GPU上の大規模格子QCDシミュレーションに向けて

- 目標と現状
- 格子QCDの問題
- これまでの結果
- オーバーラップ演算子への展望

松古 栄夫 (hideo.matsufuru@kek.jp)

In collaboration with

- 青山龍美、野秋淳一、山田憲和 (KEK)
- Special thanks to 石川健一、尾崎裕介 (広島大)

High Energy Accelerator Research Organization (KEK)

2008年6月24日 GPGPU研究会

目標

(1) 格子QCDの大規模シミュレーション

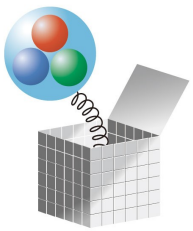
厳密なカイラル対称性を持つ格子フェルミオン

- PFlops級の計算にGPUは役に立つか？
- 現在のスパコンレベルの計算をもっと手軽に

(2) 素粒子・原子核・宇宙の計算への応用

新学術領域研究(代表：青木慎也): bridge.kek.jp

- 問題にあったアーキテクチャ、アルゴリズムを
  - GPUはどのようなタイプの問題に向いているか？
- チューニング・ノウハウの蓄積： 2割の労力で8割の性能を！
- どのようにコードを書いておくとよいか？



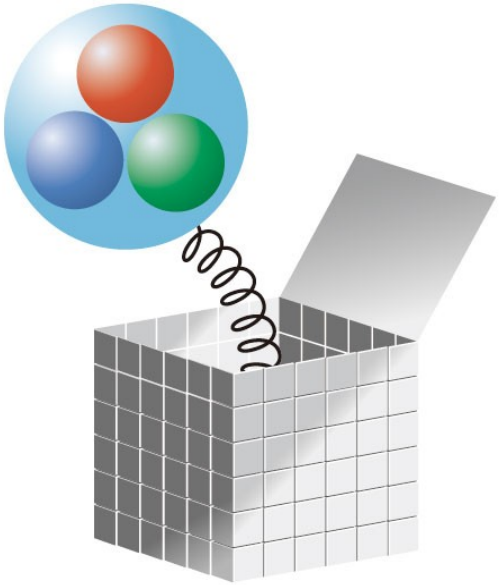
現状

- Wilson クォーク演算子でCUDAプログラミングの練習
  - 基本的なクォーク演算子
  - チューニング手法
  - 単精度と倍精度の比較
- これから：
  - シングルGPU → マルチGPUへ
  - GPU向きのアルゴリズムの探求
  - 複雑なフェルミオン演算子

計算リソース

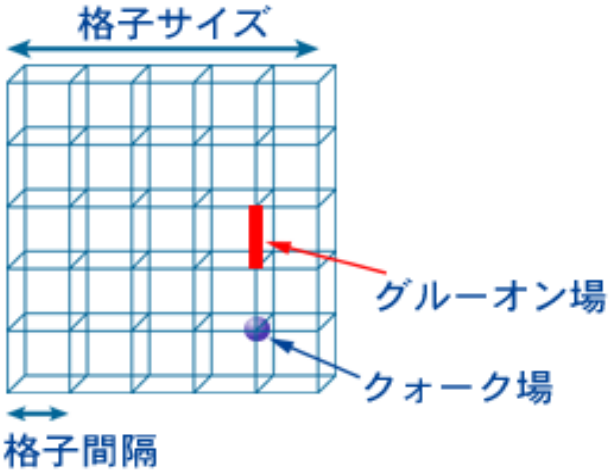
- Tesla S1070x4 (16 GPU) @KEK
- 新領域関係の人は利用できます

格子QCDの問題



格子QCDシミュレーション

- 格子QCD(量子色力学): 格子時空上の場の理論
  - グルーオン場： リンク上の3x3複素行列
  - クォーク場：サイト上のグラスマン数
    （計算機上で扱えないので手で積分）
  - 経路積分量子化 → 統計力学系と同じ形
  - モンテカルロ法によるシミュレーション
  - 低エネルギーでQCDを扱う唯一の一般的方法

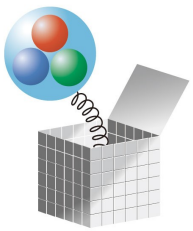


- 物理量の期待値:

$$\langle O \rangle = \int \underline{\mathcal{D}U\mathcal{D}\phi^\dagger\mathcal{D}\phi}\, O[U] \exp\left[-S_G[U] - \phi^\dagger D^{-1}[U]\phi\right]$$

グルーオン場U, 擬クォーク場 φ についての積分
→ モンテカルロ法で生成

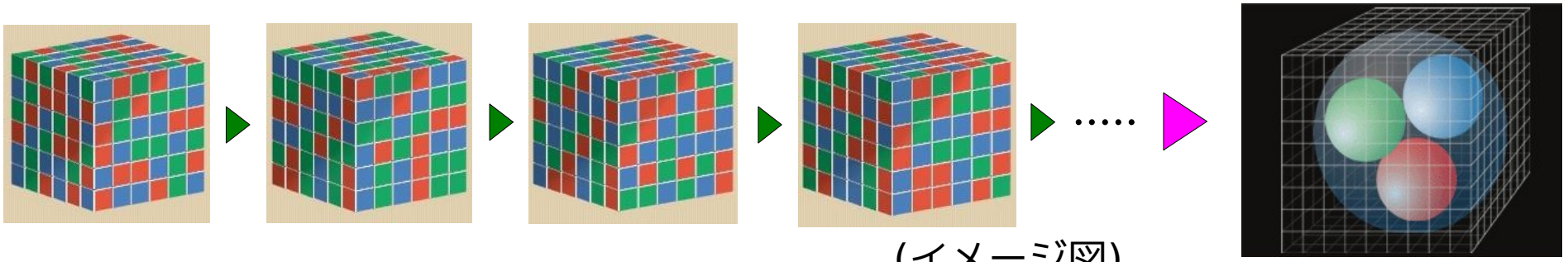
擬クォーク場の有効作用 (このDの逆を解くのに時間かかる)



格子QCDシミュレーション

モンテカルロ法: グルーオン場の配位{U} (と擬クォーク場 φ) を

$$\exp\left[-S_G[U] - \phi^\dagger D^{-1}[U]\phi\right]$$ の確率で生成



(イメージ図)

- ハイブリッド・モンテカルロ法
  - グルーオン場に共役運動量を導入
    → 分子動力学 （Hamiltonian を保存する発展方程式）
  - 擬フェルミオン場φをランダム(Gaussian)に生成
  - 各ステップで $D^{-1}\varphi$を求める必要
- 生成したグルーオン場の配位を使って、物理量を計算
  - Ex. クォークの伝播関数（D^{-1}）からハドロンの相関関数を構成
  - 統計平均→物理量の期待値

格子QCDの問題

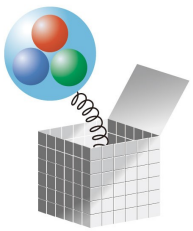
計算時間のほとんどは、線形方程式　$Dx=b$ を解いている

- x, b は N=3(カラー) x 4(スピン) x サイト (例えば 16^4)　の自由度を持つ複素ベクトル [$N \sim O(10^6\text{-}10^8)$]
- Krylov部分空間法

フェルミオン演算子 D : N x N 巨大行列 (メモリに載せるのは U)

- Wilson演算子
  - シンプルな構造、 カイラル対称性は連続極限で回復
  - 計算コストは比較的低い
- オーバーラップ演算子
  - 格子上の厳密なカイラル対称性を持つ
  - 計算コスト高： Wilson演算子をO(100)回かける必要

Dの演算のスピード、線形解法の効率を上げることが重要



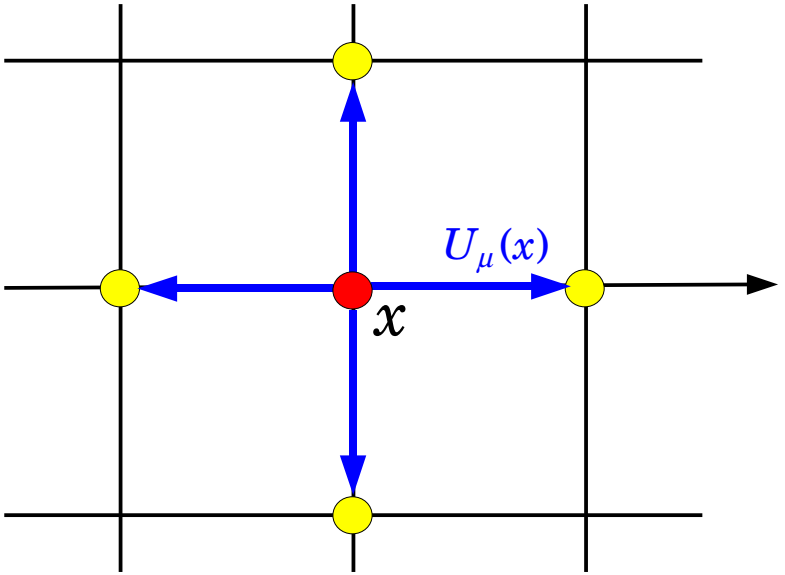
Wilson演算子の場合

- D は N x N の巨大疎行列

3x3複素行列: リンク変数 ∈SU(3)
（グルーオン場の情報）

$$D_{x,y} = \delta_{x,y} - \kappa \sum_{\mu} \{(1-\gamma_\mu) U_\mu(x) \delta_{x+\hat{\mu},y} + (1+\gamma_\mu) U_\mu(x-\hat{\mu}) \delta_{x-\hat{\mu},y}\}$$

対角成分　+μ方向の隣接サイトとの結合　−μ方向の隣接サイトとの結合

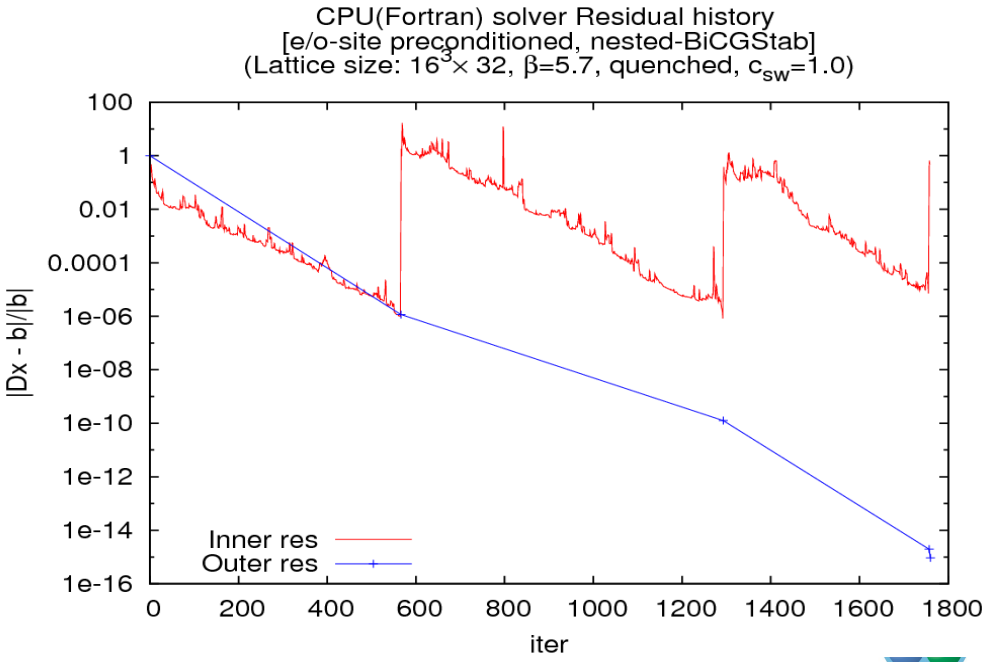


これまでの研究

- Egri, Fodor, Hoelbling, Katz, Nogradi, Szabo
  - Pioneering work, OpenGL (w/o CUDA)
  - PoS (Lat2006) 034, Comput. Phys. Commun. 177 (2007) 631
- Clark, Barros, Babich, Brower, Rebbi
  - PoS (Lattice2008) 045, etc.
- Ishikawa and Osaki
  - 尾崎さんの講演を参照

Wilson kernel solver

- 最も時間がかかる部分
- 倍精度が必要な場合:
  Single precision solver
  を前処理として使う



（尾崎-石川、物理学会2009年春より）

オーバーラップ演算子

- 我々の目標：オーバーラップ演算子を扱いたい

$$D = \frac{1}{Ra}\left[1 + \gamma_5 \,\mathrm{sign}\left(H_W(-m_0)\right)\right]$$

H_WはWilson演算子

  - Sign 関数はH_Wの固有ベクトル展開＋近似式で評価
  - Wilson演算子が速く計算できることが最低条件

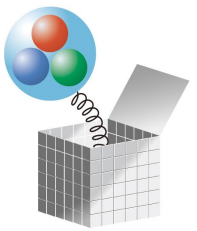
- 計算量大: DあたりWilson 演算子をO(100)回かける
- 現在は Blue Gene/L (1ラック=5.6TFlops) で計算
  - 24^3x48 格子, O(12)時間 x O(1000)
  - 16^3x48 格子，O(2) 時間 x O(2500)

実戦ではこのサイズ以上が必要！

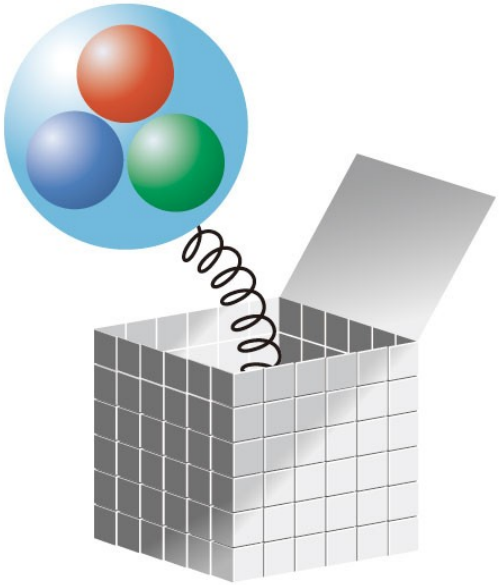
Cf. JLQCD Collaboration (jlqcd.kek.jp)

方針

- まず、Wilson演算子でいろいろやってみる
  - チューニング手法の習得
  - 線形問題の高速化： 格子QCDの最も時間のかかる部分
  - Wilson演算子は格子フェルミオンの基本
    - 他の演算子(staggered, domain-wallなど)への拡張は容易
    - オーバーラップ演算子の核部分
  - 単精度と倍精度： 性能比較＆どこで倍精度が必要か？
  - ハイブリッド・モンテカルロ法： どこまでをGPUでやるか？
  - 並列化
    - ボード内: OpenMP/MPI, ボード間: MPI
    - 演算と通信のオーバーラップ
- オーバーラップ演算子への拡張
  - 並列化は不可欠
  - アルゴリズムの改良



これまでの結果



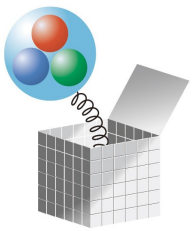
ベンチマーク問題

Wilson 演算子の線形問題

- まずはシングルGPUで実装
- 前処理 (even-oddなど) なし
- 単精度と倍精度の性能比較

現状：

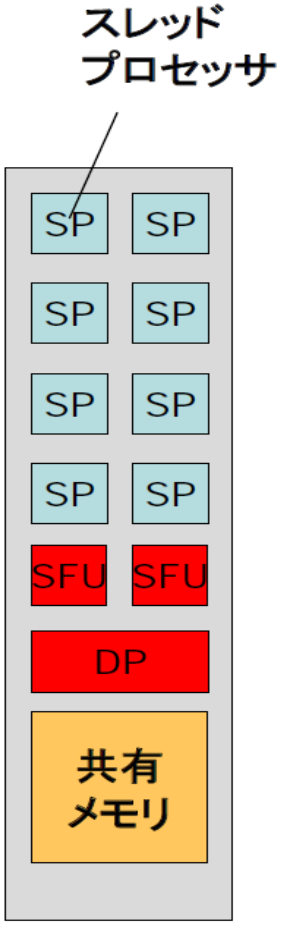
- これまでの結果
  - Wilson kernelの演算 (線形代数部分の比較はまだ)
  - 単精度と倍精度の比較
  - textureを利用
- これからの課題
  - 共有メモリ、コンスタントメモリの利用
  - 線形問題での性能比較と最適化
  - 複数GPUの利用 (ボード内、ボード間)

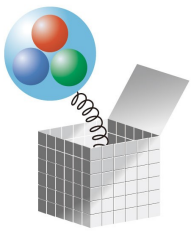


利用環境

GT200 (GeForce GTX280, Tesla C1060/S1070)

- Clock 1.3GHz
- 240 thread processor (8 x 30 multiproc.)
- Peak ~1TFlops
- Multiprocessor:
  - 8 スレッドプロセッサ/ 1倍精度演算器/ 2 Special Function Unit
  - 16384 32-bit レジスタ
  - 16KB 共有メモリ
- 4GB GDDR3
  - Memory bandwidth 102GB/s (800MHz DDR)
  - Flops >> B/s (~10 times)
- Interface: PCI Express 2.0 x16 : 8GB/s



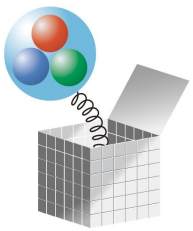


書き換え手順

GPUを使うコードに変換
- どの部分をGPUにやらせるか？
- (C/C++コードの作成)
- Thread と blockの切り方を決める
- メモリへのアクセスを減らす

→ チューニング

- Wilson kernel mult の実装
- 1 格子点を 1 スレッドに対応
  - v=UGw (U: color 空間に作用、G: スピノール空間に作用)
    - v, w: 3(color)x4(spinor)x2(Re/Im) = 24 成分 vector
    - U: Link variable: 3x3 complex matrix [SU(3)]
  - GFlops は、CPUでの同等演算に換算
    - サイトあたり、1368回の浮動小数点演算

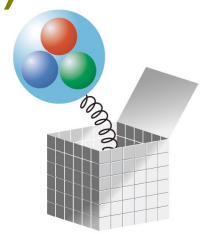


メモリアクセス

- メモリアクセス:
  - 1 vector & 1 link var. のロード x8回 (±μ) + 1 SAXPY
  - サイトあたり、1248B のデータのロード、96B のストア
    ＝1344Bのメモリアクセス
  - B/F ~1 (CPU演算換算)
  - memory b/w が律則
- Special unitarity を利用してメモリアクセスを減らす
  - 2 列(or 行) 分(12 float)をロード、残りは on-the-fly で再構成

$$\begin{pmatrix} v_1 & w_1 & z_1 \\ v_2 & w_2 & z_2 \\ v_3 & w_3 & z_3 \end{pmatrix} \Rightarrow \begin{pmatrix} v_1 & w_1 \\ v_2 & w_2 \\ v_3 & w_3 \end{pmatrix}, \quad z = (v \times w)^*$$

  - もっと凝った方法: 8個の変数で表すことも可能 (Clark et al.)

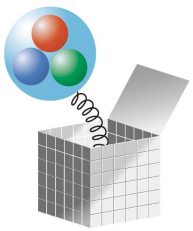


チューニングの手順

（石川さん、尾崎さんによる指南）

- Coalesced access
  - 組込みベクトル型（float4, float2 など）の利用
  - 配列構造の変更：

    float2/4で定義した変数について、スレッドに対応するindexを最内側に配置→その他の自由度→ブロックの自由度
- Texture の利用 (現在ここまで)
- 共有メモリ、constant メモリの利用
- Thread/block sizeの最適化など

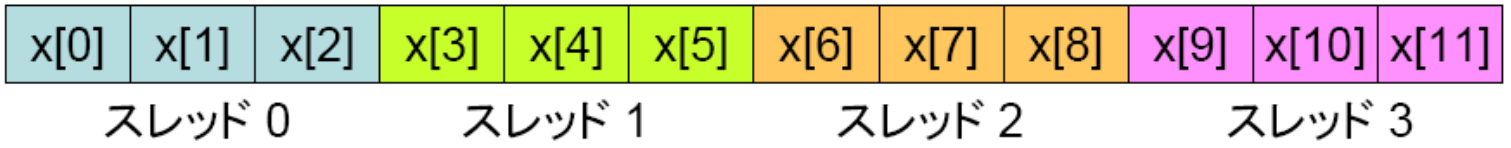


Coalesced access

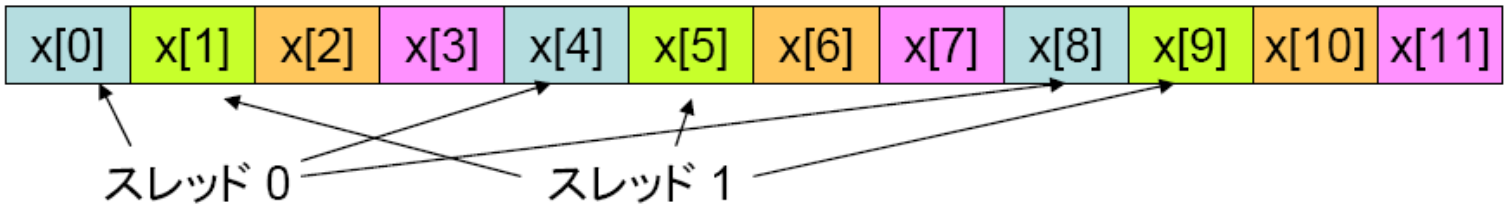
(SGI講習会資料より)

- メモリアクセス命令はハーフワープ(16)単位で実行
  - 一回でハーフワープ分のスレッドのデータを読み込める
  - 要素のデータ型が4, 8, or 16 Bであること
  - 先頭アドレスが16*sizeof(type)に整列していること

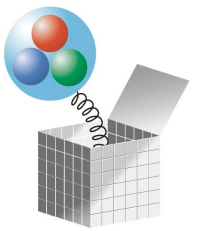
- パターン 1: まとめて均等に分割



- パターン 2: ひとつずつサイクリックに分割



こちらが速い！



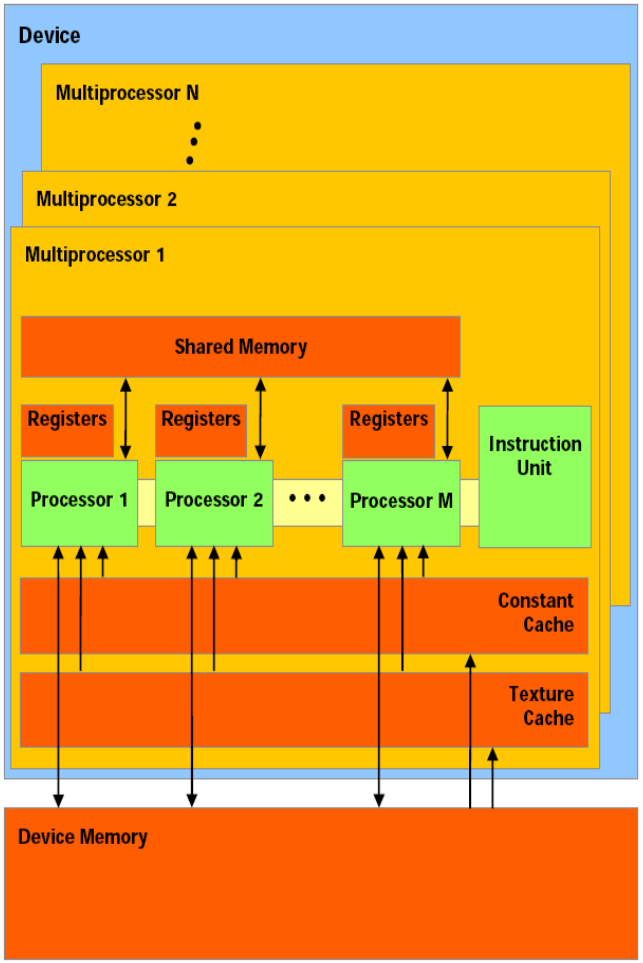
メモリ空間

ハードウェア・モデル

（SGI講習会資料、CUDAマニュアルより）

デバイスメモリ上の変数は、アクセス方法の違いで３つのメモリ空間に分類

- グローバルメモリ空間
  - 一般的、cudaMalloc, __device__
  - キャッシュされない
  - Coalescingしないと遅い
- テクスチャメモリ空間
  - テクスチャリファレンス経由でアクセス
  - カーネルからは読み込み専用
  - キャッシュされる
- コンスタントメモリ空間
  - __constant__宣言
  - 読み込み専用
  - キャッシュされる



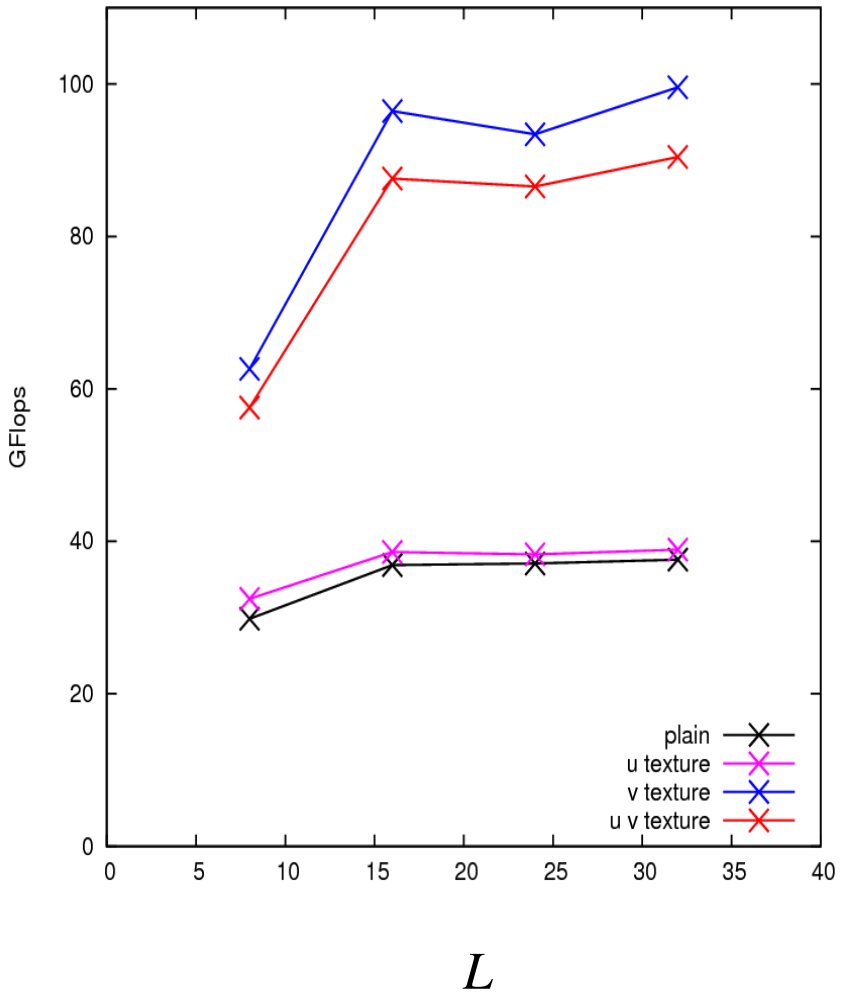
Preliminary result

Wilson kernel mult

- Lattice size: 2L x L^3
- Thread数: 128
- CPU演算に換算した性能値
  (実際の演算量：1368→1704, +25%)

Performance:

  - w/oテクスチャで~40GFlops
  - テクスチャを使うと ~100GFlops
- ベクトルにテクスチャを使うと大きな効果 (cacheの効果)
- リンク変数にはあまり効かない?
  - 同じリンク変数を2回しか使わない



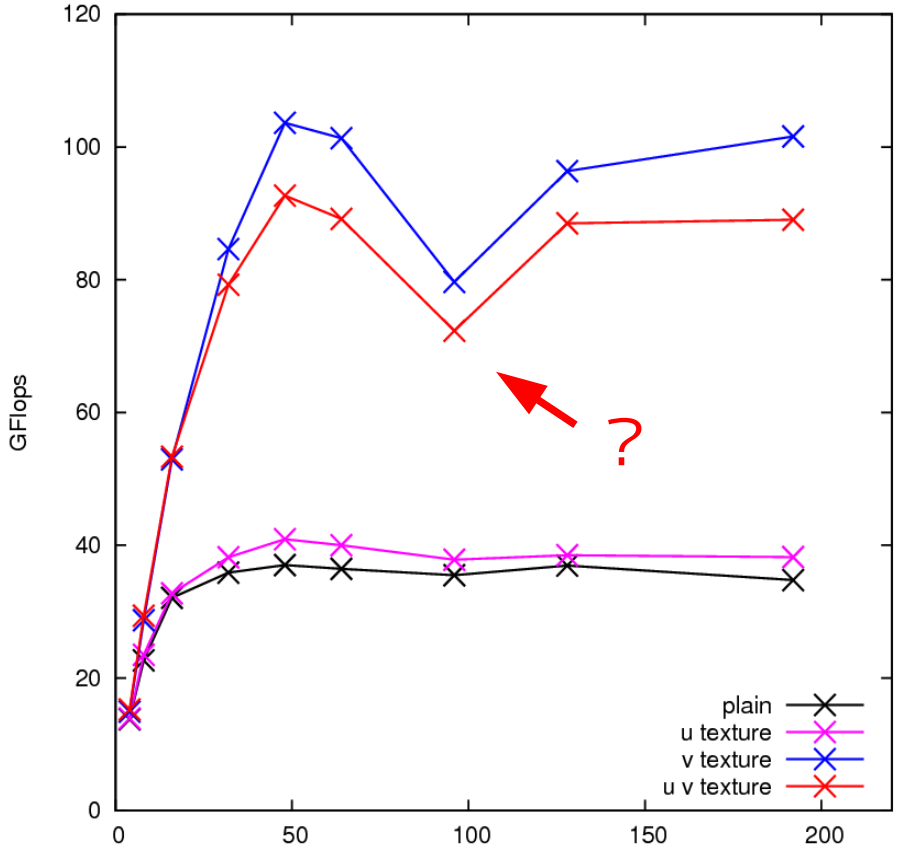
Preliminary result

スレッド数依存性

- 32x24x16x16 lattice
- スレッド(＝サイト)あたりで使うレジスタ数：66
- １マルチプロセッサあたりのレジスタ数：16384

  → スレッド数のmax: 192
- スレッド数>48で高性能
- スレッド数 ~96で性能低下

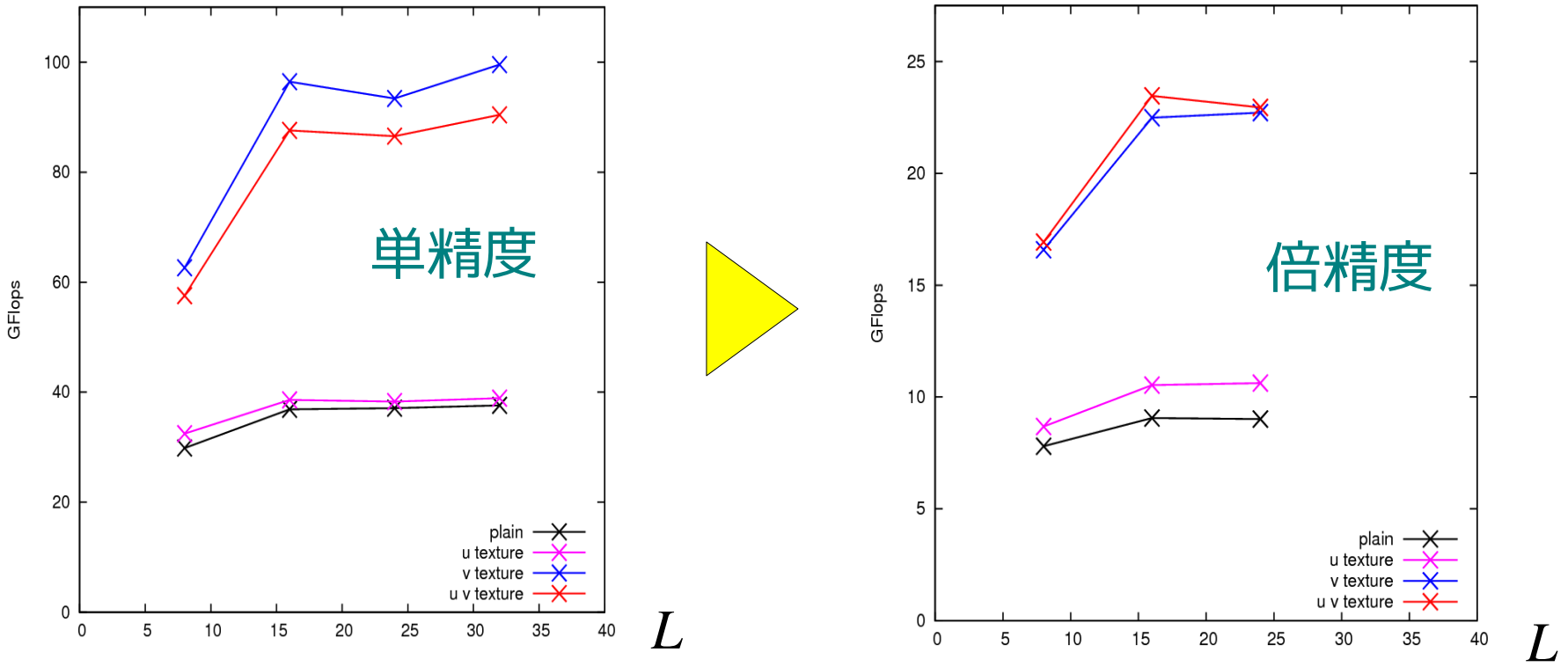
  (理由はまだ不明)



Preliminary result

倍精度： Wilson kernel mult

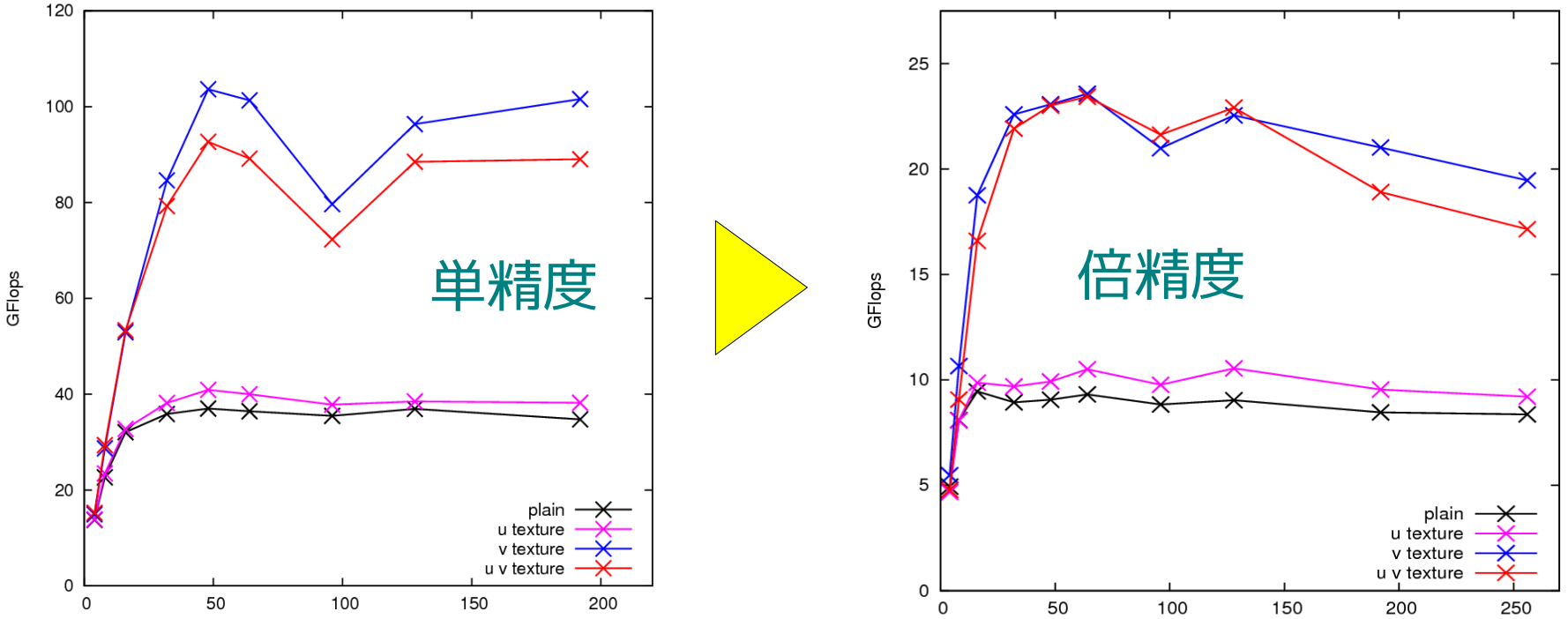
- Lattice size: 2L x L^3, thread数: 128



- Single の1/4 程度のスピード → 悪くない!!
- 傾向はsingleの場合と同様

Preliminary result

倍精度： スレッド数依存性



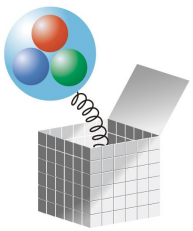
- 倍精度での最適スレッド数： ~64
- 倍精度でのスレッド数>150 での性能低下はキャッシュミスの効果？

ここまでわかったこととこれから

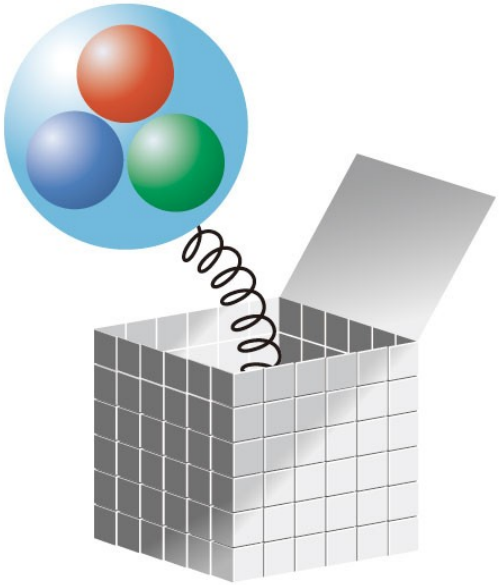
- Coalesced アクセス＋テクスチャ: CPU換算で~100GFlops
- スレッド数のチューニング
- 倍精度は単精度の1/4 程度の性能

これからやること

- 更にチューニング
  - 共有メモリ、コンスタントメモリの利用など
- 他の部分も含めた高速化
  - 線形代数: CUDA BLAS library or 実装
  - 発展方程式: 単精度でよいか (つじつま合わせできるか？)
- Double演算器をどう使うと良いか？
- 複数GPUへの拡張
  - ボード内(OpenMP/MPI)、ボード間(MPI)



オーバーラップ演算子への展望



オーバーラップ演算子

我々の目標：オーバーラップ演算子を扱いたい

$$D = \frac{1}{Ra}\left[1 + \gamma_5 \,\mathrm{sign}(H_W(-m_0))\right]$$

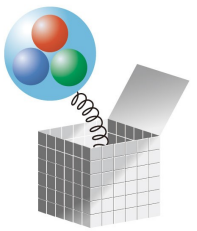
H_W はWilson演算子

- 計算量大: O(1)TFlops (sustained)が必要

- 近似式で符号関数を評価

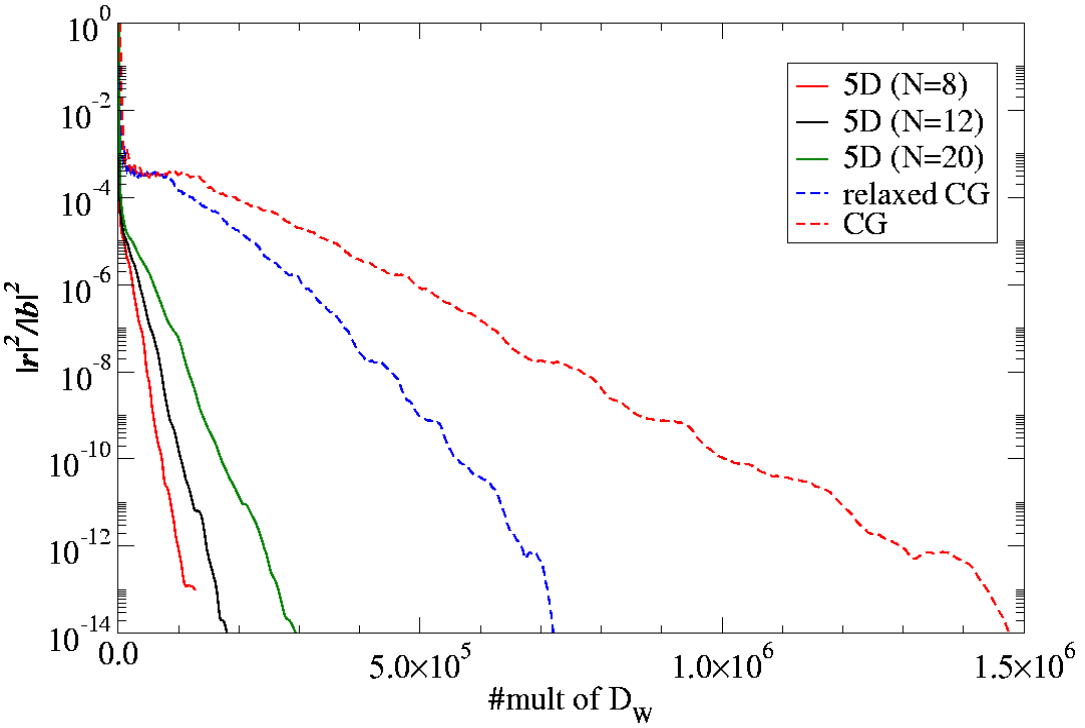
$$\mathrm{sign}(H_W) = \frac{H_W}{\sqrt{H_W^2}} = H_W\left(p_0 + \sum_{l=1}^{N} \frac{p_l}{H_W^2 + q_l}\right)$$

  - $(H_W^2 + q_l)^{-1}$ (l=1,..., N~20) を解く必要:
    マルチシフト・ソルバー (単精度による前処理が難しい)
  - H_W の小さい固有モードの射影による高速化
    - 固有値、固有ベクトルを求める必要



オーバーラップ演算子の線形方程式

- 4D solver: 上の近似式を直接使って D を構成
  - CG法を入れ子にして使う(内側はマルチシフト)
  - 単精度での問題：マルチシフトCGに対する可変前処理法の困難
- 5D solver: ５次元の行列を使って、１重のCGで解く
  - 4D solverより高速



オーバーラップ演算子の５Dソルバー

オーバーラップ演算子に関する線形方程式を、5次元に拡張した
行列を解くことによって解く

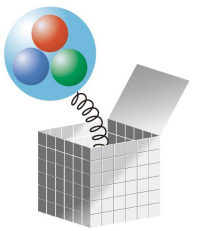
$$M_5 = \left(\begin{array}{cccc|c} H_W & -\sqrt{q_2} & & & 0 \\ -\sqrt{q_2} & -H_W & & & \sqrt{p_2} \\ & & H_W & -\sqrt{q_1} & 0 \\ & & -\sqrt{q_1} & -H_W & \sqrt{p_1} \\ \hline 0 & \sqrt{p_2} & 0 & \sqrt{p_1} & R\gamma_5 + p_0 H \end{array}\right) = \left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right)$$

(N=2の場合)

$S = D - CA^{-1}B$ がオーバーラップ演算子になるようにパラメターを選ぶ

$$M_5 \begin{pmatrix} \phi \\ \psi_4 \end{pmatrix} = \begin{pmatrix} 0 \\ \chi_4 \end{pmatrix}$$ を解くことによって$S\psi_4 = \chi_4$ の解が得られる

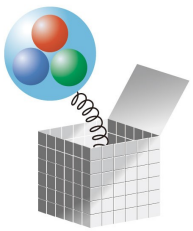
- 基本的にはWilson演算子の場合と同様
- オーバーラップ演算子のソルバー以外に発展方程式の
フォースの計算でもマルチシフトCGが必要



展望

現実的なシミュレーションでは、

- 並列化が不可欠
  - 通信をどうオーバーラップさせるか
  - 粒度の粗い並列計算向きのアルゴリズム (領域分割、マルチグリッド)
- 精度に注意する部分
  - Hamiltonianの計算 (large cancellation)
  - 分子動力学部分の reversibility
  - 固有値問題 (?)
  - 単精度ソルバーを前処理として利用するのは、マルチシフト・ソルバーに適用困難 (RHMCでも同じ問題)
  - 倍精度@GPUの有効利用



展望

期待

- ECCがつけば、ほとんどの演算をGPUでできる
- 倍精度演算器が増えればgood
- GPU間でのデータの通信ができればgood