FUJITSU THE POSSIBILITIES ARE INFINITE

2008年度理研HPCシンポジウム
第3世代PCクラスタ

GPGPUクラスタの性能評価

2009年3月12日

富士通研究所
成瀬 彰

発表の概要

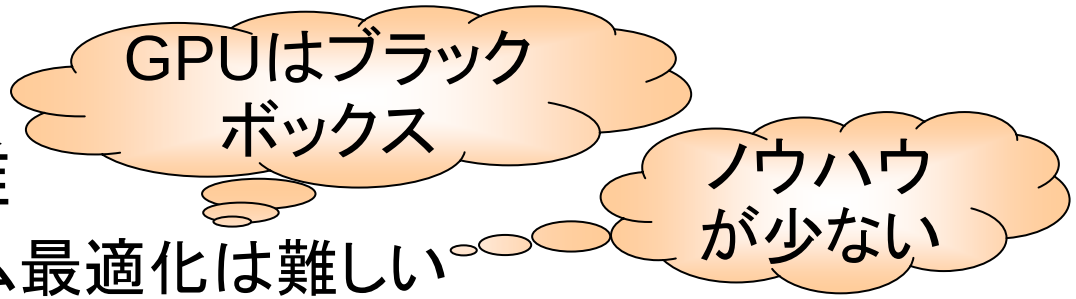
FUJITSU

- 背景
- GPGPUによる高速化
  - CUDAの概要
  - GPUのメモリアクセス特性調査
  - 姫野BMTの高速化
- GPGPUクラスタによる高速化
  - GPU・Host間のデータ転送
  - GPU-to-GPUの通信性能
  - GPGPUクラスタ上での姫野BMT性能
- まとめ

背景: GPGPU

FUJITSU

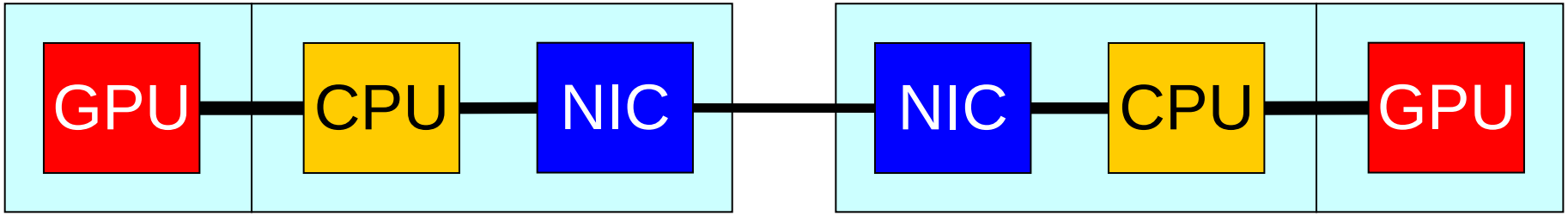
- GPUを汎用計算に
  - 高速化･汎用化が進展
    - CPUと比べて桁違いの演算性能･メモリ転送性能
  - プログラム開発環境の整備
    - CUDA … nVidiaの統合開発環境
    - GPU上のプログラム開発の簡易化
  - GPGPU対応の進展
    - 行列演算、N体問題、FFT、CFD、…
- 課題
  - チューニングが困難
    - GPU向けプログラム最適化は難しい



背景: GPGPUクラスタ

FUJITSU

- PCクラスタをGPUで加速
  - GPU搭載マシンを高速ネットワークで接続
  - 計算はGPU、通信は従来通り
    - CUDA + MPI

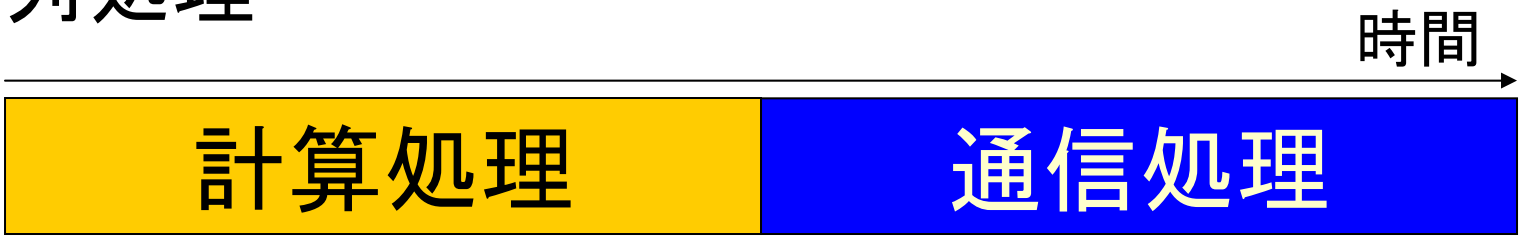


- 課題
  - 通信はCPUを経由
    - GPUで計算は速くなるが、通信は速くならない
    - GPU-to-GPUで十分な通信性能は出るのか

背景: GPGPUクラスタ

FUJITSU

- 並列処理



- 計算処理
  - GPUで加速
  - どれぐらい速くできるか?
- 通信処理
  - GPUで加速しない、むしろ遅くなる
  - どれぐらい遅くなるか?

- 姫野BMTの高速化を題材に

姫野BMT

FUJITSU

- 流体アプリのカーネルルーチン
  - Poisson方程式解法時の性能を測定

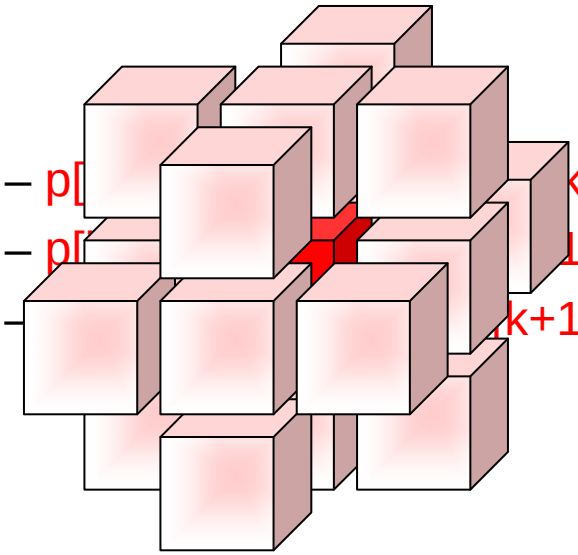
- メモリアクセス特徴
  - 14個の3D配列
    - 再利用性が低い (1配列を除く)
    - キャッシュは効かない

    →メモリバンド幅ネック
  - 14ストリームで高メモリバンド幅

姫野BMTのコア部分 (jacobi)

FUJITSU

```
for (i=1; i<imax-1; i++)
    for (j=1; j<jmax-1; j++)
        for (k=1; k<kmax-1; k++) {
            s0 = a0[i][j][k] * p[i+1][j][k]
                + a1[i][j][k] * p[i][j+1][k]
                + a2[i][j][k] * p[i][j][k+1]
                + b0[i][j][k] * (p[i+1][j+1][k] – p[                k] + p[i-1][j-1][k])
                + b1[i][j][k] * (p[i][j+1][k+1] – p[                1] + p[i][j-1][k-1])
                + b2[i][j][k] * (p[i+1][j][k+1] –                   k+1] + p[i-1][j][k-1])
                + c0[i][j][k] * p[i-1][j][k]
                + c1[i][j][k] * p[i][j-1][k]
                + c2[i][j][k] * p[i][j][k-1]
                + wrk1[i][j][k];
            ss = (s0 * a3[i][j][k] – p[i][j][k]) * bnd[i][j][k];
            wrk2[i][j][k] = p[i][j][k] + omega * ss;
        }
```



- 配列p: ステンシルアクセス
  - 再利用性有り

- 他13配列: 点アクセス
  - 再利用性無し

発表の概要

FUJITSU

- 背景
- GPGPUによる高速化
  - CUDAの概要
  - GPUのメモリアクセス特性調査
  - 姫野BMTの高速化
- GPGPUクラスタによる高速化
  - GPU・Host間のデータ転送
  - GPU-to-GPUの通信性能
  - GPGPUクラスタ上での姫野BMT性能
- まとめ

CUDA概要: ハードウェア構成

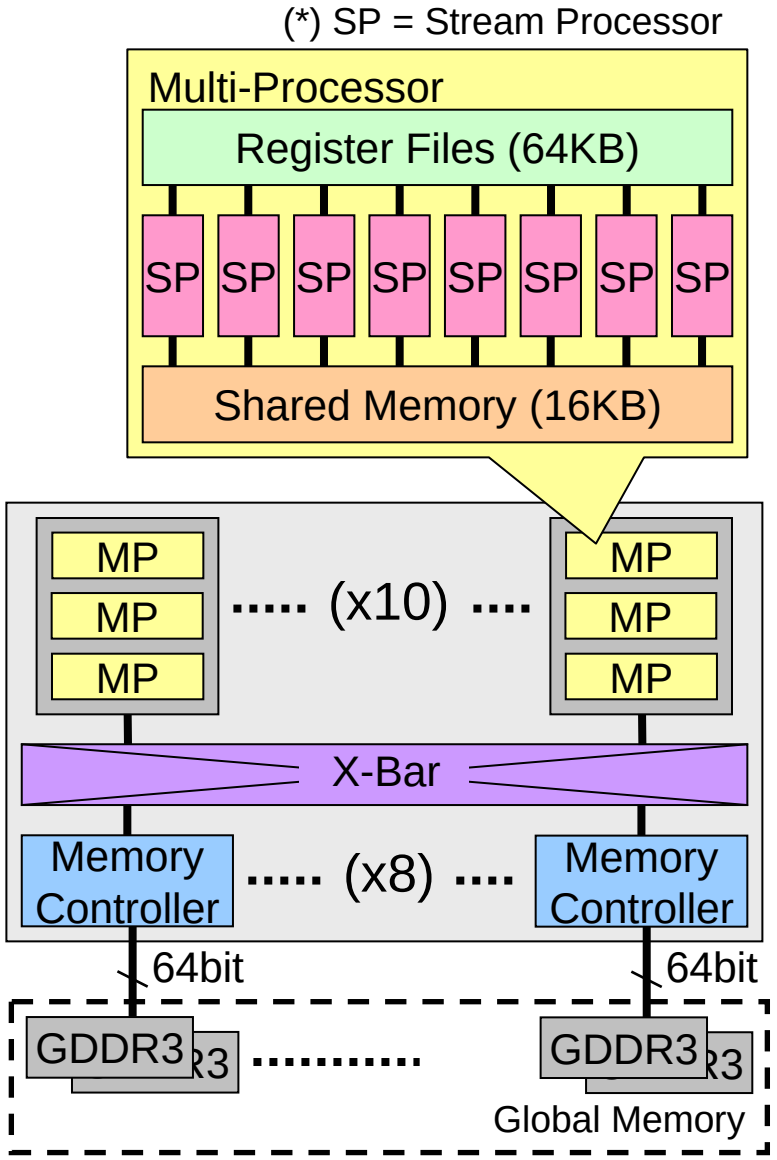
FUJITSU

- GeForce GTX280
  - 最新世代CUDA GPU
  - 理論ピークメモリバンド幅:

    141.7GB/s

    (= 64bit * 8 * 2.214GHz)

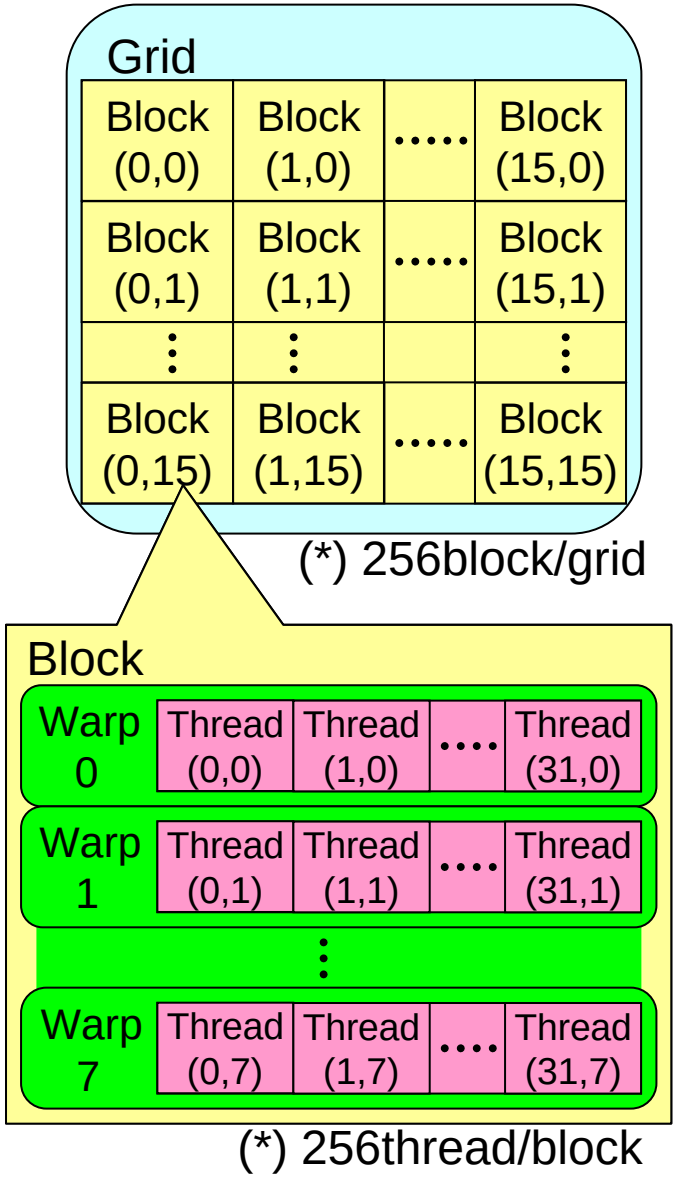
  - MP数: 30
  - MPの内部構成
    - SP数: 8 (全体で240)
    - 共有メモリ: 16KB
    - レジスタ数: 16K本 (64KB)



CUDA概要: プログラミング

FUJITSU

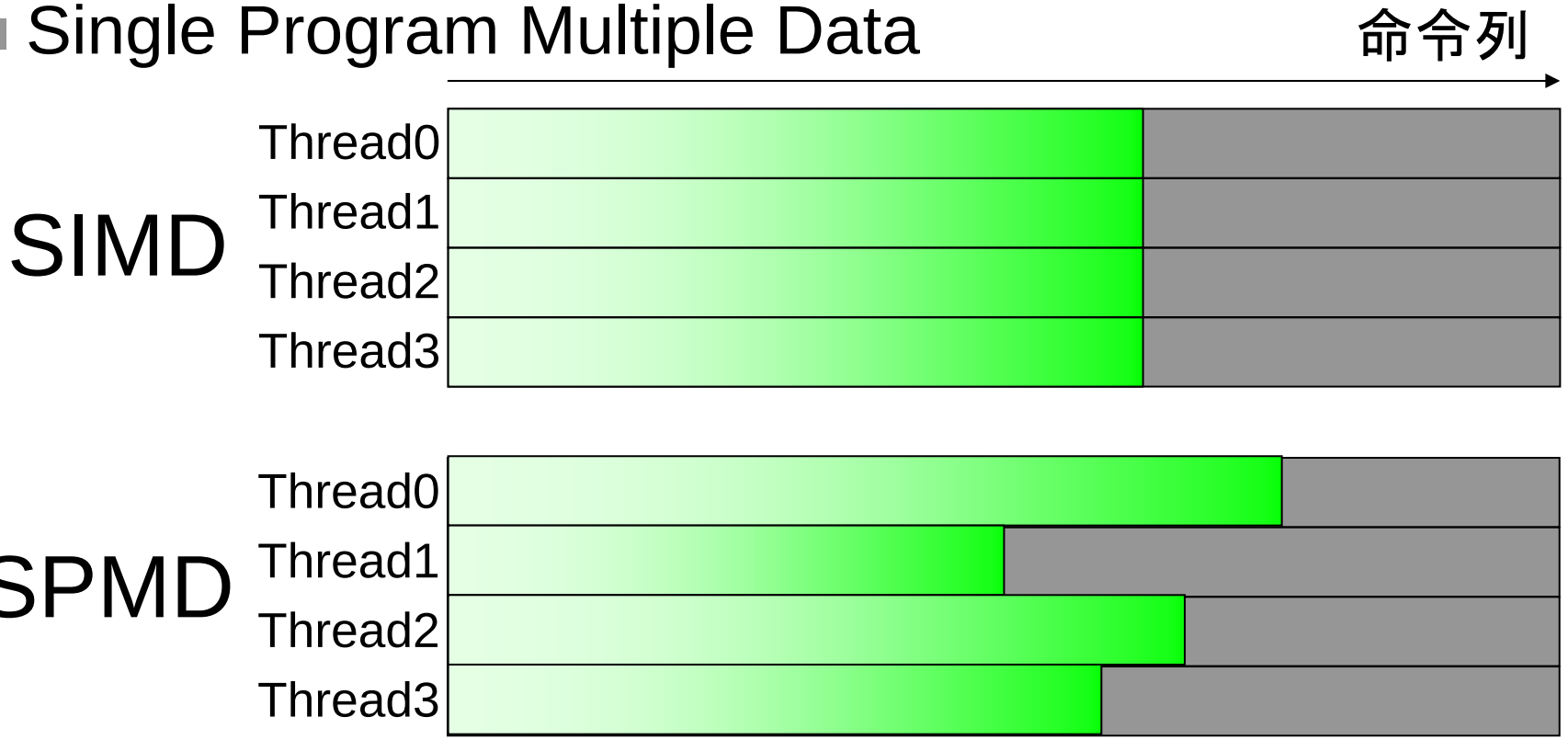
- 2段階のデータ並列
  - グリッド
    - 複数のブロックで構成
  - ブロック
    - 複数のスレッドで構成
  - スレッド
    - SP上で実行される
- MP内の処理
  - 各ブロックは、1つのMPに割当
  - MPが実行可能スレッドを選択
    - 選択単位はワープ(32スレッド)



CUDA概要: 実行モデル

FUJITSU

- CUDAの実行モデル: SPMD
  - Single Program Multiple Data



- 基本的に、スレッド間は非同期
  - 同じワープ内のスレッドだけ同期

メモリアクセス特性の調査

FUJITSU

- 高速化の対象：姫野BMT
  - 姫野BMTはメモリバンド幅ネック
  - 姫野BMTの高速化 ≒ 高メモリバンド幅の実現
- GPUの実効メモリバンド幅
  - 理論ピークの8割超も可能
  - いつでも高バンド幅を実現できる → NO
  - 高バンド幅実現の条件は?
- GPUのメモリアクセス特性を調査
  - バンド幅、アクセス遅延

メモリバンド幅の調査

FUJITSU

- メモリコピー時のメモリバンド幅を実測
  - READ:WRITE比率 = 1:1
- 以下の条件を変え、測定を実施
  - コピー量 (=転送量)
  - 同時コピー数 (=ストリーム数)

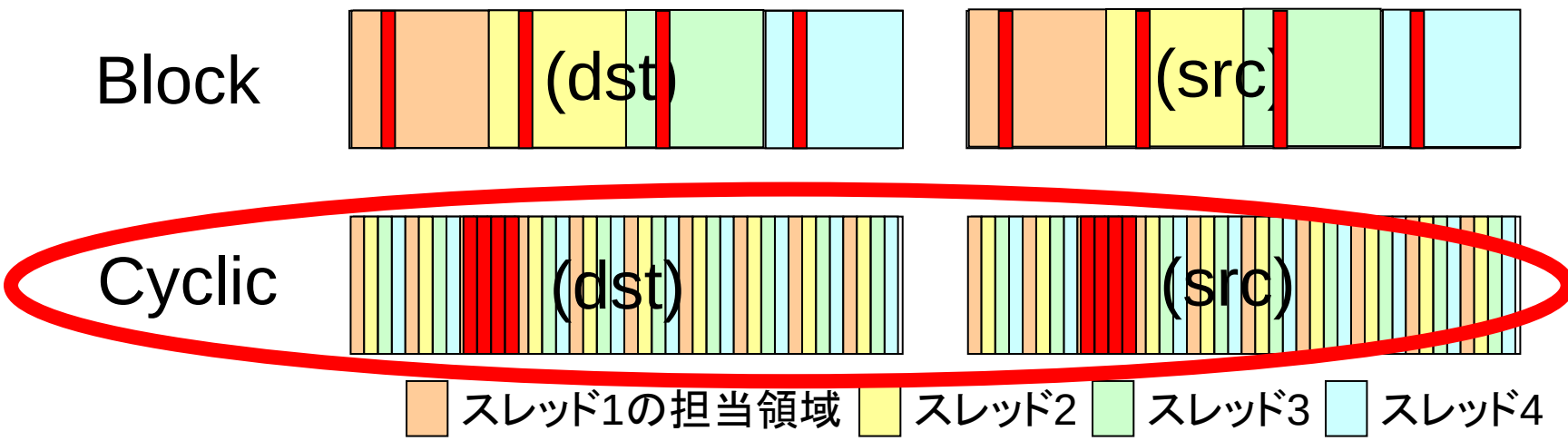
メモリコピー (基本)

FUJITSU

- 普通のメモリコピー

```
for ( i = 0 ; i < num ; i ++ ) {
        dst[ i ] = src[ i ];
}
```

- GPU：データ並列でメモリコピー
  - 各スレッドへのデータ割当（4スレッド）：



メモリコピー (基本)

FUJITSU

```
__global__ void mcopy( float *dst, float *src, int size )
{
        int  id = (各スレッド固有の番号);
        int  step = (総スレッド数);
        int  n_total = (総コピー回数);
        for ( int  i = id ; i < n_total ; i += step ) {
                dst[ i ] = src[ i ];
        }
}
```

- 配列に対するアクセスパターン
  - スレッド単体で考えるとストライドアクセス
  - スレッド全体で考えると逐次アクセス
  - READ/WRITE、各1ストリーム (計2ストリーム)

メモリコピー (同時に複数コピー)

FUJITSU

- 同時に複数のメモリコピー

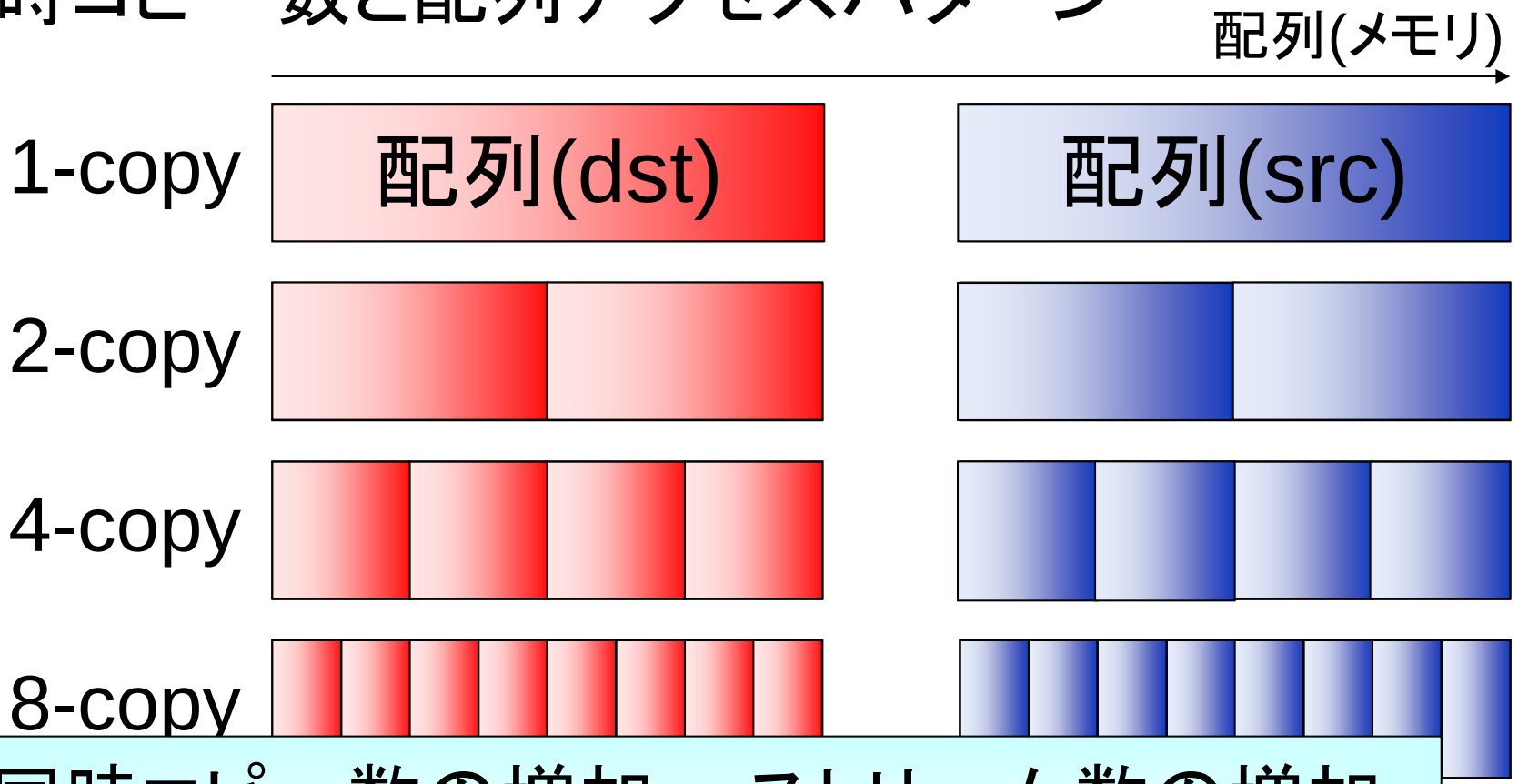
8-Copy

```
for ( i = 0 ; i < num / 8 ; i ++ ) {
        dst0[ i ] = src0[ i ];
        dst1[ i ] = src1[ i ];
        dst2[ i ] = src2[ i ];
        dst3[ i ] = src3[ i ];
        dst4[ i ] = src4[ i ];
        dst5[ i ] = src5[ i ];
        dst6[ i ] = src6[ i ];
        dst7[ i ] = src7[ i ];
}
```

メモリコピー (同時に複数コピー)

FUJITSU

- 同時コピー数と配列アクセスパターン



同時コピー数の増加 = ストリーム数の増加

ストリーム数とメモリバンド幅の関係

メモリコピー (同時に複数コピー)

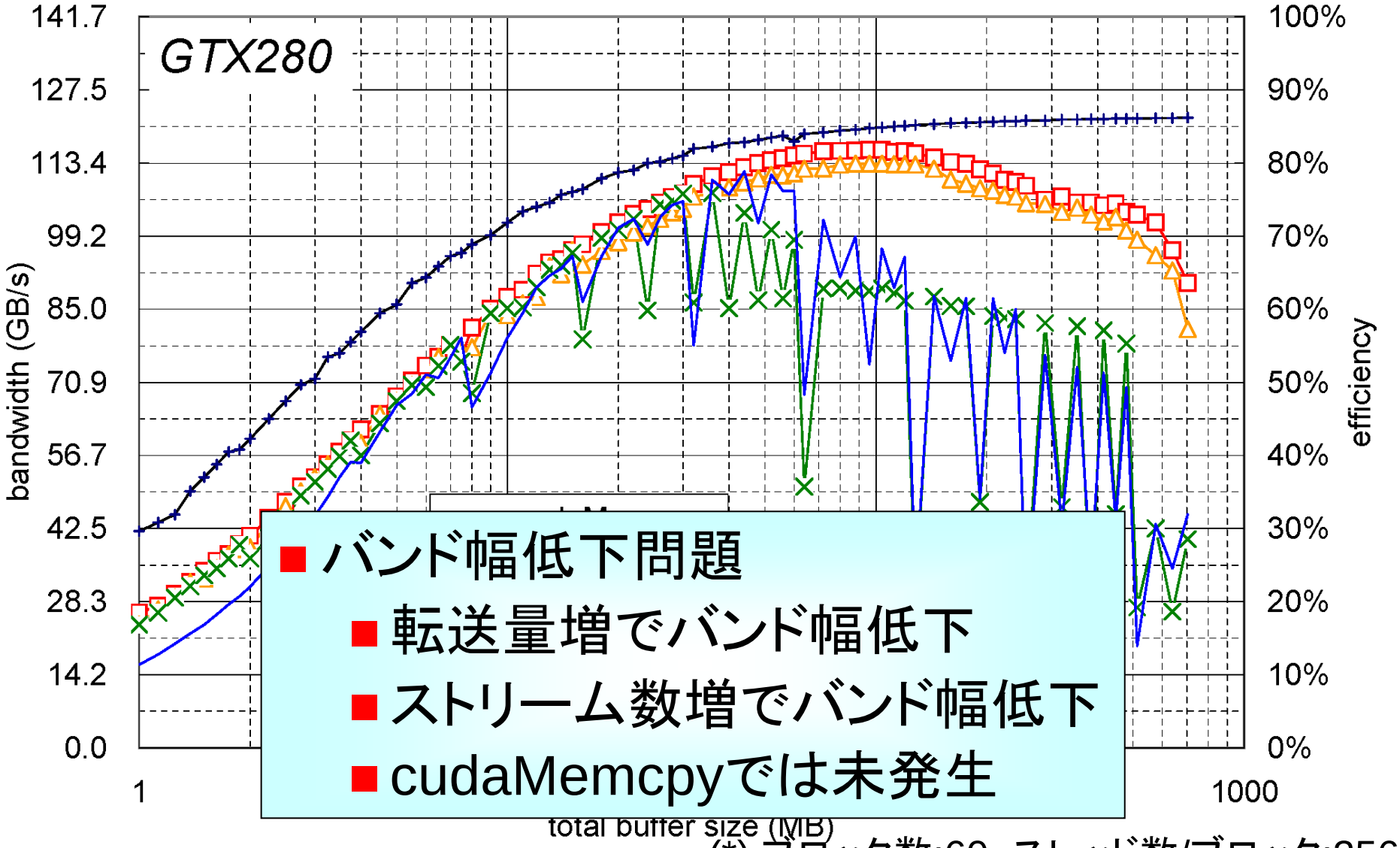
FUJITSU

```
__global__ void mcopy( float *dst, float *src, int size, int n_copy )
{
        int  id = (各スレッド固有の番号);
        int  step = (総スレッド数);
        int  n_total = (総コピー回数);
        int  n_each = n_total / n_copy;
        for ( int i = id ; i < n_each ; i += step ) {
                for ( int j = i ; j < n_total ; j += n_each ) {
                        dst[ j ] = src[ j ];
                }
        }
}
```

- 複数のメモリコピーが同時進行
  - 配列をN個に分離
  - ストリーム数： 2*N

メモリバンド幅測定結果

FUJITSU



- バンド幅低下問題
  - 転送量増でバンド幅低下
  - ストリーム数増でバンド幅低下
  - cudaMemcpyでは未発生

(*) ブロック数:60, スレッド数/ブロック:256

メモリアクセス遅延の調査

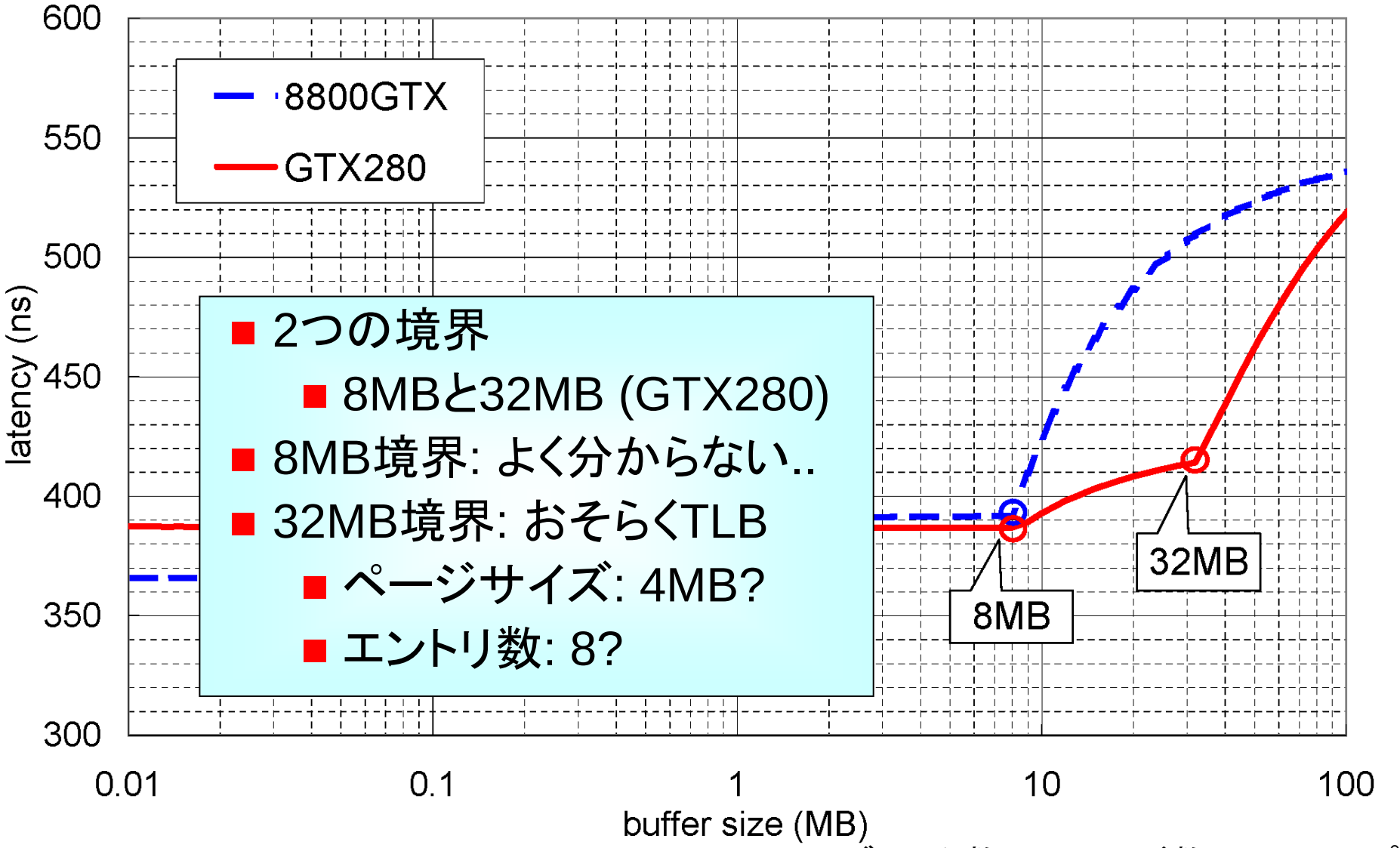
FUJITSU

- 遅延は短い方が扱いやすい
  - GPUは遅延が長いと言われている
  - 具体的に、どれぐらい長いのか
- ランダムアクセス時の遅延を測定

```
int  index = (各スレッド固有の番号);
int  num = (アクセス回数);
while ( num > 0 ) {
        index = buf[ index ];
        num--;
}
```

メモリアクセス遅延測定結果

FUJITSU



(*) ブロック数:1、スレッド数:32 (1ワープ)

調査結果の考察

FUJITSU

- バンド幅測定: バンド幅低下問題
  - 転送量増でバンド幅低下
  - ストリーム数増でバンド幅低下
  - cudaMemcpy性能に届かない
- 遅延測定: TLBの存在
  - TLBミスで～200nsの遅延増
- バンド幅低下問題の原因はTLBスラッシング?

バンド幅低下のシナリオ

FUJITSU

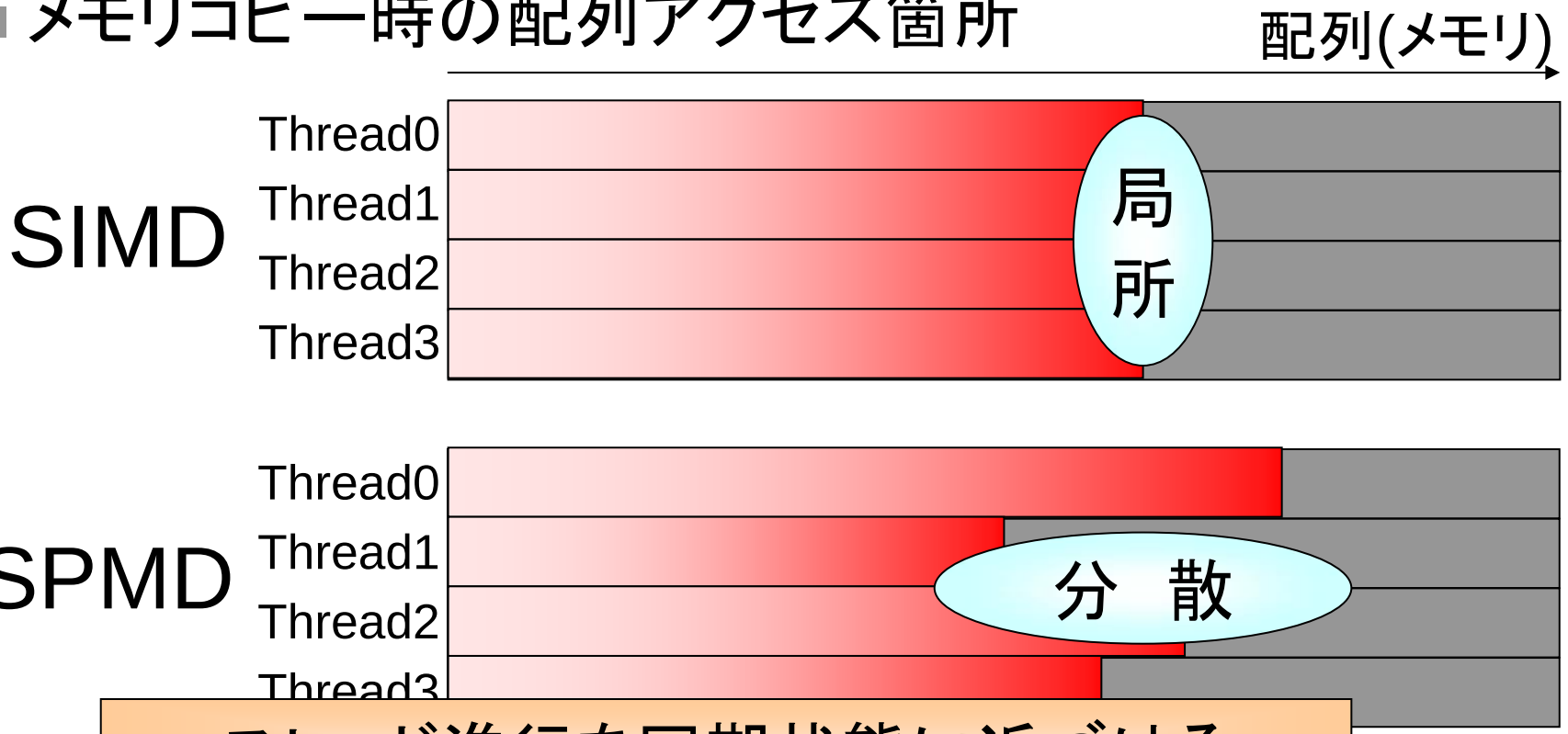
- CUDAの実行モデルはSPMD
- 進行の速いスレッド･遅いスレッドが混在
- 時間が経過、スレッド間の進行差が拡大
- メモリアクセス箇所が分散
- 単位時間あたりアクセスページ数が増加
- TLBミス発生頻度が増加 (TLBスラッシング)

→メモリバンド幅低下

バンド幅低下のシナリオ

FUJITSU

- CUDAの実行モデルはSPMD
  - メモリコピー時の配列アクセス箇所



スレッド進行を同期状態に近づける
→ バンド幅低下を回避できる?

スレッド進行の同期化

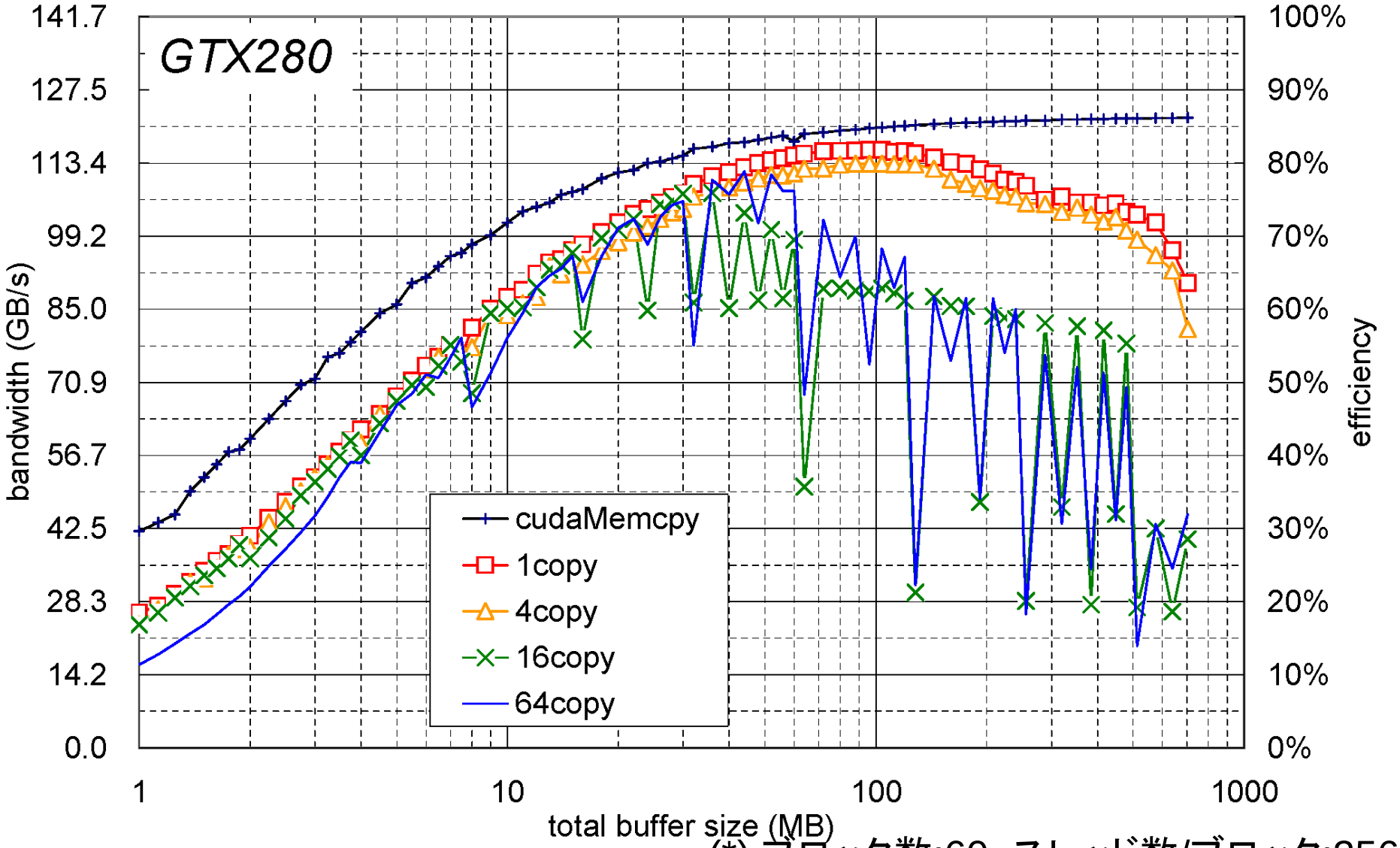
FUJITSU

- 全スレッドの同期 → CUDAでは出来ない
- 同じブロック内のスレッド、同期可能

```
__global__ void mcopy( float *dst, float *src, int size, int n_copy )
{
        int  id = (各スレッド固有の番号);
        int  step = (総スレッド数);
        int  n_total = (総コピー回数);
        int  n_each = n_total / n_copy;
        for ( int i = id ; i < n_each ; i += step ) {
                for ( int j = i ; j < n_total ; j += n_each ) {
                        __syncthreads()
                        dst[ j ] = src[ j ];
                }
        }
}
```

メモリバンド幅測定結果

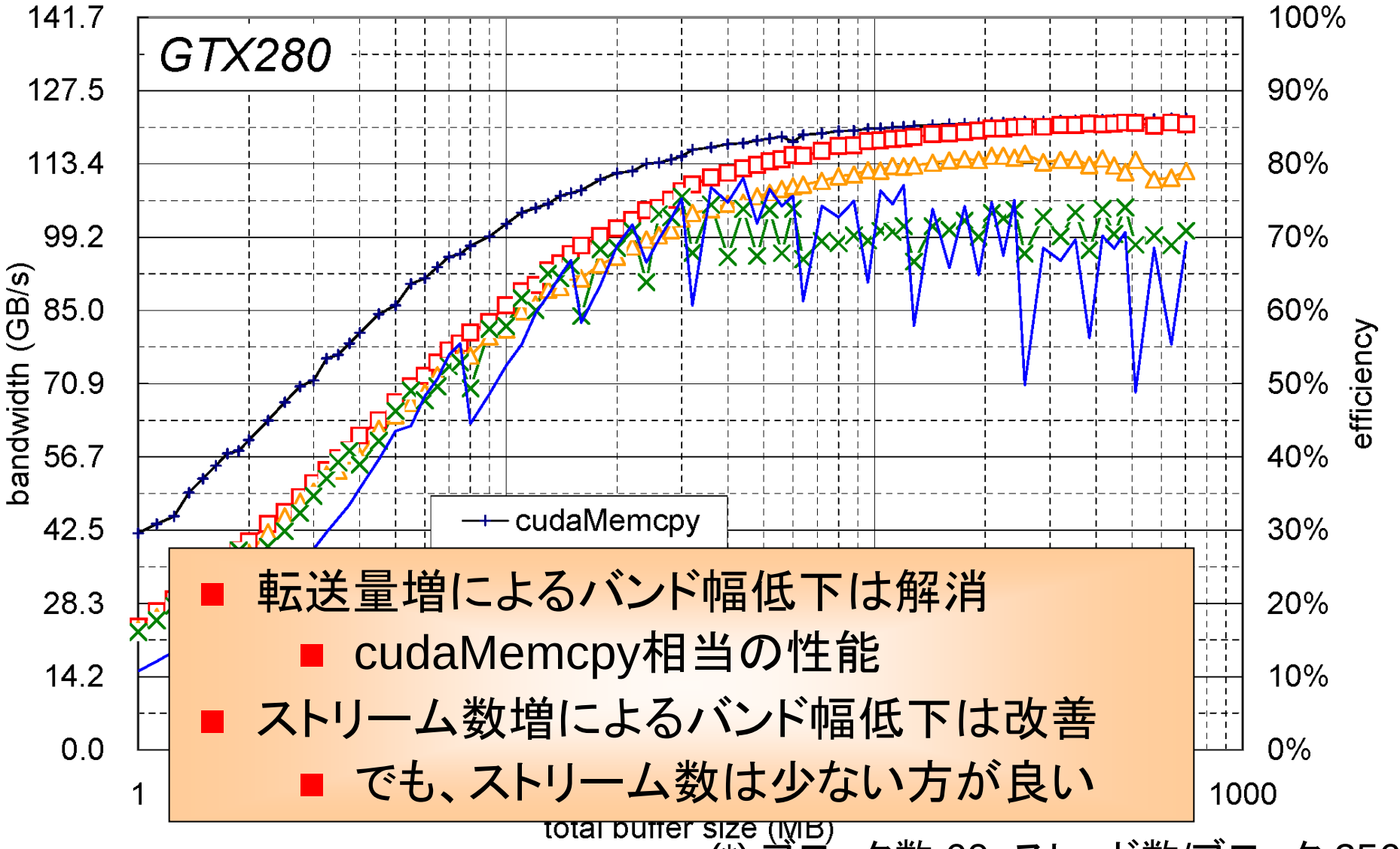
FUJITSU



(*) ブロック数:60, スレッド数/ブロック:256

メモリバンド幅測定結果 (syncthreads)

FUJITSU



- 転送量増によるバンド幅低下は解消
  - cudaMemcpy相当の性能
- ストリーム数増によるバンド幅低下は改善
  - でも、ストリーム数は少ない方が良い

(*) ブロック数:60, スレッド数/ブロック:256

高メモリバンド幅を実現する方法

FUJITSU

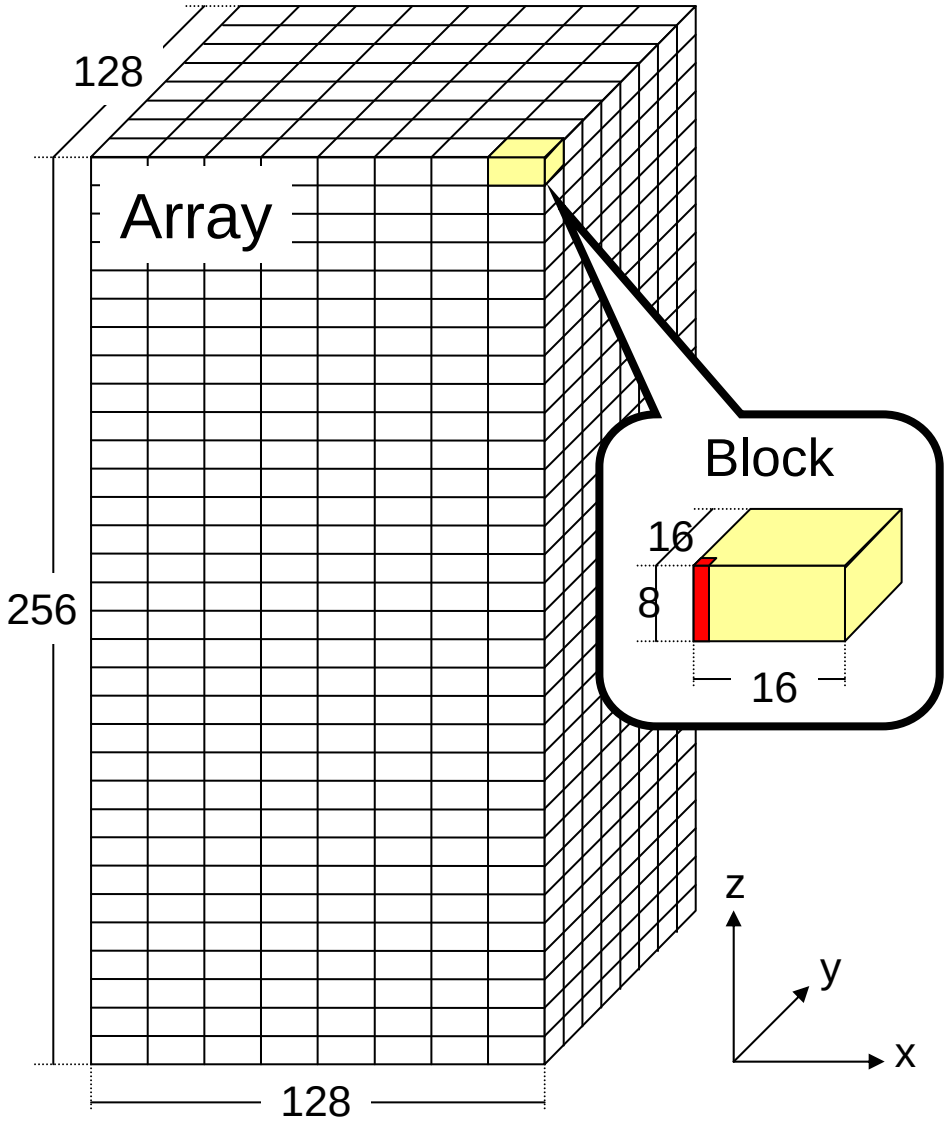
- スレッド進行の同期化
  - __syncthreads()でブロック内スレッドを同期
  - 同期ペナルティ ＜ 同期メリット
- アクセスパターンの局所化
  - アルゴリズム･データ構造を見直し、ストリーム数減
  - 単位時間あたりアクセスページ数を削減
- スレッド数の最適化
  - レジスタ･共有メモリ使用量を減らし、同時実行可能スレッド数を増加
  - 適切な総スレッド数の選択

姫野BMT on GPU

FUJITSU

- 従来実装
  - 東工大)青木教授の実装
    - 2007年度理研ベンチマークコンテスト優勝
    - HPC研究会で発表 (2008-HPC-115)
  - 姫野BMT(Mサイズ)の実行ファイルが公開

従来実装



- ブロック形状: (16,16,8)
  - ブロック数: 2,048
- 各ブロック
  - スレッド数: 256
    - 8格子点計算／スレッド
  - 格子点計算開始前に、スレッド間で共用する配列値を全て共有メモリにロード
    - 同期回数を減らすため?
  - 共有メモリ使用量: 12.7KB

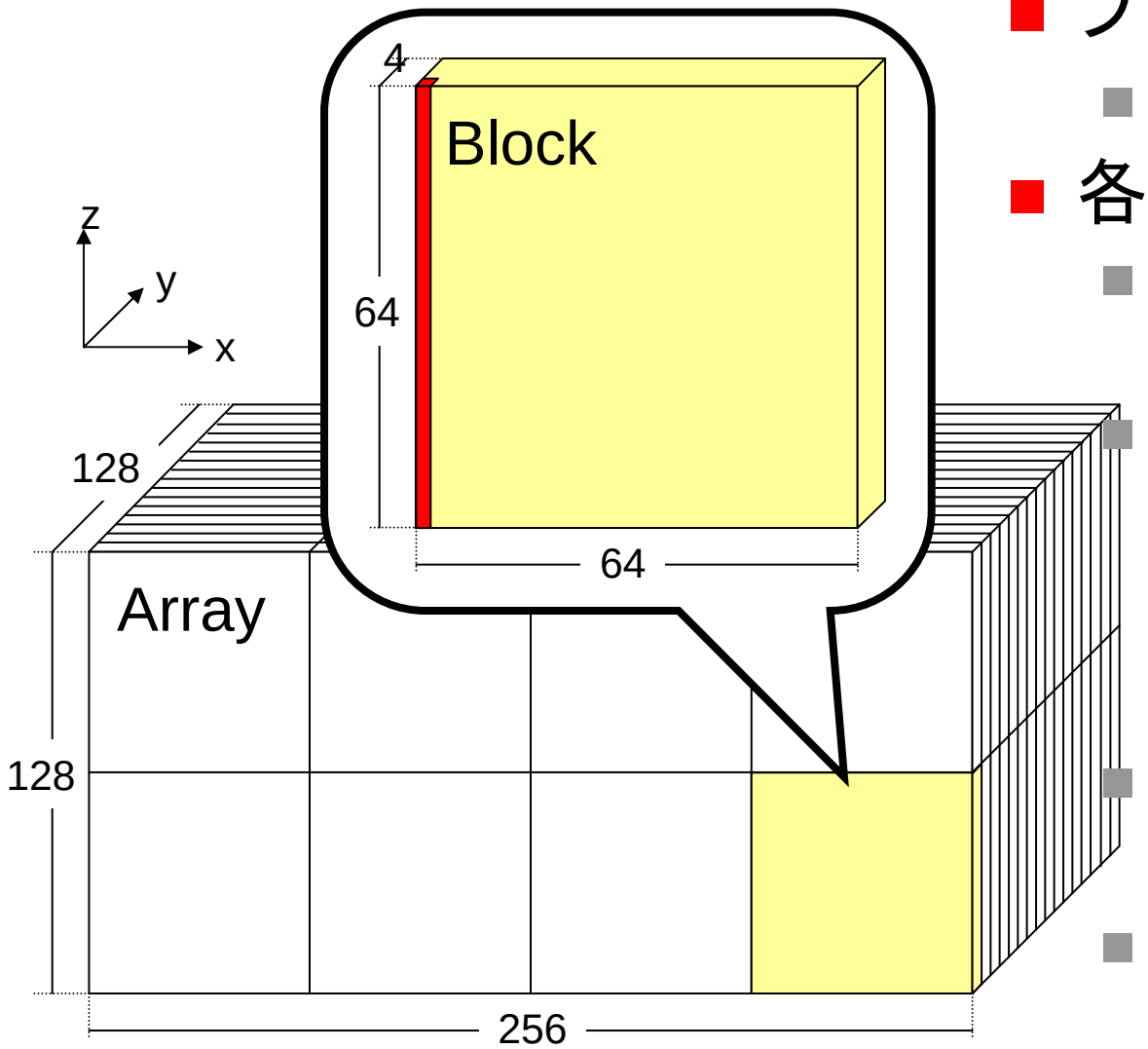
    4B*(16+2)*(16+2)*(8+2)
  - MPへの割当ブロック数: 1
- x軸とz軸の入替え
  - マルチGPU対応?

提案手法に基づく高速化

FUJITSU

- スレッド進行の同期化
  - 同期処理の多用 (__syncthreads())
- アクセスパターンの局所化
  - 配列の次元入替え
  - ブロック形状変更
- スレッド数の最適化
  - 同時実行スレッド数の増加
  - 総スレッド数調整
- その他
  - 配列間のパディング量調整

提案手法適用後

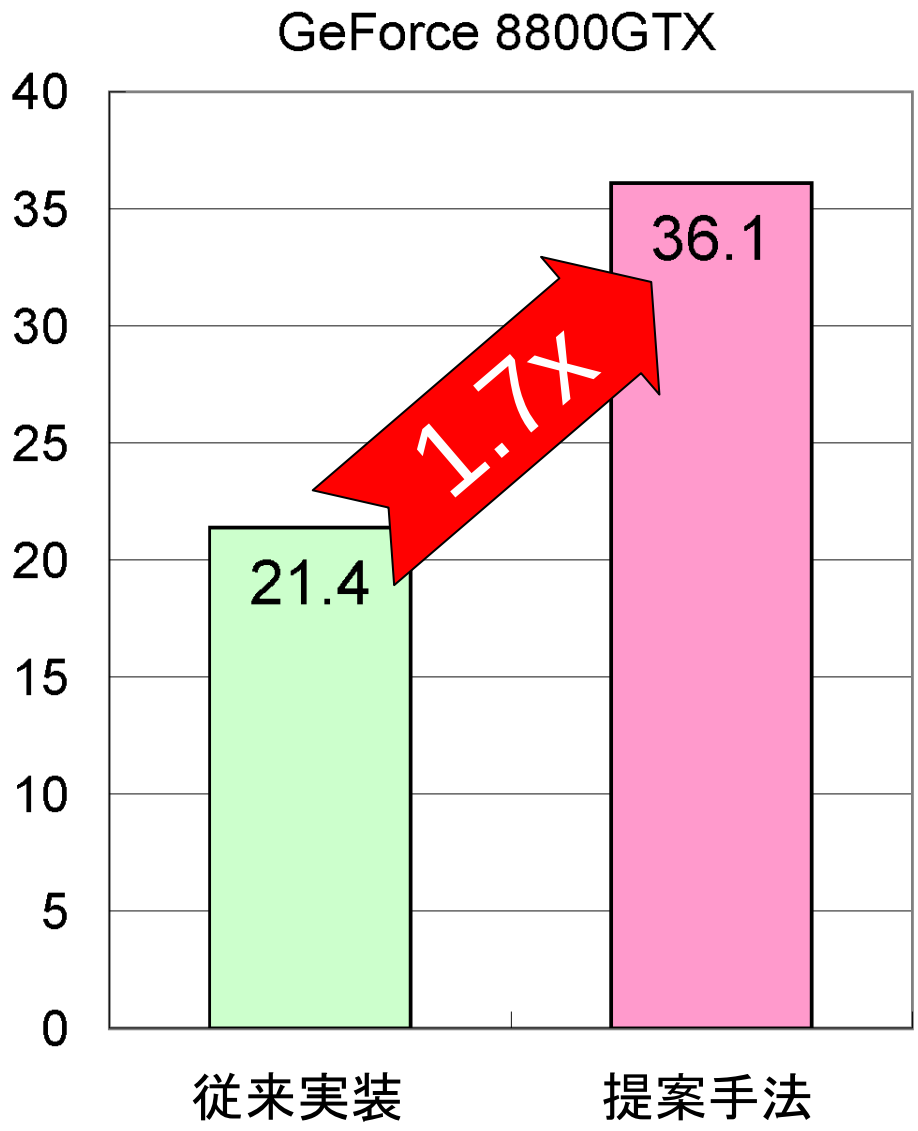


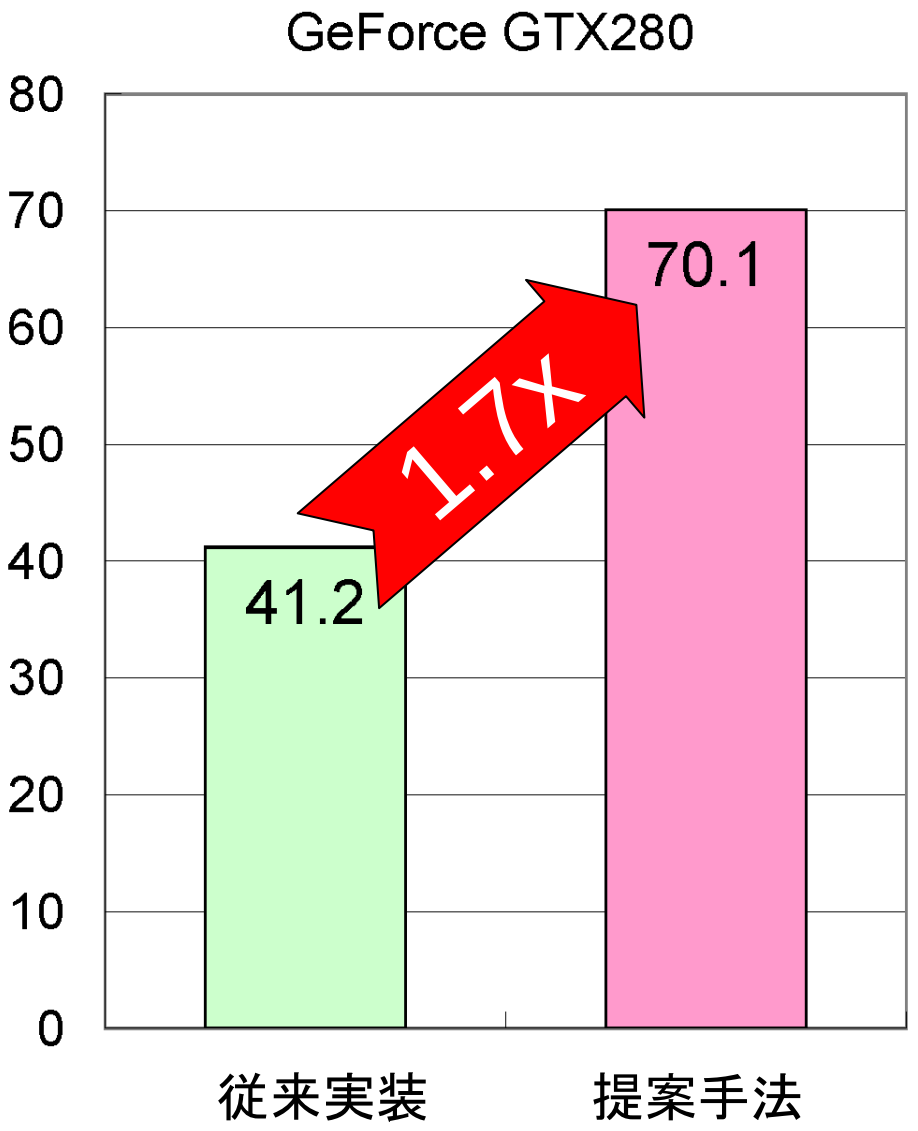
- ブロック形状: (64,4,64)
  - ブロック数: 256
- 各ブロック
  - スレッド数: 256
    - 64格子点計算／スレッド
  - スレッド間で共用する配列値、各格子点計算の開始前に、必要な分だけ共有メモリにロード
    - 同期回数増、問題無し
  - 共有メモリ使用量: 4.7KB

    4B*(64+2)*(4+2)*3
  - 1MPに3ブロック割当

姫野BMT性能(GFLOPS)

FUJITSU





GFLOPSとバンド幅の関係

FUJITSU

- 姫野BMTのメモリアクセス量

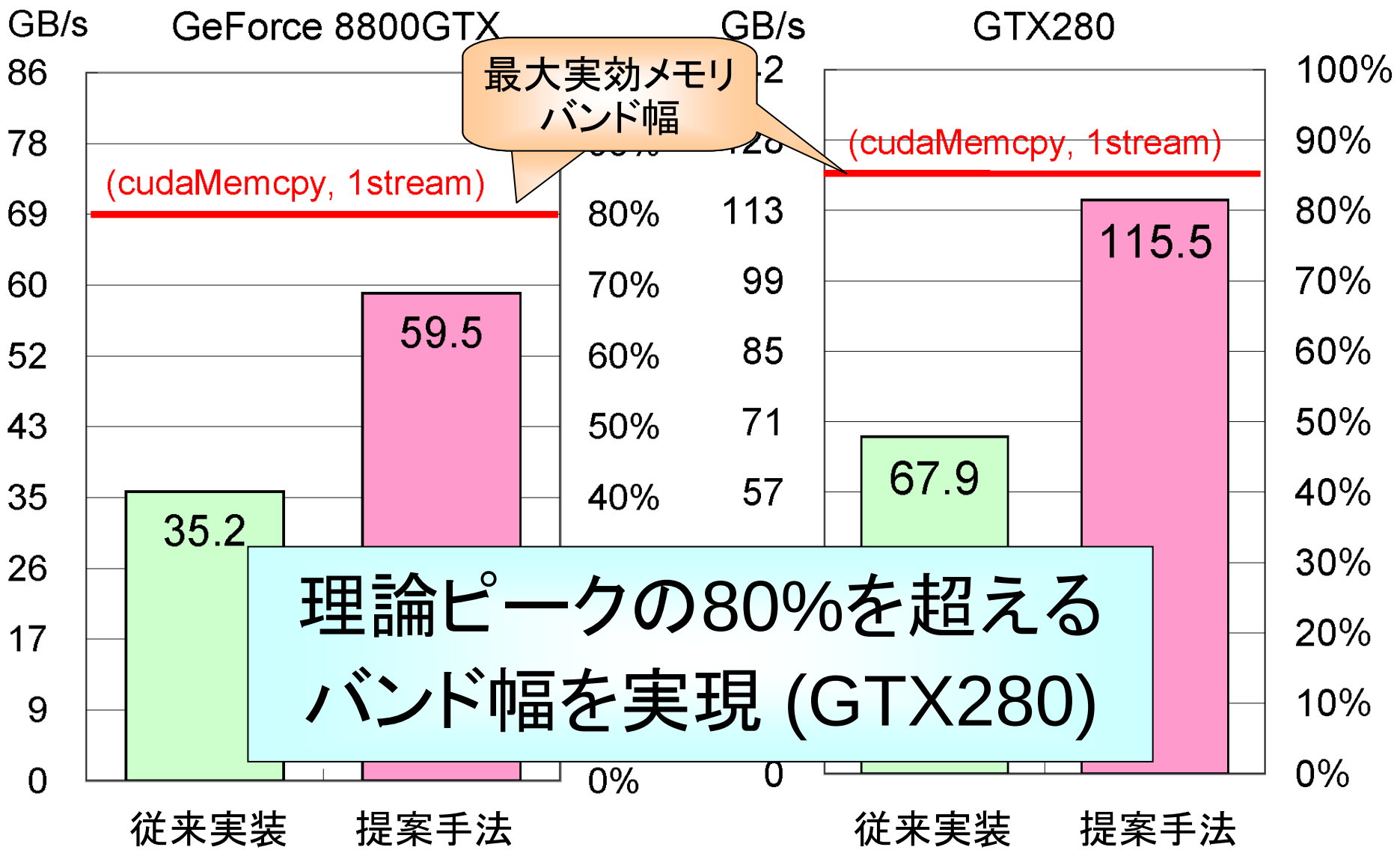
1.65 B/FLOP

(*) BF比は実装依存

  - 1格子点あたりのメモリアクセス量: 56 B
    - 1格子点あたり14変数のメモリアクセス
    - データ型はfloat (4B)
  - 1格子点あたりの演算量: 34 FLOP

姫野BMT性能(バンド幅)

FUJITSU



発表の概要

FUJITSU

- 背景
- GPGPUによる高速化
  - CUDAの概要
  - GPUのメモリアクセス特性調査
  - 姫野BMTの高速化
- GPGPUクラスタによる高速化
  - GPU・Host間のデータ転送
  - GPU-to-GPUの通信性能
  - GPGPUクラスタ上での姫野BMT性能
- まとめ

並列版の姫野BMT

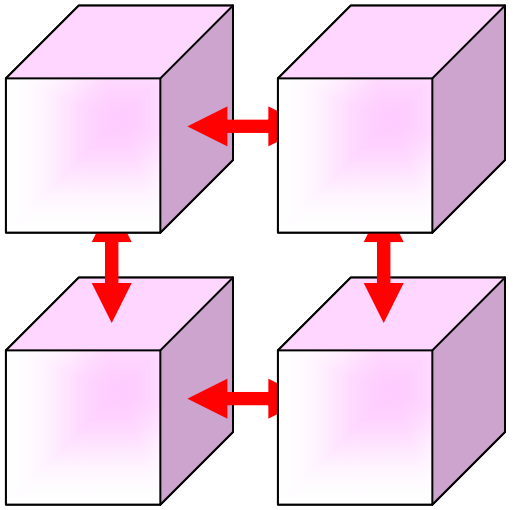
FUJITSU

- 3次元配列をプロセス数分割
- 各プロセスは分割後の配列を担当

(1) 計算処理: 各プロセスは自分の担当領域を計算

(2) 通信処理: 配列pの隣接面を隣のプロセスと送受信

(3) (1)に戻る



GPGPUクラスタ試験環境

FUJITSU

- 4台のGPU搭載マシンをInfiniBandで接続
- マシンスペック
  - GPU: nVidia GTX285 (PCIe2x16)
  - CPU: Intel Core i7 (2.66GHz)
  - NIC: Mellanox ConnectX (DDR-IB, PCIe2x8)
  - M/B: Gigabyte GA-EX58-UD5 (Intel X58)
  - Mem: DDR3-1066 2GB x 3
  - OS: RHEL 5.3 (64bit)
  - C/C++: GNU
  - CUDA: 2.1
  - MPI: OpenMPI 1.3

姫野BMT on PCクラスタ

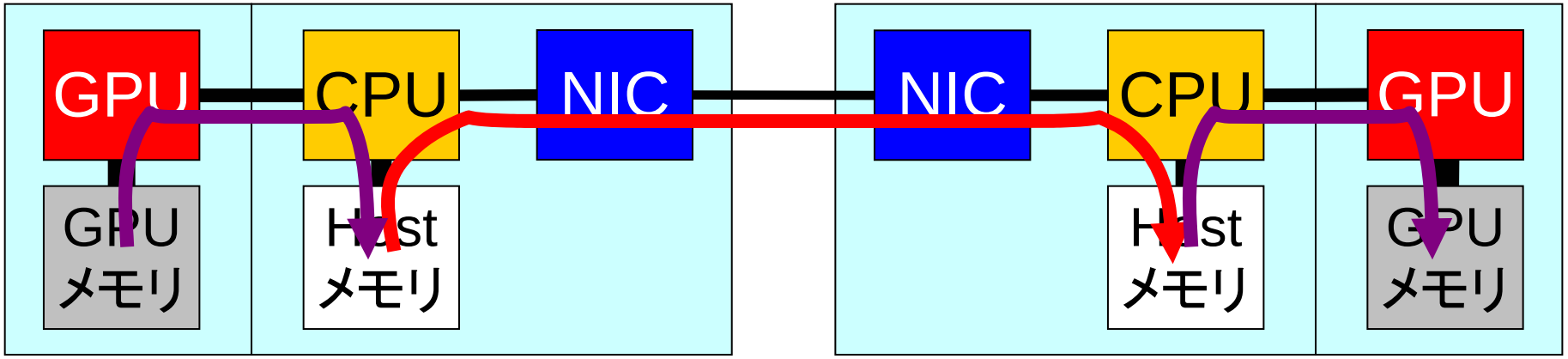
FUJITSU

- PCクラスタの姫野BMT性能（実測、Lサイズ）
  - 1ノード:　　6.5GFLOPS
  - 4ノード:　　25.5GFLOPS
  - 1→4ノードで3.9倍性能UP、スケール
- 4ノードPCクラスタの処理時間内訳

- GPGPUクラスタはスケールするか?
  - 1ノード:　　70GFLOPS程度
  - 4ノード:　　???

GPU-to-GPU通信

FUJITSU



- PCクラスタ: CPU-to-CPU通信
  - Hostメモリ → Hostメモリ (MPI)

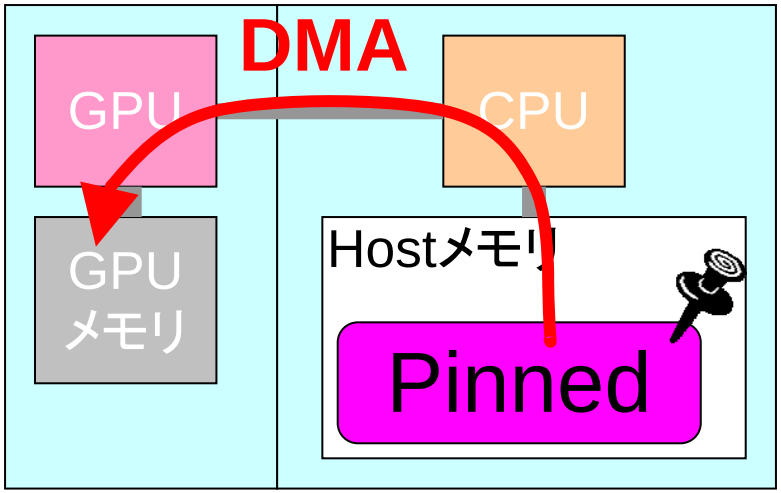
GPU・Host間のデータ転送性能が重要

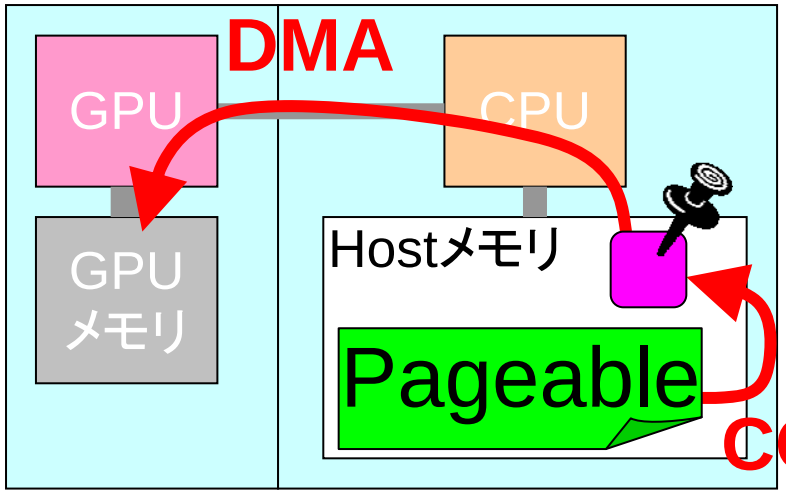
- GPGPUクラスタ: GPU-to-GPU通信
  - GPUメモリ → Hostメモリ (CUDA)
  - Hostメモリ → Hostメモリ (MPI)
  - Hostメモリ → GPUメモリ (CUDA)

GPU・Host間のデータ転送

FUJITSU

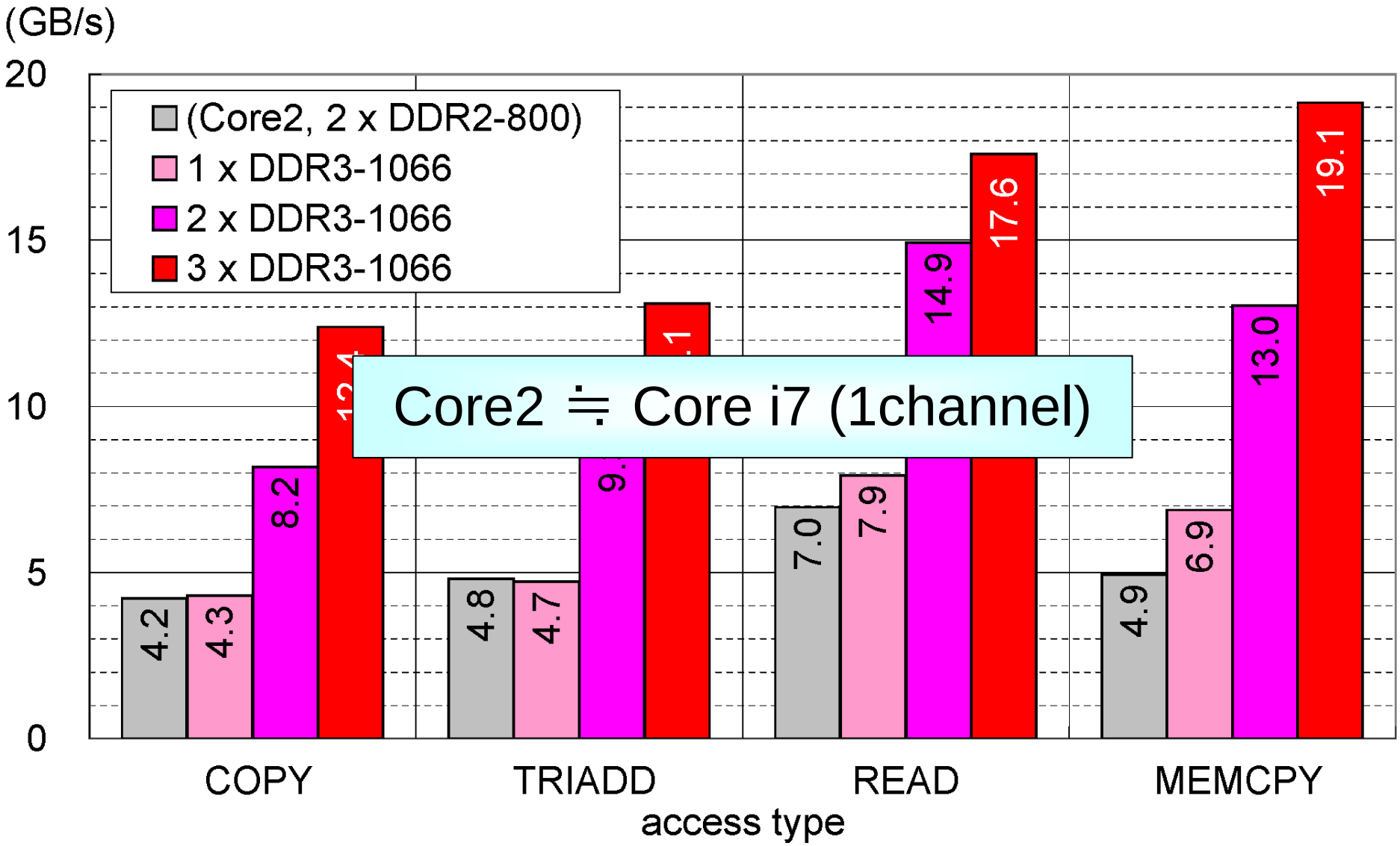
- PinnedメモリとPageableメモリ
  - Pinnedメモリ … cudaMallocHost()
    - DMA可能
  - Pageableメモリ … malloc()
    - DMA不可、Hostメモリ内でコピーが必要





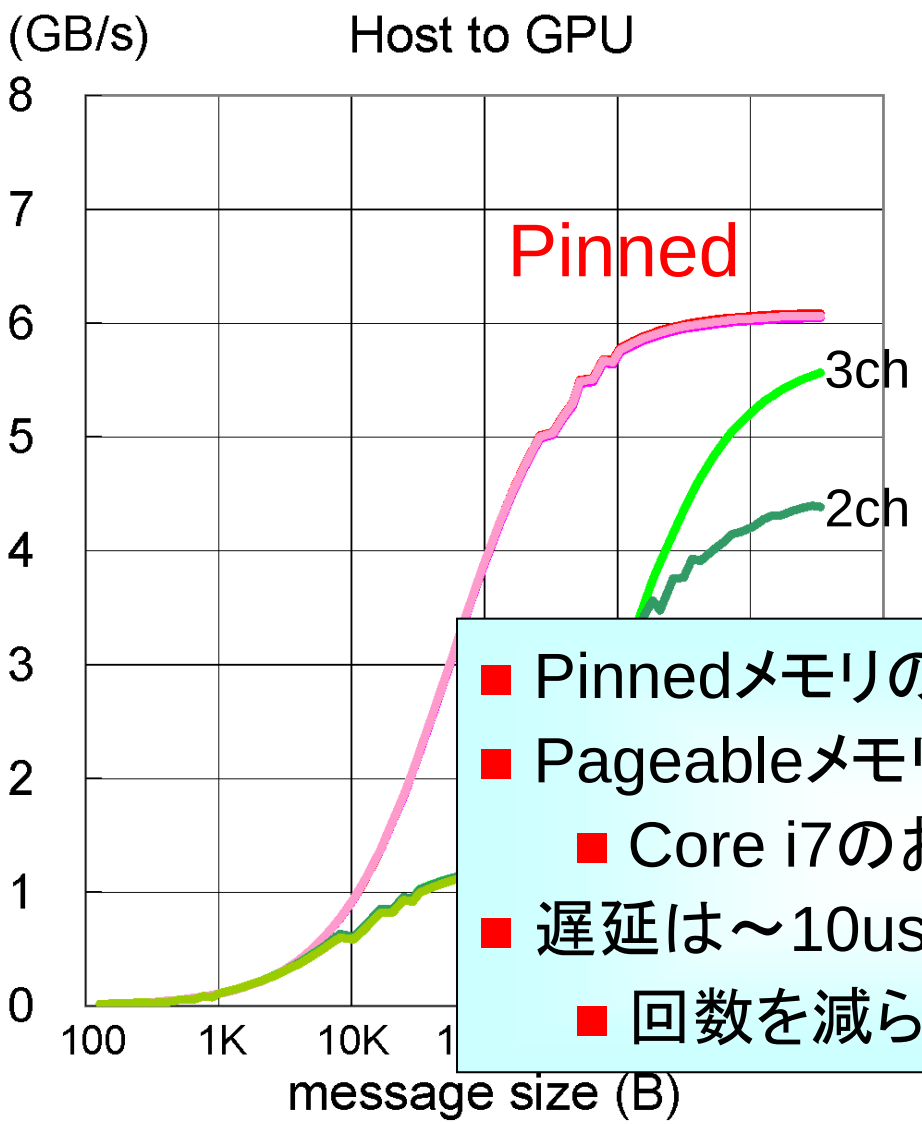
Core i7のメモリバンド幅

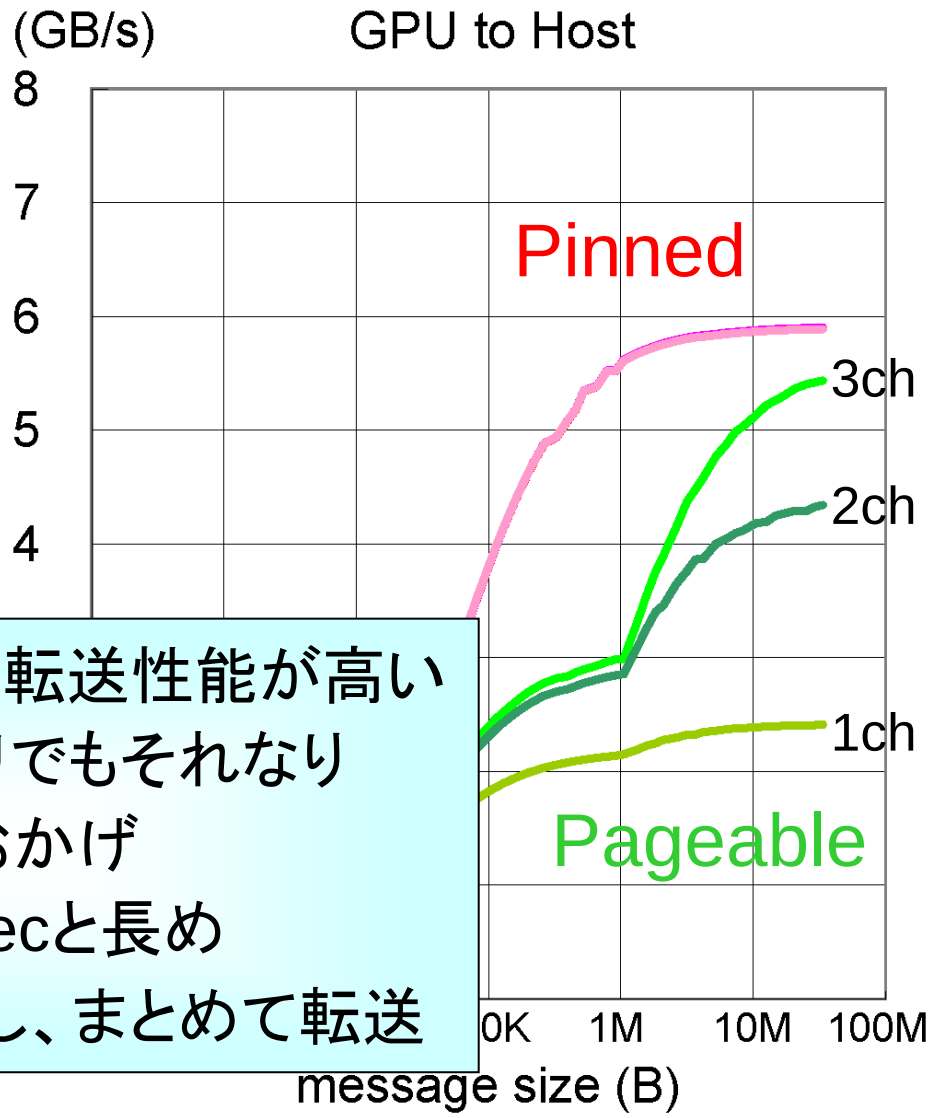
FUJITSU



GPU・Host間のデータ転送性能

FUJITSU

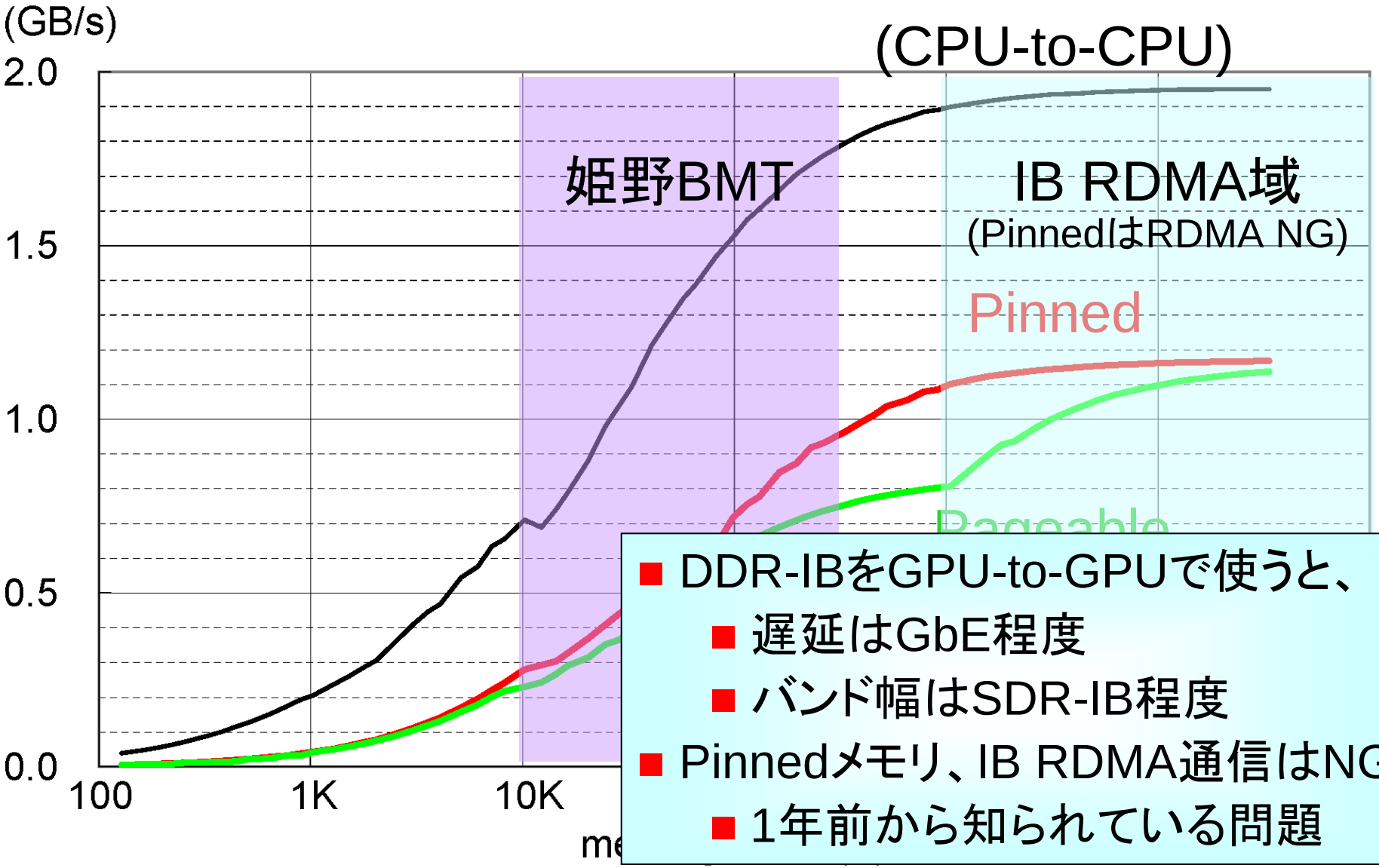




- Pinnedメモリの転送性能が高い
- Pageableメモリでもそれなり
  - Core i7のおかげ
- 遅延は～10usecと長め
  - 回数を減らし、まとめて転送

GPU-to-GPU通信性能

FUJITSU

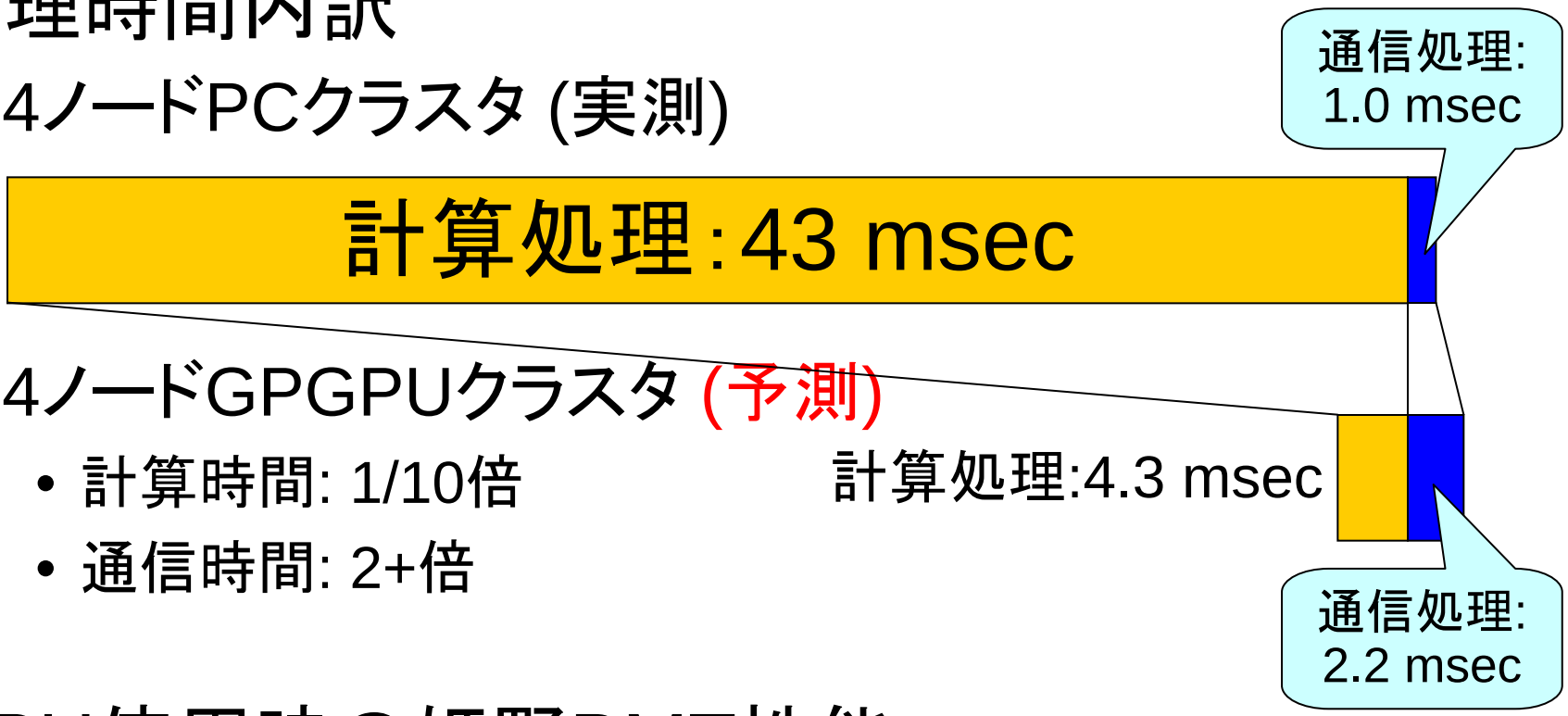


- DDR-IBをGPU-to-GPUで使うと、
  - 遅延はGbE程度
  - バンド幅はSDR-IB程度
- Pinnedメモリ、IB RDMA通信はNG
  - 1年前から知られている問題

GPGPUクラスタの性能予測

FUJITSU

- 処理時間内訳
  - 4ノードPCクラスタ (実測)



  - 4ノードGPGPUクラスタ (予測)
    - 計算時間: 1/10倍
    - 通信時間: 2+倍
- GPU使用時の姫野BMT性能
  - 1ノードGPGPU: 70GFLOPS
  - 4ノードGPGPUクラスタ: 170GFLOPS (予測)

姫野BMT on GPGPUクラスタ

FUJITSU

発表の概要

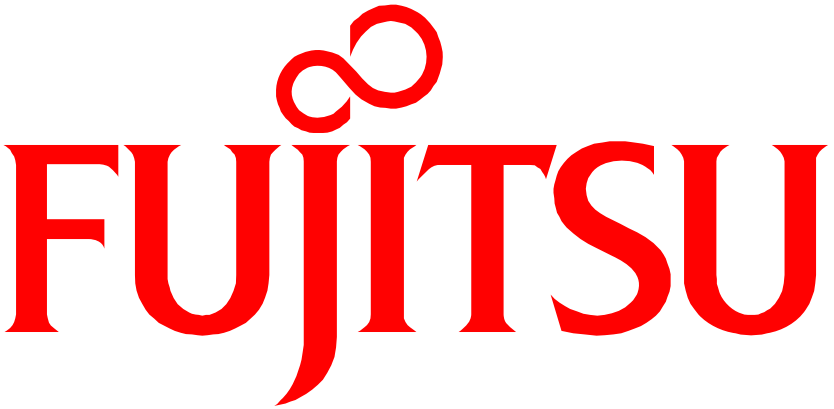
FUJITSU

- 背景
- GPGPUによる高速化
  - CUDAの概要
  - GPUのメモリアクセス特性調査
  - 姫野BMTの高速化
- GPGPUクラスタによる高速化
  - GPU・Host間のデータ転送
  - GPU-to-GPUの通信性能
  - GPGPUクラスタ上での姫野BMT性能
- まとめ

まとめ

FUJITSU

- GPGPUは使えるか？ … YES
  - GPU向けプログラム最適化ノウハウの蓄積
  - 姫野BMTで、メモリバンド幅効率80%超
    - 理論ピーク142GB/sに対して、実効で115GB/sを実現
- GPGPUクラスタは使えるか？
  - 数ノードで通信ネック
    - GPU: 計算にはアクセル、通信にはブレーキ
  - より高速な通信機構が必要
    - GPU直接通信
    - GPU・CPU統合



THE POSSIBILITIES ARE INFINITE