線形代数演算ライブラリ
BLASとLAPACKの基礎と実践2

理化学研究所 情報基盤センター
2013/5/30 13:00- 大阪大学 基礎工学部

中田真秀

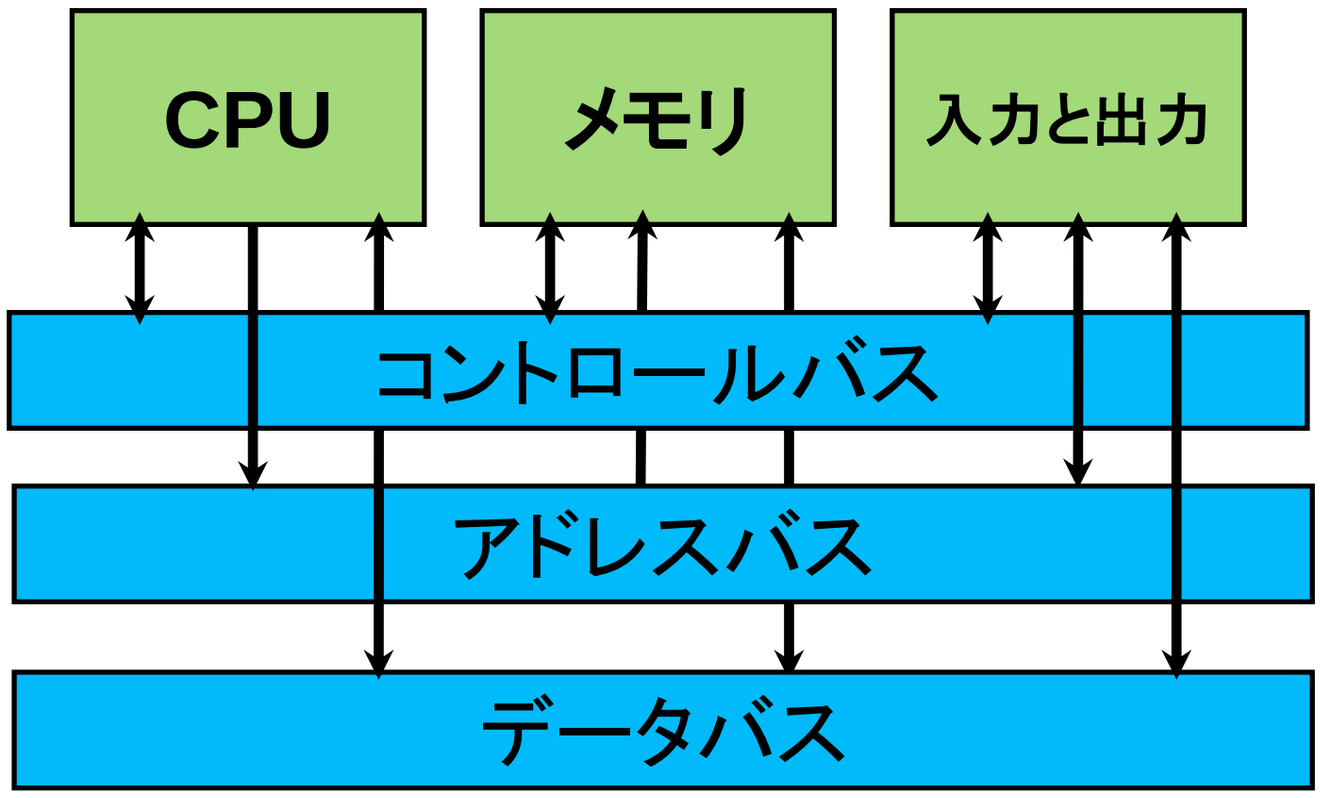
この授業の目的

- 対象者
  - 研究用プログラムを高速化したい人。
  - LAPACKについてよく知らない人。
- この講習会の目的
  - コンピュータの簡単な仕組みについて。
  - 今後、どうやってプログラムを高速化するか。
  - BLAS, LAPACKを高速なものに変えられるようにすること。

コンピュータの簡単な仕組みについて

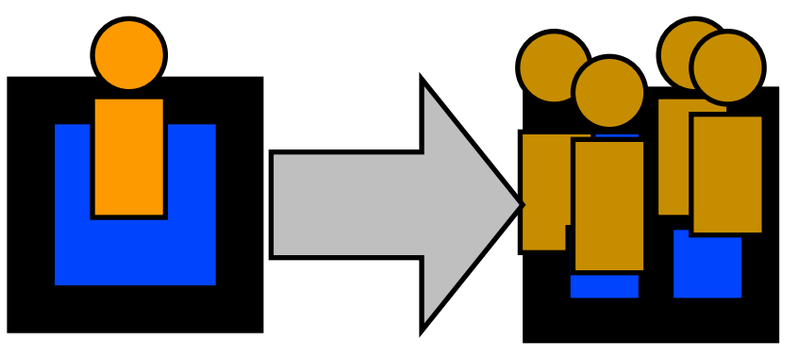
コンピュータの簡単な仕組み

- コンピュータを最も簡単にあらわすと右図のようになる(ノイマン型コンピュータ)
- CPUが高速というのは、この図では論理演算装置が高速ということ。
- フォン・ノイマンボトルネック
  - バスのスピードが(CPU-メモリの転送速度など)がスピードのボトルネックになることがある。
- CPU,メモリ,入出力が高速=必ずしもコンピュータが高速ではない。
- プログラムの高速化にはボトルネックがどこかを見極める必要あり。

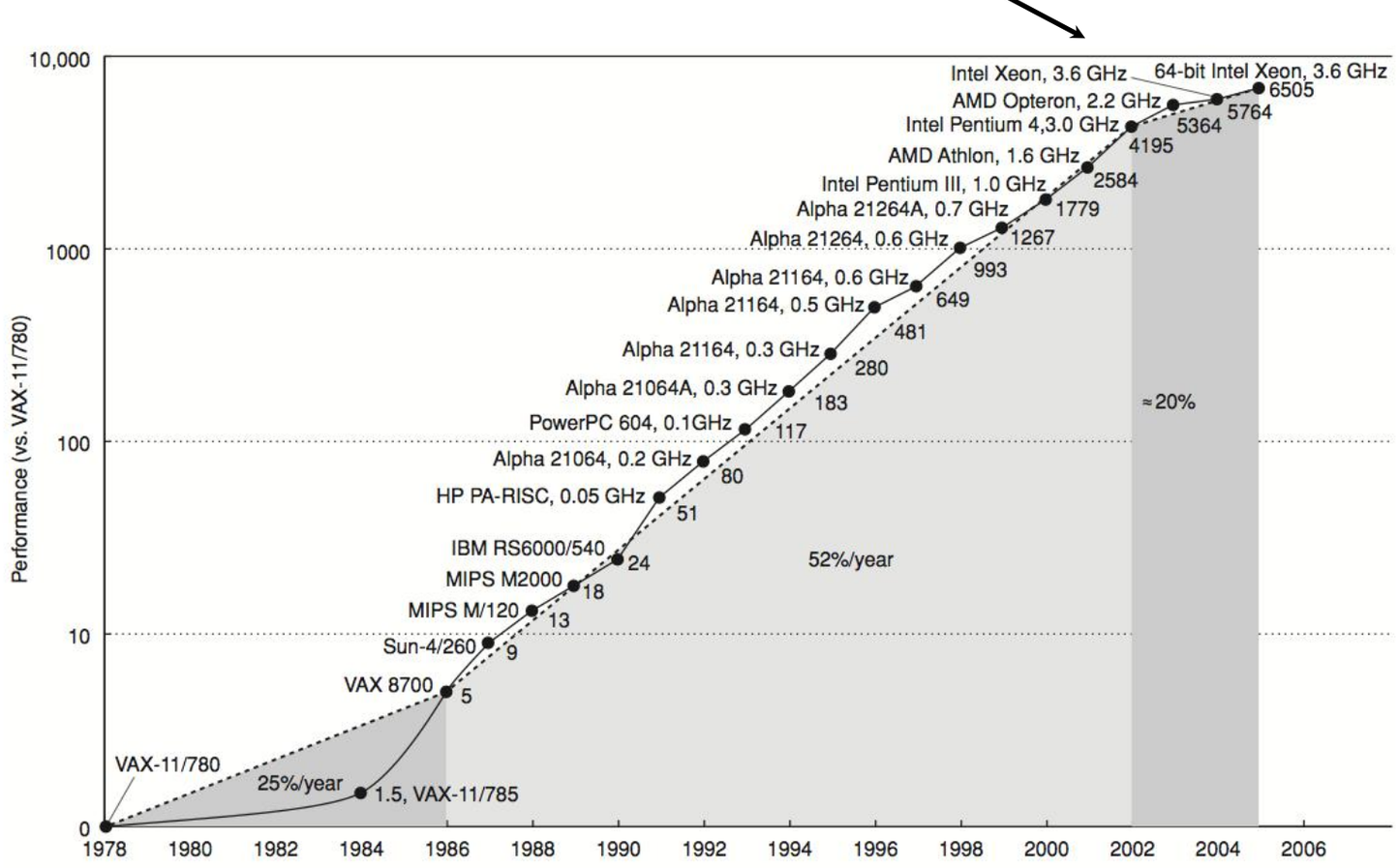


CPUのスピードについて

- コンピュータは年々高速になってきている。
- ただ、コア一個単位処理能力は落ちてきており、**2000年からマルチコア化**をしてきている。
  - 様々な物理的な限界
- マルチコアとは、
  - 下図のように、コアの処理能力を上げるのではなく、**いくつもコアを用意する**ことで、処理能力をあげている。

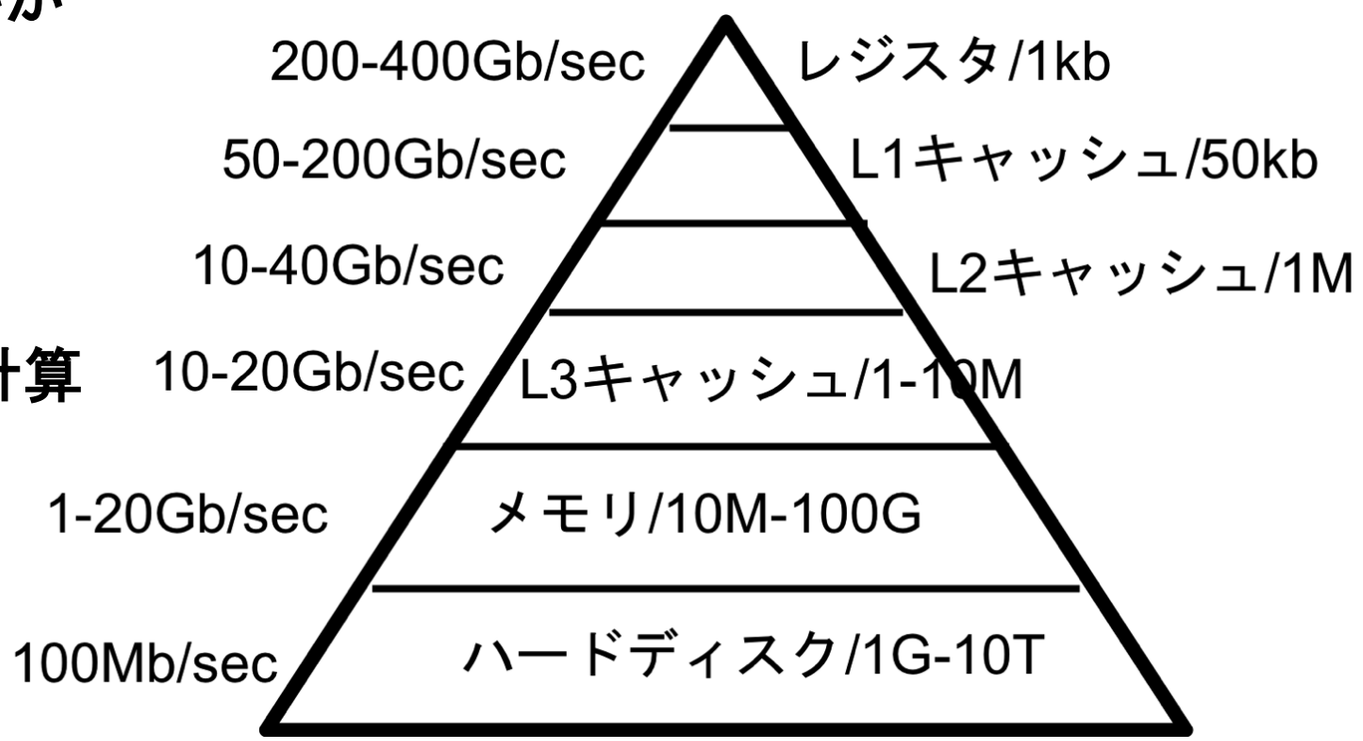


マルチコア化



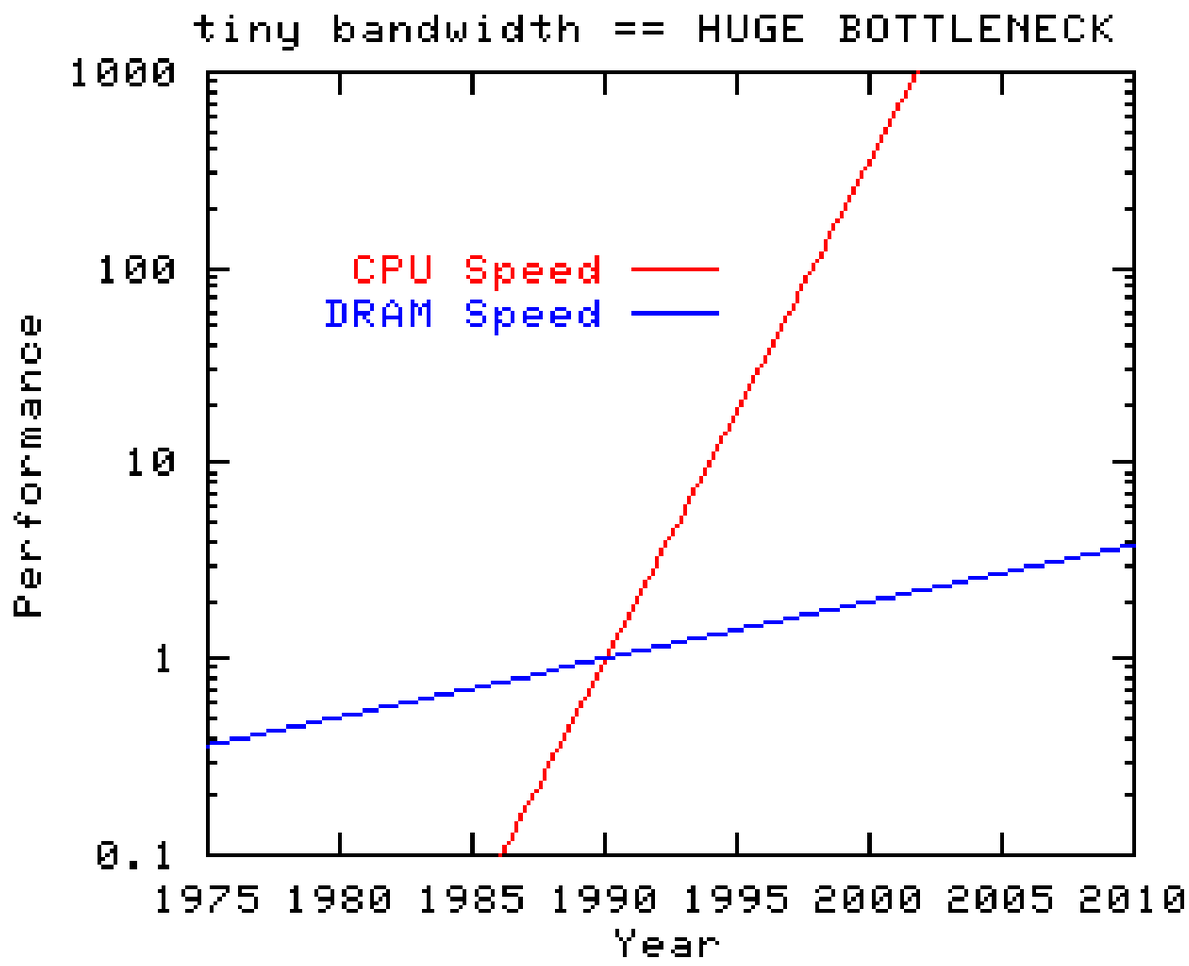
メモリ(記憶装置)のスピードについて

メモリ(記憶装置)にも幾つか種類がある。
アクセススピードが速い=コスト高、容量小
アクセススピードが遅い=コスト安、容量大
一桁容量が大きくなると、一桁遅くなる
一桁容量が小さくなると、一桁速くなる
レイテンシ:アクセスする時間
データを取ってくる命令を出してから、帰ってくるまでのことをレイテンシ(latency)。
データを一個だけ取ってくる、これは時間がかかる。
高速化するには:
アクセススピードを意識しよう。
データの移動を少なくしよう。
一度にデータを転送し、転送している間に計算をしよう(=レイテンシを隠す)
メモリバンド幅が大きい、小さいと表現



CPUとメモリのスピード比の変化

- CPUとメモリのパフォーマンス(=スピード)を年によってプロットしてみる
- 1990年くらいまでは、メモリーのスピードのほうが速く、CPUが遅かった。
  - なるべくCPUに計算させないプログラムが高速だった。
- 1990年以降、メモリよりCPUが高速
  - メモリの転送を抑えて、無駄でも計算させた方が高速。
  - このトレンドは変わらないと言われている。



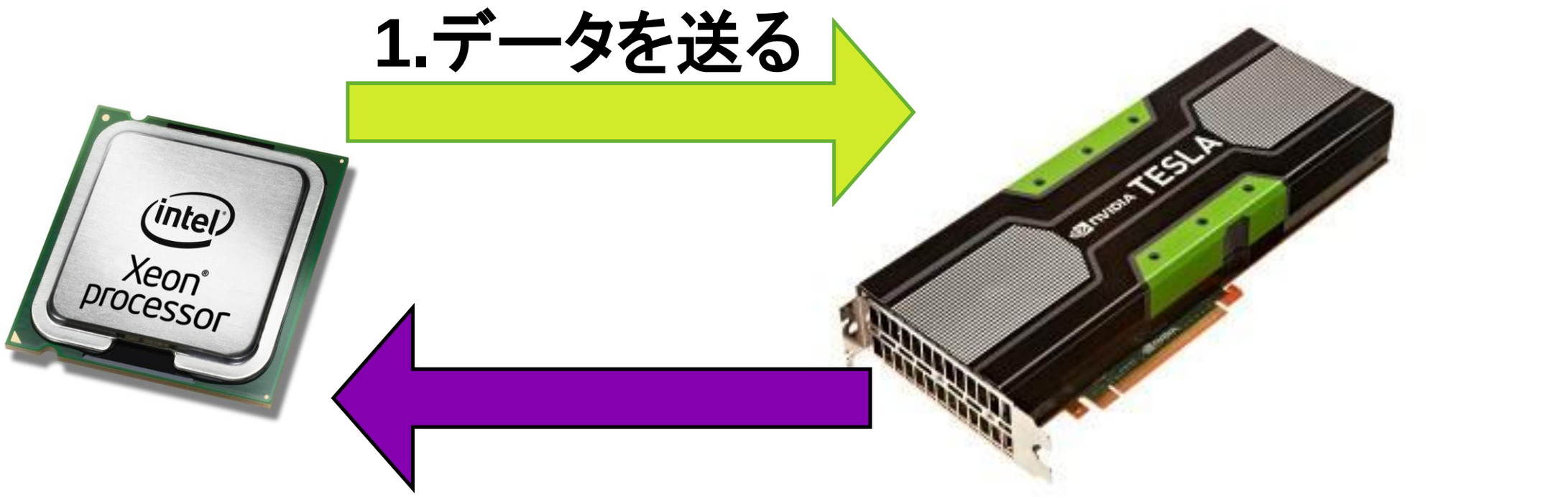
GPUについての紹介

GPUとは？GPGPUとは？

- GPUとは?
  - Graphics Processing Unit (グラフィックス処理器)のこと。
  - 本来、画像処理を担当する主要な部品
    - 例:3Dゲーム、ムービー、GUIなどの処理を高速に行える
    - 2006年からは科学計算にも使われるようになってきた。
- GPGPUとは?
  - General-Purpose computing on Graphics Processing Units
  - GPUによる、汎用目的計算
    - 画像処理でなくて科学技術計算することは
    - GPGPUといえる。
- 現在はPCI expressにつなげる形で存在。
  - バスがボトルネック
  - 将来はCPU/GPUが共有される?

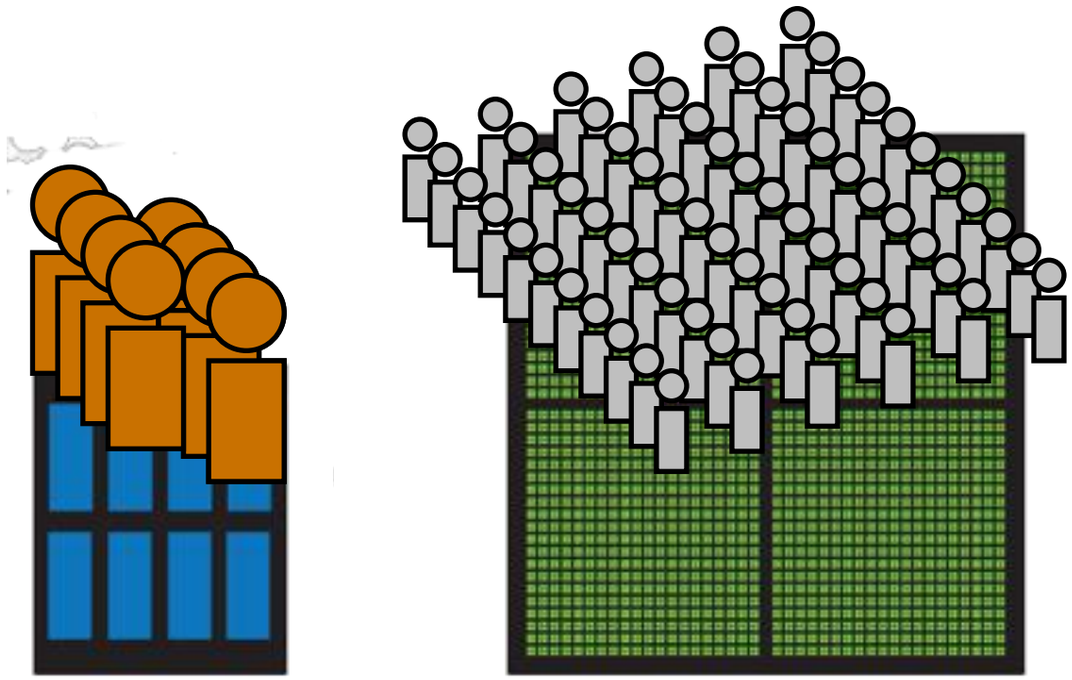
GPUの使い方

- CPUからデータを送り、GPUで計算させて、計算結果を回収
  - なるべくデータ転送を少なくした方が良い。
  - メモリは共有されない。



GPUはどうして高速か? Part I

・CPUと比べると1コ1コの処理能力は低いが、ものすごい数のコアがあって、似たような**処理を同時に沢山行えるので高速**。

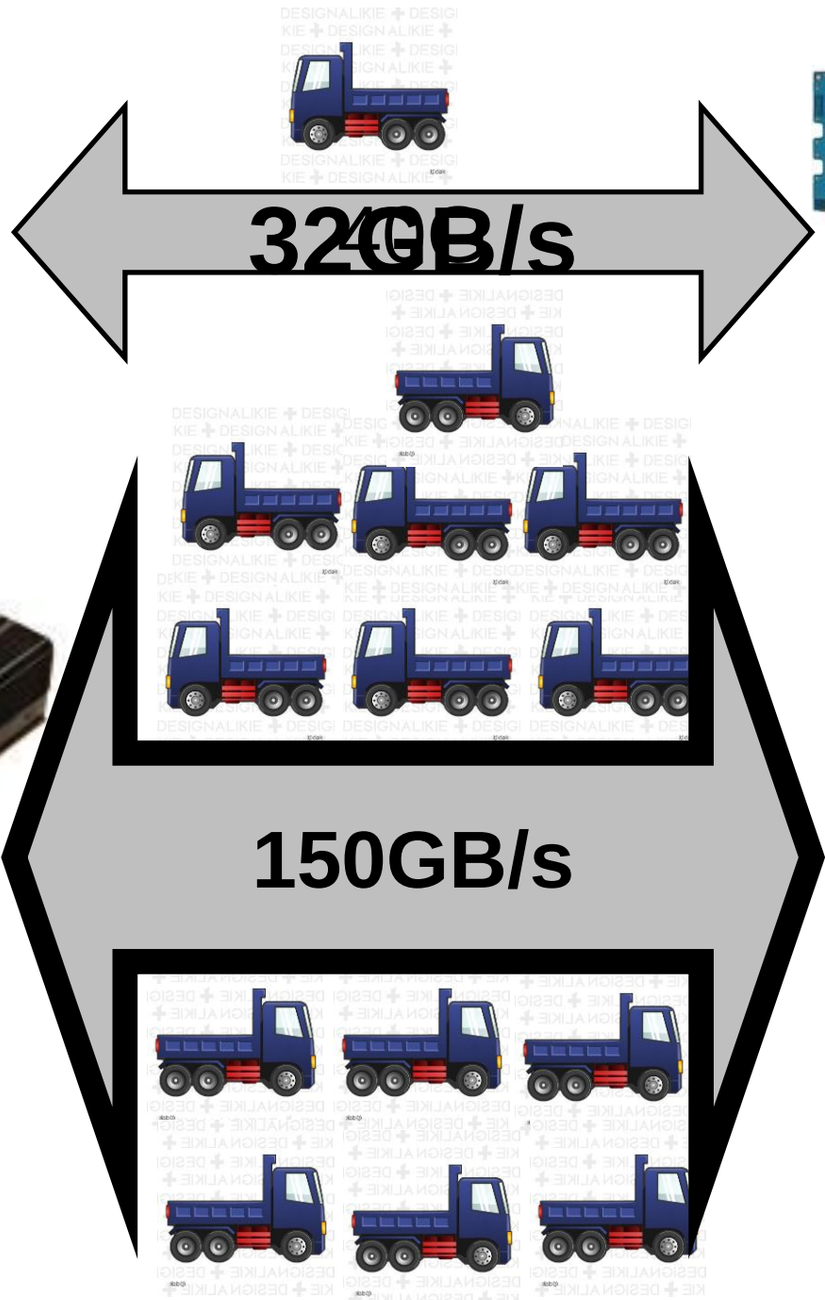


CPU　　GPU

・画像処理だと沢山独立した点に対して似たような処理をする
・CPUみたいには複雑な処理はできないが、工夫次第で色々可能

GPUはどうして高速か? Part II

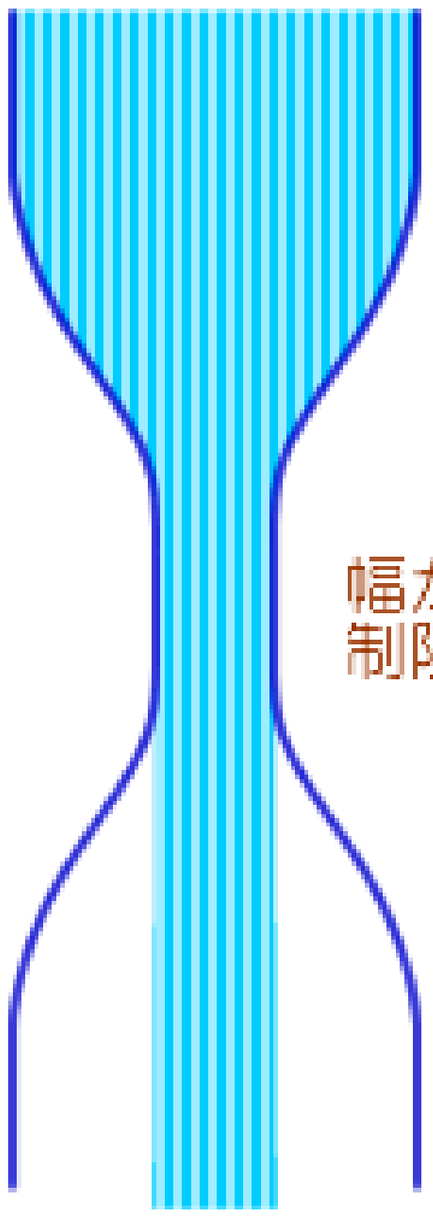
メモリバンド幅がGPUのほうが大きい



プログラムを速くするには?

プログラムを高速化する一般的な手法

ボトルネックを見極めよう

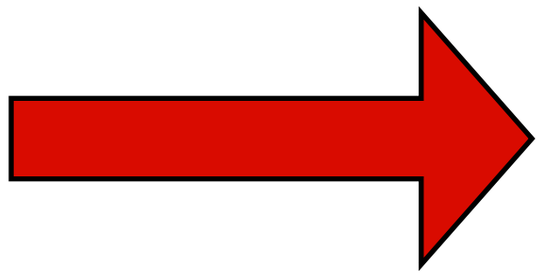


プログラムを高速化する一般的な手法

- ノイマン型のコンピュータのボトルネック
  - 演算量? バス?
- データ転送量＜演算量の場合
  - データの使い回しを行なうことで基本的には高速化する。
  - また、CPUが高速であればあるほど、高速化する。
  - 例:行列-行列積
  - 高速化はしやすい。
- データ転送量>=演算量の場合
  - メモリーCPUの転送が高速であればあるほど、高速化する。
  - 例:行列-ベクトル積
  - データの使い回しができないため、高速化は面倒。
    - メモリとCPUを比較して高速化の度合いがメモリは小さく、差も広がっている
    - 高速なメモリを搭載しているマシンが少ない。

今後、並列化プログラミングは必須になる

- CPUのコア周波数は2002年には3GHzを超えた。それ以降は横ばい
- CPUはスピードを上げるため、SIMDやマルチコア化した。
  - SIMD:1個の命令で多数の処理を行うこと。
  - マルチコア:CPUを2～8個程度一つのパッケージに詰める。
    - 1個のコアのスピード上げるのはもう限界。



並列処理が必須に

高速なBLAS LAPACKを使うには

高速なBLAS、LAPACKを使う

- コンピュータのパフォーマンスの計り方
  - ボトルネック
- DGEMM (行列-行列積), DGEMV (行列-ベクトル積)

  - 二つの典型的な例 CPU演算、メモリバンド幅がボトルネック

- 高速なBLAS, LAPACK: GotoBLAS2
- Octaveで試してみる。
- どうして高速なのか?
- GPUでcuBLASを使う

FLOPS：マシンの性能の計り方のひとつ

- FLOPS：マシンの性能の計り方のひとつ

- Floating point operations per second
- 一秒間に何回浮動小数点演算ができるか。
- カタログ値ではこのピーク性能をFLOPSで出すことが多い
  - ただし、その通りの値は実際の計算では出ない。

- 中田は多分0.0001Flops程度
  - 倍精度の計算は間違うかも。
  - そろばんやってる人は、0.1Flopsくらいあるかもしれない。

- GPUは1TFlops = 1,000,000,000,000Flpps

- 京コンピュータは10PFlops
  - 10,000,000,000,000,000 Flops

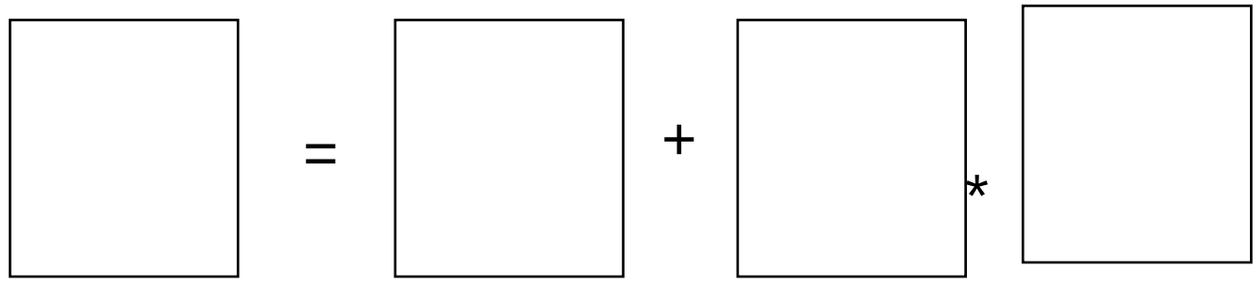
Bytes per FLOPS

- **Bytes per FLOPS:**
  - 一回の浮動小数点演算を行う際に必要なメモリアクセス量をByte/Flopで定義する。
    - (違う定義:1回の浮動小数点計算に何bytesメモリにアクセスできるか)
- たとえばdaxpyを例にとるx[i], y[i]はベクトル、a はスカラー
  y[i] ← y[i] + a x[i]

- 2n回の浮動小数点演算
- 3n回のデータの読み書きが必要
  - x[i], y[i]を読んで、y[i]に書く。
- 倍精度一つで、8bytesなので、24bytes / 2 Flops = 12 bytes/ flops.
- 小さければ小さいほど高速に処理できる。

DGEMM 行列-行列積

- マシンのパワーをみるには、DGEMM (行列-行列積)と、DGEMV (行列ベクトル積)をみればよい。
- DGEMM (行列-行列積)
  - CPUのパワーがどの程度あるかの良い目安。
  - C←αAB+βC

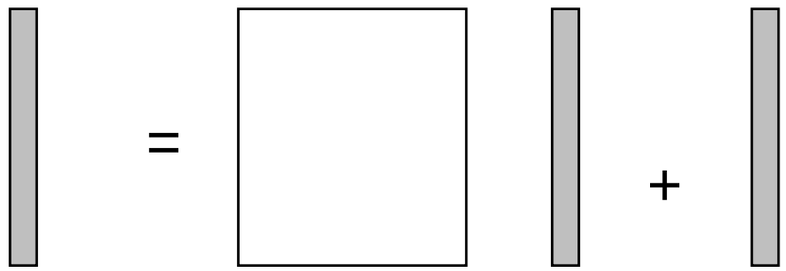


- データの量はO(n^2)
- 演算量はO(n^3)
- 大きなサイズのDGEMMは演算スピードが律速となる。
- Bytes / flop は O(1/n)なので、大きな次元でほぼゼロとなる。

DGEMV：行列ベクトル積

DGEMV (行列ベクトル積)

- メモリバンド幅がどの程度あるかの良い目安。
- y←αAx + βy



- データの量はO(n^2)
- 演算量はO(n^2)
- Bytes / flop は O(1)
  - 大きなサイズのDGEMVはメモリバンド幅が律速となる。
  - メモリバンド幅が大きいと高速になる。
  - CPUだけが高速でもDGEMVは高速にならない。

高速なBLAS、LAPACKの力を知る

- 行列-行列積DGEMM, DGEMVを試し、違いを見る。
  - Reference BLAS
    - http://netlib.org/blasをそのままコンパイルしたもの
  - Ubuntu 標準 ATLAS,
  - 自分でビルドした ATLAS バージョン 3.9.36
  - GotoBLAS2
- Octave
  - Matlabのフリーのクローン
  - かなり使える
  - 内部でBLAS, LAPACKを呼ぶ
- 使ったマシン
  - Intel Core i7 920 (2.66GHz, 理論性能値 42.56GFlops)

環境を整える

- Ubuntu 10.04 x86(Lucid Lynx) を使ってお試し。
  - 開発環境設定
  - 端末から
  - $ sudo apt-get install patch gfortran g++ libblas-dev octave3.2

- GotoBLAS2のインストール
  - http://www.tacc.utexas.edu/tacc-projects/gotoblas2/
  - $ cd ; cp <somewhere>/GotoBLAS2-1.13_bsd.tar.gz .
  - $ tar xvfz GotoBLAS2-1.13_bsd.tar.gz
  - $ cd GotoBLAS2
  - $ ./quickbuild.64bit
  - ln -fs libgoto2_nehalemp-r1.13.so libgoto2.so
  - …
  - GotoBLAS build complete.

Reference BLASのDGEMM

- Reference BLASの場合の設定
  $ LD_PRELOAD=/usr/lib/libblas.so:/usr/lib/liblapack.so; export LD_PRELOAD
- Octaveで行列-行列積
- 4000x4000の正方行列の積。値はランダム
  -$ octave
  ... 途中略...
  octave:1> n=4000; A=rand(n); B=rand(n);
  octave:2> tic(); C=A*B; t=toc(); GFLOPS=2*n^3/t*1e-9 GFLOPS
  = 1.6865
- 1.6865GFLops =>理論性能値のたった4%

Ubuntu標準ATLASの行列-行列積

- Ubuntu付属 ATLASの場合の設定

  $ LD_PRELOAD=/usr/lib/atlas/libblas.so; export LD_PRELOAD

- Octaveで行列-行列積
- 4000x4000の正方行列の積。値はランダム
- $ octave
  ... 途中略...
- octave:1> n=4000; A=rand(n); B=rand(n);
- octave:2> tic(); C=A*B; t=toc(); GFLOPS=2*n^3/t*1e-9 GFLOPS
  = 7.0291
- 7.0GFLops =>理論性能値のたった16.5%
  - 違うマシンで最適化(多くのマシンで使えるように)
  - マルチコアは使わない
    - 使ったとすると66%程度とソコソコでるはず。

ATLASの行列-行列積

- 自分でビルドしなおしたATLASの場合の設定例

  $ LD_PRELOAD=/home/maho/atlas/libblas.so; export LD_PRELOAD

- Octaveで行列-行列積
- Octave1:> n=4000; A=rand(n); B=rand(n);
- octave:2> tic(); C=A*B; t=toc(); GFLOPS=2*n^3/t*1e-9 GFLOPS
  = 32.824

- 32.8GFLops =>理論性能値の77.1%!!
  - オートチューニングは大変使える。
    - 開発コストは低い
  - 但し、マシン毎にビルドし直さなくてはならない。
    - そもそもそういうもの。
    - Linuxのディストリビューションとは相性が悪い

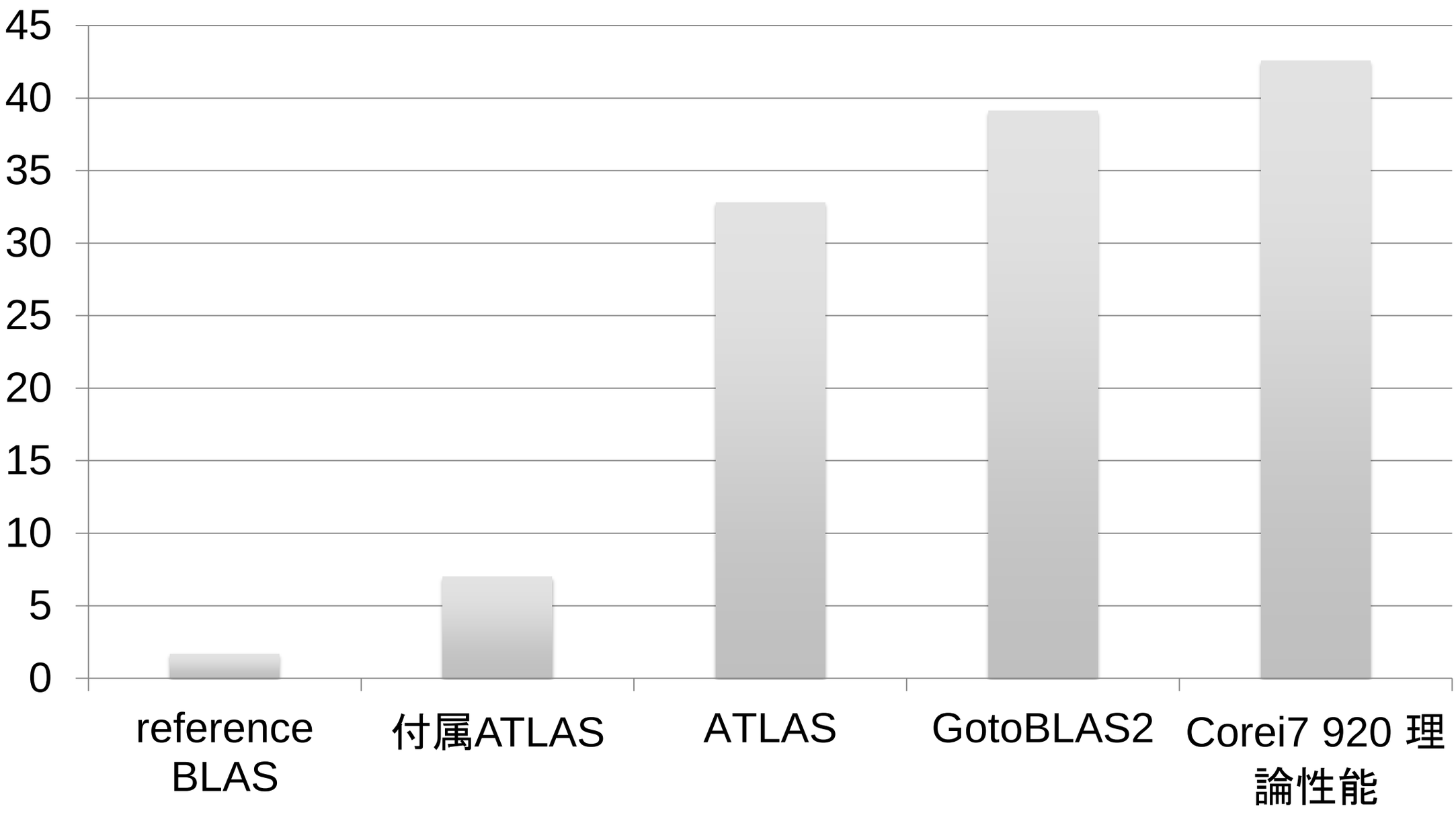
GotoBLAS2の行列-行列積

- GotoBLAS2の場合の設定例

  $ LD_PRELOAD=/home/maho/GotoBLAS2/libgoto2.so ; export LD_PRELOAD
  $ octave
- Octaveで行列-行列積
- Octave1:> n=4000; A=rand(n); B=rand(n);
- octave:2> tic(); C=A*B; t=toc(); GFLOPS=2*n^3/t*1e-9 GFLOPS = 39.068
- 39.068 GFLops =>理論性能値の91.2%
  - 一番高速。ただし、開発コストが非常に高い。
  - 開発は終了(OpenBLASに引き継がれた)
  - 標準になるまでには時間がかかる。
    - オープンソースになって日が浅いため。

高速なBLAS、LAPACKの力を知る:

Core i7 920の理論性能値は
4 FLOPS/Clock × 2.66GHz × 4コア＝42.56GFlops



DGEMVを使う:メモリバンド幅理論性能値

- Core i7 920マシンメモリバンド幅の理論性能値
  - という言い方は変だが…
  - DDR3-1066（PC3-8500）
  - メモリ帯域は25.6GB/s (トリプルチャネル)
  - http://www.elpida.com/pdfs/J1503E10.pdf
    - 133MHz x 4 (外部クロック) x 8 (I/O buffer読み込み) x 2 (8bit per ½ clock) / 8 (1bytes=8bit) * 8 (I/F data幅) = 8.53GB/sec (=1066 Mbps)
    - 8.53 x 3 (triple channel)= 25.6GB/s
- 理論性能値
  - 25.6 / 8 (bytes/1倍精度)= 3.19GFlops
  - これはCPUの速度に比べて10倍以上遅い。
  - メモリバンド幅は今後上がる見込みが少ない

DGEMVを使う:GotoBLAS2の例

- GotoBLAS2の場合の設定例

```
$ LD_PRELOAD=/home/maho/GotoBLAS2/libgoto2.so ;
export LD_PRELOAD
```

- Octaveで行列-ベクトル積

```
$ octave
Octave1:> n=10000; A=rand(n); y = rand(n,1) ; x =
 rand(n,1) ; tic(); y=A*x; t=toc(); GFLOPS=2*n^2/t*1e-9
GFLOPS =  3.0768
```

- 3.08GFlops :ピーク性能に近い値。

DGEMVを使う:付属ATLASの例

- **Ubuntu付属**ATLASの場合の設定例

```
$ LD_PRELOAD=/usr/lib/atlas-base/libatlas.so ; export LD_PRELOAD
```

- Octaveで行列-ベクトル積

```
octave:1> n=10000; A=rand(n); y = rand(n,1) ; x = rand(n,1) ; tic(); y=A*x; t=toc(); GFLOPS=2*n^2/t*1e-9
GFLOPS =  1.533
```

だいたい半分くらいでた。

DGEMVを使う:ATLASの例

- ATLASの場合の設定例

```
$ LD_PRELOAD=/home/maho/atlas/libatlas.so ; export LD_PRELOAD
```

- Octaveで行列-ベクトル積

```
octave:1> n=10000; A=rand(n); y = rand(n,1) ; x = rand(n,1) ; tic(); y=A*x; t=toc(); GFLOPS=2*n^2/t*1e-9
GFLOPS =  1.533
```

ほぼ標準のものと同じ

DGEMVを使う:Reference BLASの例

- Reference BLASの場合の設定例

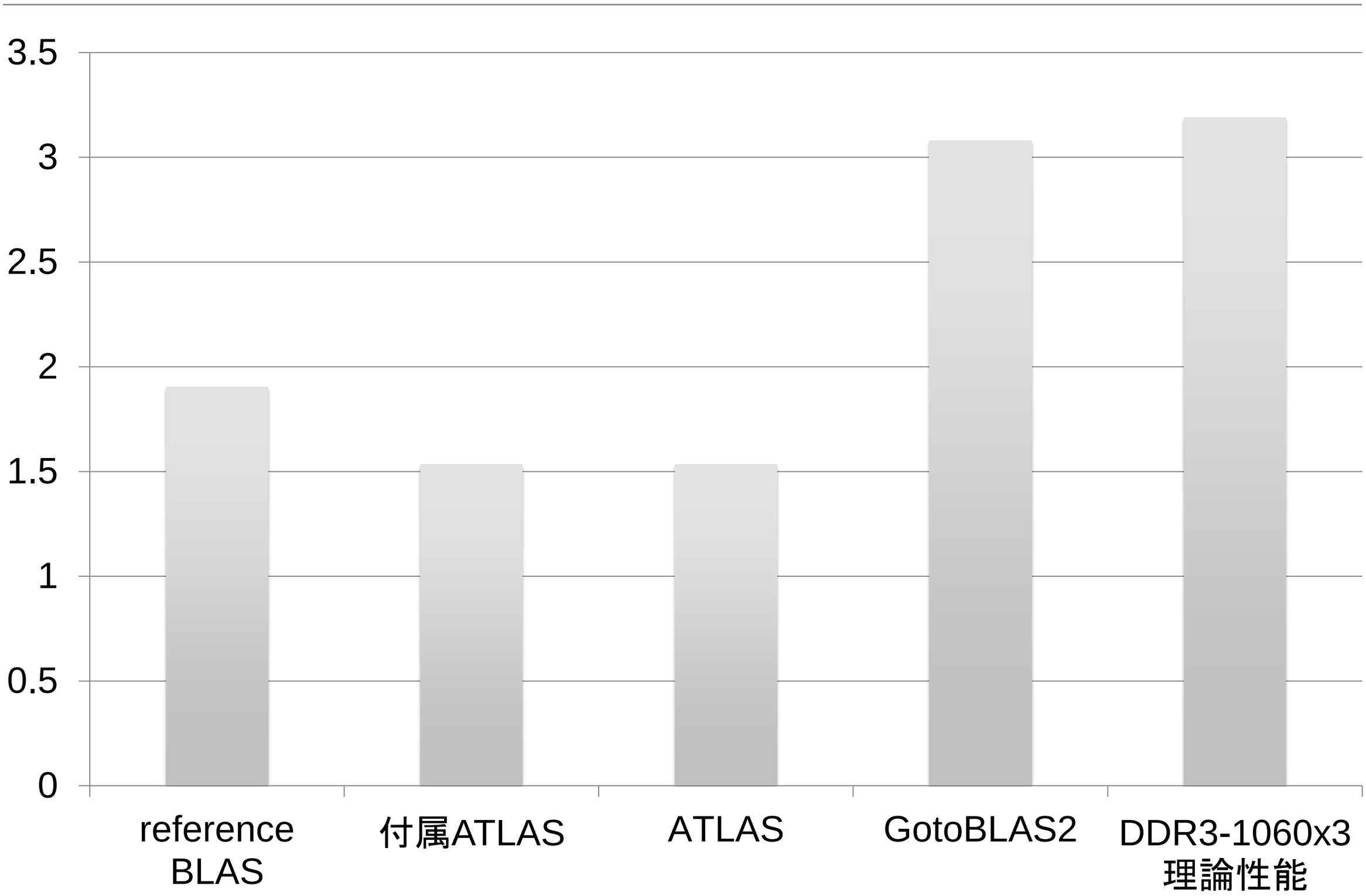
```
$ LD_PRELOAD=/usr/lib/libblas/libblas.so.3gf.0 ; export LD_PRELOAD
```

- Octaveで行列-ベクトル積

```
octave:1> n=10000; A=rand(n); y = rand(n,1) ; x = rand(n,1) ; tic(); y=A*x; t=toc(); GFLOPS=2*n^2/t*1e-9
GFLOPS =  1.9027
```

ATLASよりよい?

DGEMVを使う:表


3.5
3
2.5
2
1.5
1
0.5
0
reference BLAS
付属ATLAS
ATLAS
GotoBLAS2
DDR3-1060x3 理論性能

ここまでのまとめ

- コンピュータの仕組みを述べた
  - フォンノイマン図
- 高速化するにはどうしたらよいか
  - ボトルネックを探す。
  - 言葉: flops, byte per flops
  - メモリバンド幅、CPU性能値が重要なボトルネック

- 最適化されたBLAS (GotoBLAS2)を使ったベンチマークを行った。
- DGEMM (行列-行列積)で、CPUの理論性能と比較
- DGEMV (行列-ベクトル積)で、メモリバンド幅の理論性能と比較
- 大きな次元での結果であることに注意

高速化の手法

レジスタとアンローリング

- 8x8行列の積c=a*bを考える
- i,jループの2段のアンローリングを行う
- a[i][k],a[i+1][k],b[k][j],b[k][j+1]が2回づつ現われるので、
  レジスタの再利用が可能になり、メモリからのロードを減らせる

```
for(i=0;i<8;i++){
  for(j=0;j<8;j++){
    for(k=0;k<8;k++){
      c[i][j]+=a[i][k]*b[k][
}}}
```

通常の行列積の演算
cの要素1つの計算に、
aとbの要素が1つずつ必要

```
for(i=0;i<8;i+=2){
  for(j=0;j<8;j+=2){
    for(k=0;k<8;k++){
      c[i][j] += a[i][k]*b[k][j]
      c[i+1][j] += a[i+1][k]*b[k][j]
      c[i][j+1] += a[i][k]*b[k][j+1]
      c[i+1][j+1] += a[i+1][k]*b[k][j+1]
}}}
```

2段アンローリングを行った行列積の演算
cの要素4つの計算に、aとbの要素が2つずつ必要

キャッシュ

- キャッシュメモリではキャッシュラインの単位でデータを管理
- キャッシュラインのデータ置き換えは、Least Recently Used(LRU)方式が多い
- ダイレクトマッピング方式であるとすると、キャッシュラインを4とすると、メインメモリのデータは4毎に同じキャッシュラインに乗る
- 配列が2のベキ乗の場合は、キャッシュライン衝突、バンクコンフリクトの可能性。パティングにより回避
- 隣り合ったキャッシュラインに、隣り合ったメインメモリのデータを持ってくるメモリインタリービング機能

ブロック行列化

- キャッシュラインが4つあり、各キャッシュラインに4変数格納出来るとする
- キャッシュラインの置き換えアルゴリズムはLRUとする
- 2行2列のブロック行列に分けて計算する

```
for(i=0;i<8;i++){
  for(j=0;j<8;j++){
    for(k=0;k<8;k++){
      c[i][j]+=a[i][k]*b[k][j]
}}}
```

```
for(ib=0;ib<8;ib+=2){
  for(jb=0;jb<8;jb+=2){
    for(kb=0;kb<8;kb+=2){
      for(i=ib;i<ib+2;i++){
        for(j=jb;j<jb+2;j++){
          for(k=kb;k<kb+2;k++){
            c[i][j]+=a[i][k]*b[k][j]
}}}}}}
```

ブロック行列化2

- c[0][0],c[0][1],c[1][0],c[1][1]の計算の際のキャッシュミス回数を数える

```
c[0][0] += a[0][0] * b[0][0]
         + a[0][1] * b[1][0]
         + a[0][2] * b[2][0]
         + a[0][3] * b[3][0]
         + a[0][4] * b[4][0]
         + a[0][5] * b[5][0]
         + a[0][6] * b[6][0]
         + a[0][7] * b[7][0]
```

下線を引いた所でキャッシュミス
c[0][0]の計算で11回のキャッシュミス
4要素の計算で11x4=44回のキャッシュミス

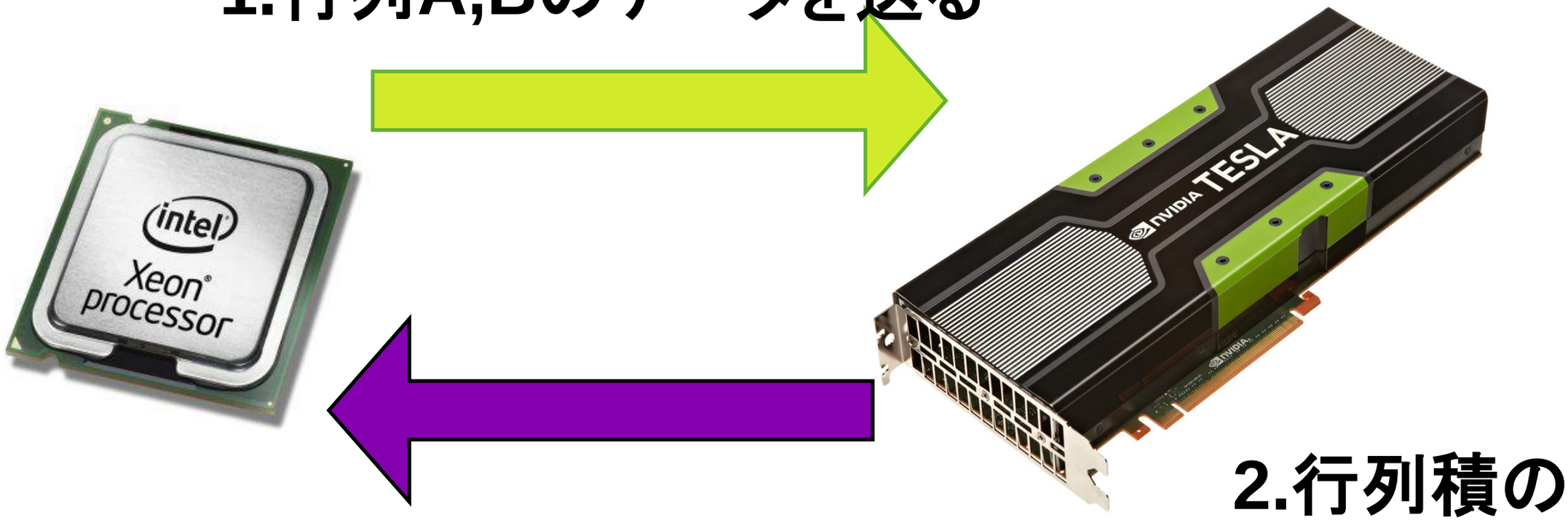
```
c[0][0] += a[0][0] * b[0][0]
         + a[0][1] * b[1][0]
c[0][1] += a[0][0] * b[0][1]
         + a[0][1] * b[1][1]
c[1][0] += a[1][0] * b[0][0]
         + a[1][1] * b[1][0]
c[1][1] += a[1][0] * b[0][1]
         + a[1][1] * b[1][1]
c[0][0] += a[0][2] * b[2][0]
         + a[0][3] * b[3][0]
...
c[1][1] += a[1][6] + b[6][1]
         + a[1][7] + b[7][1]
```

4要素の計算で20回のキャッシュミス

cuBLAS＋行列-行列積編

詳しいことは抜きにして、ライブラリを叩くのみ

- cuBLASを用いて行列-行列積を実行してみよう。
  - A,B,Cをn x n の行列として、C=ABを計算してみよう。
- 走らせ方のイメージは下図のようになる。



GPUでのBLAS (cuBLAS)

GPUの使い方:GPUの弱点

- PCIeというバスでつながっているが、この転送速度が遅い
  - GPUはPCは別のコンピュータ。
  - それをつないでいるのがPCIeバス。これが遅い!

フォン・ノイマンボトルネックの一種
他にもさまざまな制限がある。
- メモリのアクセスパターン
- スレッドの使い方

cuBLASとはなにか? (まずはBLAS)

- そのまえにBLASの復習
- BLAS (Basic Linear Algebra Subprograms) とは、基本的なベクトルや行列演算をおこなうとき基本となる「ビルディングブロック」ルーチンをあつめたもの。Level 1, 2, 3とあり、
  - Level1はスカラー、ベクトル、およびベクトル-ベクトル演算を行う。
  - Level2は行列-ベクトル演算を行う。
  - Level3は行列-行列演算を行う。
- 効率よく演算できるようになっていて、広く入手可能であるため、LAPACKなど高性能な線形代数演算ライブラリの構築に利用される。
- 高速な実装がある
  - Intel MKL (math kernel library)
  - GotoBLAS2
  - OpenBLAS
  - ATLAS
  - IBM ESSL

cuBLASとは何か

- cuBLASとは?
  - CUDAで書かれた、GPU向けに加速されたBLAS
  - ソースコードには少し変更が必要。ただし、CUDAはCPUとはアーキテクチャ(設計)かなり違うため、効率的に使うには「そのまま」ではなくソースコードの変更が必要。
  - 行列やベクトルをGPUに転送し、GPUが計算し、回収する、のような仕組みをとる.

cuBLASでの行列-行列積 (I)

- cuBLASのdgemmを行なってみる
  - 行列-行列積を行うルーチン
  - 具体的には
    - A, B, Cを行列とし、α, βをスカラーとして
    - C←αAB＋βC
    - を行う。
    - 他にもA,Bをそれぞれ転置するかしないかを選択できる(が今回はやらない)。
  - 今回試して見ること: 3x3の行列A,B,Cを下のようにして、
    - スカラは、α=3.0, β=-2.0とした。

$$A=\begin{pmatrix}1 & 8 & 3\\ 2 & 10 & 8\\ 9 & -5 & -1\end{pmatrix}\quad B=\begin{pmatrix}9 & 8 & 3\\ 3 & 11 & 2.3\\ -8 & 6 & 1\end{pmatrix}\quad C=\begin{pmatrix}3 & 3 & 1.2\\ 8 & 4 & 8\\ 6 & 1 & 2\end{pmatrix}$$

$$\alpha AB+\beta C=\begin{pmatrix}21 & 336 & 70.8\\ -64 & 514 & 95\\ 210 & 31 & 47.5\end{pmatrix}$$

cuBLASでの行列-行列積 (II)

```
// dgemm CUDA test public domain
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "cublas.h"

//Matlab/Octave format
void printmat(int N, int M, double *A, int LDA) {
  double mtmp;
  printf("[ ");
  for (int i = 0; i < N; i++) {
    printf("[ ");
    for (int j = 0; j < M; j++) {
      mtmp = A[i + j * LDA];
      printf("%5.2e", mtmp);
      if (j < M - 1) printf(", ");
    } if (i < N - 1) printf("]; ");
    else printf("] ");
  } printf("]");
}
int main()
{
  int n = 3; double alpha, beta;
  cublasStatus statA, statB, statC;
  double *devA, *devB, *devC;
  double *A = new double[n*n];
  double *B = new double[n*n];
  double *C = new double[n*n];
```

```
cublasInit();
statA = cublasAlloc (n*n, sizeof(*A), (void**)&devA);
statB = cublasAlloc (n*n, sizeof(*B), (void**)&devB);
statC = cublasAlloc (n*n, sizeof(*C), (void**)&devC);
if (statA != CUBLAS_STATUS_SUCCESS
 || statB != CUBLAS_STATUS_SUCCESS
 || statC != CUBLAS_STATUS_SUCCESS) {
  printf ("device memory allocation failed\n");
  cublasShutdown();
  return EXIT_FAILURE;
}

A[0+0*n]=1; A[0+1*n]= 8; A[0+2*n]= 3;
A[1+0*n]=2; A[1+1*n]=10; A[1+2*n]= 8;
A[2+0*n]=9; A[2+1*n]=-5; A[2+2*n]=-1;

B[0+0*n]= 9; B[0+1*n]= 8; B[0+2*n]=3;
B[1+0*n]= 3; B[1+1*n]=11; B[1+2*n]=2.3;
B[2+0*n]=-8; B[2+1*n]= 6; B[2+2*n]=1;

C[0+0*n]=3; C[0+1*n]=3; C[0+2*n]=1.2;
C[1+0*n]=8; C[1+1*n]=4; C[1+2*n]=8;
C[2+0*n]=6; C[2+1*n]=1; C[2+2*n]=-2;

statA = cublasSetMatrix (n, n, sizeof(*A), A, n, devA, n);
statB = cublasSetMatrix (n, n, sizeof(*B), B, n, devB, n);
statC = cublasSetMatrix (n, n, sizeof(*C), C, n, devC, n);
if (statA != CUBLAS_STATUS_SUCCESS
 || statB != CUBLAS_STATUS_SUCCESS
 || statC != CUBLAS_STATUS_SUCCESS) {
```

GPU側にメモリを確保する

行列のセット(ホスト側)

column majorな並びになっている

ことに注意!!

行列をGPUに

転送

cuBLASでの行列-行列積 (III)

```
  printf ("data download failed\n");
  cublasFree (devA); cublasFree (devB);
  cublasFree (devC);
  cublasShutdown();
  return EXIT_FAILURE;
}

printf("# dgemm demo...\n");
printf("A =");printmat(n,n,A,n);printf("\n");
printf("B =");printmat(n,n,B,n);printf("\n");
printf("C =");printmat(n,n,C,n);printf("\n");
alpha = 3.0; beta = -2.0;

cublasDgemm('n', 'n', n, n, n, alpha, devA, n, devB, n, beta,
devC, n);

statA = cublasGetMatrix (n, n, sizeof(*A), devA, n, A, n);
statB = cublasGetMatrix (n, n, sizeof(*B), devB, n, B, n);
statC = cublasGetMatrix (n, n, sizeof(*C), devC, n, C, n);
if (statA != CUBLAS_STATUS_SUCCESS
 || statB != CUBLAS_STATUS_SUCCESS
 || statC != CUBLAS_STATUS_SUCCESS) {
  printf ("data upload failed\n");
  cublasFree (devA);
  cublasFree (devB);
  cublasFree (devC);
  cublasShutdown();
  return EXIT_FAILURE;
}
```

GPUで行列-行列積!!

結果の回収

```
printf("alpha = %5.3e\n", alpha);
printf("beta  = %5.3e\n", beta);
printf("ans="); printmat(n,n,C,n);
printf("\n");
printf("#you can check by Matlab by:\n");
printf("alpha * A * B + beta * C =\n");

cublasFree (devA);
cublasFree (devB);
cublasFree (devC);
cublasShutdown();
delete[]C; delete[]B; delete[]A;
}
```

後片付け

cuBLASでの行列-行列積 (IV)

- 先ほどのプログラムをdgemm_demo.cppとしてセーブ
- ricc.riken.jpへログイン
- 以下コマンドライン

行列積のテスト用ファイル

コンパイルにはaccelに入る必要がある

コンパイル

サブミット用スクリプト

```
[maho@ricc1 ~]$ ssh accel
Last login: Thu Mar  7 14:12:51 2013 from ricc1
[maho@upc0000 ~]$ ls dgemm_demo.cpp
dgemm_demo.cpp
[maho@upc0000 ~]$ gcc dgemm_demo.cpp  -I/usr/local/cuda/include -L/usr/local/cuda/lib64  -lcudart -lcublas
[maho@upc0000 ~]$ cat test.sh
#!/bin/sh
#--- qsub option ---#
#MJS: -accel
#MJS: -proc 1
#MJS: -time 1:00:00
#MJS: -eo
#MJS: -cwd
#--- FTL command ---# #FTLDIR: $MJS_CWD
#--- Program execution ---#

./a.out
[maho@upc0000 ~]$
```

cuBLASでの行列-行列積 (VI)

- 注意点
  - コンパイルにはaccelに入る必要がある (ssh accel)
  - ジョブのサブミットには ricc に入る必要がある (ssh ricc)
  - インタラクティブノードはない。
    - 従ってaccelでコンパイルし、ジョブをriccに入ってサブミットしなければならない…
  - A, B, Cの行列の値の並びはFORTRANのようにcolumn majorになっている

column major

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

row major

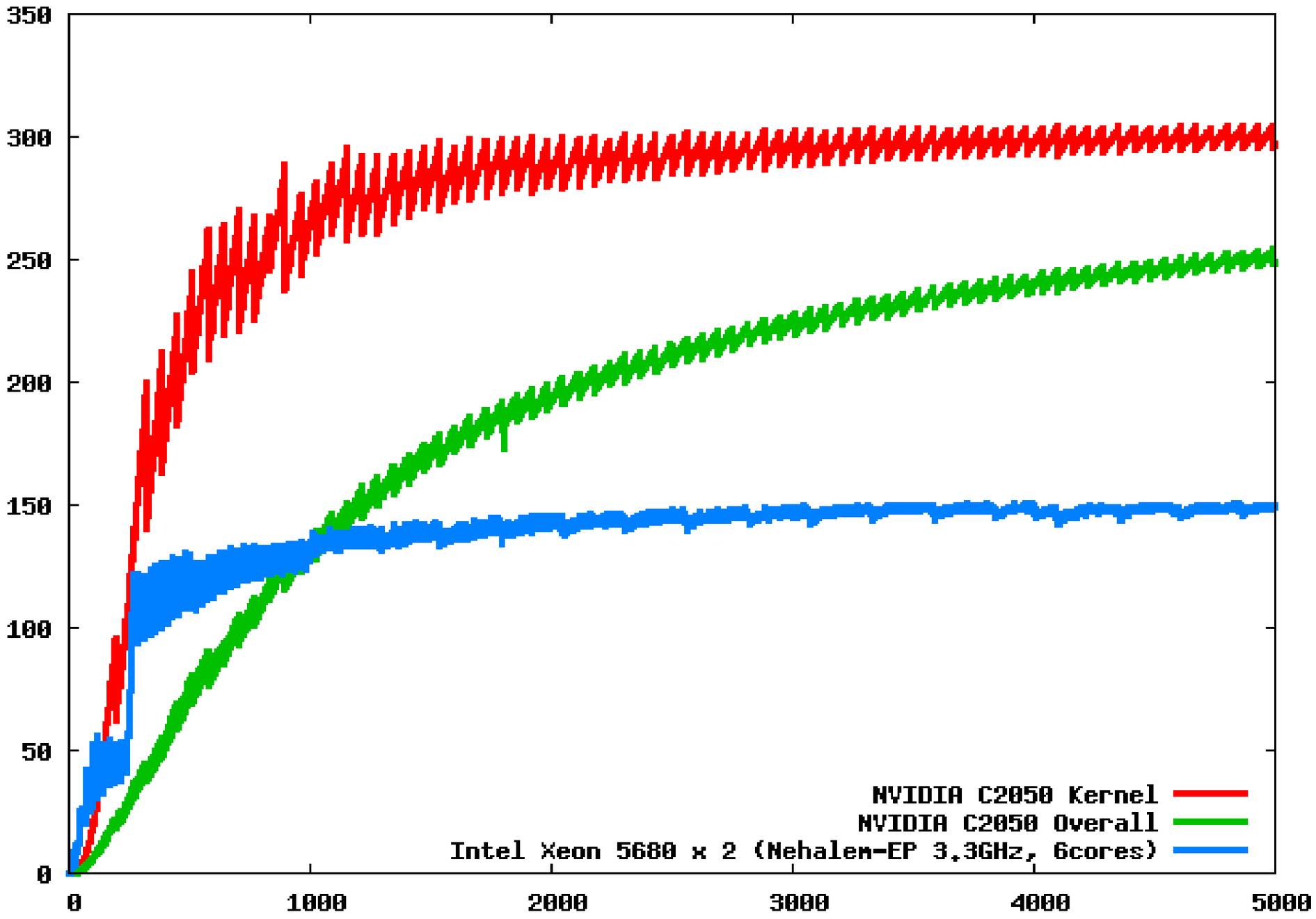
$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

cuBLASでの行列-行列積 (VIII)

- 行列-行列積の計算dgemmを呼んだ場合の結果
  - 正方行列A, B, Cおよびスカラーα, βについて、
    - C←αAB+βC
    - 行列のサイズnを1-5000まで変えた場合。
    - 縦軸はFLOPS (Floating-point Operations Per Second)
- 先のグラフ、赤い線は、GPUのみのパフォーマンス
  - 理論性能値515GFlops中300Glops程度出ている
- 緑の線は、CPU-GPUを含んだ場合のパフォーマンス
  - GPUをアクセラレータとしてみた場合
  - PCIeバスでのデータ(行列の)転送速度が遅い。
- 青の線は、Intel Xeon 5680 (Nehalem 3.3GHz) 6 core x 2 のパフォーマンス
  - 理論性能値 158.4GFlops/ノード
  - RICCは理論性能値 93.76GFlops/ノード

cuBLASでの行列-行列積 (VII)

- どの程度パフォーマンスが出るか見てみよう。



難しい問題

- サイズの大きな問題は比較的簡単に性能評価できる。
- サイズの小さな問題を多数解きたい場合は性能が下がる傾向にある。
  - これはメモリバンド幅の問題となる。