CUDA テクニカルトレーニング

Vol II:
CUDA ケーススタディ

制作および提供: NVIDIA

Q2 2008

目次

セクション　スライド

CUDAの計算ファイナンス

ブラック・ショールズとモンテカルロによるオプション価格の算出

概要

- CUDAは計算ファイナンスに適している
  - ファイナンスデータの超並列性
  - 高度に独立した計算
  - 高い計算集約度 (入出力に対する計算の割合)
  - 上のすべてが高い拡張性につながる
- CUDAでのヨーロピアンオプション価格の算出を2つの方式で説明する
  - ブラック・ショールズ
  - モンテカルロシミュレーション

CUDAを使用したブラック・ショールズによるヨーロピアンオプションの価格算出

概要

このプレゼンテーションでは、CUDAを使用してヨーロピアンのプットおよびコールオプションの価格算出を実装する方法を説明する

- ホストでランダムな入力データを生成する
- GPUにデータを転送する
- GPU上で価格を計算する
- ホストに価格を送り返す
- CPUで価格を計算する
- 結果をチェックする

各オプション価格は個別に計算されるため、マップするための簡単な問題がある

ヨーロピアンオプション

$$V_{call} = S \cdot CND(d_1) - X \cdot e^{-rT} \cdot CND(d_2)$$

$$V_{put} = X \cdot e^{-rT} \cdot CND(-d_2) - S \cdot CND(-d_1)$$

$$d_1 = \frac{\log(\frac{S}{X}) + (r + \frac{v^2}{2})T}{v\sqrt{T}}$$

$$d_2 = \frac{\log(\frac{S}{X}) + (r - \frac{v^2}{2})T}{v\sqrt{T}}$$

$$CND(-d) = 1 - CND(d)$$

*S*は現在の株価、*X*は行使価格、*CND*は正規分布の累積分布関数、*r*は無リスク金利、*v*は変動率

正規分布の累積分布関数

$$N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{u^2}{2}} du$$

多項近似で計算する(Hullを参照)

5次多項式による6桁の小数点精度

```
__host__ __device__ float CND(float d)
{
        float K = 1.0f / (1.0f + 0.2316419f * fabsf(d));
        float CND = RSQRT2PI * expf(- 0.5f * d * d) *
                (K * (A1 + K * (A2 + K * (A3 + K * (A4 + K * A5)))));
        if(d > 0)
        CND = 1.0f - CND;

return CND;
}
```

- floatセーフなプログラムを設計すること
- コンパイラによってホスト用とデバイス用の2つの関数が生成される

実装ステップ

以下のステップを実装する必要がある

1. ホストに配列を割り当てる:　hOptPrice(N)、hOptStrike (N) 、..
2. デバイスに配列を割り当てる: dOptPrice(N)、dOptStrike (N) 、…
3. 入力配列を初期化する
4. ホストメモリから対応するデバイスメモリの配列に配列を転送する
5. GPU上で固定構成を使用してオプション価格を計算する
6. GPUからホストに結果を送り返す
7. GPU上でオプション価格を計算する
8. 結果を計算する
9. メモリをクリアする

コードのウォークスルー (ステップ1〜3)

```
/* ホストに配列を割り当て */
   float *hOptPrice, *hOptStrike, *hOptYear;
   hOptPrice = (float *) malloc(sizeof(float*N);
   hOptStrike = (float *) malloc(sizeof(float*N);
   hOptYear = (float *) malloc(sizeof(float*N);

/* cudaMallocでGPUに配列を割り当て */
   float *dOptPrice, *dOptStrike, *dOptYear;
   cudaMalloc( (void **) &dOptPrice, sizeof(float)*N);
   cudaMalloc( (void **) &dOptStrike, sizeof(float)*N);
   cudaMalloc( (void **) &dOptYear , sizeof(float)*N);
   ……

/* ホストのhOptPrice、hOptStrike、hOptYearを初期化 */
   ……
```

コードのウォークスルー (ステップ4〜5)

```
/* ホストからデバイスにデータを転送
cudaMemcpy (target, source, size, direction)*/
cudaMemcpy (dOptPrice, hOptPrice, sizeof(float)*N,
                    cudaMemcpyHostToDevice);
cudaMemcpy (dOptStrike, hOptStrike, sizeof(float)*N,
                    cudaMemcpyHostToDevice);
cudaMemcpy (dOptYears, hOptYears, sizeof(float)*N,
                    cudaMemcpyHostToDevice);

/* 固定構成を使用してGPU上でオプション価格を計算
   <<<ブロック数, スレッド数>>>*/
BlackScholesGPU<<<128, 256>>>(
            dCallResult, dPutResult,
            dOptionStrike, dOptionPrice, dOptionYears,
            RISKFREE, VOLATILITY, OPT_N);
```


コードのウォークスルー (ステップ6〜9)

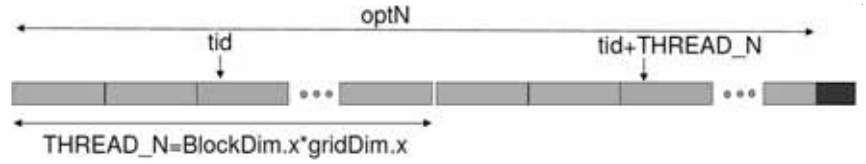
```
/* デバイスからホストにデータを転送
 cudaMemcpy(target, source, size, direction)*/
   cudaMemcpy (hCallResult , dCallResult , sizeof(float)*N,
                              cudaMemcpyDeviceToHost);
   cudaMemcpy (hPutResult , dPutResult , sizeof(float)*N,
                             cudaMemcpyDeviceToHost);

/* CPU上でオプション価格を計算 */
   BlackScholesCPU(……);

/* 結果を比較 */
   ……
/* ホストとデバイスのメモリをクリア*/
   free( hOptPrice);
   ……
   cudaFree(dOptPrice);
   ……
```

BlackScholesGPU

オプションの総数OptNをどのように処理するか?
　ブロックの最大数 = 65,536、1ブロックあたりの最大スレッド数 = 512
　(これによってOptNは33Mに制限される)

解決策: 各スレッドで複数のオプションを処理する

```
 __global__ void BlackScholes (float *…., int OptN)
{
     const int   tid = blockDim.x * blockIdx.x + threadIdx.x;
     const int   THREAD_N = blockDim.x * gridDim.x;

     for(int opt = tid; opt < OptN; opt += THREAD_N)
        BlackScholesBody(
          d_CallResult[opt], d_PutResult[opt], d_OptionPrice[opt],
          d_OptionStrike[opt],d_OptionYears[opt], Riskfree,Volatility );
}
```



コンパイルと実行

- **サンプルBlackScholes.cuのコンパイル**

```
nvcc –O3 –o BlackScholes BlacScholes.cu \\
-I../../common/inc/ -L ../../lib/ -lcutil -lGL –lglut
```

(ライブラリとインクルードのパスは使用するシステムによって異なる)

- **サンプルの実行: BlackScholes**

ブラック・ショールズ式のヨーロピアンオプション(1,000,000オプション)

入力データをGPUメモリにコピー	転送時間: 10.496000ミリ秒
GPUカーネルを実行	GPU時間: 0.893000ミリ秒
GPUの結果を読み直す	転送時間: 15.471000ミリ秒
結果をチェック CPU計算を実行	CPU時間: 454.683990ミリ秒

結果を比較
L1ノルム: 5.984729E-08 最大絶対誤差: 1.525879E-05

PCIe転送速度の改良

通常のホストメモリからのPCIe転送の帯域幅は1〜1.5 GB/s (ホストCPUのチップセットによって異なる)

ページロックのメモリ割り当てを使用すると転送速度は3 GB/s 以上 (専用のNVIDIAチップセットで「LinkBoost」を使用すると 4 GB)

CUDAにはこのための特殊なメモリ割り当て関数がある

pinnedメモリの使用方法

- cudaMallocをcudaMallocHostで置き換える
- cudaFreeをcudaFreeHostで置き換える

```
/* メモリの割り当て (通常のmallocの代わり) */
   cudaMallocHost ((void **) &h_CallResultGPU, OPT_SZ);

/* メモリのクリア (通常のfreeの代わり)
   cudaFreeHost(h_CallResultGPU);
```

コンパイルと実行

- **サンプルBlackScholesPinned.cuのコンパイル**

```
nvcc –O3 –o BlackScholesPinned BlacScholesPinned.cu\\
-I../../common/inc/ -L ../../lib/ -lcutil -lGL –lglut
```

(ライブラリとインクルードのパスは使用するシステムによって異なる)

- **サンプルの実行: BlackScholesPinned**

ブラック・ショールズ式のヨーロピアンオプション (1,000,000オプション)

入力データをGPUメモリにコピー	転送時間: 3.714000ミリ秒 (以前は10.4)
GPUカーネルを実行	GPU時間: 0.893000ミリ秒
GPUの結果を読み直す	転送時間: 2.607000ミリ秒 (以前は15.4)
結果をチェック CPU計算を実行	CPU時間: 454.683990ミリ秒

結果を比較
L1ノルム: 5.984729E-08 最大絶対誤差: 1.525879E-05

CUDAを使用したヨーロピアンオプションのモンテカルロシミュレーション

概要

このプレゼンテーションでは、CUDAを使用してヨーロピアンコールオプションのモンテカルロシミュレーションを実装する方法を説明する

モンテカルロシミュレーションは並列化に非常に適している

- 高い規則性と局所性
- 入出力に対する計算の割合が非常に高い
- 高度な拡張性

基本的なモンテカルロの実装（相反変数や制御変量などの分散低減技術は実装しない）

モンテカルロ法

モンテカルロシミュレーションによるオプション価格の算出

i. 基本となる資産価格のサンプルパスをシミュレートする
ii. 各サンプルパスの対応するオプションペイオフを計算する
iii. シミュレーションのペイオフを平均して平均値を割り引き、オプションの価格を算出する

```
% MATLABでのヨーロピアンコールのモンテカルロ評価
% An Introduction to Financial Option Valuation: Mathematics, Stochastics
% and Computation、D. Higham著

S = 2; E = 1; r = 0.05; sigma = 0.25; T = 3; M = 1e6;
Svals = S*exp((r-0.5*sigma^2)*T +
sigma*sqrt(T)*randn(M,1));
Pvals = exp(-r*T)*max(Svals-E,0);
Pmean = mean(Pvals)
width = 1.96*std(Pvals)/sqrt(M);
conf = [Pmean - width, Pmean + width]
```

モンテカルロの例

i. 乱数Mを生成する:M=200,000,000
   i. メルセンヌツイスター (MT) を使用した一様分布
   ii. ガウス分布を生成するボックスミューラー変換
ii. N個のオプションの対数正規分布を計算する: N=128
iii. 各オプションの合計と二乗の合計を計算して平均と分散を得る
iv. シミュレーションのペイオフを平均して平均値を割り引き、オプションの価格を算出する

注: MTの最初のシードを適切に使用すると、この例を簡単に拡張して複数のGPU上で実行できる。CUDA SDK v1.1のサンプル「MonteCarloMultiGPU」を参照

一様分布乱数の生成

この例で使用されている乱数ジェネレータ (RNG) は松本氏、西村氏によって開発されたメルセンヌツイスターの並列バージョンで、DCMTと呼ばれるものです
- 高速
- 統計的特性に優れている
- 多くの独立したメルセンヌツイスターを生成する

最初のパラメータはオフラインで計算され、ファイルに保存される

```
RandomGPU<<<32,128>>>( d_Random, N_PER_RNG, seed);
```

32ブロック×128スレッド: 4096の独立したランダムなストリーム

Tesla C870 (シングルGPU) 上で
80ミリ秒で2億のサンプル
1秒で25億のサンプル!!!!

ガウス標準分布の生成

ボックスミューラー変換を使用して一様分布の乱数からガウス標準分布を生成する

```
BoxMullerGPU<<<32,128>>>( d_Random, N_PER_RNG, seed);
```

Tesla C870 (シングルGPU) 上で
120ミリ秒で2億のサンプル

```
#define PI 3.14159265358979323846264338327950288f
__device__ void BoxMuller(float& u1, float& u2){
  float r = sqrtf(-2.0f * logf(u1));
  float phi = 2 * PI * u2;
  u1 = r * cosf(phi);
  u2 = r * sinf(phi);
}
```

- その他の選択肢：逆正規を近似するBeasley-Springer-Moroアルゴリズム

対数正規分布と部分和

```
void MonteCarloGPU(d_Random,….)
{
  // 64×256 (16,384) の和を部分和に分割する
  MonteCarloKernelGPU<<<64, 256, 0>>>(d_Random);

  // 部分和をホストに読み戻す

  cudaMemcpy(h_Sum, d_Sum, ACCUM_SZ, cudaMemcpyDeviceToHost) ;
  cudaMemcpy(h_Sum2, d_Sum2, ACCUM_SZ, cudaMemcpyDeviceToHost) ;

  // ホスト上で合計と2乗の合計を計算

  double dblSum = 0, dblSum2 = 0;
  for(int i = 0; i < ACCUM_N; i++){
    dblSum += h_Sum[i];
    dblSum2 += h_Sum2[i];
  }
}
```

対数正規分布と部分和

```
__global__ void MonteCarloKernelGPU(…)
{
 const int tid = blockDim.x * blockIdx.x + threadIdx.x;
 const int threadN = blockDim.x * gridDim.x;
 //...

 for(int iAccum = tid; iAccum < accumN; iAccum += threadN) {
  float sum = 0, sum2 = 0;

  for(int iPath = iAccum; iPath < pathN; iPath += accumN) {
   float r = d_Random[iPath];
   //...
   sum += endOptionPrice;
   sum2 += endOptionPrice * endOptionPrice;
 }

 d_Sum[iAccum] = sum;
 d_Sum2[iAccum] = sum2;
}
```


浮動小数点数の正確な合計

連続するN個の数 a_iを合計する一般的な方法は、以下の再帰的な式

$S_0=0$

$S_i=S_{i-1}+a_1$

$S=S_n$

誤差分析 (Wilkinson、1963) により、浮動小数点数の演算における丸め誤差の累積はN^2の速度で増加することが証明されている。

複数の中間集計を作成すると、累積される丸め誤差は大幅に減少する。
並列の加算は、まさしくこのように動作する。

コンパイルと実行

- **サンプルMontecarlo.cuのコンパイル:**

```
nvcc –O3 –o Montecarlo Montecarlo.cu\\
-I../../common/inc/ -L ../../lib/ -lcutil -lGL –lglut
```

(ライブラリとインクルードのパスは使用するシステムによって異なる)

- **サンプルの実行: Montecarlo**

ヨーロピアンコールオプションのモンテカルロシミュレーション (200,000,000パス)

GPU上で乱数を生成	80.51499ミリ秒
GPU上のボックスミューラー	122.62799ミリ秒
オプションの平均時間	15.471000ミリ秒

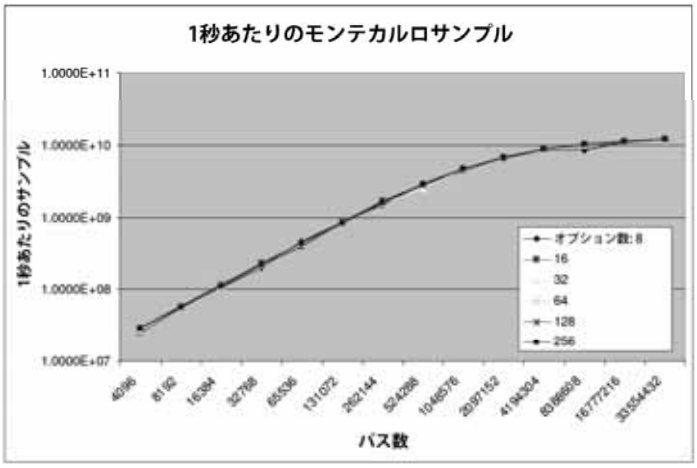
結果をチェック
モンテカルロ: 6.621926; ブラック・ショールズ: 6.621852
絶対: 7.343292e-05; 相対: 1.108948e-05;

モンテカルロ: 8.976083; ブラック・ショールズ: 8.976007
絶対: 7.534027e-05; 相対: 8.393517e-06;

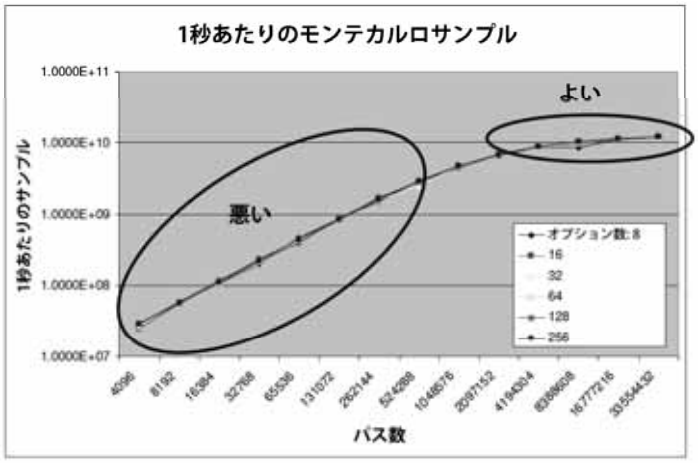
小規模な問題用に最適化

- このモンテカルロの実装は大規模な問題用に最適化されている
  - 例:256のオプションに対して16Mのパス
- 実際のシミュレーションの大部分はずっと小規模
  - 例: 64のオプションに対して256Kのパス
- 最適化の前にパフォーマンスの詳細を確認する
  - さまざまな規模の問題でパフォーマンスを比較すると、いろいろなことが分かる

1秒あたりのモンテカルロサンプル

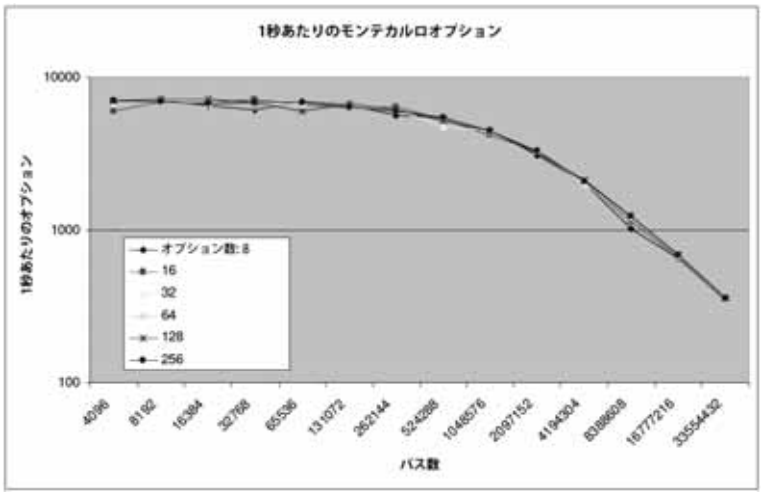


1秒あたりのモンテカルロサンプル

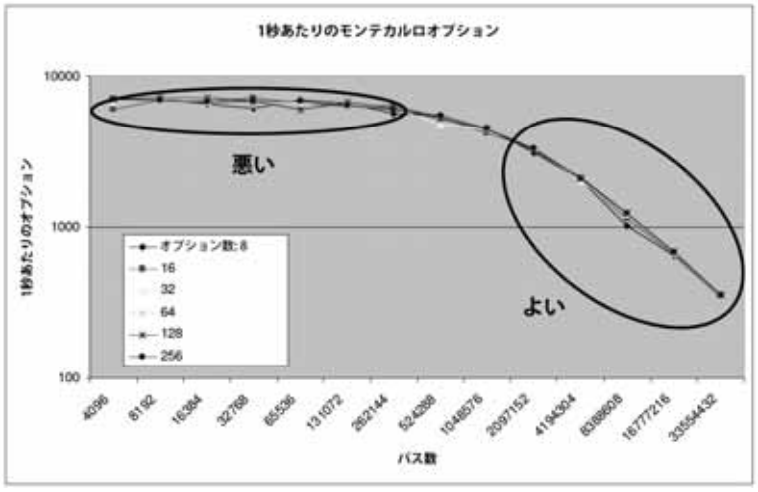


グラフは水平線にならなくてはいけない

1秒あたりのモンテカルロオプション



1秒あたりのモンテカルロオプション



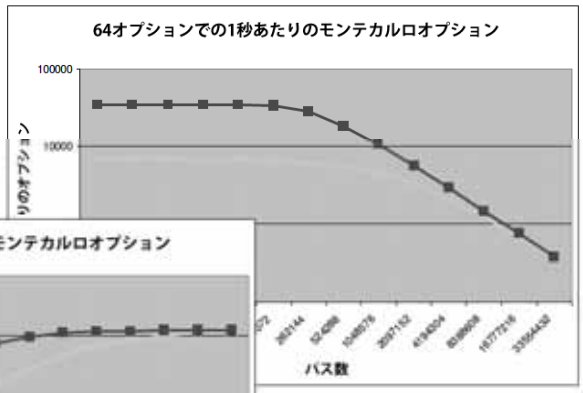
グラフは直線の対角線にならなくてはいけない

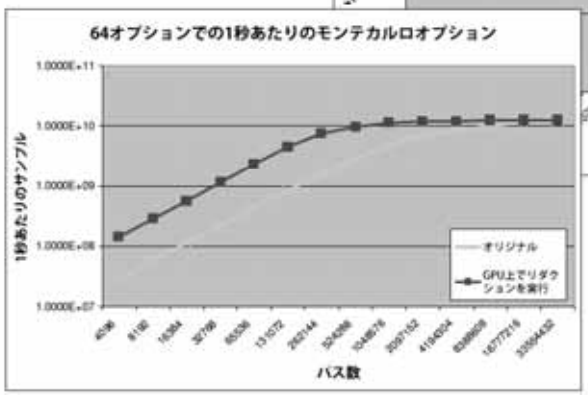
効率の悪い部分

- コードを見ると効率の悪い部分が何点かある
  - 最終的な和のリダクションがGPUでなくCPUで実行されている
  - オプションのループがGPUでなくCPUで実行されている
  - 1オプションあたり複数のスレッドブロックが起動されている
- 大規模な問題では、計算が制限となるために明らかにならない
- 注: 今後の比較では、64のオプションを最適化のケースとして使用する

リダクションをGPUに移動する

並列リダクションを使用してGPU上で最終的な合計を実行すると、大幅に速度が向上する



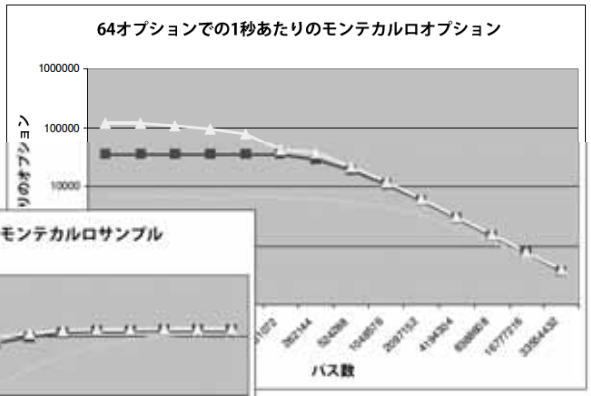


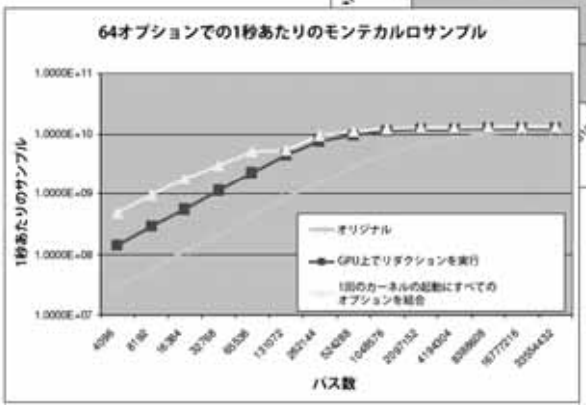
各スレッドブロックの1つの合計と2乗の合計を読み戻す

1回のカーネルの起動にすべてのオプション

- ホスト上のループの最初のコードでオプションごとにカーネルを1回起動する
  - 1次元のグリッド。1オプションあたり複数のスレッドブロック
- ループせずに2次元のグリッドを起動する
  - 1オプションあたり1行のスレッドブロック

1回のカーネルの起動にすべてのオプション





1オプションあたり1つのスレッドブロック

- 複数ブロックを使用するオプションの価格算出では、複数のカーネルを起動する必要がある
  - 最初のカーネルは部分和を生成する
  - 次のカーネルは和のリダクションを実行して最終的な値を取得する
- 非常に小規模な問題では、カーネルの起動のオーバーヘッドがコストの大部分を占める
  - CPU上でリダクションを実行する場合には、cudaMemcpyが大部分を占める
- 解決策: パス数が少ない場合は、新しいカーネルで1オプションあたり1つのスレッドブロックを使用する
  - オプション全体の和はこのカーネルで実行される

サイズの基準

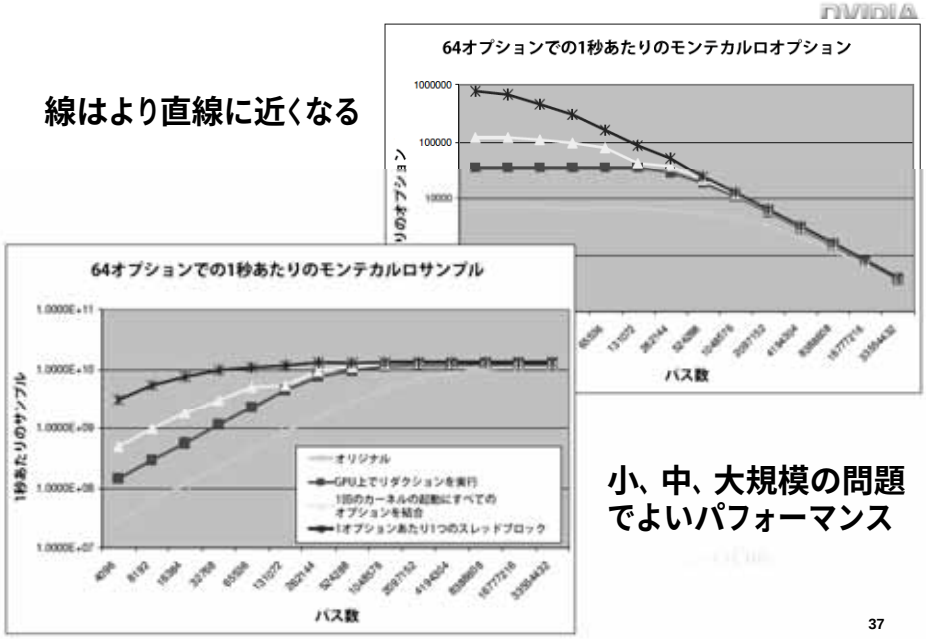
- パス数が多い場合は、引き続き古い2つのカーネルの方法を使用する
- 切り替えるべきときの判断基準は?
- NVIDIAで独自に作成した基準

```
bool multiBlock = ((numPaths / numOptions) >= 8192);
```

- multiBlockがfalseの場合はシングルブロックのカーネルを実行する
- それ以外の場合はマルチブロックのカーネルを実行してカーネルでリダクションする

1オプションあたり1つのスレッドブロック

線はより直線に近くなる



小、中、大規模の問題
でよいパフォーマンス

詳細

- この最適化の詳細についてはCUDA SDK 1.1の「MonteCarlo」サンプルコードを参照
  - 最新のバージョンはこの実験に基づいて最適化されている
- 付属のホワイトペーパーに詳細が記載されている
- 問題をシステム上の複数のGPUに分散する方法については「MonteCarloMultiGPU」の例を参照

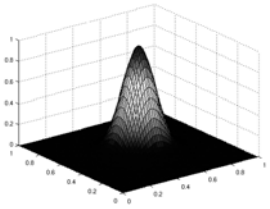
結論

- CUDAは計算ファイナンスに適している
- 特定の問題に合わせてコードを調整することが重要
  - さまざまな問題のサイズの相対的なパフォーマンスを研究する
  - サイズが異なる問題ごとに、ボトルネックの原因を理解する
  - 最適化はニーズによって異なる場合がある

スペクトラルのポアソン方程式ソルバ

概要

この例では、フーリエスペクトラル方式を使用して、周期的境界条件を持つ長方形の領域のポアソン方程式を解く



この例ではFFTライブラリの使用方法、GPUとのデータ転送方法、GPUでの簡単な計算の実行方法について説明する

数理的な背景

$$\nabla^2\phi = r \xrightarrow{\text{FFT}} -(k_x^2 + k_y^2)\hat{\phi} = \hat{r}$$

1. rに2次元の順方向FFTを適用し、r(k)を取得する。kは波数
2. r(k)にラプラス演算子の逆数を適用し、u(k)を取得する。簡単なフーリエ空間の要素ごとの除算

$$\hat{\phi} = -\frac{\hat{r}}{(k_x^2 + k_y^2)}$$

3. 2次元の逆方向FFTをu(k)に適用し、uを取得する

MATLAB実装のリファレンス

```
% フーリエノードの数
    N = 64;
% ドメインサイズ（正方形が前提）
    L = 1;
% fの特性幅（<< 1にする）
    sig = 0.1;
% 波数のベクトル
    k = (2*pi/L)*[0:(N/2-1) (-N/2):(-1)];
% フーリエモード(m,n)に対応する
% 波数の(x,y) 行列
    [KX KY] = meshgrid(k,k);
% 波数で動作するラプラス行列
    delsq = -(KX.^2 + KY.^2);
% 波数(0,0)のゼロ除算を避けるための
% テクニック
% （この波数はいずれにしても0になる）
    delsq(1,1) = 1;
% グリッド間隔
    h = L/N;
    x = (0:(N-1))*h ;
    y = (0:(N-1))*h;
    [X Y] = meshgrid(x,y);
```

```
% フーリエグリッドポイントでRHS f(x,y)を構築
    rsq = (X-0.5*L).^2 + (Y-0.5*L).^2;
    sigsq = sig^2;
    f = exp(-rsq/(2*sigsq)).*…
      (rsq - 2*sigsq)/(sigsq^2);
% ラプラスのスペクタル逆関数
    fhat = fft2(f);
    u = real(ifft2(fhat./delsq));
% u = 0に強制的に設定して
% 任意の定数を指定
    u = u - u(1,1);
% L2およびLinfのエラー誤差を計算
    uex = exp(-rsq/(2*sigsq));
    errmax = norm(u(:)-uex(:),inf);
    errmax2 = norm(u(:)-uex(:),2)/(N*N);
% L2およびLinfのエラー誤差を出力
    fprintf('N=%d¥n',N);
    fprintf('Solution at (%d,%d):
    ',N/2,N/2);
    fprintf('computed=%10.6f …
      reference = %10.6f¥n',u(N/2,N/2),
    uex(N/2,N/2));
    fprintf('Linf err=%10.6e L2 norm
        err = %10.6e¥n',errmax, errmax2);
```

http://www.atmos.washington.edu/2005Q2/581/matlab/pois_FFT.m

実装ステップ

以下の手順を実行する必要がある

1. **ホストにメモリを割り当てる**: r (NxN)、u (NxN)、kx (N)、ky (N)
2. **デバイスにメモリを割り当てる**: r_d、u_d、kx_d、ky_d
3. **ホストメモリから、対応するデバイスメモリの配列に**kx**と**ky**を転送する**
4. FFT**のプランを初期化する**
5. **実行構成を計算する**
6. **実数の入力を複素数の入力に変換する**
7. 2**次元順方向**FFT
8. **フーリエ空間のポアソン方程式を解く**
9. 2**次元逆方向**FFT
10. **複素数の出力を実数の出力に変換してスケーリングを適用する**
11. GPU**からホストに結果を転送する**

コードを簡単にするために、シンメトリ（実数のC2C**変換）は利用しない**

コードのウォークスルー (ステップ1〜3)

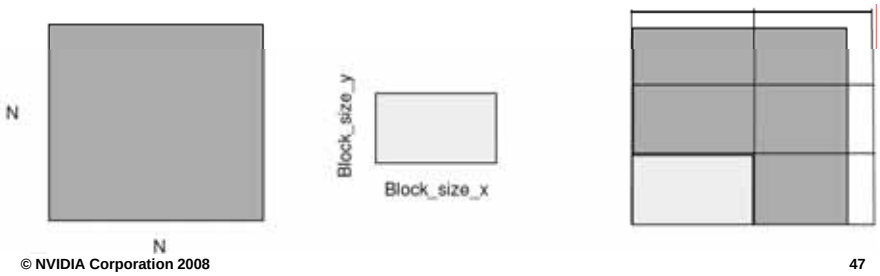
```
/* ホストに配列を割り当て */
     float *kx, *ky, *r;
     kx = (float *) malloc(sizeof(float*N);
     ky = (float *) malloc(sizeof(float*N);
     r = (float *) malloc(sizeof(float*N*N);

/* cudaMallocでGPUに配列を割り当て */
     float *kx_d, *ky_d, *r_d;
     cudaMalloc( (void **) &kx_d, sizeof(cufftComplex)*N);
     cudaMalloc( (void **) &ky_d, sizeof(cufftComplex)*N);
     cudaMalloc( (void **) &r_d , sizeof(cufftComplex)*N*N);

     cufftComplex *r_complex_d;
     cudaMalloc( (void **) &r_complex_d, sizeof(cufftComplex)*N*N);
```


コードのウォークスルー (ステップ3〜4)

```
/* ホストのr、kx、kyを初期化 */
  ……
/* ホストからデバイスにデータを転送
 cudaMemcpy (target, source, size, direction)*/
 cudaMemcpy (kx_d, kx, sizeof(float)*N , cudaMemcpyHostToDevice);
 cudaMemcpy (ky_d, ky, sizeof(float)*N , cudaMemcpyHostToDevice);
 cudaMemcpy (r_d , r , sizeof(float)*N*N, cudaMemcpyHostToDevice);

/* CUDA FFTのプランを作成(FFTWに似たインターフェース) */
  cufftHandle plan;
  cufftPlan2d( &plan, N, N, CUFFT_C2C);
```

コードのウォークスルー (ステップ5)


```
/* 実行構成を計算する
   注: block_size_x*block_size_y = スレッド数
       G80ではスレッド数は512未満 */
  dim3 dimBlock(block_size_x, block_size_y);
  dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);

/*ハンドルNはblock_size_xまたはblock_size_yの倍数でない*/
  if (N % block_size_x !=0 ) dimGrid.x+=1;
  if (N % block_size_y !=0 ) dimGrid.y+=1
```



コードのウォークスルー (ステップ6〜10)

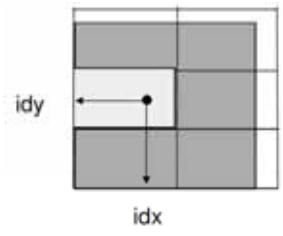

```
/* 実数の入力を複素数の入力に変換*/
  real2complex<<<dimGrid, dimBlock>>> (r_d, r_complex_d, N);

/* インプレースで順方向FFTを計算*/
  cufftExecC2C (plan, r_complex_d, r_complex_d, CUFFT_FORWARD);

/* フーリエ空間でポアソン方程式を解く*/
  solve_poisson<<<dimGrid, dimBlock>>> (r_complex_d, kx_d, ky_d,N);

/* インプレースで逆方向FFTを計算*/
  cufftExecC2C (plan, r_complex_d, r_complex_d, CUFFT_INVERSE);

/* 解を実数の配列にコピーバックしてスケーリングを適用
     (FFTおよび後続のiFFTは、同じ長さの配列に変換する) */
  scale = 1.f / ( (float) N * (float) N );
  complex2real_scaled<<<dimGrid, dimBlock>>> (r_d, r_complex_d, N, scale);
```

コードのウォークスルー (ステップ11)

```
/* デバイスからホストにデータを転送
  cudaMemcpy(target, source, size, direction)*/
  cudaMemcpy (r , r_d , sizeof(float)*N*N, cudaMemcpyDeviceToHost);

/* プランを破棄し、デバイスのメモリをクリア*/
  cufftDestroy( plan);
  cudaFree(r_complex_d);
  ……
  cudaFree(kx_d);
```


real2complex

```
/* 実数のデータを複素数のデータにコピー */

 __global__ void real2complex (float *a, cufftComplex *c, int N)
{
  /* オリジナルのNxN配列の要素の位置のidxとidyを計算 */
  int idx = blockId.x*blockDim.x+threadIdx.x;
  int idy = blockId.y*blockDim.y+threadIdx.y;

  if ( idx < N && idy <N)
      {
      int index = idx + idy*N;
      c[index].x = a[index];
      c[index].y = 0.f;
      }
}
```



solve_poisson

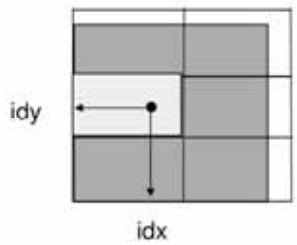
```
__global__ void solve_poisson (cufftComplex *c, float *kx, float *ky, int N)
{
    /* オリジナルのN×N配列の要素の位置のidxとidyを計算 */
    int idx = blockId.x*blockDim.x+threadIdx.x;
    int idy = blockId.y*blockDim.y+threadIdx.y;

  if ( idx < N && idy <N)
      {
      int index = idx + idy*N;
      float scale = - ( kx[idx]*kx[idx] + ky[idy]*ky[idy] );
      if ( idx ==0 && idy == 0 ) scale =1.f;
      scale = 1.f / scale;
      c[index].x *= scale;
      c[index].y *= scale;
      }
}
```

$$\hat{\phi} = -\frac{\hat{r}}{(k_x^2 + k_y^2)}$$

complex2real_scaled

/* 複素数の実数部分を実数の配列にコピーしてスケーリングを適用 */

```
__global__ void complex2real_scaled (cufftComplex *c, float *a, int N,
                                     float scale)
{
      /* オリジナルのN×N配列の要素の位置のidxとidyを計算 */
      int idx = blockId.x*blockDim.x+threadIdx.x;
      int idy = blockId.y*blockDim.y+threadIdx.y;

      if ( idx < N && idy <N)
          {
          int index = idx + idy*N;
          a[index] = scale*c[index].x ;
          }
}
```



poisson_1のコンパイルと実行

- **例poisson_1.cuのコンパイル**

```
nvcc –O3 –o poisson_1 poisson_1.cu\\
-I/usr/local/cuda/include –L/usr/local/cuda/lib -lcufft –lcudart
```

- **例を実行**

./poisson_1 -N64
ドメイン64×64**のポアソンソルバ**
dimBlock 32 16 (512**スレッド**)
dimGrid 2 4
L2**誤差**9.436995e-08:
時間 0.000569:
I/O**時間** 0.000200 (0.000136 + 0.000064):
(32,32)**の解**
計算=0.975879 **リファレンス**=0.975882

- **MATLABによるリファレンス値**

N=64
(32,32)**の解**: **計算**= 0.975879 **リファレンス**= 0.975882
Linf**誤差**=2.404194e-05 L2**ノルム誤差**= 9.412790e-08

プロファイリング

CUDA**での関数呼び出しのプロファイリングは非常に簡単**
以下の環境変数によって制御される

- CUDA_PROFILE:**有効／無効**
  - 1 (**プロファイラを有効化**)
  - 0 (**デフォルト、プロファイラなし**)
- CUDA_PROFILE_LOG: **ファイル名を指定**
  - **設定すると**「filename」**に書き込まれる**
  - **設定しないと**cuda_profile.log**に書き込まれる**
- CUDA_PROFILE_CSV: **形式を制御**
  - 1 (**カンマ区切りファイルを有効**)
  - 0 (**カンマ区切りファイルを無効**)

Poisson_1からのプロファイラ出力

./poisson_1 –N1024

method=[memcopy] gputime=[1427.200]
method=[memcopy] gputime=[10.112]
method=[memcopy] gputime=[9.632]

method=[real2complex] gputime=[1654.080] cputime=[1702.000] occupancy=[0.667]

method=[c2c_radix4] gputime=[8651.936] cputime=[8683.000] occupancy=[0.333]
method=[transpose] gputime=[2728.640] cputime=[2773.000] occupancy=[0.333]
method=[c2c_radix4] gputime=[8619.968] cputime=[8651.000] occupancy=[0.333]
method=[c2c_transpose] gputime=[2731.456] cputime=[2762.000] occupancy=[0.333]

method=[solve_poisson] gputime=[6389.984] cputime=[6422.000] occupancy=[0.667]

method=[c2c_radix4] gputime=[8518.208] cputime=[8556.000] occupancy=[0.333]
method=[c2c_transpose] gputime=[2724.000] cputime=[2757.000] occupancy=[0.333]
method=[c2c_radix4] gputime=[8618.752] cputime=[8652.000] occupancy=[0.333]
method=[c2c_transpose] gputime=[2767.840] cputime=[5248.000] occupancy=[0.333]

method=[complex2real_scaled] gputime=[2844.096] cputime=[3613.000] occupancy=[0.667]

method=[memcopy] gputime=[2461.312]

パフォーマンスの改善

- pinnedメモリを使用してCPU／GPU転送時間を改善する

```
#ifdef PINNED
    cudaMallocHost((void **) &r,sizeof(float)*N*N); // rhs、2次元配列
#else
    r = (float *) malloc(sizeof(float)*N*N); // rhs、2次元配列
#endif
```

$ ./poisson_1
ドメイン1,024×1,024**のポアソソルバ**
合計時間: 69.929001 (ms)
解決時間: 60.551998 (ms)
I/O**時間** : **8.788000 (5.255000 + 3.533000) (ms)**

$ ./poisson_1_pinned
ドメイン1,024×1,024**のポアソソルバ**
合計時間: 66.554001 (ms)
解決時間: 60.736000 (ms)
I/O**時間** : **5.235000 (2.027000 + 3.208000) (ms)**

その他の改良

- solve_poissonの配列kxとkyで共有メモリを使用する
- 高速な整数演算を使用する(__umul24)

solve_poisson (共有メモリを使用)

```
__global__ void solve_poisson (cufftComplex *c, float *kx, float *ky, int N)
{
   unsigned int idx = __umul24(blockIdx.x,blockDim.x)+threadIdx.x;
   unsigned int idy = __umul24(blockIdx.y,blockDim.y)+threadIdx.y;
// 共有メモリを使用して同じkの値へのアクセス回数を最小限に抑える
   __shared__ float kx_s[BLOCK_WIDTH], ky_s[BLOCK_HEIGHT]
   if (threadIx.x < 1) kx_s[threadIdx.x] = kx[idx];
   if (threadIx.y < 1) ky_s[threadIdx.y] = ky[idy];
   __syncthreads();
   if ( idx < N && idy <N)
      {
      unsigned int index = idx +__umul24(idy ,N);
      float scale = - ( kx_s[threadIdx.x]*kx_s[threadIdx.x]
               +  ky_s[threadIdy.y]*ky_s[threadIdy.y] );
      if ( idx ==0 && idy == 0 ) scale =1.f;
      scale = 1.f / scale;
      c[index].x *= scale;
      c[index].y*= scale;
      }
}
```

Poisson_2からのプロファイラ出力

./poisson_2 –N1024 –x16 –y16

method=[memcopy] gputime=[1426.048]
method=[memcopy] gputime=[9.760]
method=[memcopy] gputime=[9.472]

method=[real2complex] gputime=[1611.616] cputime=[1662.000] occupancy=[0.667] (was 1654)

method=[c2c_radix4] gputime=[8658.304] cputime=[8689.000] occupancy=[0.333]
method=[c2c_transpose] gputime=[2731.424] cputime=[2763.000] occupancy=[0.333]
method=[c2c_radix4] gputime=[8622.048] cputime=[8652.000] occupancy=[0.333]
method=[c2c_transpose] gputime=[2738.592] cputime=[2770.000] occupancy=[0.333]

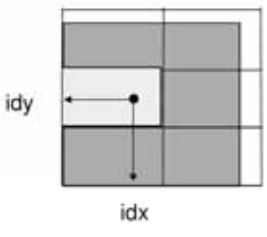
method=[solve_poisson] gputime=[2760.192] cputime=[2792.000] occupancy=[0.667] (was 6389)

method=[c2c_radix4] gputime=[8517.952] cputime=[8550.000] occupancy=[0.333]
method=[c2c_transpose] gputime=[2729.632] cputime=[2766.000] occupancy=[0.333]
method=[c2c_radix4] gputime=[8621.024] cputime=[8653.000] occupancy=[0.333]
method=[c2c_transpose] gputime=[2770.912] cputime=[5252.000] occupancy=[0.333]

method=[complex2real_scaled] gputime=[2847.008] cputime=[3616.000] occupancy=[0.667]
???????
method=[memcopy] gputime=[2459.872]

complex2real_scaled (高速バージョン)

```
__global__ void complex2real_scaled (cufftComplex *c, float *a, int N,
                                                       floatscale)
{
   /* オリジナルのNxN配列の要素の位置のidxとidyを計算 */
   int idx = blockId.x*blockDim.x+threadIdx.x;
   int idy = blockId.y*blockDim.y+threadIdx.y;
   volatile float2 c2;
       if ( idx < N && idy <N)
       {
       int index = idx + idy*N;
       c2.x= c[index].x;
       c2.y= c[index].y;
       a[index] = scale*c2.x ;
       }
}
```



ptxファイルを見るとコンパイラがメモリの結合を妨げるベクトルのロードを最適化していることがわかる。ベクトルのロードでは強制的に揮発性メモリを使用

Poisson_3からのプロファイラ出力

```
method=[ memcopy ] gputime=[ 1427.808 ]
method=[ memcopy ] gputime=[ 9.856 ]
method=[ memcopy ] gputime=[ 9.600 ]

method=[ real2complex] gputime=[ 1614.144 ] cputime=[ 1662.000 ] occupancy=[ 0.667 ]

method=[ c2c_radix4]       gputime=[ 8656.800 ] cputime=[ 8688.000 ] occupancy=[ 0.333 ]
method=[ c2c_transpose] gputime=[ 2727.200 ] cputime=[ 2758.000 ] occupancy=[ 0.333 ]
method=[ c2c_radix4 ]      gputime=[ 8607.616 ] cputime=[ 8638.000 ] occupancy=[ 0.333 ]
method=[ c2c_transpose] gputime=[ 2729.888 ] cputime=[ 2761.000 ] occupancy=[ 0.333 ]

method=[ solve_poisson ] gputime=[ 2762.656 ] cputime=[ 2794.000 ] occupancy=[ 0.667 ]

method=[ c2c_radix4 ]      gputime=[ 8514.720 ] cputime=[ 8547.000 ] occupancy=[ 0.333 ]
method=[ c2c_transpose] gputime=[ 2724.192 ] cputime=[ 2760.000 ] occupancy=[ 0.333 ]
method=[ c2c_radix4 ]      gputime=[ 8620.064 ] cputime=[ 8652.000 ] occupancy=[ 0.333 ]
method=[ c2c_transpose] gputime=[ 2773.920 ] cputime=[ 4270.000 ] occupancy=[ 0.333 ]
method=[ complex2real_scaled ] gputime=[ 1524.992 ] cputime=[ 1562.000 ] occupancy=[ 0.667 ]

method=[ memcopy ] gputime=[ 2468.288 ]
```


パフォーマンスの改良

	pinned**メモリ 不使用**	pinned**メモリ**
最初の実装 (r2c、**ポアソン**、c2r)	67ms (10.8ms)	63ms
+共有メモリ +整数での高速乗算	63.4ms (7.1ms)	59.4ms
+c2r**での読み込みの結合**	62.1ms (5.8ms)	58.2ms

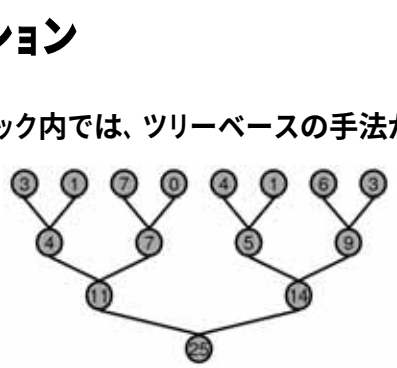
Tesla C870、pinned**メモリ**、**最適化バージョン**: 10.4ms

並列リダクション

並列リダクション

- 共通かつ重要なデータ並列プリミティブ
- CUDAでの実装は簡単
  - 正しく理解することの方が難しい
- 最適化の好例
  - 7つの異なるバージョンで学習
  - 重要な最適化戦略の実例をいくつか検証する

並列リダクション



- 各スレッドブロック内では、ツリーベースの手法が使用されている

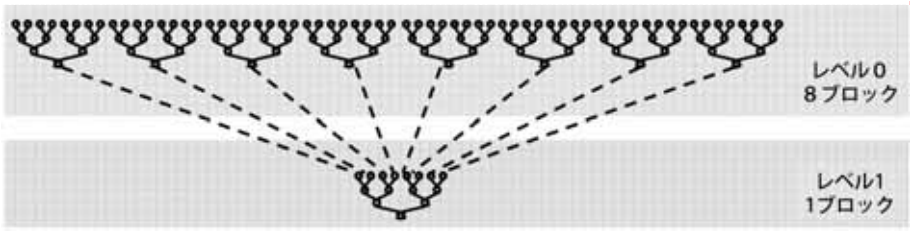
- 以下の目的で、複数のスレッドブロックを使用する必要がある
  - 非常に大規模な配列を処理する
  - GPU上のすべてのプロセッサをビジーにする
  - 各スレッドブロックで配列の一部をリダクションする
- しかし、部分的な結果をスレッドブロック間で伝達するにはどうしたらよいか?

グローバルな同期についての「神話」

- すべてのスレッドブロックを同期できるとすれば、非常に大規模な配列を簡単にリダクションできるのか?
  - 各ブロックが結果を生成した後にグローバルで同期
  - すべてのブロックが同期に達したら、再帰的に継続する
- 問題: GPUのリソースの制限
  - GPUは*M*個のマルチプロセッサを持つ
  - 各マルチプロセッサは最大*b*個のブロックをサポートできる
  - 合計のブロック数*B* > *M* * *b*… この場合には**デッドロック**となる
- また、GPUはメモリのレイテンシをカバーするため*独立した*並列性に大きく依存
  - グローバルな同期は独立を損なう

解決策: カーネルの分解

- 計算を複数のカーネルの呼び出しに分解して、グローバルな同期を避ける



- リダクションではすべてのレベルのコードが同じ
  - 再帰的なカーネルの呼び出し

最適化の目的とは?

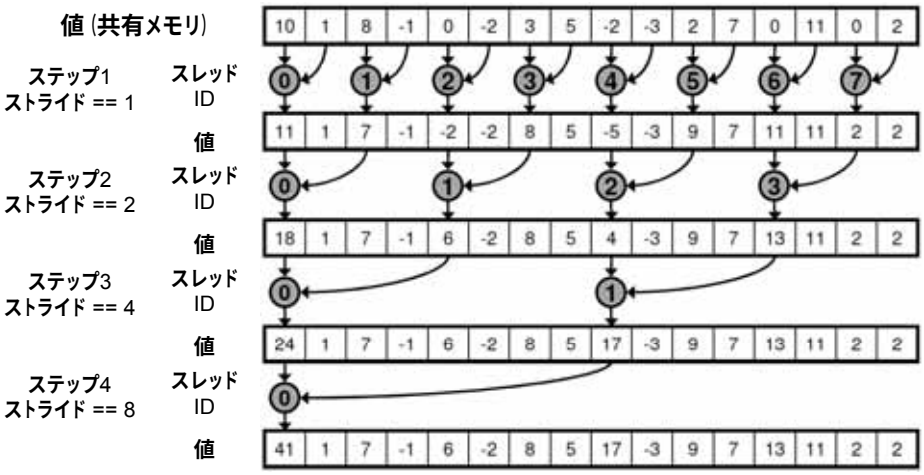
- GPUがピークパフォーマンスに到達することが目標
  - GFLOP/s: 計算能力が制約となるカーネルの場合
  - 帯域幅: メモリが制約となるカーネルの場合
- リダクションの計算集約度は非常に低い
  - 1要素のロードは1フロップ (帯域幅-最適)
- そのために最大の帯域幅を追求

- この例ではG80 GPUを使用
  - 384ビットのメモリインターフェース、900 MHz DDR
  - 384 * 1800 / 8 = 86.4 GB/s

リダクション1: インターリーブ アドレス指定

```
__global__ void reduce0(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];

  // 各スレッドはグローバルメモリから1つの要素を共有メモリに読み込む
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  __syncthreads();

  // 共有メモリでリダクションを実行
  for(unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }

  // このブロックの結果をグローバルメモリに書き込む
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```


並列リダクション: インターリーブ アドレス指定



リダクション1: インターリーブアドレス指定

```
__global__ void reduce1(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];

  // 各スレッドはグローバルメモリから1つの要素を共有メモリに読み込む
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  __syncthreads();

  // 共有メモリでリダクションを実行
  for (unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
    sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }

  // このブロックの結果をグローバルメモリに書き込む
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

問題: 発散が大きな分岐では パフォーマンスが非常に低下する

4Mの要素のリダクションのパフォーマンス

カーネル1: インターリーブアドレス指定 (分岐の発散あり)	8.054 ms	2.083 GB/s

注: ブロックサイズ＝すべてのテストで128スレッド

リダクション2: インターリーブアドレス指定

内部ループの発散する分岐を置き換えるのみ

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
    sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

ストライドされたインデックスおよび発散のない分岐を使用

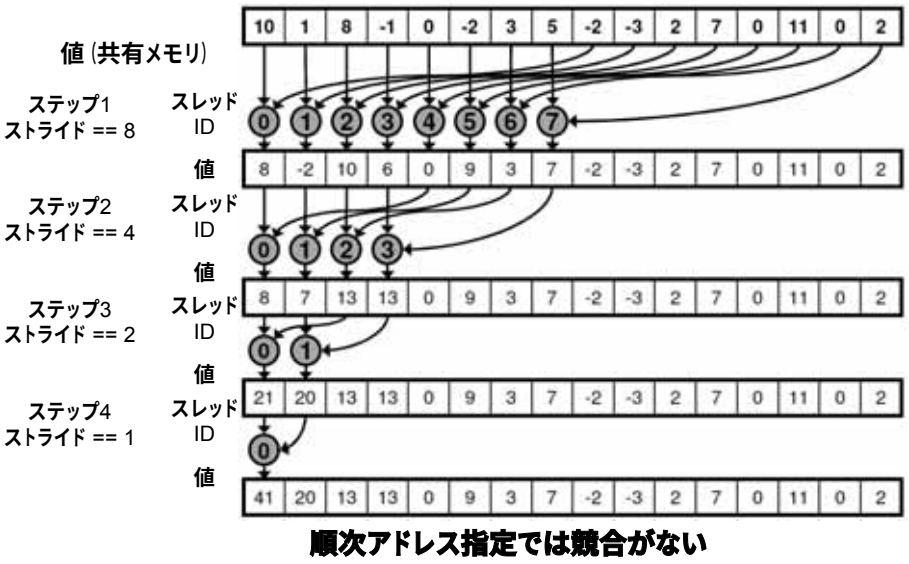
```
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
    sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

新しい問題:
共有メモリでの
バンクの競合

4Mの要素のリダクションのパフォーマンス

	時間 (2^{22}int)	帯域幅	速度向上
カーネル1: インターリーブアドレス指定 (分岐の発散あり)	8.054 ms	2.083 GB/s	
カーネル2: インターリーブアドレス指定 (バンク競合あり)	3.456 ms	4.854 GB/s	2.33倍

並列リダクション: 順次アドレス指定



順次アドレス指定では競合がない

リダクション3: 順次アドレス指定

内部ループのストライドされたインデックスを置き換えるだけ

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
    sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

逆ループとスレッドIDベースのインデックスを使用

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
    sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

4Mの要素のリダクションのパフォーマンス

	時間 (2^{22}int)	帯域幅	速度向上
カーネル1: インターリーブアドレス指定（分岐の発散あり）	8.054 ms	2.083 GB/s	
カーネル2: インターリーブアドレス指定（バンク競合あり）	3.456 ms	4.854 GB/s	2.33倍
カーネル3: 順次アドレス指定	1.722 ms	9.741 GB/s	2.01倍

アイドル状態のスレッド

問題:

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
  if (tid < s) {
  sdata[tid] += sdata[tid + s];
  }
  __syncthreads();
}
```

最初のループのイテレーションで、半分のスレッドがアイドル状態になる!

その部分が無駄になってしまう

リダクション4: ロード時に最初の加算

ブロックを半数にし、1回のロードを置き換える

```
// 各スレッドは1つの要素をグローバルメモリから共有メモリに読み込み
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

2回のロードと最初のリダクションの加算を使用

```
// 最初のレベルのリダクションを実行
// グローバルメモリから読み込み、共有メモリに書き込む
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```


4Mの要素のリダクションのパフォーマンス

	時間 (2^{22}int)	帯域幅	速度向上
カーネル1: インターリーブアドレス指定 (分岐の発散あり)	8.054 ms	2.083 GB/s	
カーネル2: インターリーブアドレス指定 (バンク競合あり)	3.456 ms	4.854 GB/s	2.33倍
カーネル3: 順次アドレス指定	1.722 ms	9.741 GB/s	2.01倍
カーネル4: グローバルのロード時に最初の加算	0.965 ms	17.377 GB/s	1.78倍

命令のボトルネック

- 17 GB/sでは帯域幅の限界には到達しない
  - 前述したように、リダクションは計算集約度が低い
- そのため、考えられるボトルネックは命令のオーバーヘッドである
  - コアの計算のロード、保存、演算を行わない補助的な命令
  - すなわち対処すべきは、アドレス指定の演算とループのオーバーヘッド
- 対策: ループの展開

最後のワープの展開

- リダクションが進むと「アクティブな」スレッド数は減少する
  - s <= 32になると1つのワープしか残っていない
- 命令はワープ内でSIMD同期
- つまりs <= 32の場合
  - __syncthreads()は不要
  - 作業を保存しないため、「if (tid < s)」は不要
- 内部ループの最後の6つのイテレーションを展開しましょう

リダクション5: 最後のワープの展開

```
for (unsigned int s=blockDim.x/2; s>32; s>>=1)
{
    if (tid < s)
    sdata[tid] += sdata[tid + s];
    __syncthreads();
}

if (tid < 32)
{
    sdata[tid] += sdata[tid + 32];
    sdata[tid] += sdata[tid + 16];
    sdata[tid] += sdata[tid + 8];
    sdata[tid] += sdata[tid + 4];
    sdata[tid] += sdata[tid + 2];
    sdata[tid] += sdata[tid + 1];
}
```

注: これにより最後のワープだけでなくすべてのワープ内の不要な作業が省略される

展開しないと、すべてのワープがforループステートメントとifステートメントですべてのイテレーションを実行する

4Mの要素のリダクションのパフォーマンス

	時間 (2^{22}int)	帯域幅	速度向上
カーネル1: インターリーブアドレス指定 (分岐の発散あり)	8.054 ms	2.083 GB/s	
カーネル2: インターリーブアドレス指定 (バンク競合あり)	3.456 ms	4.854 GB/s	2.33倍
カーネル3: 順次アドレス指定	1.722 ms	9.741 GB/s	2.01倍
カーネル4: グローバルのロード時に最初の加算	0.965 ms	17.377 GB/s	1.78倍
カーネル5: 最後のワープを展開	0.536 ms	31.289 GB/s	1.8倍

完全な展開

- コンパイル時に繰り返し数がわかっていれば、完全にリダクションを展開できる
  - 幸運にも、ブロックサイズはGPUによって512スレッドに制限されている
  - また2の累乗のブロックサイズを遵守している
- そのため、固定ブロックサイズで簡単に展開できる
  - しかし汎用的でなくてはならない: どうすればコンパイル時に不明なブロックサイズを展開できるのか?
- このようなときに役立つのがテンプレート
  - CUDAは、デバイス関数とホスト関数でC++テンプレートパラメータをサポートする

テンプレートでの展開

- ブロックサイズを関数テンプレートのパラメータで指定する

```
template <unsigned int blockSize>
__global__ void reduce5(int *g_idata, int *g_odata)
```

リダクション6: 完全な展開

```
    if (blockSize >= 512) {
      if (tid < 256) { sdata[tid] += sdata[tid + 256]; }
        __syncthreads();
    }
    if (blockSize >= 256) {
      if (tid < 128) { sdata[tid] += sdata[tid + 128]; }
        __syncthreads();
    }
    if (blockSize >= 128) {
      if (tid < 64) { sdata[tid] += sdata[tid +  64]; }
        __syncthreads();
    }

    if (tid < 32) {
      if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
      if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
      if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
      if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
      if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
      if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
    }
```

注: 太字のコードはコンパイル時に評価される

内部ループが非常に効率的になる

テンプレートカーネルの起動

- コンパイル時にブロックサイズが必要ではないのか?
  - 想定されるブロックサイズ10個分について、switchステートメントを記述するだけで済みます

```
switch (threads)
    {
    case 512:
      reduce5<512><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 256:
      reduce5<256><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 128:
      reduce5<128><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 64:
      reduce5< 64><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 32:
      reduce5< 32><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
      case 16:
      reduce5< 16><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 8:
      reduce5< 8><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 4:
      reduce5< 4><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 2:
      reduce5< 2><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 1:
      reduce5< 1><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    }
```

4Mの要素のリダクションのパフォーマンス

	時間 (2^{22}int)	帯域幅	速度向上
カーネル1: インターリーブアドレス指定（分岐の発散あり）	8.054 ms	2.083 GB/s	
カーネル2: インターリーブアドレス指定（バンク競合あり）	3.456 ms	4.854 GB/s	2.33倍
カーネル3: 順次アドレス指定	1.722 ms	9.741 GB/s	2.01倍
カーネル4: グローバルのロード時に最初の加算	0.965 ms	17.377 GB/s	1.78倍
カーネル5: 最後のワープを展開	0.536 ms	31.289 GB/s	1.8倍
カーネル6: 完全に展開	0.381 ms	43.996 GB/s	1.41倍

並列リダクションの計算量

- Log(*N*)の並列ステップでは、各ステップ*S*は*N*/2の独立した演算を実行する
  - ステップの計算量はO(log *N*)である
- N=2^Dの場合$\sum_{S\in[1..D]} 2^{D-S} = N-1$の演算を実行する
  - 仕事計算量はO(*N*): 仕事効率が良い
  - すなわち、順次アルゴリズムよりも実行する演算が多いわけではない
- 物理的に並列な*P*個のスレッドでは（*P*個のプロセッサ）、時間計算量はO(*N*/*P* + log *N*)
  - 順次リダクションのO(*N*)と比較する
  - スレッドブロックではN=P、そのためO(log *N*)

コストとは

- 並列アルゴリズムのコストはプロセッサxの時間計算量
  - プロセッサの代わりにスレッドを割り当てる: O(*N*)スレッド
  - 時間計算量はO(log *N*)なのでコストはO(*N* log *N*):コスト効率性はよくない
- ブレント理論ではO(*N*/log *N*)スレッドが提唱されている
  - 各スレッドはO(log *N*)の順次作業を実行する
  - 次にすべてのO(*N*/log *N*)スレッドはO(log *N*)ステップを協調する
  - コスト = O((*N*/log *N*) * log *N*) = O(*N*)
- アルゴリズムのカスケードと呼ばれる
  - 実際に大幅な速度の向上につながる

アルゴリズムのカスケード

- 順次と並列のリダクションを組み合わせる
  - 各スレッドは複数の要素をロードして加算し、共有メモリに読み込む
  - 共有メモリでツリーベースのリダクション
- ブレント理論によると、各スレッドはO(log *n*)要素を加算する
  - 1ブロックあたり1,024または2,048の要素 (対して256要素)
- 経験によると、カスケードにはより大きな利点があった
  - スレッドあたりの仕事が多くなると、よりレイテンシを隠すことができる
  - 1ブロックあたりのスレッドが増えると、再帰的なカーネル呼び出しのツリーのレベルが減少する
  - ブロック数が少ない最後のレベルでのカーネル起動のオーバーヘッドが高い
- G80では128スレッドの64～256ブロックが最もよいパフォーマンス
  - 1スレッドあたり1,024～4,096の要素

リダクション7: スレッドあたり、複数の加算

2要素のロードと追加を置き換える

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

必要なだけ追加するwhileループに置き換える

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;

do {
   sdata[tid] += g_idata[i] + g_idata[i+blockSize];
   i += gridSize;
}  while (i < n);
__syncthreads();
```


4Mの要素のリダクションのパフォーマンス

	時間 (2^{22}int)	帯域幅	速度向上
カーネル1: インターリーブアドレス指定（分岐の発散あり）	8.054 ms	2.083 GB/s	
カーネル2: インターリーブアドレス指定（バンク競合あり）	3.456 ms	4.854 GB/s	2.33倍
カーネル3: 順次アドレス指定	1.722 ms	9.741 GB/s	2.01倍
カーネル4: グローバルのロード時に最初の加算	0.965 ms	17.377 GB/s	1.78倍
カーネル5: 最後のワープを展開	0.536 ms	31.289 GB/s	1.8倍
カーネル6: 完全に展開	0.381 ms	43.996 GB/s	1.41倍
カーネル7: 1スレッドあたり複数要素	0.268 ms	62.671 GB/s	1.42倍

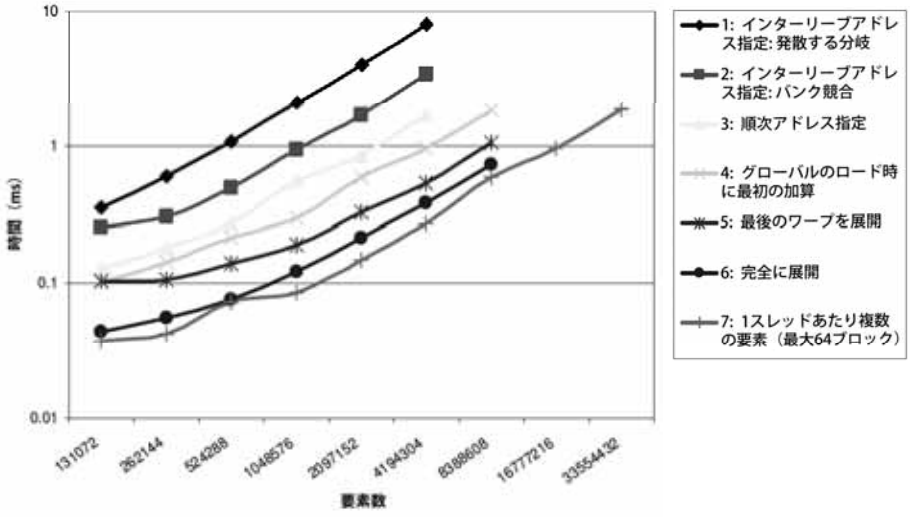
32M要素のカーネル7: 72 GB/s!

合計の速度向上: 30倍!

```
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n)
{
  extern __shared__ int sdata[];

  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*(blockSize*2) + tid;
  unsigned int gridSize = blockSize*2*gridDim.x;
  sdata[tid] = 0;

  do { sdata[tid] += g_idata[i] + g_idata[i+blockSize]; i += gridSize; } while (i < n);
  __syncthreads();

  if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
  if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
  if (blockSize >= 128) { if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads(); }

  if (tid < 32) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
    if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
    if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
    if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
  }

  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

最適化された最終的なカーネル

パフォーマンスの比較



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