倍々精度 Rgemm の nVidia C2050 上への実装と応用

中田真秀 †

maho@riken.jp

http://accc.riken.jp/maho/

高雄 保嘉 ††, **野田 茂穂** †, **姫野 龍太郎** †

理化学研究所 情報基盤センター †, **株式会社エクサ** ††

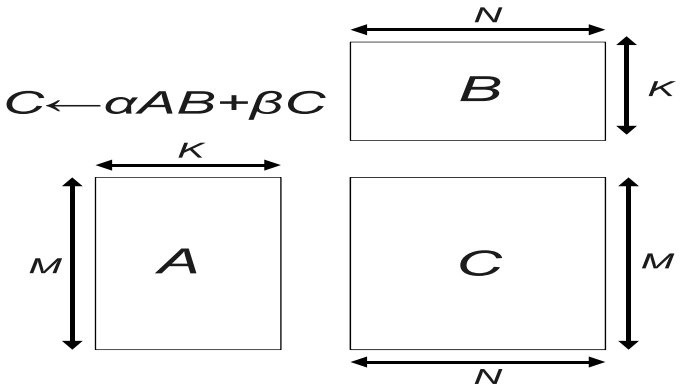
理研シンポ 2011/2/16

概観

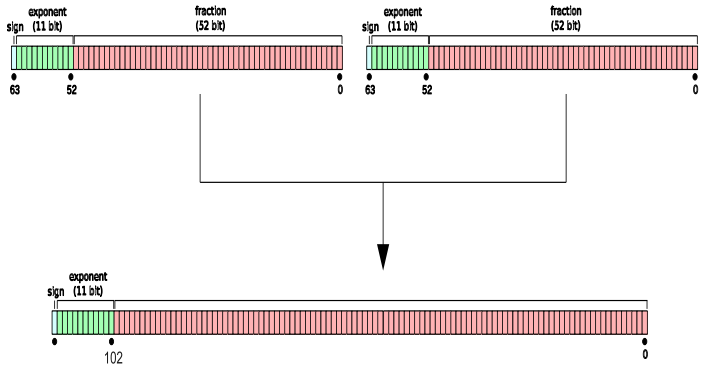
- 研究内容を 1 スライドで紹介
- 線形代数の行列-行列積
- 高精度化の重要性
- 倍々精度:手軽な四倍精度
- GPU 上での実装と評価
- 応用計算:半正定値計画問題ソルバ “SDPA-DD” の 10 倍の加速
- まとめと今後の展望

研究内容を 1 スライドで紹介

行列-行列積 Rgemm 完全実装

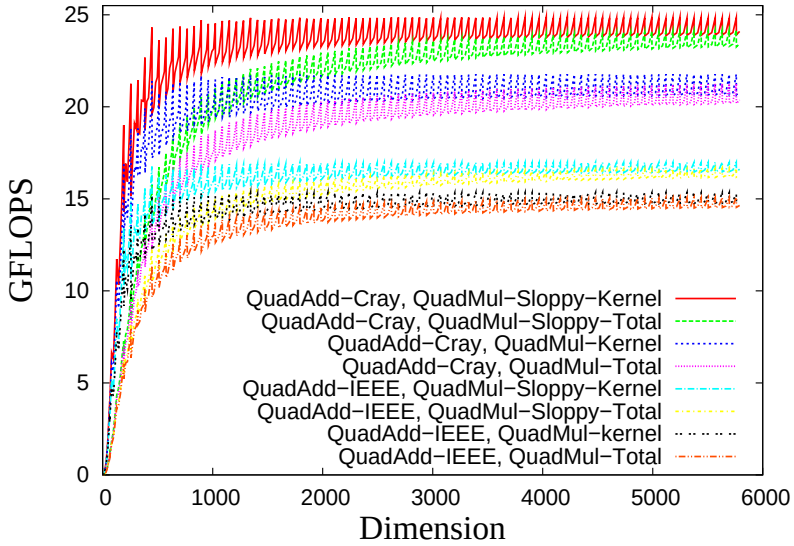


倍々精度:約 4 倍精度 (32 桁)



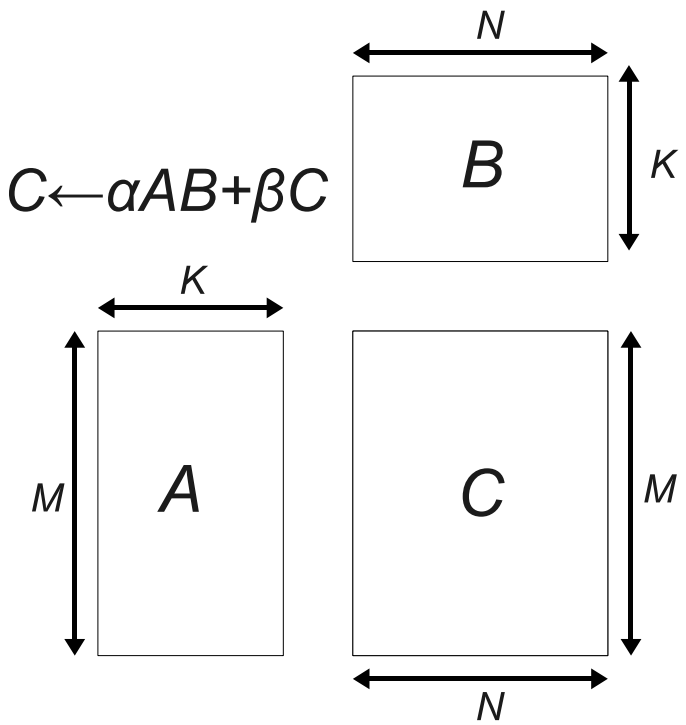
nVidia C2050, GPU の利用

CPU 比 150 倍, 最高 24GFlops



線形代数の行列-行列積

コンピュータでの行列-行列積は重要かつ、様々なアルゴリズムに使われている。



- **線形連立一次方程式を解く → 工学、理学、物理、あらゆるところから出てくる。**
- BLAS **の** “dgemm” **というルーチンが広く使われている。**
- LINPACK : **予算獲得、国家威信の発揚のため政治的に重要。**

高精度の重要性

- エクサ、ペタフロップスの時代へ向けて規模の巨大化に伴い、誤差が 16 桁 (IEEE 754 double) では足りなくなってくるのではないか。
  - エクサ、ペタのマシンで一週間計算すると $10^{23}, 10^{20}$ 程度の演算!!
- 小規模でも誤差が出やすい問題がある。
  - Krylov 部分空間法 (や共役勾配法): 精度をあげると収束したり、収束回数が減ったりする。
  - 半正定値計画法: 最適解付近で条件数が高くなり、精度が足りなくなる。
- LAPACK チームのサーベイ (2007):16%のユーザが、高精度なものを、「よく使っている」、「時々使っている」

高精度の重要性

じゃあなぜ高精度計算はあんまりやられてないか? 遅いという思い込みがある。

- 高精度計算は 100 倍から 1000 倍遅いという根拠の無い思い込み。
- 特に倍々精度なら 8-20 倍程度で計算できるはずで、10Pflops マシンなら 1PFlops 程度で倍々精度ができるはず。
- CPU、プロセッサ最適化が進んでない...BLAS も reference BLAS と GotoBLAS2 の DGEMM 性能では 20 倍以上パフォーマンスが違うこともある。
- 近年の CPU-Memory の転送は非常に遅い... むしろ演算密度の高い高精度演算は最近のハードに向いている。
- FMA などハードのサポートが無いと高速化しにくい...nVidia, AMD の GPU は持っている。

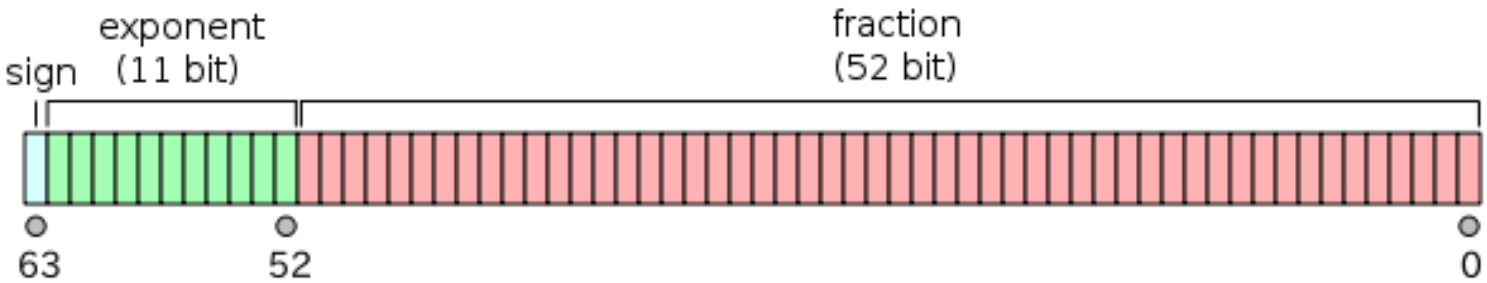
GPU で 100 倍から 200 倍高速化して CPU とコンパラになれば

やってみようという気になるはず → やってみた

倍々精度:手軽な四倍精度

まず浮動小数点から。

- コンピュータで実数は扱えない。浮動小数点数を使うのが一般的。ただ誤差が入る。これが大問題。
- 規格名 754-2008 IEEE Standard for Floating-Point Arithmetic
- binary64 (倍精度) 仮数部が 64-bit ある: 有効数字約 16 桁

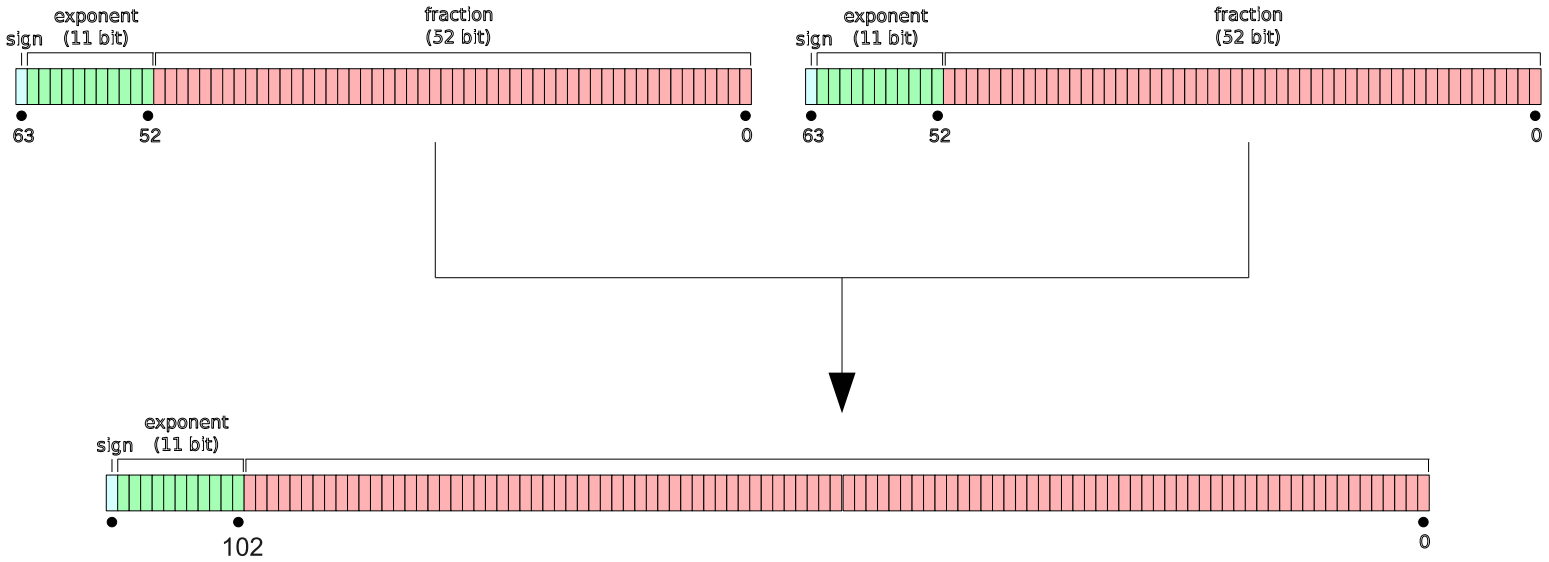


- $a = \pm\left(\frac{1}{2} + \frac{d_2}{2^2} + \frac{d_3}{2^3} + \cdots + \frac{d_{52}}{2^{52}}\right) \times 2^e$, $d = 0$ or 1, $e = -1022 \sim 1023$
- 一番よく使われ、高速 (Core i7 920: ~40GFlops; RADEON HD5970 ~900GFlops)

(Partially taken from Wikipedia: http://en.wikipedia.org/wiki/IEEE_754-2008).

倍々精度:手軽な四倍精度

倍精度二つくっつけて四倍精度でいいんじゃない?*



倍々精度型 a は二つの倍精度 a_{hi}, a_{lo} で定義される:

$$a = (a_{hi}, a_{lo}).$$

* 規格化されている binary128(4 倍精度) と比較して若干精度は落ちる

倍々精度:手軽な四倍精度

有名なライブラリ: QD **ライブラリ**
特徴: C++**のクラス。倍々精度** “dd_real” **だけでなく、倍精度** 4 **つ並べた** “qd_real” **もある。フリーソフトウェア** (BSD)
作者: Yozo Hida, Xiaoye S. Li, David H. Bailey
ダウンロード:

`http://crd.lbl.gov/~dhbailey/mpdist/`

論文:

`http://crd.lbl.gov/~dhbailey/dhbpapers/arith15.pdf`

Yozo Hida, Xiaoye S. Li, David H. Bailey, “Quad-Double Arithmetic: Algorithms, Implementation, and Application”, *Technical Report LBNL-46996, Lawrence Berkeley National Laboratory*, 2000.

倍々精度:手軽な四倍精度

演算の定義の基礎:Knuth **の定理**
浮動小数点数 a, b **について**

$$a + b = (a \oplus b) + e$$

$\oplus$ **は誤差を含む浮動小数点演算の加算、** $+$ **は数学的な意味での加算、** e **は浮動小数点数。実は厳密に計算できる**

倍々精度:手軽な四倍精度

演算の定義の基礎:Knuth **の定理**
a, b **を浮動小数点数**, $|a| \geq |b|$ **のとき** $\oplus, \ominus$ **は誤差を含む浮動小数点演算として、** $s = (a \oplus b), e = a + b - (a \oplus b)$ **を厳密に** 3 **演算で計算する。**

Quick-Two-Sum (a, b):

1. $s \leftarrow a \oplus b$
2. $e \leftarrow b \ominus (s \ominus a)$
3. $\mathbf{return}(s, e)$

$(s, e) =$ Quick-Two-Sum (a, b)

倍々精度:手軽な四倍精度

演算の定義の基礎:Knuth **の定理**
a, b **を浮動小数点数**, $\oplus, \ominus$ **は誤差を含む浮動小数点演算として、**
$s = (a \oplus b), e = a + b - (a \oplus b)$ **を厳密に** 6 **演算で計算する。**

Two-Sum (a, b):

1. $s \leftarrow a \oplus b$
2. $v \leftarrow s \ominus a$
3. $e \leftarrow (a \ominus (s \ominus v)) \oplus (b \ominus v)$
4. $\mathbf{return}(s, e)$

$(s, e) =$ Two-Sum (a, b)

倍々精度:手軽な四倍精度

演算の定義の基礎:Dekker **の定理**
浮動小数点数 a, b **について**

$$a \times b = (a \otimes b) + e$$

⊗ は誤差を含む浮動小数点演算の積算、 $\times$ **は数学的な意味での積算、** e **は浮動小数点数。これも厳密に計算できる**

倍々精度:手軽な四倍精度

演算の定義の基礎:Dekker **の定理**
$s = (a \otimes b), e = a \times b - (a \otimes b)$ **を計算する。**$\otimes$ **は浮動小数点数の積 (誤差あり)**
補助となる 4 **演算の** Split(a) **と** 17 **演算の** Two-Prod(a,b)**。**

Split (a):

1. $t \leftarrow (2^{27} + 1) \otimes a$
2. $a_{hi} \leftarrow t \ominus (t \ominus a)$
3. $a_{lo} \leftarrow a \ominus a_{hi}$
4. $\mathbf{return}(a_{hi}, a_{lo})$

Two-prod (a, b):

1. $p \leftarrow a \otimes b$
2. $(a_{hi}, a_{lo}) \leftarrow \mathbf{Split}(a)$
3. $(b_{hi}, b_{lo}) \leftarrow \mathbf{Split}(b)$
4. $e \leftarrow ((a_{hi} \otimes b_{hi} \ominus p) \oplus a_{hi} \otimes b_{lo} \oplus a_{lo} \otimes b_{hi}) \oplus a_{lo} \otimes b_{lo}$
5. $\mathbf{return}(p, e)$

(s, e) =Two-Prod(a, b)

倍々精度:手軽な四倍精度

倍々精度型の加算。20 演算

QuadAdd-IEEE (a, b):

1. $(s_{hi}, e_{hi}) =$ Two-Sum(a_{hi}, b_{hi})
2. $(s_{lo}, e_{lo}) =$ Two-Sum(a_{lo}, b_{lo})
3. $e_{hi} = e_{hi} \oplus s_{lo}$
4. $(s_{lo}, e_{lo}) =$ Quick-Two-Sum(s_{hi}, e_{hi})
5. $e_{hi} = e_{hi} \oplus s_{lo}$
6. $(s_{hi}, e_{lo}) =$ Quick-Two-Sum(s_{hi}, e_{hi})
7. **return**(c)

倍々精度:手軽な四倍精度

24 演算かかる倍々精度型の乗算

QuadMul (a, b):

1. $(p_{hi}, p_{lo}) =$ Two-Prod(a_{hi}, b_{hi})
2. $p_{lo} = p_{lo} \oplus (a_{hi} \otimes b_{lo} \oplus a_{lo} \otimes b_{hi})$
3. $(c_{hi}, c_{lo}) =$ Quick-Two-Sum(p_{hi}, p_{lo})
4. **return**(c)

倍々精度:手軽な四倍精度

より高速にする方法 1. FMA **を使う**
FMA (fused multiply-add) **とは**??

$$a \times b + c$$

を一命令で行い、かつ $a \times b + c$ **は厳密に行い、最後に浮動小数点に丸める。**

Note: **現在** Intel/AMD **の** CPU **とも** FMA **は未サポート** (Intel **は** 2013 **年の** AVX **から**, AMD **は** 2011 **年の** Bulldozer **から**)。IBM **など** Power **系**, nVidia C1060, C2050 **や** AMD RADEON HD5xxx, HD6xxx **は** FMA **を持っている。**

倍々精度:手軽な四倍精度

FMA で Two-Prod が 17 演算から 3 演算へ、それに伴い QuadMul が 24 演算から 10 演算へ!

Two-prod-FMA (a, b):

1. $p \leftarrow a \otimes b$
2. $e \leftarrow \mathrm{FMA}(a \times b - p)$
3. $\mathbf{return}(p, e)$

倍々精度:手軽な四倍精度

より高速にする方法 2. 若干精度を落とす

QuadAdd-Cray (a, b):

1. $(c_{hi}, c_{lo}) =$ Two-Sum(a_{hi}, b_{hi})
2. $c_{lo} = c_{lo} \oplus (a_{lo} \oplus b_{lo})$
3. $(c_{hi}, c_{lo}) =$ Quick-Two-Sum(c_{hi}, c_{lo})
4. **return**(c)

QuadMul-Sloppy (a, b):

1. $p = (a_{hi} \otimes b_{lo})$
2. $q = (a_{lo} \otimes b_{hi})$
3. $t = p \oplus q$
4. $c_{hi} = \mathrm{FMA}(a_{hi} \times b_{hi} + t)$
5. $e = \mathrm{FMA}(a_{hi} \times b_{hi} - c_{hi})$
6. $c_{lo} = e \oplus t$
7. **return**(c)

倍々精度:手軽な四倍精度

演算量のまとめ

アルゴリズム	演算回数
Quick-Two-Sum	3
Two-Sum	6
Split	4
Two-Prod	17
Two-Prod-FMA	3*
QuadAdd-IEEE	20
QuadAdd-Cray	11
QuadMul	24
QuadMul-FMA	10*
QuadMul-FMA-Sloppy	8*

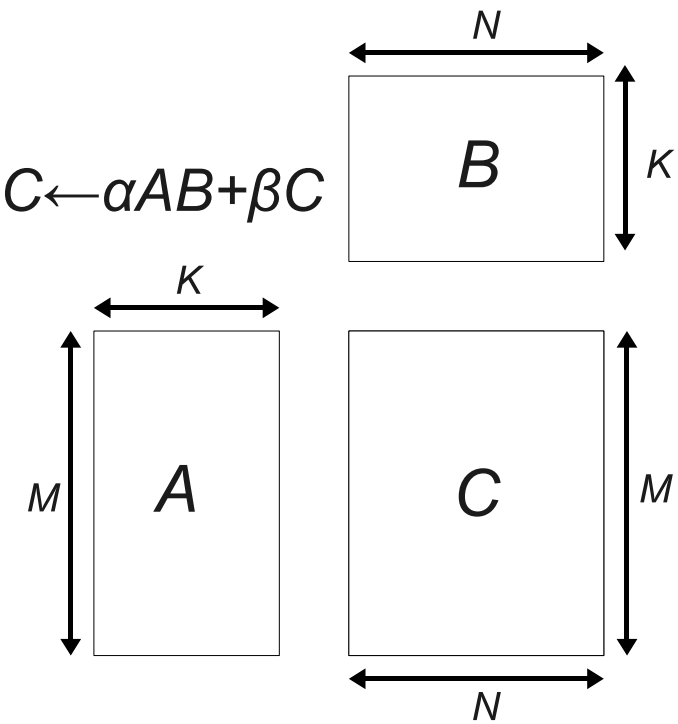
*FMA **命令は** 2Flops **と数えた**.

特に断らない限り、QuadAdd-IEEE, QuadMul-FMA **を用いた。**

GPU 上での実装と評価

行列-行列積ルーチンの Rgemm **の高速化を行った。倍々精度** Rgemm **のプロトタイプ宣言**.

```
void Rgemm(const char *transa, const char *transb,
mpackint m, mpackint n, mpackint k, dd_real alpha,
dd_real * A, mpackint lda, dd_real * B, mpackint ldb,
dd_real beta, dd_real * C, mpackint ldc)
```



BLAS, LAPACK(**デファクトスタンダードの線形代数ライブラリ**) **の高精度版である中田による** “MPACK”

http://mplapack.sourceforge.net

のルーチン Rgemm (BLAS **の** dgemm, sgemm **に対応**)

GPU 上での実装と評価

過去の研究

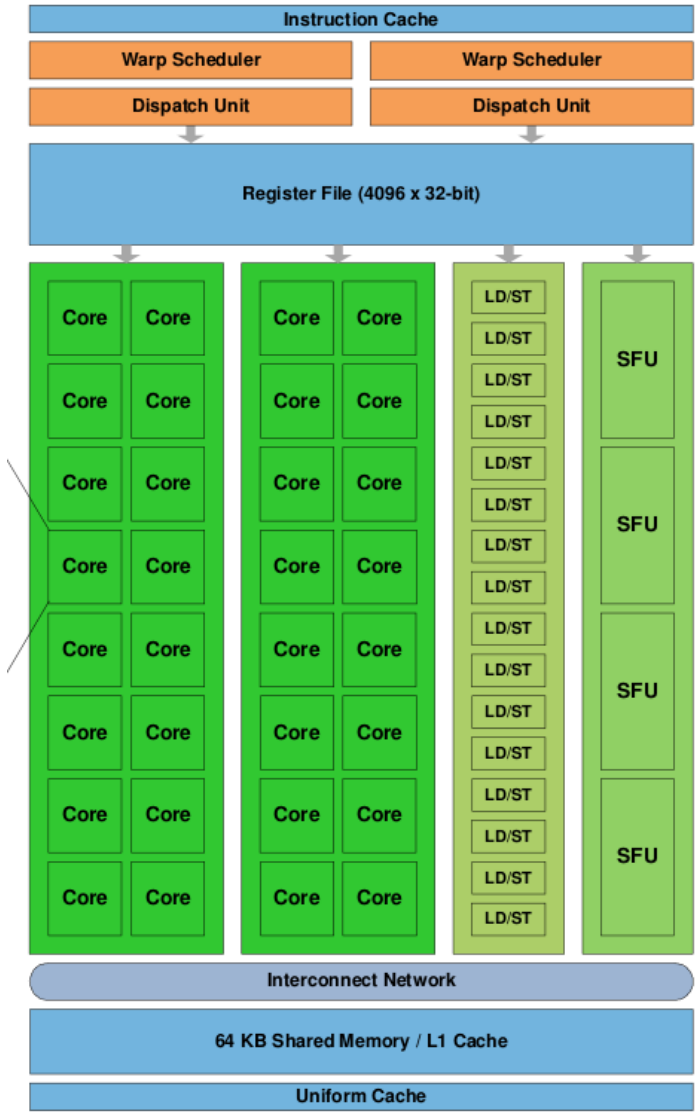
- **椋木大地, 高橋大介** :GPU **による** 4 **倍精度** BLAS **の実装と評価**, **情報処理学会研究報告**, HPC, Vol.123, pp.13, 2009, **椋木大地, 高橋大介**: GPU **による** 4 **倍・**8 **倍精度** BLAS **の実装と評価，**2011 **年ハイパフォーマンスコンピューティングと計算科学シンポジウム** HPCS201 **論文集**, pp. 148-156, (2011). → **行列のサイズが** 64 **の倍数でなければらない, 少し遅い**
- Nakasato, N.:, “A Fast GEMM Implementation On a Cypress GPU, Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems”, Louisiana, USA, 2010. → **行列のサイズが** 64 **の倍数でなければらないが** 31GFlops RADEON 58xx **は** C2050 **より二倍以上高速**!

どちらも実用的ではない → **完全実装と応用を目標**

GPU 上での実装と評価

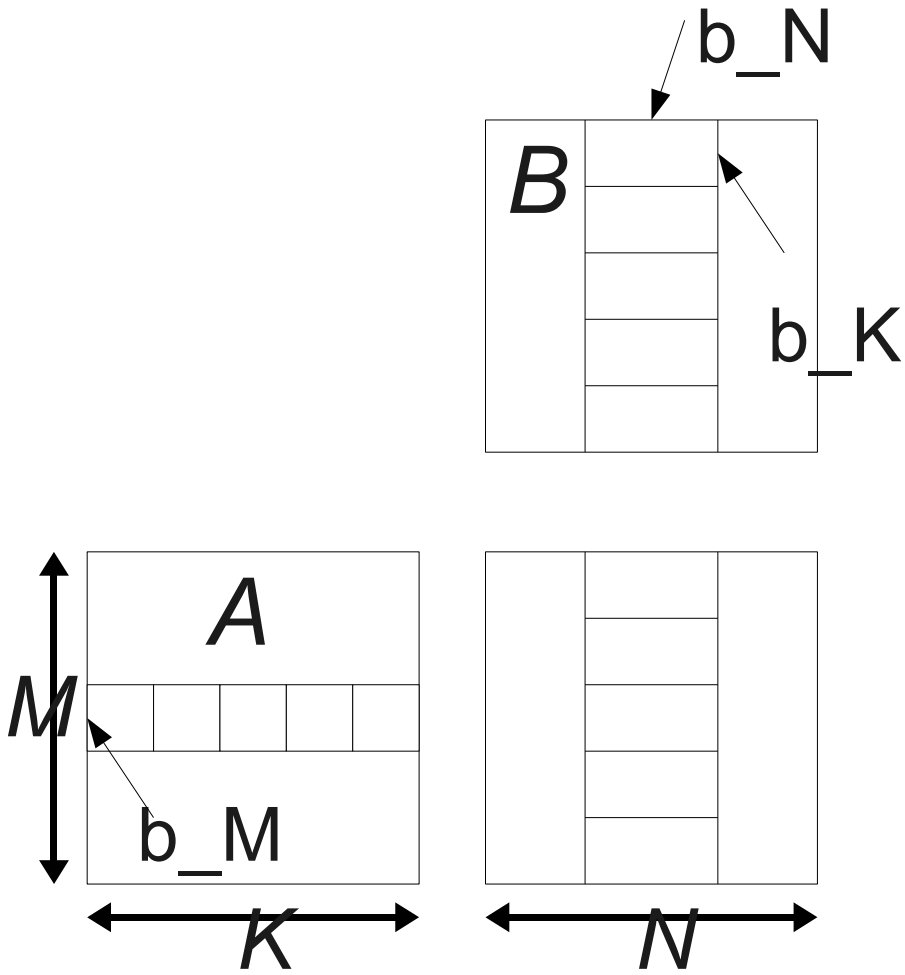
nVidia C2050

アーキテクチャ



GPU 上での実装と評価

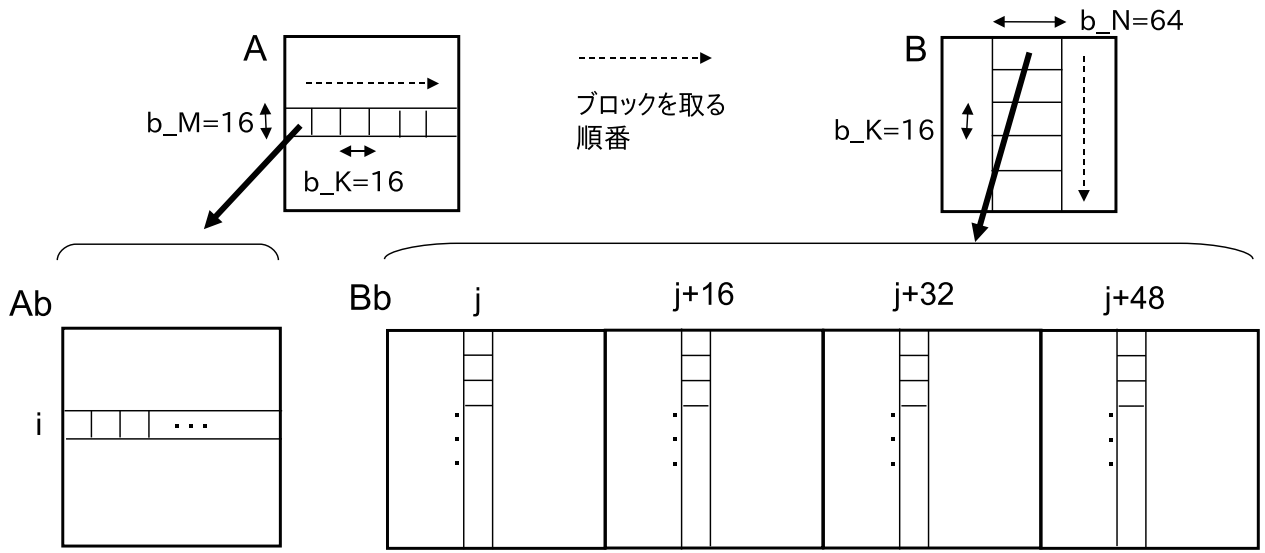
行列のブロック化の模式図. b_K, b_M, b_N **のように小さなブロックに分ける**. b_M = b_K = 16 **を用い**, b_N **には** 64 **を用いた**.



GPU 上での実装と評価

基本アルゴリズム:

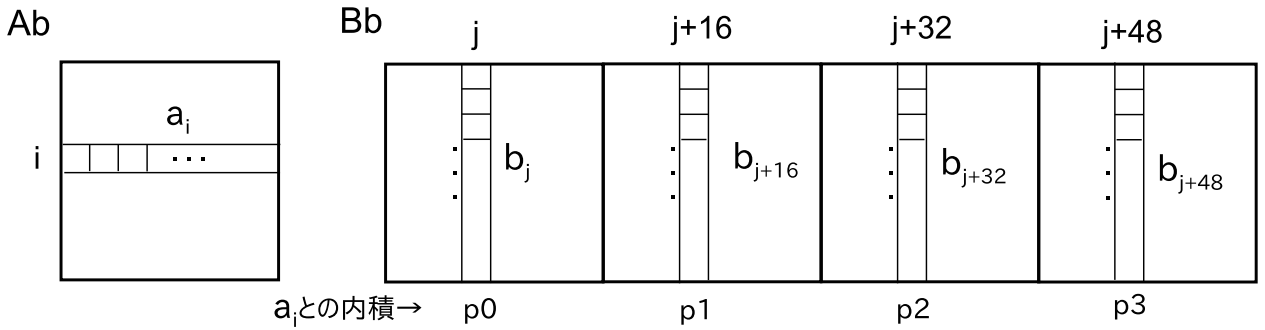
1. **行列** A,B,C **を転送する：**CPU **メモリ** → GPU **グローバルメモリ**
2. A **のブロック** Ab **は** 16×16、B **のブロック** Bb **は** 16×64 **が一番効率が良かった。**
3. **このブロックに対し、** $16 \times 16 = 256$ **個のスレッドブロックを対応させる。スレッドブロック中の** (i, j) **スレッドは、**Ab **の** i **行、および** Bb **の** $j, j+16, j+32, j+48$ **列の** 4 **列を計算する。**



GPU 上での実装と評価

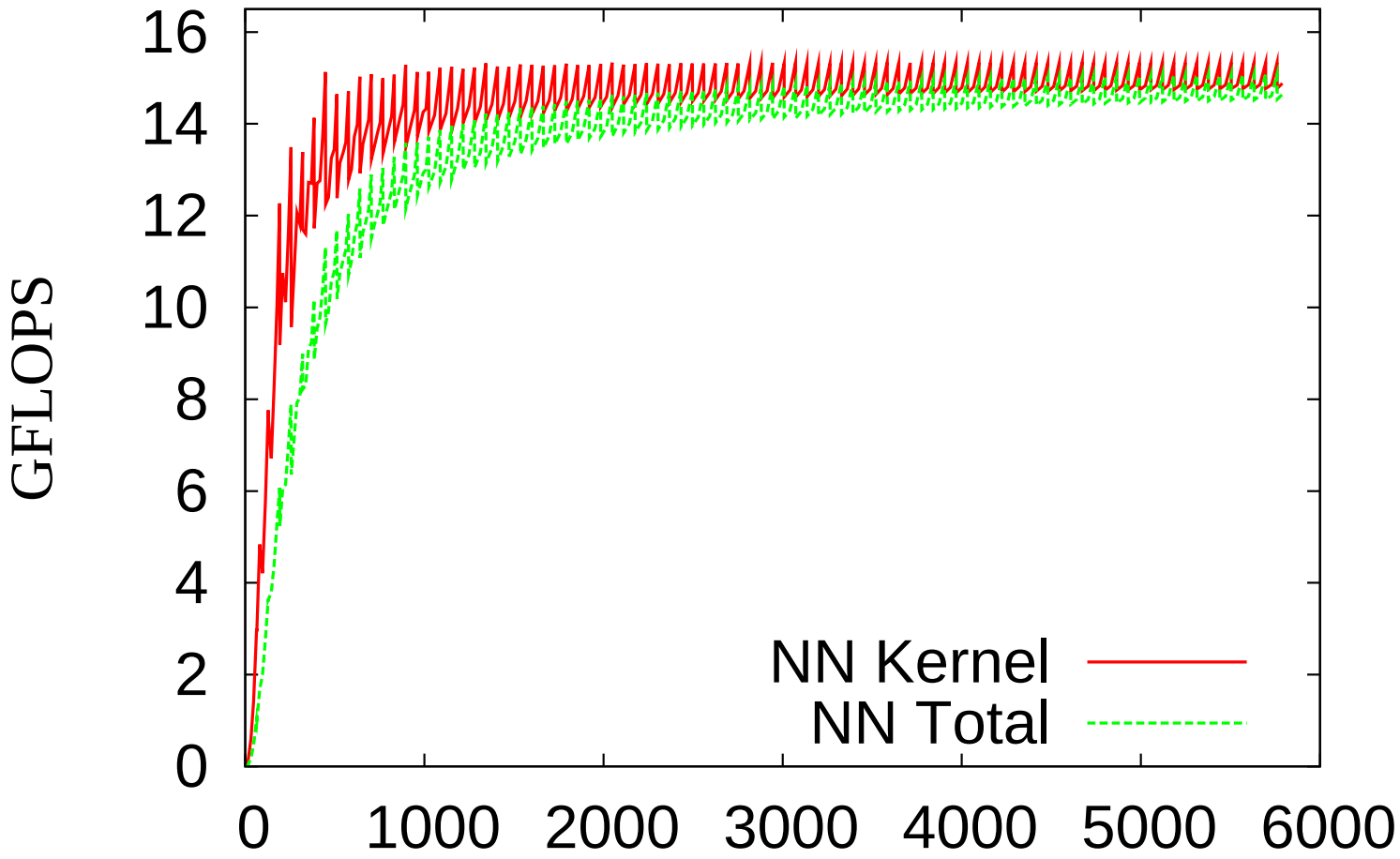
各スレッドの処理:

1. 変数 $c0, c1, c2, c3$ に、Ab の i 行、および Bb の $j, j+16, j+32, j+48$ 列の位置に対応する、行列 C の成分を読み, β をかける。
2. 行列 A,B の最初のブロック Ab,Bb を、グローバルメモリから共有メモリ上に読み込む。ブロックの中の各スレッドが共同で行う。 各スレッドは自分の担当分のみを読む。
3. Ab の行ベクトル ai と、Bb の列ベクトル b_i , b_{i+16} , b_{i+32} , b_{i+48} の内積を計算し p_0, p_1, p_2, p_3 とする。
4. $c0 \leftarrow c0 + \alpha p0$ のように変数 $c0, c1, c2, c3$ の値を更新する
5. 次のブロック Ab, Bb を読み、3, 4 を繰り返し, をブロックが無くなるまで繰り返す。
6. $c0, c1, c2, c3$ で行列 C の更新。
7. 最後に 計算結果の C 行列を GPU グローバルメモリ →CPU に転送.



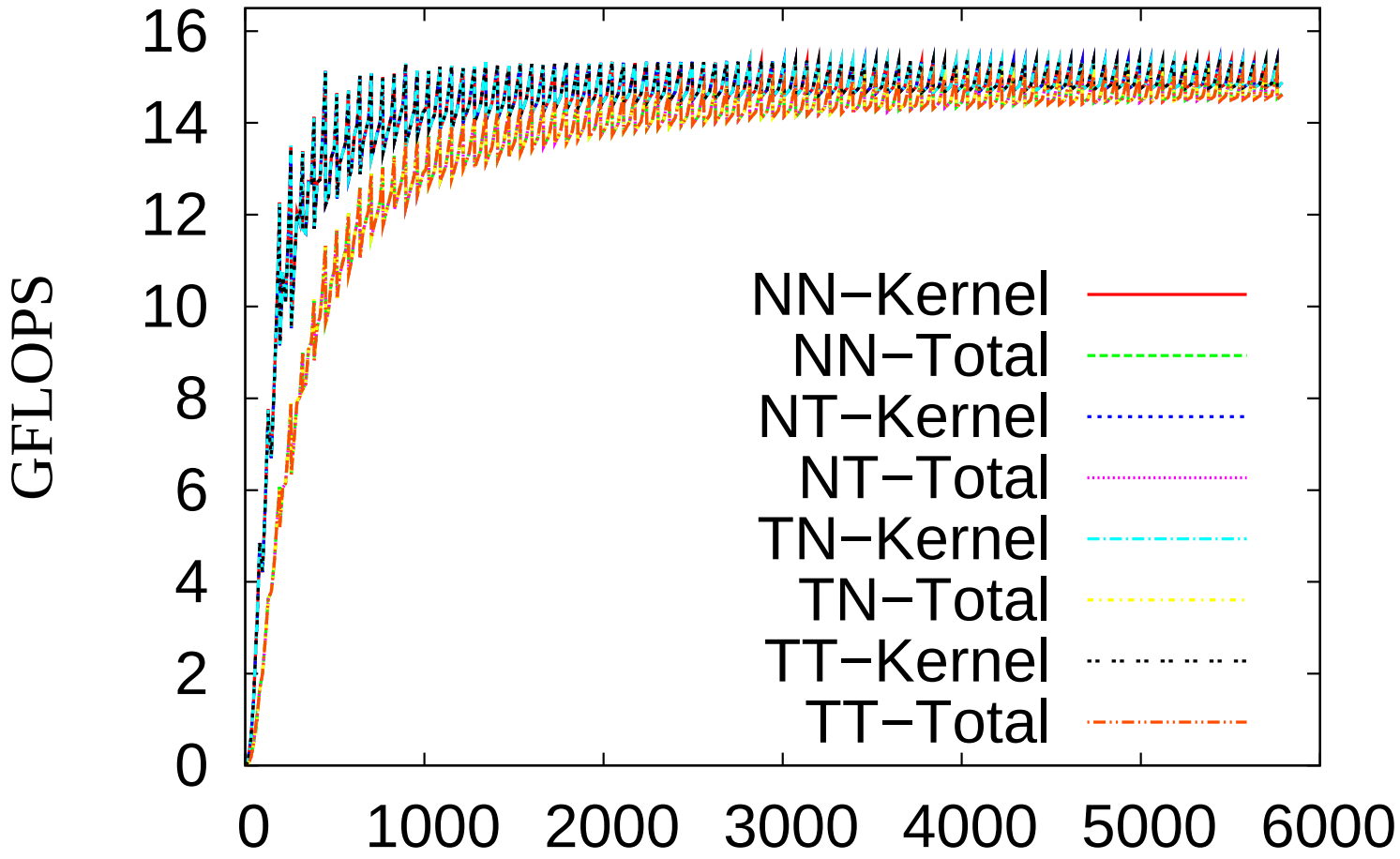
GPU 上での実装と評価

倍々精度行列行列積, 正方行列 $(m = n = k)$ で m を変化させたときの性能。最大カーネル性能 15.3GFlops, **転送含み最大性能** 15.1GFlops



GPU 上での実装と評価

倍々精度行列行列積, 正方行列 $(m = n = k)$ で m を変化させたときの性能。転置有り無し全ての組み合わせを計算した。転置なしとほとんど変わらない。



GPU 上での実装と評価

転置有り無し全ての組み合わせほとんど変わらない秘訣はテクスチャメモリの利用にある。

- グローバルメモリとテクスチャメモリはどちらも本質的には一緒
- コアレッシング (連続的) なメモリアクセスでなくても、テクスチャメモリだとロスが小さい。

	スレッド番号順		スレッド番号順でない	
	グローバルメモリ	テクスチャメモリ	グローバルメモリ	テクスチャメモリ
所要時間 [s]	3.0897	2.7730	9.1265	3.9740
転送性能 [GB/s]	86.9 (60.3%)	96.8 (67.2%)	29.4 (20.4%)	67.5 (46.9%)

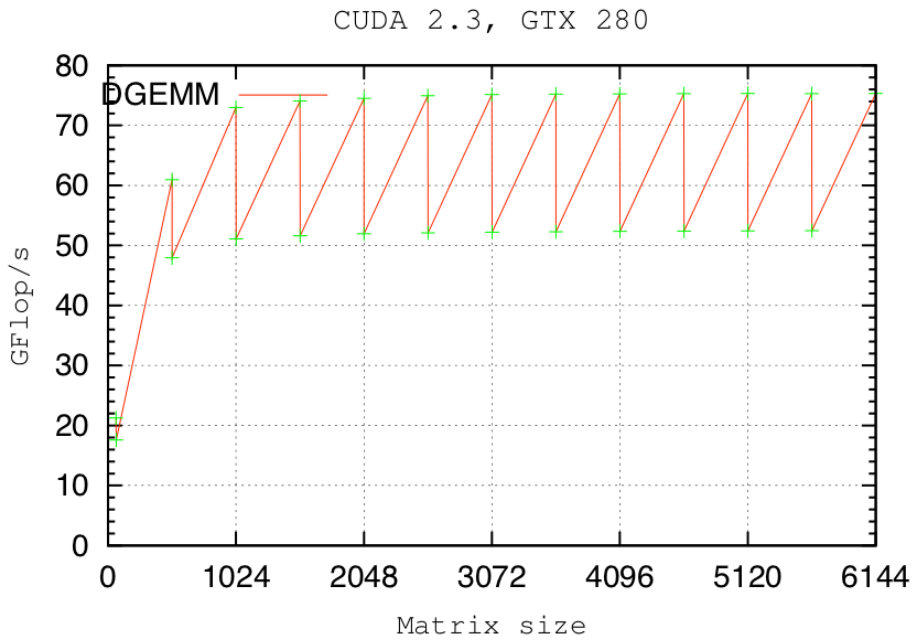
() 内は、最大性能（144 [GB/s] ）比

- 倍々精度だと演算回数が多いため、(cf. QuadAdd-IEEE 20 演算, QuadMul-FMA 10 演算) メモリ転送のレイテンシが隠蔽しやすい。

GPU 上での実装と評価

“Accelerating GPU kernels for dense linear algebra”, Rajib Nath, Stanimire Tomov, and Jack Dongarra **による** Pointer Redirecting **の手法**

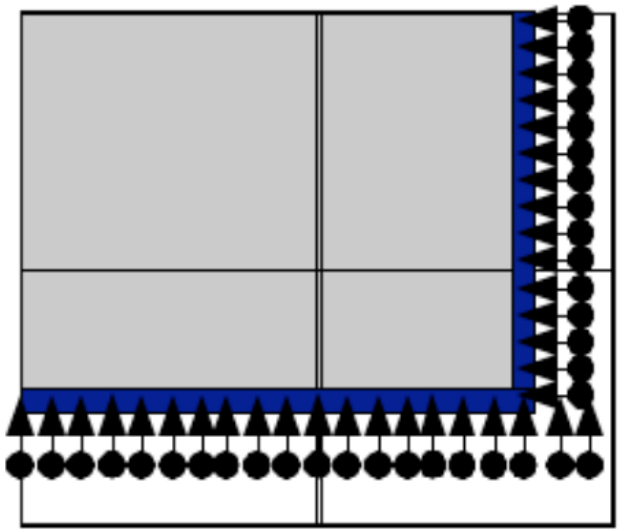
- **問題意識:行列サイズが** 64 **の倍数とか出ないとパフォーマンスが出ないのを何とかしたい。**



GPU 上での実装と評価

“Accelerating GPU kernels for dense linear algebra”, Rajib Nath, Stanimire Tomov, and Jack Dongarra **による** Pointer Redirecting **の手法**

- **超シンプルにとりあえずブロック化の外に出たらその端っこを返すようにする。**

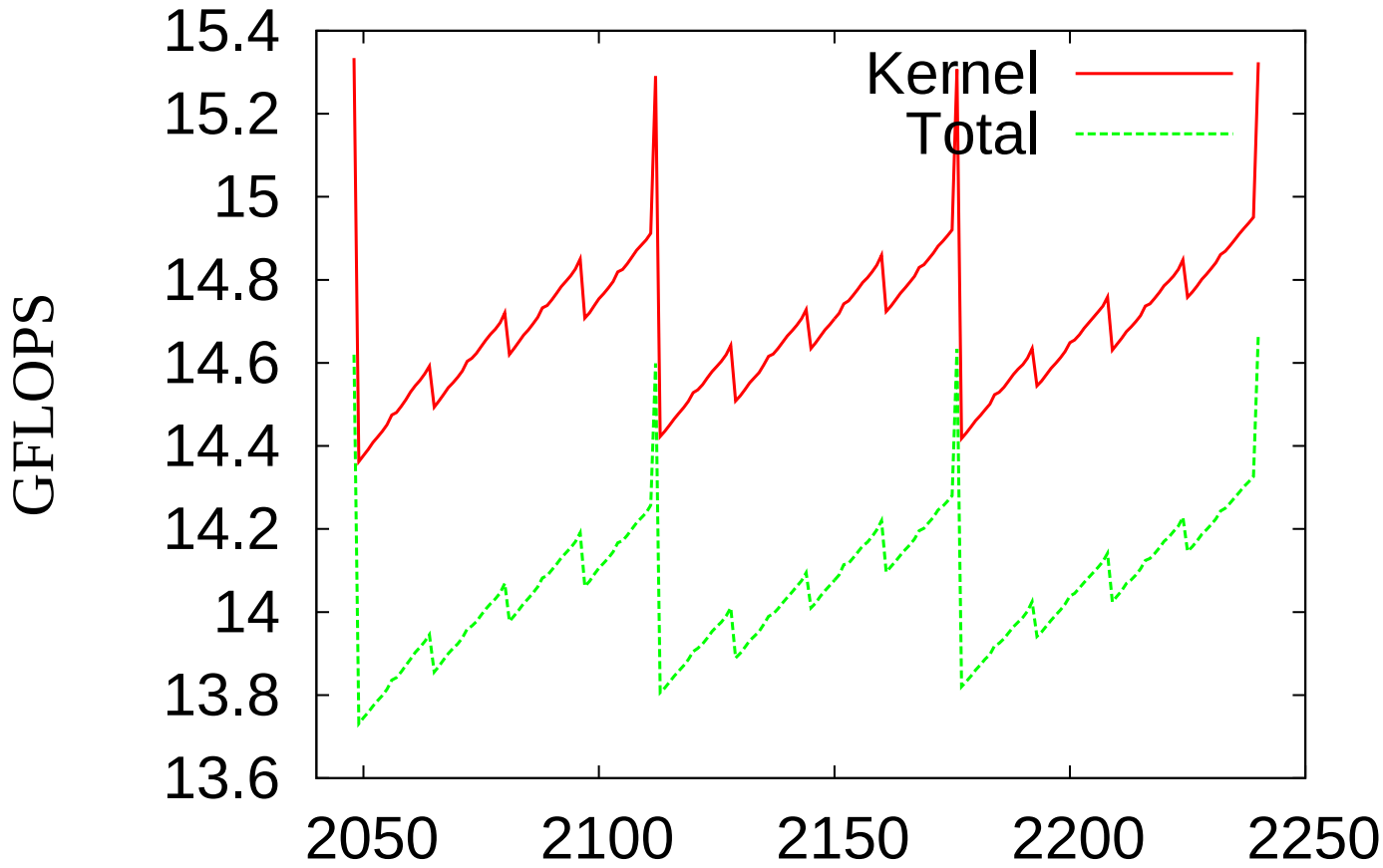


- **プログラムは超簡単。**
- **若干の計算の無駄がでる。**

コロンブスの卵的ブレークスルー!!

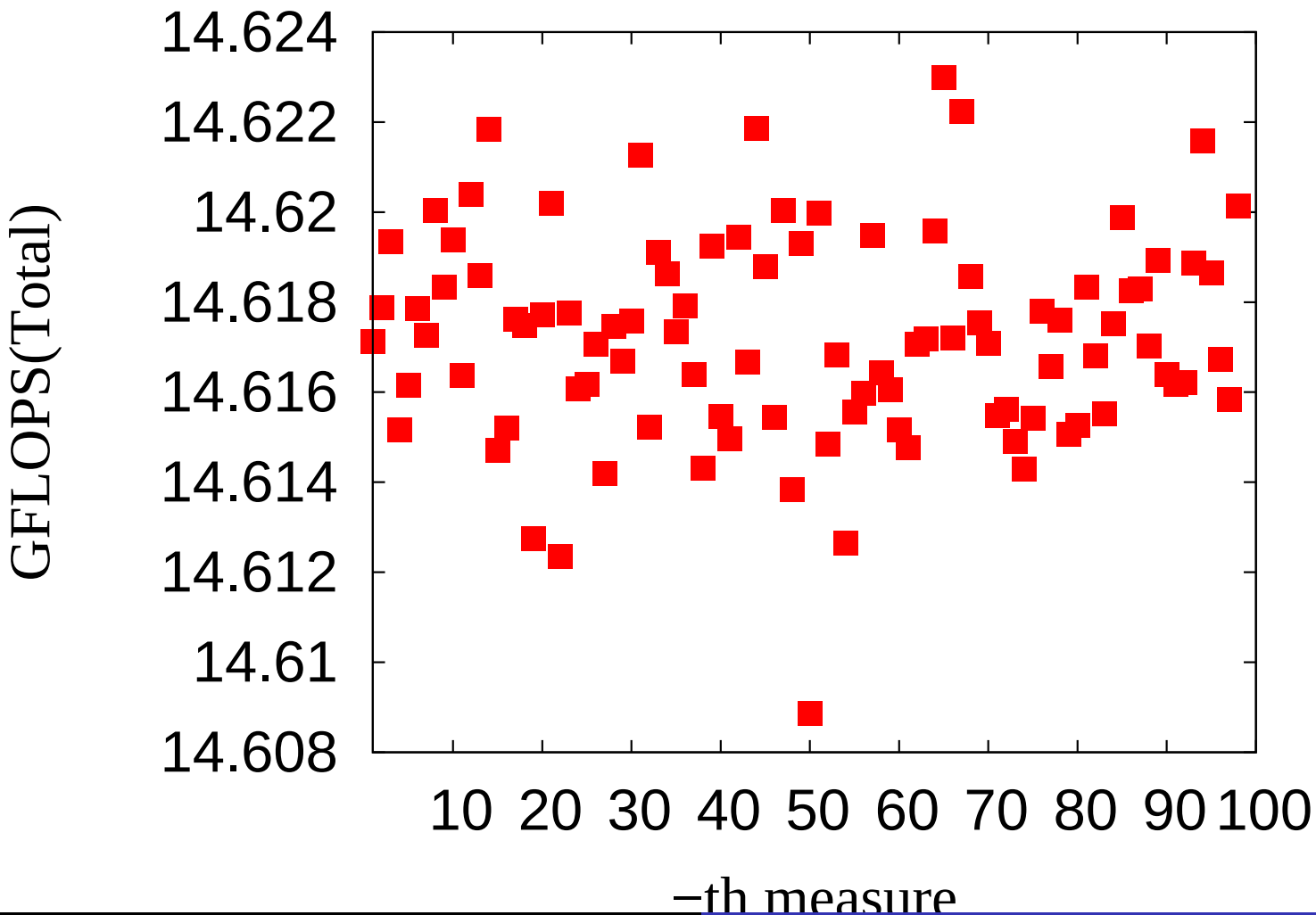
GPU 上での実装と評価

Nath らによる Pointer Redirecting の手法を用いた場合のパフォーマンスロスは 6%程度だった。正方行列で $m = 2048$ から $n = 2248$ まで動かした場合の性能。$n = 64$ の倍数の時, 特にパフォーマンスが良かった.



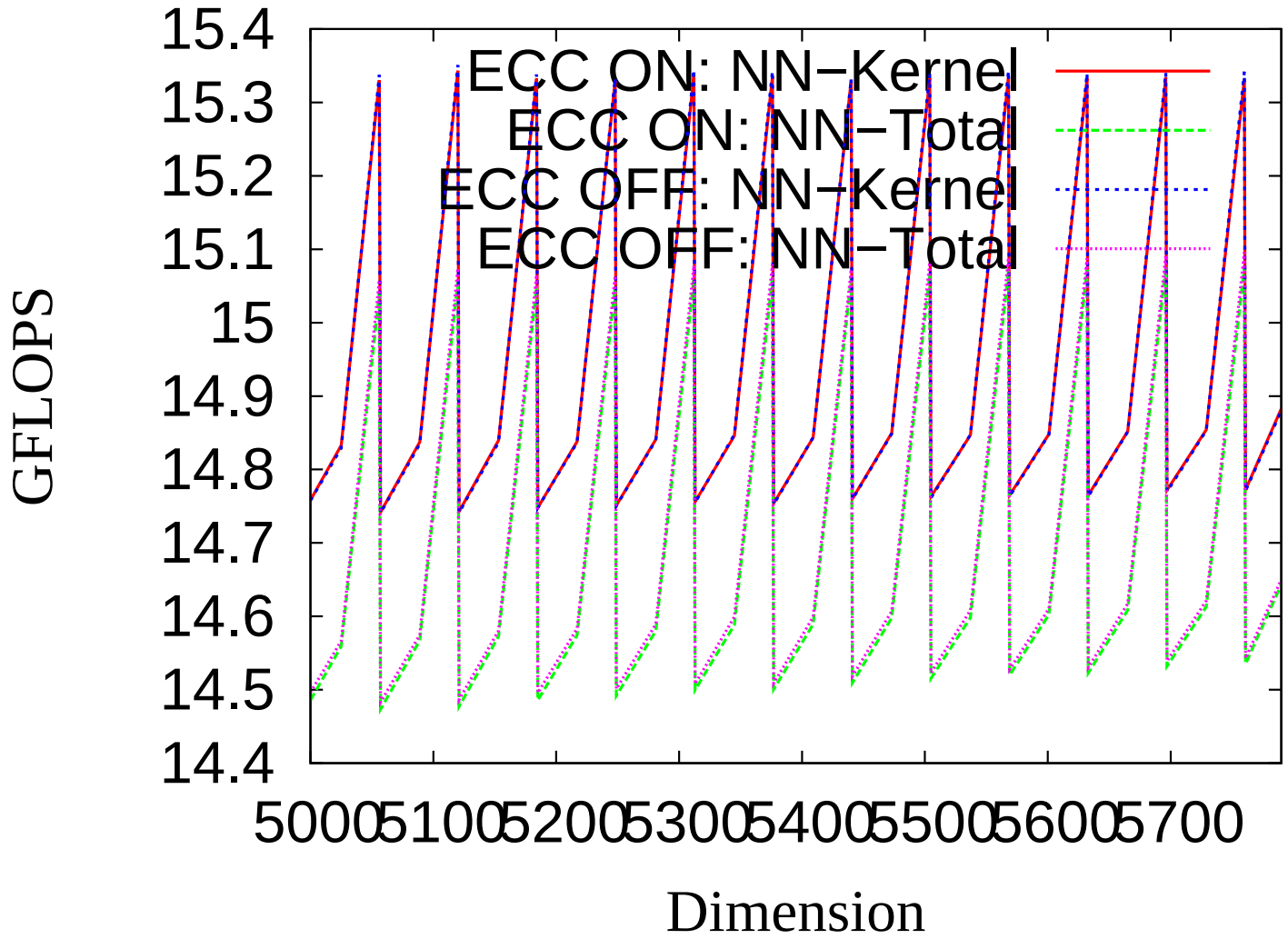
GPU 上での実装と評価

$n = 2048$ のとき 100 回連続して測定したときのパフォーマンス。
性能のぶれは 0.1% 以内だった。



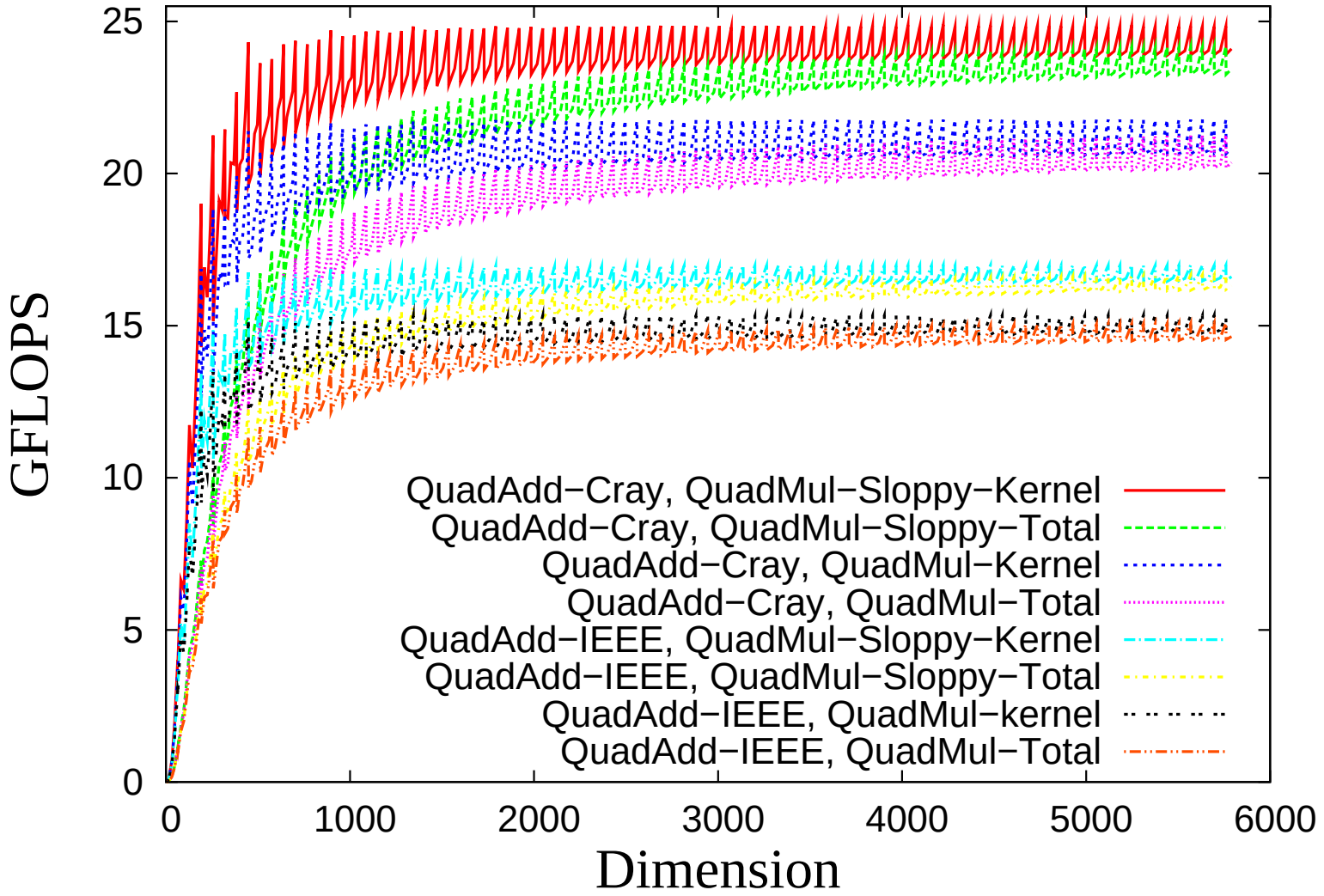
GPU 上での実装と評価

ECC On/Off **の結果。**Off **にしてもパフォーマンスの向上は無かった**



GPU 上での実装と評価

演算を精度を落として高速化し, 最大で 24.9GFlops **得た.**



GPU 上での実装と評価

演算を精度を落として高速化した場合の最大性能。CPU **は** Xeon 3470 + DDR3-1066

アルゴリズム	**パフォーマンス**
QuadAdd-Cray, QuadMul-Sloppy **カーネル**	24.9GFlops
QuadAdd-Cray, QuadMul-Sloppy **トータル**	24.2GFlops
QuadAdd-Cray, QuadMul **カーネル**	21.8GFlops
QuadAdd-Cray, QuadMul **トータル**	21.3GFlops
QuadAdd-IEEE, QuadMul-Sloppy **カーネル**	17.0GFlops
QuadAdd-IEEE, QuadMul-Sloppy **トータル**	16.7GFlops
QuadAdd-IEEE, QuadMul **カーネル**	15.3GFlops
QuadAdd-IEEE, QuadMul **トータル**	15.1GFlops
QuadAdd-IEEE, QuadMul CPU	100MFlops
QuadAdd-IEEE, QuadMul OpenMP CPU	400MFlops

GPU 上での実装と評価

- 15.1GFlops = 85% (or 43%) **の性能** (QuadAdd-IEEE, QuadMul-FMA)
- **平均** Clock **の計算**:QuadAdd-IEEE 20 **演算**, QuadMul-FMA 10 **演算**, Rgemm **では和積が半分ずつ**

$$(20 + 10 - 1)/2 = 14.5$$

- **理論性能値は**...

$$515\text{GFlops}/14.5 = 35.5\text{GFlops}$$

- **最大性能の見積りは大雑把に**...nVidia C2050 **だと** FMA **をフルに使って** 515GFlops **なので、**

$$515\text{GFlops}/14.5/2 = 17.75\text{GFlops}$$

となる。

応用計算:半正定値計画問題ソルバ “SDPA-DD” の 10 倍の加速

Eat your own dog food (**自作自演**) :-)

応用計算:半正定値計画問題ソルバ “SDPA-DD” の 10 倍の加速

半正定値計画問題 (SDP) **とは**

主問題　最小化:

$$A_0 \bullet X$$

s.t.:

$$A_i \bullet X = b_i \qquad (i = 1, 2, \cdots, m)$$

$$X \succeq 0$$

双対問題　最大化:

$$\sum_{i=1}^{m} b_i z_i$$

s.t.:

$$\sum_{i=1}^{m} A_i z_i + Y = A_0$$

$$Y \succeq 0$$

A_i は $n \times n$ 実対称行列、X $n \times n$ は実対称の変数行列、b_i は m-次元定数ベクトル、Y は $n \times n$ 実対称の変数行列、$X \bullet Y := \sum X_{ij} Y_{ij}$.
$X \succeq 0$ は X が半正定値、つまり固有値がゼロ以上

応用計算:半正定値計画問題ソルバ “SDPA-DD” の 10 倍の加速

最適解の性質と最適解付近で誤差が溜まりやすいことについて

Theorem (相補性定理)

(X^*, Y^*, z^*) の組が内点実行可能解とする。従って SDP の主問題、双対問題の条件を満たすとき、(X^*, Y^*, z^*) が最適解である必要十分条件は

$$X^* \bullet Y^* = 0$$

である。

応用計算:半正定値計画問題ソルバ “SDPA-DD” の 10 倍の加速

最適解で解が特異的になる:X^*, Y^* **が最適解のとき**,

$$X^* \bullet Y^* = 0$$

が成立する。そして線形代数の定理から

$$\mathrm{rank}X^* + \mathrm{rank}Y^* \leq n \qquad (1)$$

つまり

X^*, Y^* はどちらかが少なくとも特異的

大抵は X^*, Y^* 両方とも特異的 → 数値計算誤差がたまりやすい。

応用計算:半正定値計画問題ソルバ “SDPA-DD” の 10 倍の加速

SDPLIB から大きないくつか問題を解いたときのベンチマーク
CPU: Xeon 3470, DDR3 -1066

問題名	CPU(秒)	GPU(秒)	加速率
equalG51	6531.9	573.2	11.4
gpp500-1	902.0	72.2	12.5
gpp500-4	638.0	74.8	8.5
maxG32	36284.4	4373.1	8.3
maxG55	521575.4	53413.1	9.8
mcp500-4	539.1	65.2	8.3
qpG11	16114.7	1408.0	11.4
qpG51	39678.9	3299.2	12.0
ss30	310.7	138.6	2.2
theta5	3250.0	239.8	13.6
theta6	9028.2	623.6	14.5
thetaG51	49161.5	4870.4	10.1

まとめと今後の展望

- 高精度計算、行列-行列積は科学技術にとって重要。
- 倍々精度で手軽に高精度化 (32 桁、倍精度は 16 桁なのでその倍の精度)。
- nVidia C2050 GPU を使うと Xeon 3470 での参照実装の 150 倍高速化が達成された。
  - 性能は 15GFlops から (精度落として)24GFlops 理論性能値比 85% (or 43%)。Note:Core i7 920 倍精度は 43GFlops !!倍々精度は十分高速!!
- Rgemm の完全実装を目指した。Nath らの Pointer redirecting 手法で、パフォーマンスを 6%程度しか落とさず、一般の次元に対応。転置の組み合わせにも対応。パフォーマンスは非転置と同じ。OS などによる性能のぶれは 0.1%程度と非常に安定した実装。
- 倍々精度半正定値計画法;SDPA-DD に組み込んだ。SDPLIB の大きめの問題のパフォーマンスは 2.2 倍から 14.5 倍。
- 他の BLAS 関数、LAPACK の関数も作成したい。
- !!MPACK 次期バージョンで公開予定!!

謝辞

この研究の一部は科学研究費補助金 基盤研究 (B)(一般) 21300017「マルチプラットフォームの大規模数値シミュレーションを支援するフレームワークの構築」および「第 6 回 マイクロソフト産学連携研究機構 研究プロジェクト」によってサポートされた.