23ビット	なし	12ビット	なし

G8xとIEEE 754の違い

- 加算と乗算はIEEEに準拠
  - 最大0.5 ulpの誤差
- ただし多くの場合に積和演算 (FMAD) に結合される
  - 中間の結果は切り捨てられる
- 除算は準拠しない (2 ulp)
- すべての丸めモードがサポートされるわけではない
- 非正規化処理はサポートされない
- 浮動小数点数の例外を検出するメカニズムはない

floatセーフなプログラムを設計すること！

- 将来のハードウェアは倍精度をサポート
  - G8xは単精度のみ
  - 倍精度ではコストが増える
- 必要がない場所では倍精度の使用を避け、floatセーフにすることが重要
  - floatのリテラルに「f」指定子を追加
    - `foo = bar * 0.123;` // 倍制度と認識される
    - `foo = bar * 0.123f;` // floatを明示
  - 標準のライブラリ関数のfloatバージョンを使用する
    - `foo = sin(bar);` // 倍精度と認識される
    - `foo = sinf(bar);` // floatを明示

フロー命令の制御

- 分岐におけるパフォーマンス上の主な懸念事項は分散化
  - 1つのワープ内のスレッドは異なる経路を通る
  - 異なる実行経路はシリアル化する必要がある
- 分岐条件がスレッドIDの関数の場合は分散化を避ける
  - 分散化の例
    - `if (threadIdx.x > 2) { }`
    - 分岐の粒度＜ワープサイズ
  - 分散なしの例
    - `if (threadIdx.x / WARP_SIZE > 2) { }`
    - 分岐の粒度はワープサイズ全体の倍数

まとめ

- 以下の簡単なガイドラインに従えば、GPUハードウェアはデータの並列計算で優れたパフォーマンスを発揮できる
  - 並列性を効率よく使用する
  - 可能な限りメモリアクセスを結合する
  - 共有メモリを活用する
  - 他のメモリ空間を活用する
    - テクスチャ
    - コンスタント
  - バンクの競合を低減する

CUDAのライブラリ

概略

- CUDAには、広い用途に使用できる2つのライブラリがある
  - CUBLAS: BLASの実装
  - CUFFT:　FFTの実装

CUBLAS

- CUDAドライバ上でのBLAS（Basic Linear Algebra Subprograms: 線形代数の基本サブルーチン）の実装
  - APIレベルで内蔵され、CUDAドライバとの直接的なやり取りは発生しない
- 基本的な使用モデル
  - GPUメモリ空間で行列オブジェクトとベクトルオブジェクトを作成する
  - オブジェクトにデータを挿入する
  - 一連のCUBLAS関数を呼び出す
  - GPUからデータを取得する
- CUBLASライブラリのヘルパー関数
  - GPU空間でのオブジェクトの作成と破棄
  - オブジェクトへのデータの読み書きとオブジェクトからのデータの取得

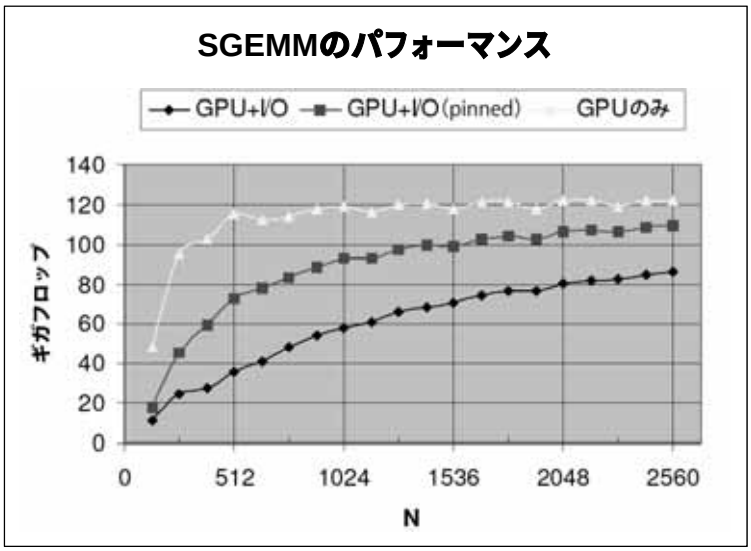
サポートされる機能

- 単精度のBLAS関数
  - 実データ
    - レベル1 (ベクトル-ベクトル O(N))
    - レベル2 (行列-ベクトル $O(N^2)$)
    - レベル3 (行列-行列 $O(N^3)$)
  - 複素数データ
    - レベル1
    - CGEMM
- BLASの規則に従い、CUBLASは列優先のストレージを使用

CUBLASの使用

- CUBLASライブラリのインターフェースはcublas.h
- 関数の命名規則
  - cublas + BLAS名
  - cublasSGEMMなど
- エラー処理
  - CUBLASコア関数はエラーを返さない
    - CUBLASは記録されている最後のエラーを取得する関数を提供する
  - CUBLASヘルパー関数はエラーを返す
- CベースのCUDAツールチェインを使用して実装される
  - C／C++アプリケーションとの連結は簡単

CUBLASのパフォーマンス



cublasInit()、cublasShutdown()

- cublasStatus cublasInit()
  - CUBLASライブラリを初期化する
  - GPUにアクセスするために必要なハードウェアリソースを割り当てる
  - 他のCUBLAS API関数よりも前に呼び出す必要がある
- cublasStatus cublasShutdown()
  - CUBLASライブラリで使用されるCPU側リソースを解放する
  - GPU側リソースの解放は、アプリケーションが終了するまで先送りされる

cublasGetError()、cublasAlloc()、cublasFree()

- cublasStatus cublasGetError()
  - **CUBLASコア関数で発生した、最後のエラーを返す**
  - **内部エラー状態を**CUBLAS_STATE_SUCCESS**にリセットする**
- cublasStatus cublasAlloc(int n, int elemSize,
  Void **devPtr)
  - **要素が**n**個の配列のオブジェクトをGPUメモリに作成する**
  - **各要素で**elemSize**バイトのストレージが必要**
  - cudaMalloc()**のラッパーなので**devPtr**を使用できる**
- cublasStatus cublasFree(const void *devPtr)
  - **GPU空間中の**devPtr**のメモリ領域を破棄する**

cublasSetVector()、cublasGetVector()

- cublasStatus cublasSetVector(int n, int elemSize, const void *x,
  int incx, void *y, int incy)
  - **CPUメモリ空間のベクトル**x**から**n**個の要素をGPUメモリ空間のベクトル**y**にコピーする**
  - **各要素は**elemSize**バイトを占める**
  - **配列**x**と**y**の連続する要素間のストレージの間隔は、それぞれ**incx**と**incy
- cublasStatus cublasGetVector(int n, int elemSize, const void *x,
  int incx, void *y, int incy)
  - **GPUメモリ空間のベクトル**x**から**n**個の要素をCPUメモリ空間のベクトル**y**にコピーする**

cublasSetMatrix()、cublasGetMatrix()

- cublasStatus cublasSetMatrix(int rows, int cols, int elemSize,
  const void *A, int lda, void *B, int ldb)
  - **CPUメモリ空間の行列**Aから**rows*cols個の要素のタイルをGPUメモリ空間の行列**Bにコピーする
  - **各要素は**elemSize**バイトを占める**
  - **どちらの行列も列優先形式で保存され、行列Aの第一列を**lda**、行列Bの第一列を**ldb**とする**
- cublasStatus cublasGetMatrix(int rows, int cols, int elemSize,
  const void *A, int lda, void *B, int ldb)
  - **GPUメモリ空間の行列**Aから**rows*cols個の要素のタイルをCPUメモリ空間の行列**Bにコピーする

FORTRANからのCUBLASの呼び出し

- Fortranからの呼び出し規則は標準化されておらず、プラットフォームとツールチェインによって異なる違いの例
  - シンボル名（大文字小文字、名前の装飾）
  - 引数の渡し方（値渡しまたは参照渡し）
  - 文字列引数の渡し方（長さの情報）
  - ポインタ引数の渡し方（ポインタのサイズ）
  - 浮動小数点数または複合データ型（単精度や複素数のデータ型など）を返す
- CUBLASはユーザーが選択したツールチェインでのコンパイルに必要な、ラッパー関数を提供する（fortran.cファイル）
  - 特定のプラットフォームとツールチェインで必要な変更を可能にするソースコードを提供

FORTRANからのCUBLASの呼び出し

- 2つのインターフェース
  - サンク (fortran.cのコンパイルでCUBLAS_USE_THUNKINGを定義)
    - 変更なしに、既存のアプリケーションとのインタフェースが可能
    - 各呼び出し中にラッパーがGPUメモリを割り当て、CPUメモリ空間からGPUメモリ空間にソースデータをコピーし、CUBLASを呼び出し、結果をCPUメモリ空間にコピーバックすると同時にGPGPUメモリを解放する
    - 呼び出しのオーバーヘッドのための軽量のテスト用途
  - サンクなし (デフォルト)
    - 本稼動コード用途
    - すべてのBLAS関数で、ベクトルと行列の引数のデバイスポインタを置き換える
    - GPGPUメモリ空間でデータ構造を割り当ておよび解放し (CUBLAS_ALLOCとCUBLAS_FREEを使用)、GPUメモリ空間とCPUメモリ空間の間でデータをコピー (CUBLAS_SET_VECTOR、CUBLAS_GET_VECTOR、CUBLAS_SET_MATRIX、CUBLAS_GET_MATRIXを使用) するには既存のアプリケーションを若干変更する必要がある

FORTRAN 77コードの例

```
program matrixmod
implicit none
integer M, N
parameter (M=6, N=5)
real*4 a(M,N)
integer i, j

do j = 1, N
 do i = 1, M
   a(i,j) = (i-1) * M + j
 enddo
enddo

call modify (a, M, N, 2, 3, 16.0, 12.0)

do j = 1, N
 do i = 1, M
   write(*,"(F7.0$)") a(i,j)
 enddo
 write (*,*) ""
enddo

stop
end
```

```
subroutine modify (m, ldm, n, p, q, alpha, beta)
implicit none
integer ldm, n, p, q
real*4 m(ldm,*), alpha, beta

external sscal

call sscal (n-p+1, alpha, m(p,q), ldm)

call sscal (ldm-p+1, beta, m(p,q), 1)

return
end
```

FORTRAN 77コードの例: サンクなしインターフェース

```
program matrixmod
implicit none
integer M, N, sizeof_real, devPtrA
parameter (M=6, N=5, sizeof_real=4)
real*4 a(M,N)
integer i, j, stat
external cublas_init, cublas_set_matrix,cublas_get_matrix
external cublas_shutdown, cublas_alloc
integer cublas_alloc

do j = 1, N
 do i = 1, M
   a(i,j) = (i-1) * M + j
 enddo
enddo

call cublas_init
stat = cublas_alloc(M*N, sizeof_real, devPtrA)
if (stat .NE. 0) then
 write(*,*) "device memory allocation failed"
 stop
endif

call cublas_set_matrix (M, N, sizeof_real, a, M, devPtrA, M)
call modify (devPtrA, M, N, 2, 3, 16.0, 12.0)
call cublas_get_matrix (M, N, sizeof_real, devPtrA, M, a, M)
call cublas_free(devPtrA)
call cublas_shutdown
```

```
do j = 1, N
 do i = 1, M
   write(*,"(F7.0$)") a(i,j)
 enddo
 write (*,*) ""
enddo

stop
end

#define IDX2F(i,j,ld) ((((j)-1)*(ld))+((i)-1)

subroutine modify (devPtrM, ldm, n, p, q, alpha, beta)
implicit none
integer ldm, n, p, q
integer sizeof_real, devPtrM
parameter (sizeof_real=4)
real*4 alpha, beta
call cublas_sscal (n-p+1, alpha,
                   devPtrM+IDX2F(p,q,ldm)*sizeof_
                   real,ldm)
call cublas_sscal (ldm-p+1, beta,
                   devPtrM+IDX2F(p,q,ldm)*sizeof_
                   real,1)
return
end
```

固定形式を使用する場合は行の長さが72列以下に制限されることに注意！！

CUFFT

- FFT (Fast Fourier Transform: **高速フーリエ変換**) **は、複素数のデータセットまたは実数のデータセットの個別のフーリエ変換を効率よく計算する分割統治法アルゴリズム**
- CUFFT**は**CUDA FFT**ライブラリ**
  - NVIDIA GPU**上で並列**FFT**計算を行うためのシンプルなインターフェースを提供する**
  - **カスタムの**GPU**ベースの**FFT**の実装を開発する必要なく、強力な浮動小数点数と**GPU**の並列性を活用できる**

サポートされる機能

- 複素数データと実数データの1次元、2次元、3次元変換
- 複数の1次元の変換を並列で実行するバッチ
- 1次元の変換サイズは最大8M要素
- 2Dと3Dの変換サイズの範囲は[2,16384]
- 実数データと複素数データのインプレースおよびアウトプレース変換

CUFFT型と定義

- cufftHandle
  - CUFFTプランの保存とアクセスに使用されるハンドル型
- cufftResults
  - API関数戻り値の列挙体
  - CUFFT_SUCCESS、CUFFT_INVALID_PLANなど

変換の種類

- ライブラリは実数または複素数の変換をサポート
  - CUFFT_C2C、CUFFT_C2R、CUFFT_R2C
- 定義
  - CUFFT_FORWARD (-1)とCUFFT_BACKWARD (1)
    - 複素数の指数項の符号によって決定
- 複素数の入力および出力で実数と虚数の部分はインターリーブされる
  - cufftComplex型はこのための定義
- 実数から複素数へのFFTでは、出力配列は非冗長の係数のみを維持
  - N -> N/2+1
  - N0 × N1 × … × Nn -> N0 × N1 × … × (Nn/2+1)
  - インプレース変換では入力／出力配列でパディングが必要

変換の詳細

- 2Dおよび3Dの変換では、CUFFTは行優先で変換を実行する（C順序）
- FORTRANまたはMATLABからの呼び出しでは、プラン作成中にサイズパラメータの順序を変更することに注意
- CUFFTは非正規化の変換を実行する

  IFFT(FFT(A))= length(A)*A
- CUFFT APIはFFTWの後にモデルされる特定のサイズのFFTを実行するオプションの構成を完全に指定する、プランに基づく
- プランを作成した後は、ライブラリは構成を再コンパイルせずに複数回プランを実行するために、必要な状態を保存する
  - 別の種類のFFTは別のスレッド構成とGPUリソースを必要とするため、CUFFTで適切に動作する

cufftPlan1d()

cufftResult cufftPlan1d(cufftHandle *plan, int nx, cufftType type, int batch)

- **指定された信号サイズとデータ型で、1次元FFTのプラン構成を作成する**
- batch**入力パラメータによって、構成する1次元変換の数をCUFFTに通知する**

- **入力**

plan	**cufftHandleオブジェクトのポインタ**
nx	**変換サイズ (256ポイントのFFTでは256)**
type	**変換データ型 (CUFFT_C2Cなど)**
batch	**サイズnxの変換数**

- **出力**

plan	**CUFFT 1次元プランのハンドル値を格納**

cufftPlan2d()

cufftResult cufftPlan2d(cufftHandle *plan, int nx, int ny, cufftType type)

- **指定された信号サイズとデータ型で2次元FFTのプラン構成を作成する**

- **入力**

plan	**cufftHandleオブジェクトのポインタ**
nx	**X方向の変換サイズ**
ny	**Y方向の変換サイズ**
type	**変換データ型 (CUFFT_C2Cなど)**

- **出力**

plan	**CUFFT 2次元プランハンドル値を格納**

cufftPlan3d()

cufftResult cufftPlan3d(cufftHandle *plan, int nx, int ny, int nz, cufftType type)

- **指定された信号サイズとデータ型で3次元FFTのプラン構成を作成する**
- **入力**
  - plan **cufftHandleオブジェクトのポインタ**
  - nx **X方向の変換サイズ**
  - ny **Y方向の変換サイズ**
  - nz **Z方向の変換サイズ**
  - type **変換データ型 (CUFFT_C2Cなど)**
- **出力**
  - plan **CUFFT 3次元プランハンドル値を格納**

cufftDestroy()

cufftResult cufftDestroy(cufftHandle plan)

- **CUFFTプランに関連付けられているすべてのGPUリソースを解放して、内部のプランデータ構造を破棄する**
- **GPUメモリを無駄に消費しないために、プランが必要なくなったら呼び出す必要がある**
- **入力**
  - plan **cufftHandleオブジェクト**

cufftExecC2C()

cufftResult cufftExecC2C(cufftHandle plan, cufftComplex *idata,
cufftComplex *odata, int direction)

- **CUFFT複素数から複素数の変換プランを実行する**
- idata**パラメータでポイントされているGPUメモリを入力データに使用する**
- **フーリエ係数を**odata**配列に保存する**
  - idata**と**odata**が同じ場合はインプレース変換を実行する**

- **入力**
  - plan **cufftHandleオブジェクト**
  - idata **変換する入力データのポインタ (GPUメモリ内)**
  - odata **出力データのポインタ (GPUメモリ内)**
  - direction **変換の方向 (**CUFFT_FORWARD**または**CUFFT_BACKWARD**)**
- **出力**
  - odata **複素数のフーリエ係数を格納する**

cufftExecR2C()

cufftResult cufftExecR2C(cufftHandle plan, cufftReal *idata,
cufftComplex *odata)

- **CUFFT実数から複素数の変換プランを実行する**
- idata**パラメータでポイントされているGPUメモリを入力データに使用する**
- **非冗長のフーリエ係数を**odata**配列に保存する**
  - idata**と**odata**が同じ場合はインプレース変換を実行する**

- **入力**
  - plan **cufftHandleオブジェクト**
  - idata **変換する入力データのポインタ (GPUメモリ内)**
  - odata **出力データのポインタ (GPUメモリ内)**
- **出力**
  - odata **複素数のフーリエ係数を格納する**

cufftExecC2R()

cufftResult cufftExecC2R(cufftHandle plan, cufftReal *idata,
cufftComplex *odata)

- **CUFFT複素数から実数の変換プランを実行する**
- idata**パラメータでポイントされているGPUメモリを入力データに使用する**
  - idata**には非冗長の複素数フーリエ係数のみを格納する**
- **実数の出力データを**odata**配列に保存する**
  - idata**と**odata**が同じ場合はインプレース変換を実行する**

- **入力**
  - plan **cufftHandleオブジェクト**
  - idata **変換する複素数の入力データのポインタ (GPUメモリ内)**
  - odata **実数の出力データのポインタ (GPUメモリ内)**
- **出力**
  - odata **実数の出力データを格納する**

精度とパフォーマンス

CUFFTライブラリは複数のFFTアルゴリズムを実装し、各アルゴリズムでパフォーマンスと精度が異なる

最適なパフォーマンスのパスは、以下の変換サイズに一致する

1. CUDAの共有メモリに適合
2. 1つの因数の累乗 (2の累乗など)

条件1だけが満たされた場合、CUFFTは汎用的な混合基数アルゴリズムを使用。速度が遅くなり、数値上の精度が下がる

どちらの条件も満たされない場合、CUFFTは中間的な結果をグローバルのGPUメモリに保存する、アウトプレースの混合基数アルゴリズムを使用する

注目すべき例外の1つは、長精度の1次元変換において、CUFFTは2次元FFTを使用する1次元FFTを実行する、分散アルゴリズムを使用すること

CUFFTは実数の専用アルゴリズムを実装しておらず、複素数から複素数の代わりに実数から複素数 (または複素数から実数) を使用した際にもパフォーマンス上の直接的な利点はないこのリリースでは実数のAPIは主に便宜上の目的で存在する。

コードの例:
1次元複素数から複素数への変換

```
#define NX 256
#define BATCH 10

cufftHandle plan;
cufftComplex *data;
cudaMalloc((void**)&data, sizeof(cufftComplex)*NX*BATCH);
…
/* 1次元FFTのプランを作成 */
 cufftPlan1d(&plan, NX, CUFFT_C2C, BATCH);

 /* CUFFTプランを使用して信号をインプレース変換 */
 cufftExecC2C(plan, data, data, CUFFT_FORWARD);

/* 信号をインプレース逆変換 */
 cufftExecC2C(plan, data, data, CUFFT_INVERSE);

/* 注:
  (1) データセット内の要素の数で除算してもとのデータに戻す
  (2) 入力配列と出力配列のポインタが同じ場合はインプレース変換を示す
 */

/* CUFFTプランを破棄 */
 cufftDestroy(plan);

 cudaFree(data);
```


コードの例:
2次元複素数から複素数への変換

```
#define NX 256
#define NY 128

cufftHandle plan;
cufftComplex *idata, *odata;
cudaMalloc((void**)&idata, sizeof(cufftComplex)*NX*NY);
cudaMalloc((void**)&odata, sizeof(cufftComplex)*NX*NY);
…
/* 1次元FFTのプランを作成 */
 cufftPlan2d(&plan, NX,NY, CUFFT_C2C);

 /* CUFFTプランを使用して信号をアウトプレース変換 */
 cufftExecC2C(plan, idata, odata, CUFFT_FORWARD);

/* 信号をインプレース逆変換 */
 cufftExecC2C(plan, odata, odata, CUFFT_INVERSE);

/* 注:
   入力配列と出力配列のポインタが異なる場合はアウトプレース変換を示す
 */

/* CUFFTプランを破棄 */
 cufftDestroy(plan);

 cudaFree(idata), cudaFree(odata);
```

その他のCUDAトピック

概略

- テクスチャ機能
- Fortran相互運用性
- イベントAPI
- デバイス管理
- グラフィックス相互運用性

CUDAのテクスチャ機能

CUDAのテクスチャ

- メモリへの異なるハードウェアパス
- CUDAのテクスチャの利点
  - テクスチャフェッチはキャッシュ可能
    - 2次元の局所性に最適化
  - テクスチャは2次元でアドレス指定可能
    - 整数または正規化された座標の使用
    - コード内でのアドレス計算が減少
  - コストなしでフィルタリングを提供
  - 自由なラップモード（境界条件）
    - エッジにクランプ／繰り返し
- CUDAのテクスチャの制限
  - 読み取り専用
  - 現在は1次元または2次元（3次元は追加予定）
  - 9ビット精度のフィルタ加重

2種類のCUDAテクスチャ

- 線形メモリにバインド
  - グローバルメモリがテクスチャにバインドされる
  - 1次元のみ
  - 整数のアドレス指定
  - フィルタリングとアドレス指定のモードはなし
- CUDA配列にバインド
  - CUDA配列がテクスチャにバインドされる
  - 1次元または2次元
  - floatのアドレス指定（サイズベースまたは正規化）
  - フィルタリング
  - アドレス指定モード（クランプ／繰り返し）
- 両方
  - 要素型または正規化されたfloatを返す

CUDAにおけるテクスチャリングのステップ

- ホスト（CPU）コード
  - メモリの割り当て／確保を実行する（グローバルの線形またはCUDA配列）
  - テクスチャ参照オブジェクトを作成する
    - 現在はフルスコープでなくてはならない
  - テクスチャの参照をメモリ／配列にバインドする
  - 終了後
    - テクスチャの参照をアンバインドしてリソースを解放する
- デバイス（カーネル）コード
  - テクスチャ参照を使用してフェッチする
  - 線形メモリのテクスチャ
    - tex1Dfetch()
  - 配列のテクスチャ
    - tex1D()またはtex2D()

テクスチャの参照

- 不変のパラメータ (コンパイル時)
  - 型: フェッチで返される型
    - 基本的なint、float型
    - CUDAの1要素、2要素、4要素のベクトル
  - 次元
    - 現在は1次元または2次元 (3次元は将来的にサポートされる予定)
  - 読み取りモード
    - cudaReadModeElementType
    - cudaReadModeNormalizedFloat (8ビットまたは16ビットのintで有効)
      - 符号付で[-1,1]、符号なしで[0,1]を返す
- 可変パラメータ (実行時、配列テクスチャのみ)
  - 正規化
    - 0以外=アドレス指定範囲[0, 1]
  - フィルタモード
    - cudaFilterModePoint
    - cudaFilterModeLinear
  - アドレスモード
    - cudaAddressModeClamp
    - cudaAddressModeWrap

例: 線形メモリのホストコード

```
// テクスチャ参照を宣言 (ファイルスコープ内にすること)
texture<unsigned short, 1, cudaReadModeNormalizedFloat> texRef;
        ...

// 線形メモリのセットアップ
unsigned short *dA = 0;
cudaMalloc((void**)&d_A, numbytes);
cudaMemcpy(dA, hA, numBytes, cudaMemcpyHostToDevice);

// テクスチャの参照を配列にバインド
cudaBindTexture(NULL, texRef, dA);
```

cudaArray型

- チャネル形式、幅、高さ
- cudaChannelFormatDesc構造体
  - int x、y、z、w: 各コンポーネントのビット
  - enum cudaChannelFormatKind – 以下のいずれか
    - cudaChannelFormatKindSigned
    - cudaChannelFormatKindUnsigned
    - cudaChannelFormatKindFloat
  - 定義済みのコンストラクタ
    - `cudaCreateChannelDesc<float>(void);`
    - `cudaCreateChannelDesc<float4>(void);`
- 管理関数
  - `cudaMallocArray`、`cudaFreeArray`、`cudaMemcpyToArray`、`cudaMemcpyFromArray`など

例: 2次元配列テクスチャのホストコード

```
// テクスチャ参照を宣言 (ファイルスコープ内にすること)
texture<float, 2, cudaReadModeElementType> texRef;
        ...

// CUDA配列をセットアップ
cudaChannelFormatDesc cf = cudaCreateChannelDesc<float>();
cudaArray *texArray = 0;
cudaMallocArray(&texArray, &cf, dimX, dimY);
cudaMempcyToArray(texArray, 0,0, hA, numBytes, cudaMemcpyHostToDevice);

// 可変テクスチャ参照パラメータを指定
texRef.normalized = 0;
texRef.filterMode = cudaFilterModeLinear;
texRef.addressMode = cudaAddressModeClamp;

// テクスチャの参照を配列にバインド
cudaBindTextureToArray(texRef, texArray);
```

CUDAのテクスチャリングの詳細

- 線形（双線形）のフィルタリング
  - CUDA配列にバインドされるテクスチャのみ
  - floatを返すテクスチャのみ
  - 8ビットまたは16ビットの整数をフィルタリング可能
    - cudaReadModeNormalizedFloatテクスチャ参照
    - フェッチ後にカーネルで値をスケーリング
- 実行時APIとドライバAPI
  - ドライバAPIはhalf float型（16ビット）のストレージが可能
    - フェッチされる値は32ビット
    - 将来はランタイムAPIでサポートされる
- 線形メモリとCUDA配列間でコピー可能

CUDAのFortran相互運用性

Fortranの例

- Fortranからの CUBLASの呼び出し
- Fortranでの pinnedメモリの使用
- Fortranからの CUDAカーネルの呼び出し

SGEMMの例

```
! 3つの単精度の行列A、B、Cを定義
real , dimension(m1,m1):: A, B, C
……
! 初期化
……
#ifdef CUBLAS
 ! サンクインターフェースを使用してCUBLASライブラリのSGEMMを呼び出し
 !（デバイスでのメモリの割り当てとデータの移動を管理するライブラリ）
 call cublas_SGEMM ('n','n',m1,m1,m1,alpha,A,m1,B,m1,beta,C,m1)
#else
! ホストのBLASライブラリでSGEMMを呼び出し
 call SGEMM ('n','n',m1,m1,m1,alpha,A,m1,B,m1,beta,C,m1)
#endif
```

ホストのBLASルーチンを使用するには：

```
g95 –O3 code.f90 –L/usr/local/lib –lblas
```

CUBLASルーチンを使用するには（fortran.cはNVIDIAで提供）：

```
gcc -O3 -DCUBLAS_USE_THUNKING -I/usr/local/cuda/include -c fortran.c
g95 -O3 -DCUBLAS code.f90 fortran.o -L/usr/local/cuda/lib -lcublas
```

pinnedメモリの例

pinnedメモリは高速のPCIe転送速度を実現し、ストリームの使用を有効にする
•領域の割り当てはcudaMallocHostで実行する必要がある
•Cとの相互運用性には新しいFortran 2003機能を使用する

```
iso_c_bindingの使用
! 割り当てはCの関数呼び出しで実行。Cポインタをtype (C_PTR)で定義
 type(C_PTR) :: cptr_A, cptr_B, cptr_C
! Fortran配列をポインタとして定義
 real, dimension(:,:), pointer :: A, B, C

! cudaMallocHostでメモリを割り当て
! ここでポインタとして定義されるFortran配列は、iso_c_bindingで定義される新しい相互
!運用性を使用してCポインタに関連付けられる。これは(A(m1,m1))に等しい
 res = cudaMallocHost ( cptr_A, m1*m1*sizeof(fp_kind) )
 call c_f_pointer ( cptr_A, A, (/ m1, m1 /) )

! Aを通常通り使用
! cudaMallocHostインターフェースコードのサンプルコードを参照
```


CUDAカーネルの呼び出し

FortranからCUDAカーネルを呼び出すC関数を呼び出す

```
! Fortran -> C -> CUDA ->C ->Fortran
call cudafunction(c,c2,N)
```

```
/* 注: Fortranではサブルーチンの引数は参照で渡される */
extern "C" void cudafunction_(cuComplex *a, cuComplex *b, int *Np)
{
...
int N=*np;
cudaMalloc ((void **) &a_d , sizeof(cuComplex)*N);
cudaMemcpy( a_d, a, sizeof(cuComplex)*N ,cudaMemcpyHostToDevice);
dim3 dimBlock(block_size); dim3 dimGrid (N/dimBlock.x); if( N % block_size != 0 ) dimGrid.x+=1;
square_complex<<<dimGrid,dimBlock>>>(a_d,a_d,N);
cudaMemcpy( b, a_d, sizeof(cuComplex)*N,cudaMemcpyDeviceToHost);
cudaFree(a_d);
}
```

```
complex_mul: main.f90 Cuda_function.o
    $(FC) -o complex_mul main.f90 Cuda_function.o -L/usr/local/cuda/lib –lcudart

Cuda_function.o: Cuda_function.cu
    nvcc -c -O3 Cuda_function.cu
```

CUDAのイベントAPI

- イベントはCUDA呼び出しストリームに挿入（記録）される
- 使用例
  - CUDA呼び出しの経過時間の測定（クロックサイクル精度）
  - 非同期のCUDA呼び出しの状態の照会
  - イベント前のCUDA呼び出しが完了するまで、CPUをブロック
  - CUDA SDKのasyncAPIサンプル

```
cudaEvent_t start, stop;
cudaEventCreate(&start);          cudaEventCreate(&stop);
cudaEventRecord(start, 0);
kernel<<<grid, block>>>(...);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float et;
cudaEventElapsedTime(&et, start, stop);
cudaEventDestroy(start);          cudaEventDestroy(stop);
```


デバイス管理

- CPUはGPUデバイスを問い合わせして選択できる
  - cudaGetDeviceCount(int* count)
  - cudaSetDevice(int device)
  - cudaGetDevice(int *current_device)
  - cudaGetDeviceProperties(cudaDeviceProp* prop, int device)
  - cudaChooseDevice(int *device, cudaDeviceProp* prop)
- マルチGPUのセットアップ
  - デフォルトではデバイス0が使用される
  - 1つのCPUスレッドが制御できるのは1つのGPUのみ
    - 複数のCPUスレッドで同じGPUを制御できる
      - – 呼び出しはドライバでシリアル化される

複数のCPUスレッドとCUDA

- CPUスレッドに割り当てられるCUDAリソースは、同じCPUスレッドからのCUDA呼び出しでのみ消費できる
- 違反の例
  - CPUスレッド2がGPUメモリを割り当て、アドレスをpに保管する
  - スレッド3がCUDA呼び出しを発行して、pからメモリにアクセスする

CUDAのグラフィックスにおける相互運用性

OpenGL相互運用性

- OpenGLバッファオブジェクトはCUDAアドレス空間にマッピングでき、グローバルメモリとして使用できる
  - 頂点バッファオブジェクト
  - ピクセルバッファオブジェクト
- Direct3D9の頂点オブジェクトはマップ可能
- データはデバイスコードで他のグローバルデータと同様にアクセスできる
- イメージデータはglDrawPixels / glTexImage2Dを使用して、ピクセルバッファオブジェクトから表示できる
  - ビデオメモリへのコピーが必須(それでも高速)

OpenGLの相互運用のステップ

- CUDAでバッファオブジェクトを登録
  - `cudaGLRegisterBufferObject(GLuint buffObj);`
  - OpenGLでは登録済みバッファをソースとしてのみ使用できる
  - OpenGLによってレンダリングする前にバッファを登録解除する
- バッファオブジェクトをCUDAメモリにマップする
  - `cudaGLMapBufferObject(void **devPtr, GLuint buffObj);`
  - グローバルメモリのアドレスを返す
  - バッファはマップする前に登録する必要がある
- CUDAカーネルを起動してバッファを処理する
- OpenGLで使用する前にバッファをアンマップする
  - `cudaGLUnmapBufferObject(GLuint buffObj);`
- バッファオブジェクトを登録解除する
  - `cudaGLUnregisterBufferObject(GLuint buffObj);`
  - オプション: バッファがレンダリングのターゲットの場合は必要
- OpenGLコードでバッファオブジェクトを使用する

相互運用のシナリオ:
動的なCUDA生成のテクスチャ

- CUDAでテクスチャPBOを登録する
- フレームごとに以下を実行
  - バッファをマップする
  - CUDAカーネルでテクスチャを生成する
  - バッファをアンマップする
  - テクスチャを更新する
  - テクスチャ済みオブジェクトをレンダリングする

```
unsigned char *p_d=0;
cudaGLMapBufferObject((void**)&p_d, pbo);
prepTexture<<<height,width>>>(p_d, time);
cudaGLUnmapBufferObject(pbo);
glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, pbo);
glBindTexture(GL_TEXTURE_2D, texID);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0,0, 256,256,
                GL_BGRA, GL_UNSIGNED_BYTE, 0);
```


相互運用のシナリオ:
CUDAによるフレームポストプロセス

- フレームごとに以下を実行
  - OpenGLでPBOにレンダリングする
  - CUDAでPBOを登録する
  - バッファをマップする
  - CUDAカーネルでバッファを処理する
  - バッファをアンマップする
  - CUDAからPBOを記録解除する

```
unsigned char *p_d=0;
cudaGLRegisterBufferObject(pbo);
cudaGLMapBufferObject((void**)&p_d, pbo);
postProcess<<<blocks,threads>>>(p_d);
cudaGLUnmapBufferObject(pbo);
cudaGLUnregisterBufferObject(pbo);
...
```

注記

NVIDIA のデザイン仕様、リファレンスボード、ファイル、図面、診断、リスト、およびその他のドキュメント（集合的および単独の「マテリアル」）はすべて「現状のまま」提供されます。NVIDIA は、本マテリアルについて、明示的、暗示的、法定的またはその他の保証を一切行わず、権利の不侵害、商品性、および特定目的への適合性に関するあらゆる黙示保証を明示的に放棄するものとします。

記載された情報の正確性、信頼性には万全を期しておりますが、NVIDIA Corporation はこれらの情報の使用の結果、もしくはこれらの情報の使用に起因する第三者の特許またはその他の権利の侵害に対して、一切の責任を負いません。暗示的に、もしくは NVIDIA Corporation が所有する特許または特許権に基づき、付与されるライセンスは一切ありません。本書に記載の仕様は予告なしに変更されることがあります。本書は、過去に提供されたすべての情報よりも優先されます。NVIDIA Corporation の製品は、NVIDIA Corporation の明示的な書面による許可なくしては、生命維持装置の重要な部品として使用することはできません。

商標について

NVIDIA、NVIDIA ロゴ、CUDA、および Tesla は、米国およびその他の国における NVIDIA Corporation の商標または登録商標です。その他の会社名および製品名は、各社の登録商標または商標です。

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com