まだCPUで消耗してるの？ Jubatusによる近傍探索のGPUを利用した高速化

ヱヂリウム株式会社
渡邉卓也 室井浩明

Jubatus Casual Talks #4
2016年6月18日

やりたいこと

- Jubatusの近傍探索を商用利用したい
  - LSHによる近傍探索をリアルタイムなレコメンデーションに活用したい
- 性能面の制約がきつい
  - 規定の応答時間に納める為に、データ件数やLSHのハッシュ数を抑える必要があった
  - CPUの機能の活用や無駄な処理の削除で徐々に高速化している
  - 0.9.1でさらに高速化したと聞く
- GPGPUで一気に高速化できないか
  - 近傍探索は各ベクトルについて独立に処理でき、最もGPU向きな処理の一つである
  - 探索対象のビットベクトルは予めGPU側メモリに格納しておけばよいので、近傍探索時のデータ転送は少ない
  - 目標は2000万件を100 ms以内

GPGPUとは

- Graphics Processing Unit (GPU)
  - 主にゲームの映像を描画する為に発展してきたプロセッサ
  - 中身は高並列度のベクトル計算機
    - 最近はCPUにもベクトル演算器が入っているが、それらよりもずっと並列度が高い（cf. SSE, AVX）
- General Purpose GPU (GPGPU)
  - GPUを汎用のベクトル計算機として利用
    - かつてのベクトル型スーパーコンピュータは科学技術計算に利用
    - GPUは安価なので手軽にビジネス用途にも活用できる
  - nVidiaのCUDA（C言語ベース）を利用するのが一般的
    - 実行方式やメモリアクセスパターンについて慎重に検討する必要がある
      - CPUで高速な方式がGPUで高速とは限らない（むしろ大抵遅い）
    - 高い並列度を活かして力押しするのが基本
      - できるだけメモリ帯域を活かすようメモリアクセスパターンを設計する

基本方針

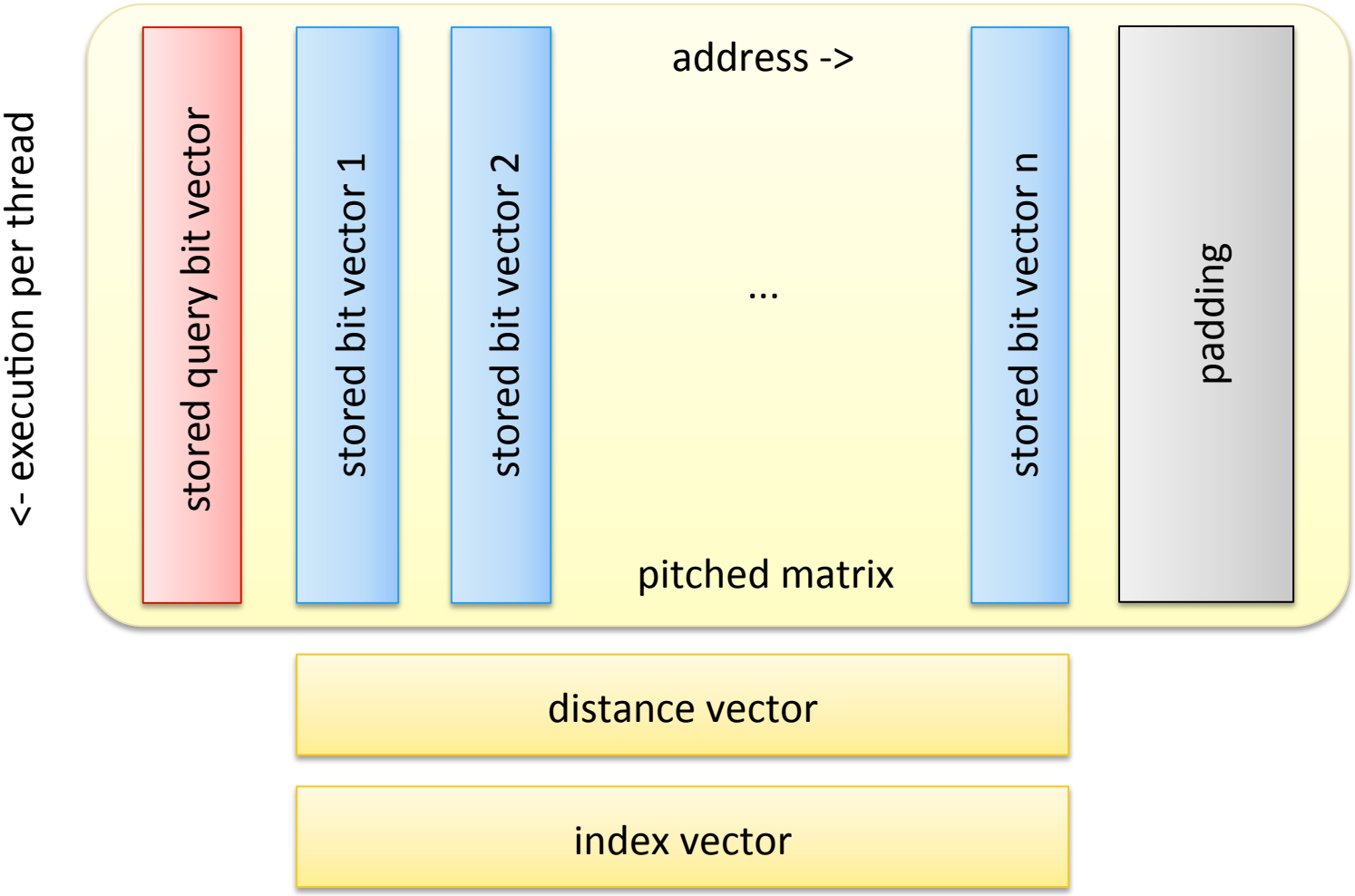
- 対象
  - jubanearest_neighborを改造
    - bit_vector_nearest_neighbor_base.cppを.cuにリネームして改造
      - .cuはCUDAソースファイルの拡張子
      - 本来ならstorageとして実装すべきであろうか
  - LSHのみ実装
- とりあえず動く物を作る
  - どの程度の性能が見込めるのかを計測する為なので動けばよい
  - 性能に関係する箇所は方式通りきちんと実装する
- できるだけ手を抜く
  - 既存の実装があればそれを利用
  - 性能に関係が薄いところは非効率でもそのまま
  - エラー処理は最小限（すらしない）

プロジェクトへの組み込み

- wafで.cuをビルド可能にする
  - wafはJubatusの利用しているmakeのようなもの
  - wafは標準ではCUDAに対応していない
  - 既存の実装を利用
    - https://github.com/krig/waf/blob/master/playground/cuda/cuda.py
  - Cプログラムに対する処理を拡張して.cuの場合にnvcc（CUDAコンパイラ）を利用してくれる
  - 作りかけっぽくそのままでは動かない
    - 数カ所修正の必要があった
- ライブラリやライブラリパスの指定
  - conf.env.LINKFLAGSに-lcudaや-lcudartを追加
  - 適宜CUDAのライブラリパスも追加

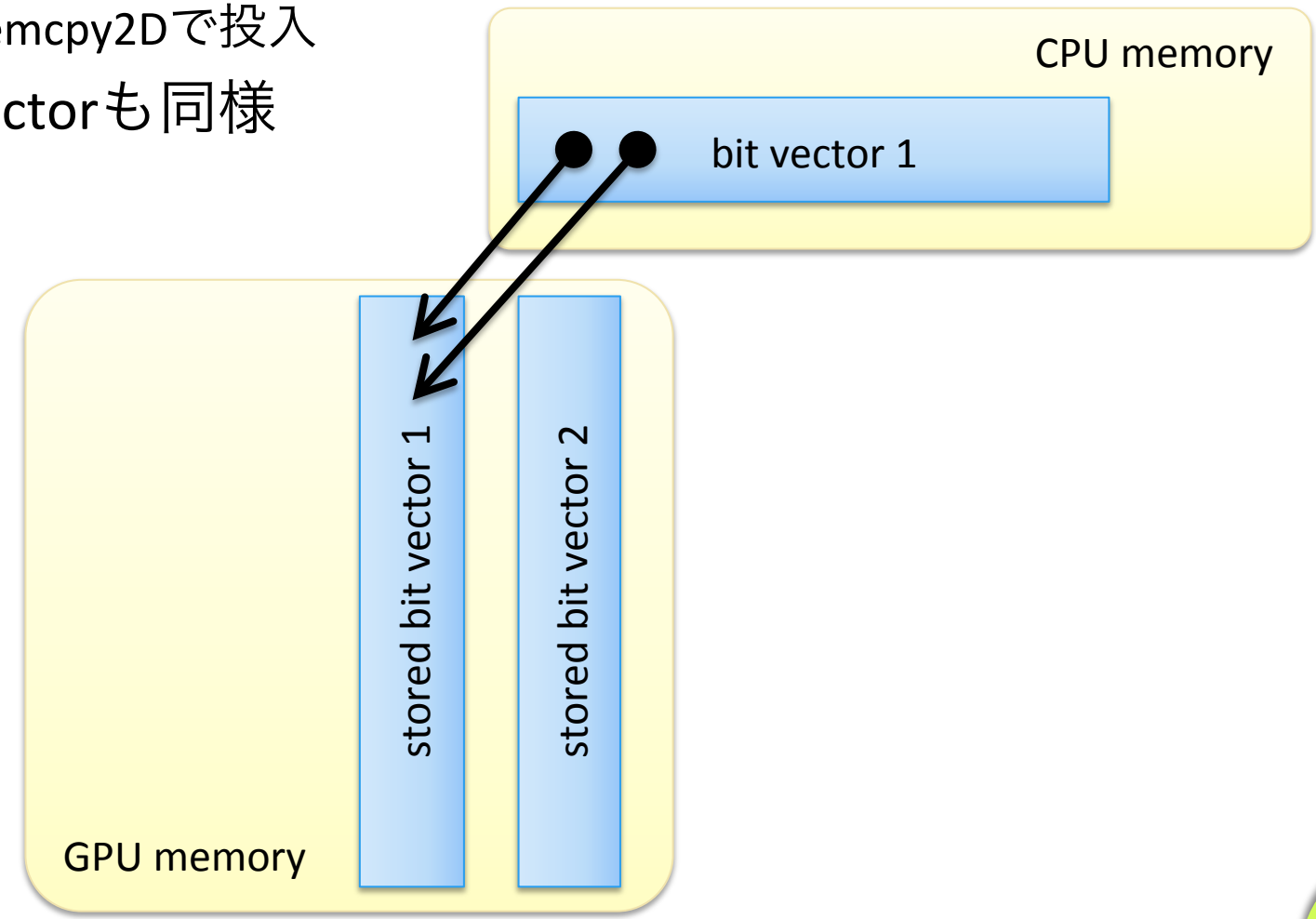
メモリレイアウト

- column-major orderでGPU側メモリに格納
  - cudaMallocPitchで確保



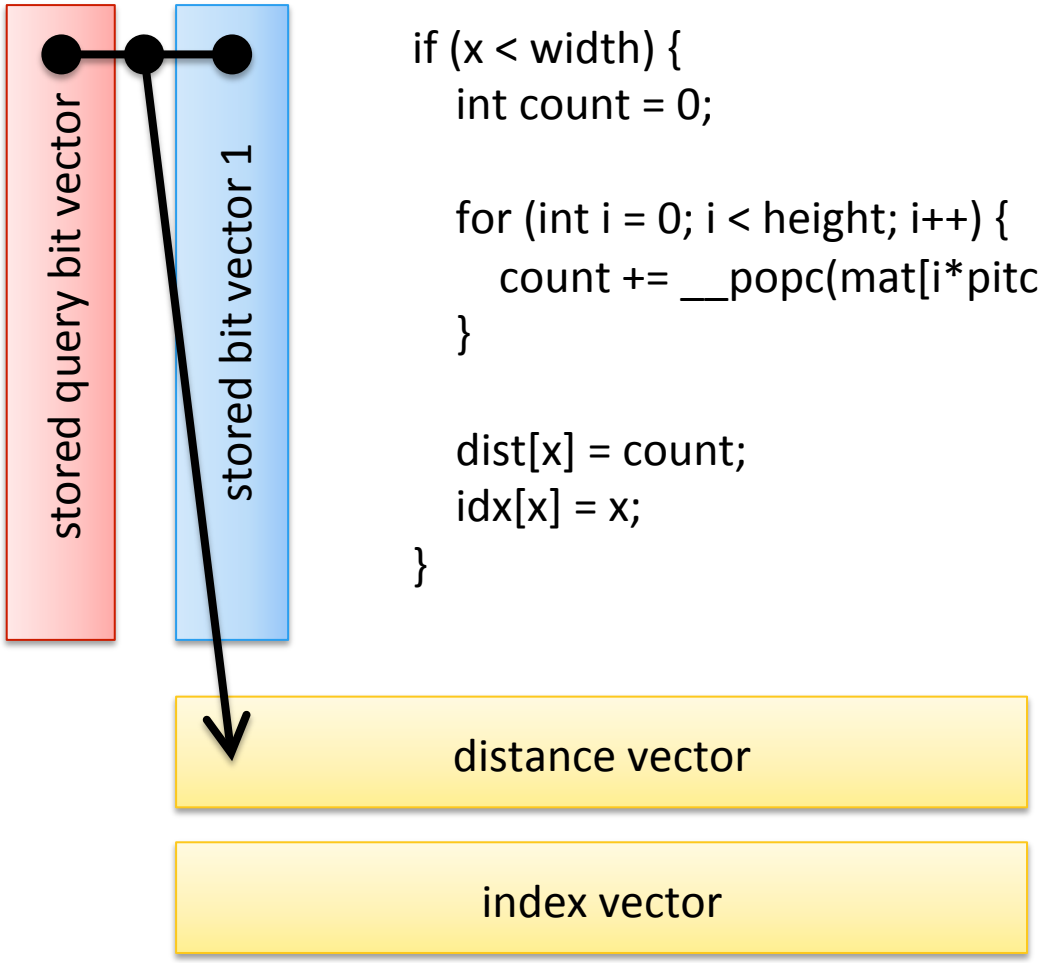
ビットベクトルの投入

- set_row時にベクトルの向きの変換が必要
  - column-major orderとpitched memory layoutへの対応が必要
  - cudaMemcpy2Dで投入
- query vectorも同様



距離の算出

- ハミング距離をxorしてpopcountで求める
  - __popcプリミティブを利用




```
if (x < width) {
    int count = 0;

    for (int i = 0; i < height; i++) {
        count += __popc(mat[i*pitch] ^ mat[i*pitch+x+1]);
    }

    dist[x] = count;
    idx[x] = x;
}
```

後処理

- 整列
  - Thrust（GPUで使えるSTLのようなもの）を利用
    - distance vectorとindex vectorへのポインタをthrust::device_ptrでラップ
      - ラップするとGPU側と認識される
    - thrust::sort_by_keyを用い、distanceをキーとして整列
      - 実装はradix sortらしい
      - 400万件からエラーを吐く
  - GPUではよくbitonic sortやradix sortが用いられる
    - 多くの実装が公開されている
- CPU側メモリへ結果をコピー
  - 呼び出し元へ返す分のdistanceとindexをCPU側へコピーする
- 距離を正規化
  - LSHのハッシュ数で割る

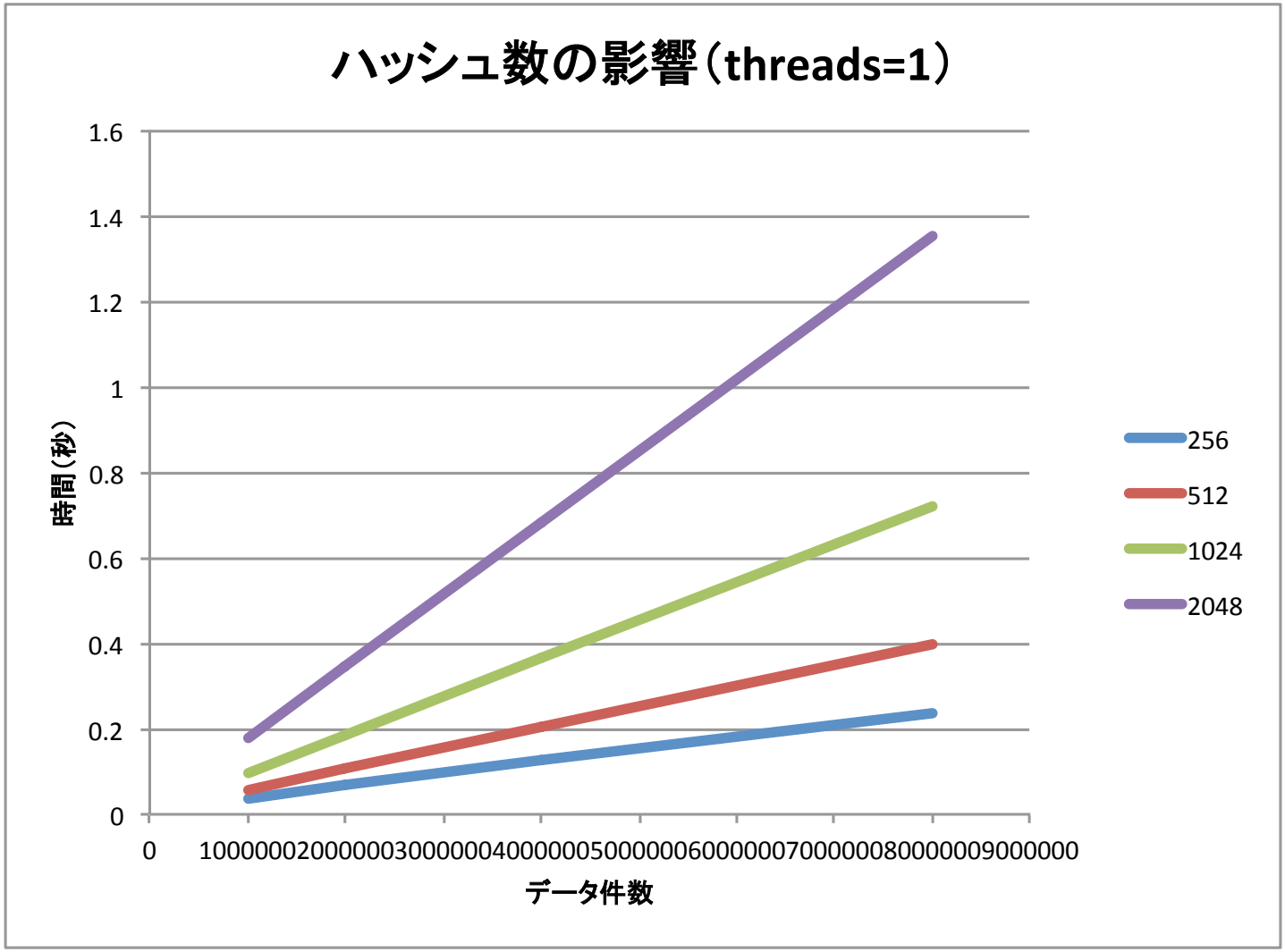
性能計測環境

- AWSで性能計測
  - インスタンス: g2.2xlarge
  - CPU: Intel Xeon E5-2670 2.6 GHz x 8（Sandy Bridge世代）
  - GPU: nVidia GRID K520（Kepler世代、GeForce GTX 680に近い）
  - GPUのメモリ: 4 GB
  - 料金: 時間あたり$0.898
- ソフトウェア
  - OS: Ubuntu 14.04
  - CUDA: 7.5.18
  - nVidiaディスプレイドライバ: 361.45.11
  - Jubatus: 0.9.1
    - 通常版はパッケージで導入
    - CUDA版は標準のコンパイルオプションでコンパイル

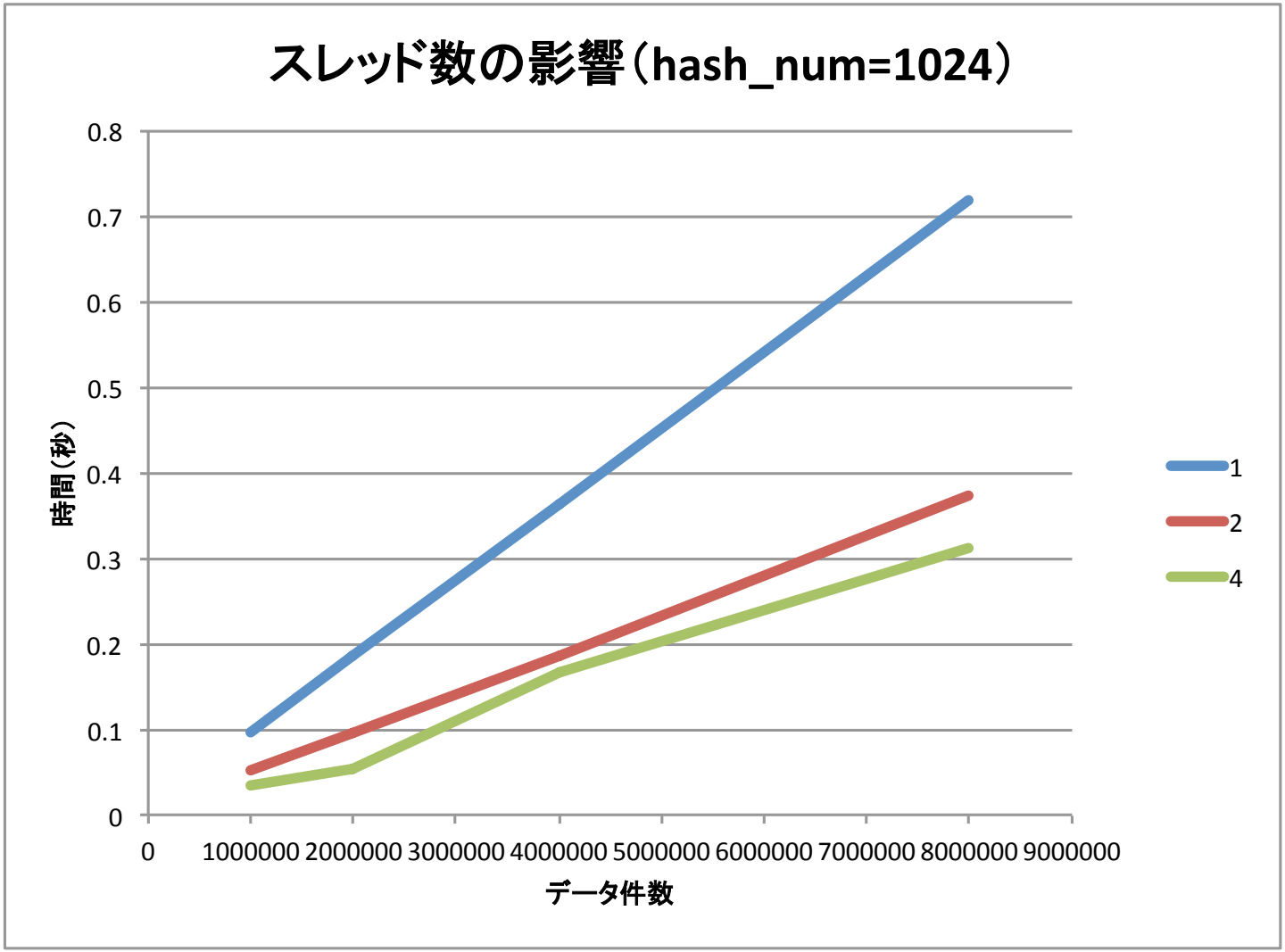
性能計測条件

- jubanearest_neighborのパラメータ
  - method: lsh
  - hash_num: 256, 512, 1024, 2048
  - threads: 1, 2, 4
    - ただし2以上は通常版のみ
      - CUDA版はマルチスレッドに対応していない為
- 投入データ
  - [0, 1)の一様分布から生成した100次元のベクトル
  - データ件数: 100万、200万、400万、800万
    - ただし400万以上は通常版のみ
- 近傍探索の条件
  - PythonクライアントからRPCで近傍1000件を取得
    - neighbor_row_from_id
  - ランダムなIDで100回実行し、実行時間の平均と標準偏差を求めた

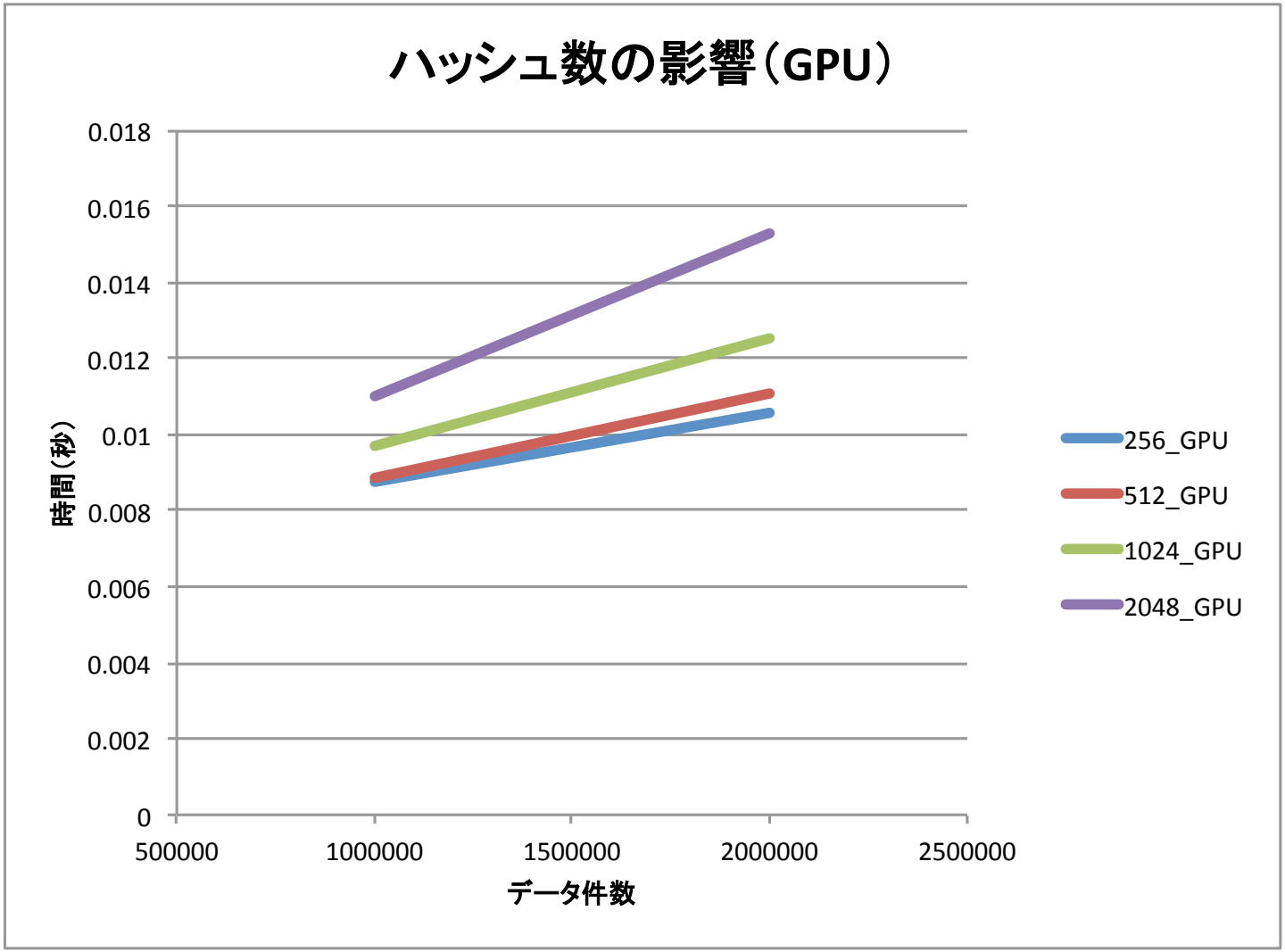
性能計測結果（通常版・ハッシュ数可変）



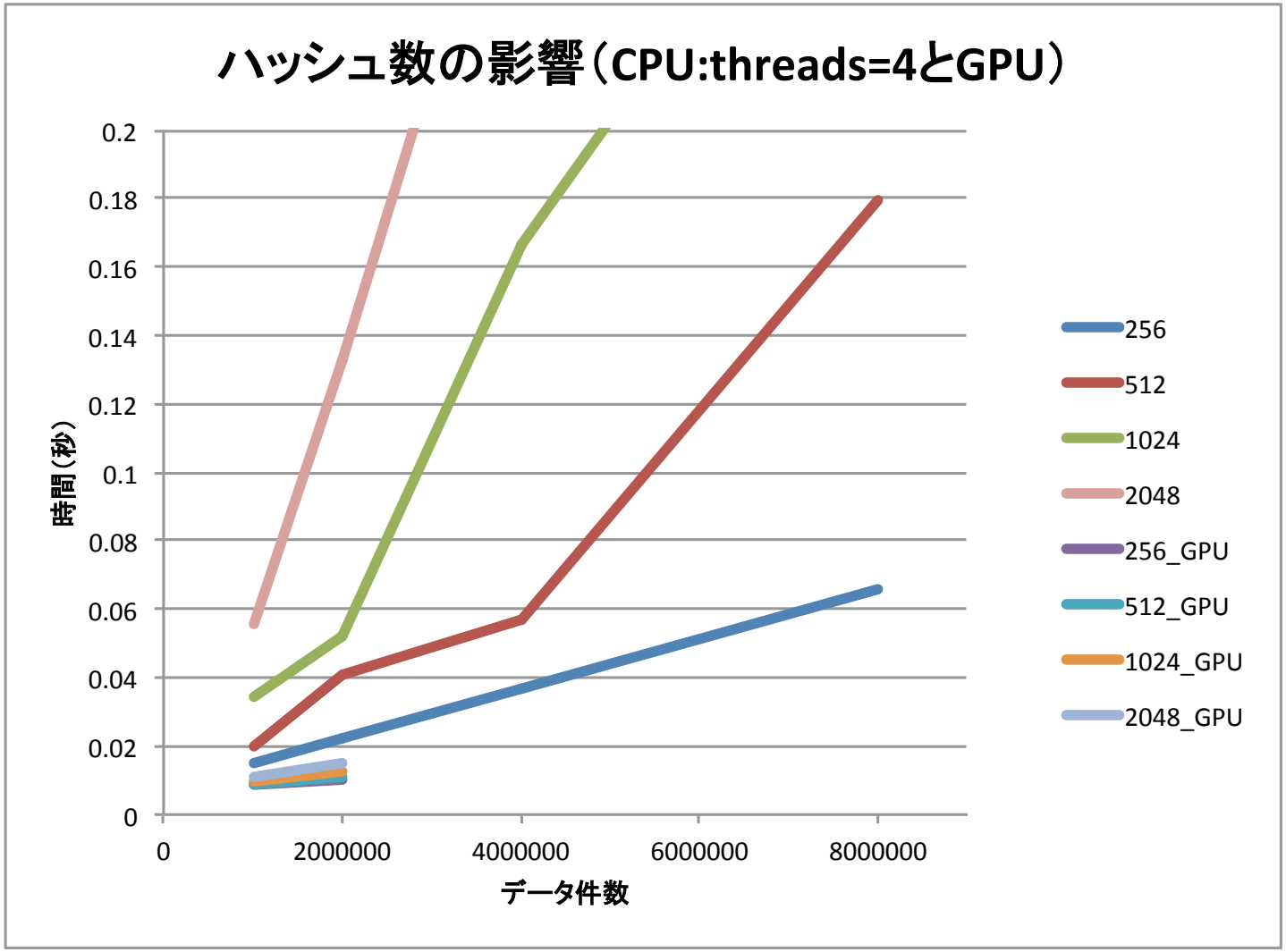
性能計測結果（通常版・スレッド数可変）



性能計測結果（CUDA版）



通常版とCUDA版の性能比較



まとめ

- Jubatusの近傍探索をGPUで実行するよう改修
  - CUDAで実装
  - jubanearest_neighborのLSHのみ対応
- 性能計測を実施
  - データ件数やLSHのハッシュ数を変えて計測
    - データ件数はほぼ線形な影響
    - ハッシュ数の影響も線形に近いように見える
    - なお、どの場合も計測値の標準偏差はかなり小さい
  - 0.9.1で高速化されていることは確認
    - マルチスレッドは有効
- GPUにより大幅な高速化が達成できることを確認
  - 並列度を活かせる処理はGPUは速い
  - 目標（2000万件を100 ms以内）は達成の見込み
    - ハッシュ数の影響が相対的に小さいので精度も改善できる