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

中田真秀

計算科学技術特論A

2017/6/1 13:00-大阪大学豊中キャンパス 基礎工学研究科 G 棟５０８

授業の内容

- コンピュータの簡単な仕組みについて。
  - フォン・ノイマン型ボトルネック、メモリ、CPUのスピードのトレンド
- なぜそのコードは低速/高速に動くのか。
  - メモリの階層構造
  - 行列の積の例： CPUがボトルネックとなる
    - 行列をブロックに分ける。
  - 行列-ベクトル積の例：メモリのスピードがボトルネック
    - スレッドを使うこと
  - この二つは超重要+基礎となる例!
- 性能評価のやりかた
- BLAS, LAPACKを高速に使うにはどうしたらよいか。
  - 最近はデフォルトで高速なものを使うので気にする必要なし…
  - 参照実装のBLASは遅い。
  - OpenBLAS/Intel MKLにリンク先を変えること
- GPUでBLASを使う

しくみを知ることが大切

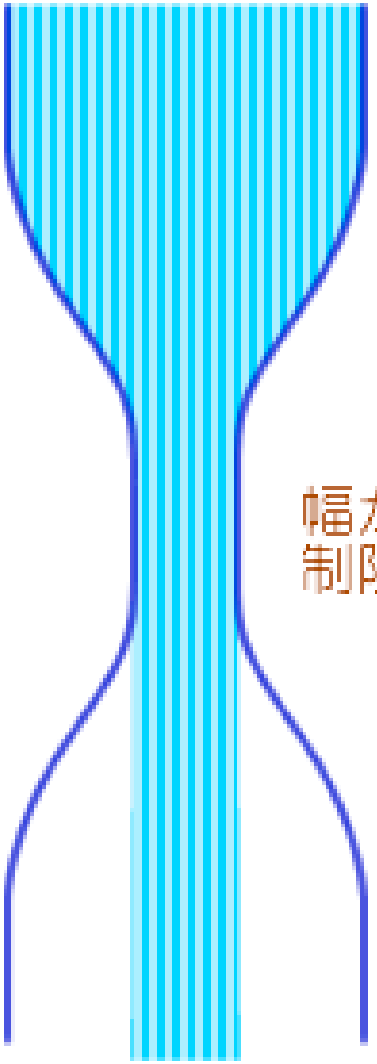
- 現在高速なBLAS, LAPACKを使うのは、ものすごく簡単

  Ubuntuではもはや標準で高速なBLAS/LAPACKが使えるので、なんで何も考える必要なし

- なぜ高速になるか、の、大まかな仕組みを知っておくことが重要。

コンピュータの簡単な仕組みについて

ボトルネック



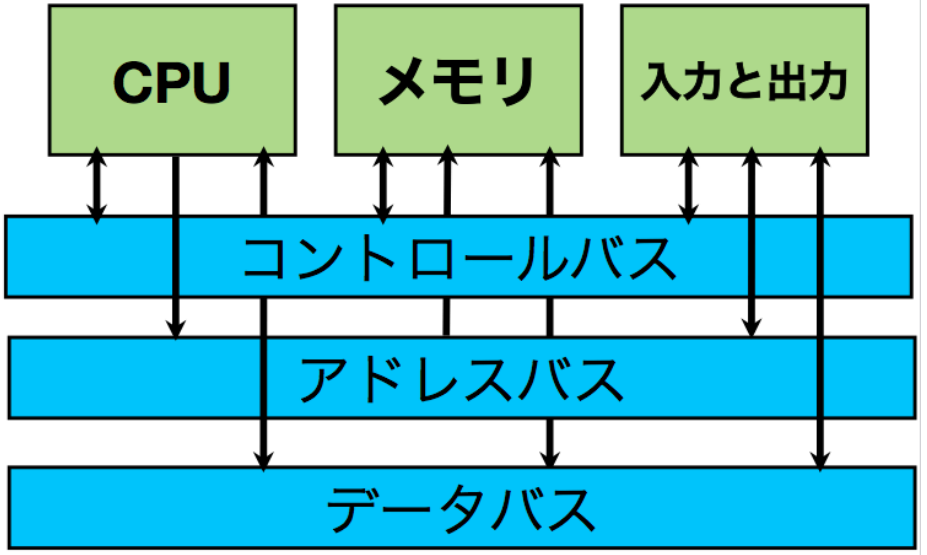
幅が広いが狭部により
最大流量が制限される

幅が狭いため流量が
制限される

幅が広いが狭部により
最大流量が制限される

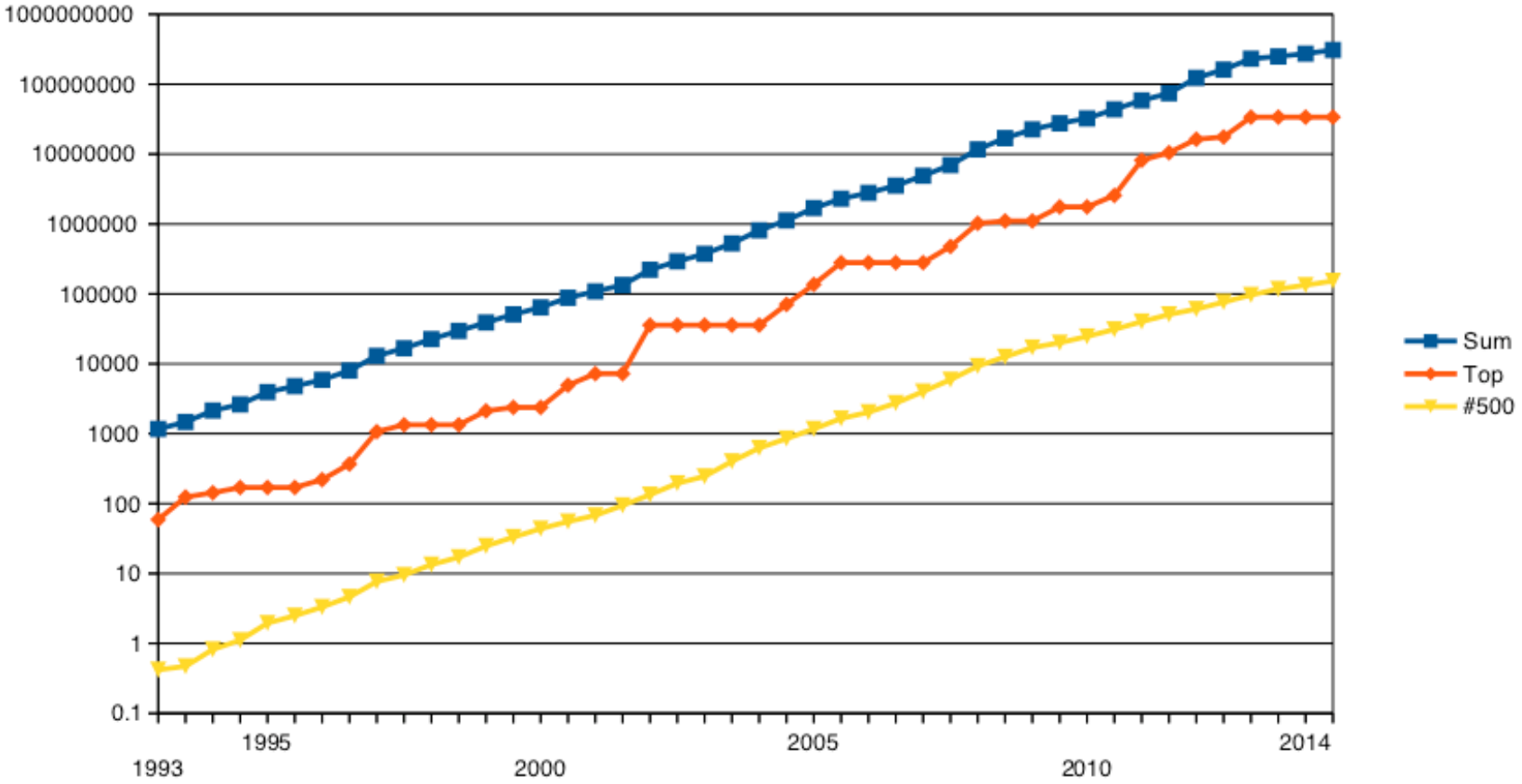
コンピュータの仕組み:ノイマン型コンピュータ

- 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
- 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。
    - 単にメモリのバンド幅、メモリのスピードと呼ばれることが多い
  - 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。
- バスをとにかく埋めておくことが高速化につながる

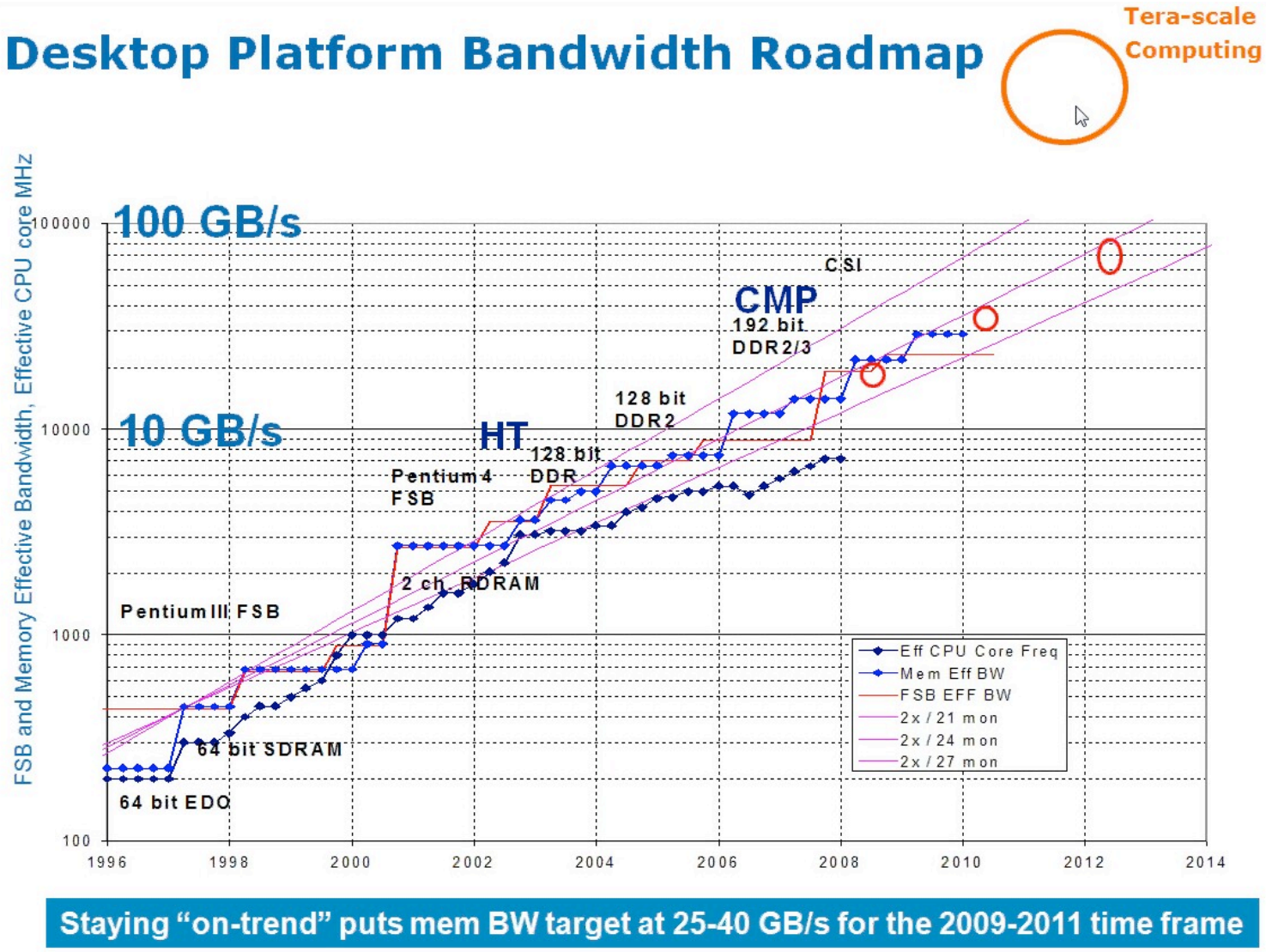


CPUのスピード(バンド幅)についてのトレンド

CPUのスピードは年々増加している



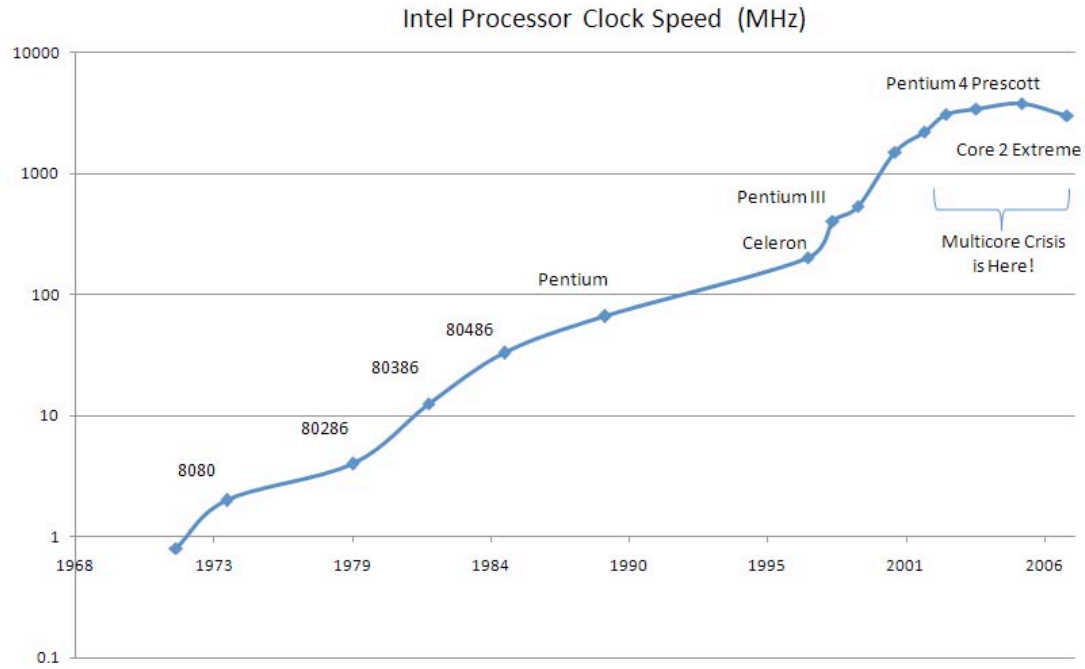
メモリのスピード(バンド幅)も年々高速になってきている

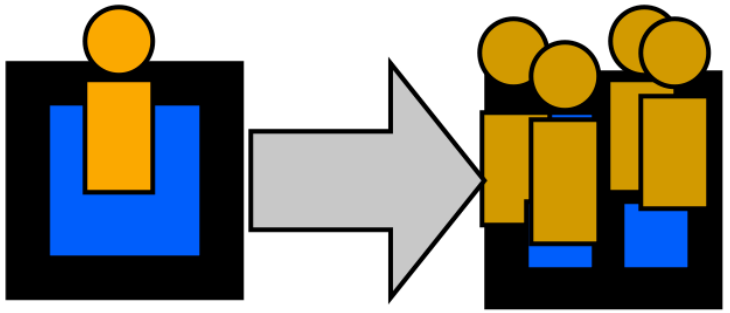


最近のコンピュータの傾向について

CPUのスピードのトレンドについて

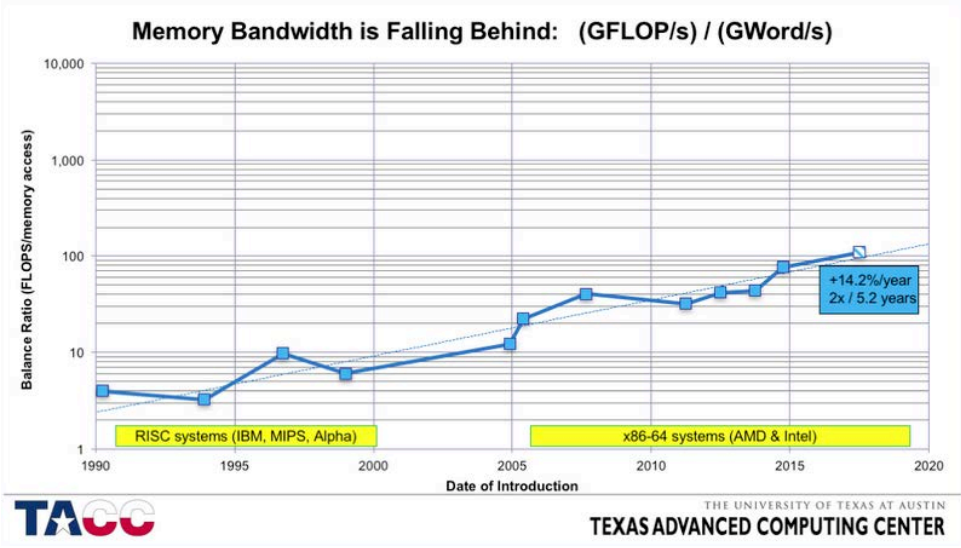
- 近年コア一個単位処理能力は落ちてきており、2000年からマルチコア化をしてきている。
  - 様々な物理的な限界：微細加工の限界(量子的ノイズ)、熱の発生…
- マルチコアとは?
  - 下図のように、コアの処理能力を上げるのではなく、いくつもコアを用意することで、処理能力をあげている。





メモリのスピードについて

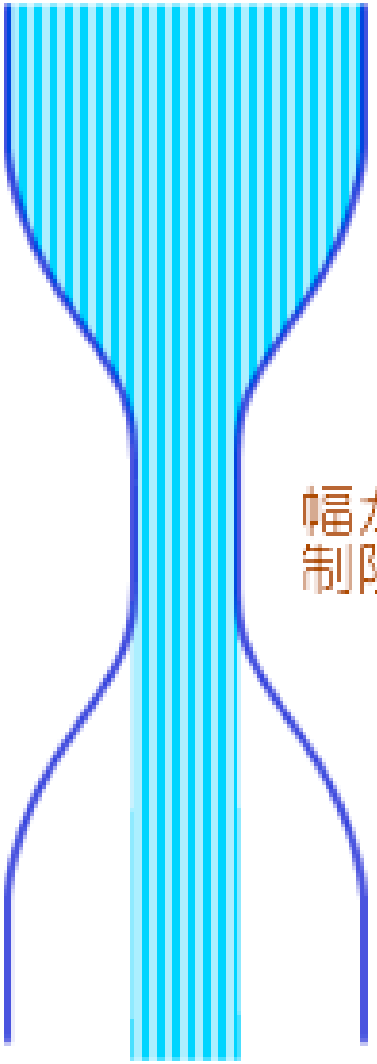
- CPUとメモリのパフォーマンス(=スピード)を年によってプロットしてみる
- 1990年くらいまでは、メモリーのスピードのほうが速く、CPUが遅かった。
  - なるべくCPUに計算させないプログラムが高速だった。
- 1990年以降、メモリよりCPUが高速
  - メモリの転送を抑えて、無駄でも計算させた方が高速。
- 三次元積層型が実用化、磁界結合メモリーなど検討されているが、根本的解決ではない



64bitのデータを転送するのに何FLOPSかかるか、というデータ

対策:プログラムを速くするには?

ボトルネック



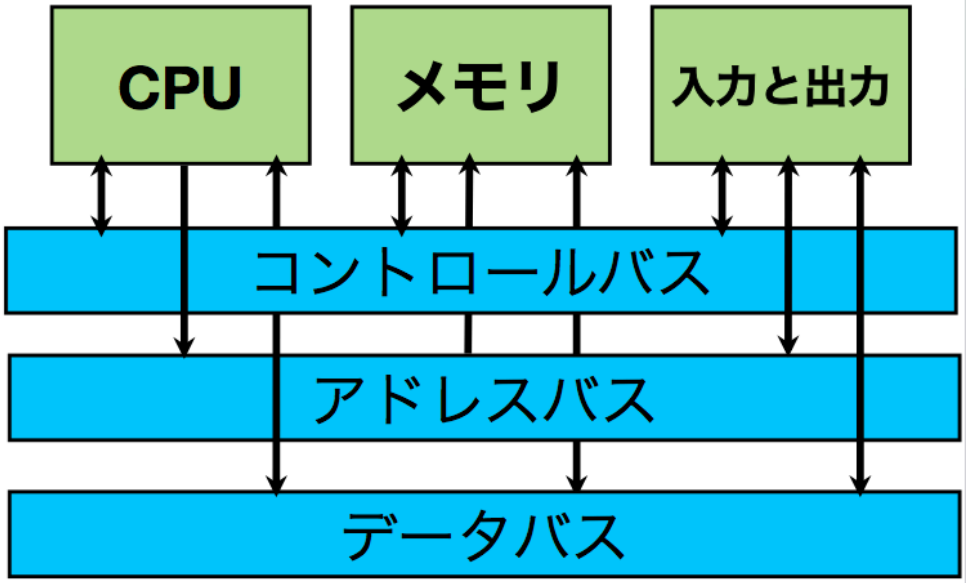
幅が広いが狭部により
最大流量が制限される

幅が狭いため流量が
制限される

幅が広いが狭部により
最大流量が制限される

ノイマン型コンピュータとボトルネック

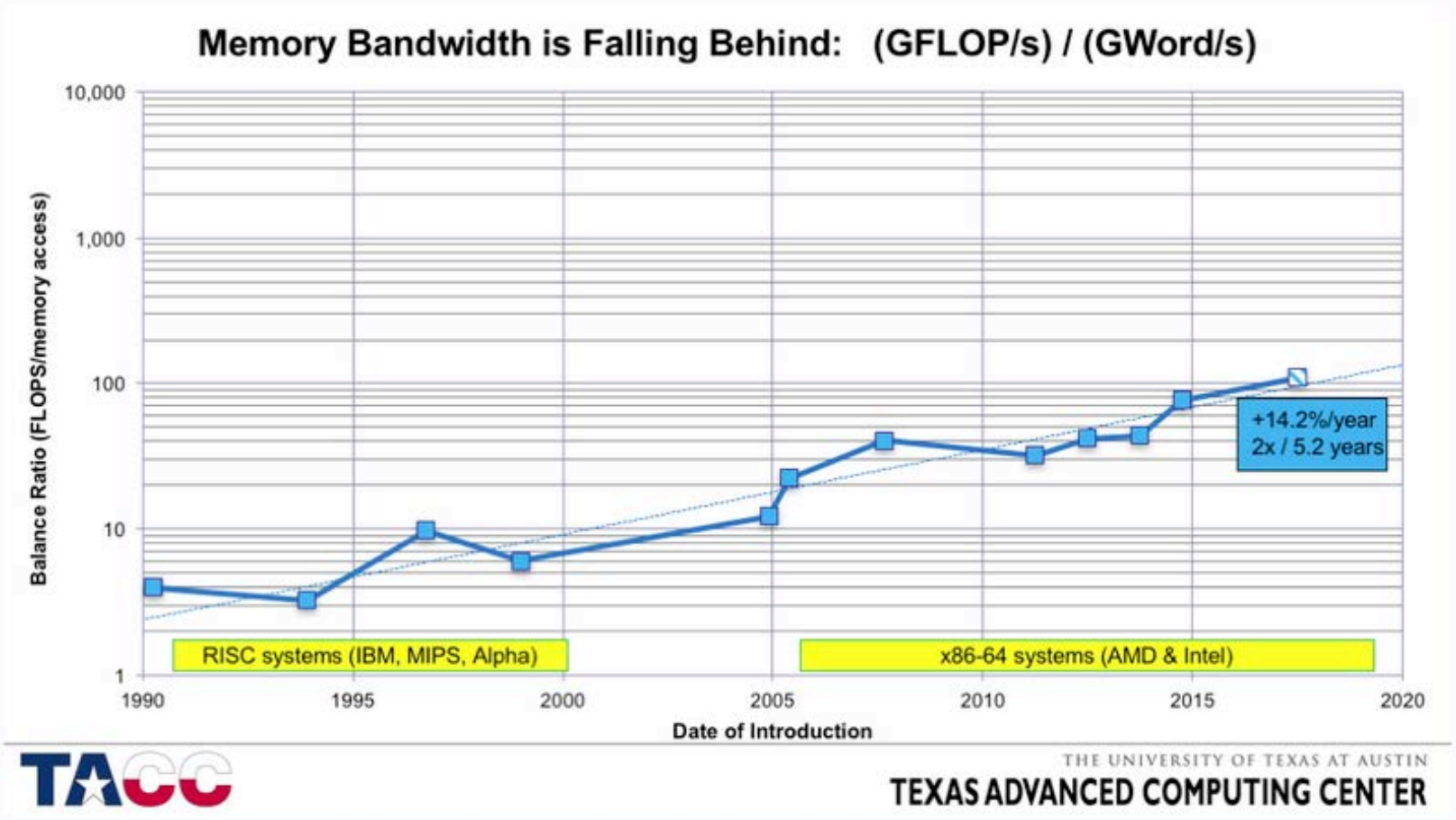
ボトルネックはどこ？
プログラムの性質を見極める



- CPUがボトルネックの場合
  - CPUの利用率をどうやって上げるか?
- メモリがボトルネックの場合
  - メモリバンド幅をどうやって満たすか?
  - CPU-メモリの転送速度がボトルネック
  - メモリの速度 or メモリのバンド幅と呼ぶ

メモリを効率的に使うため

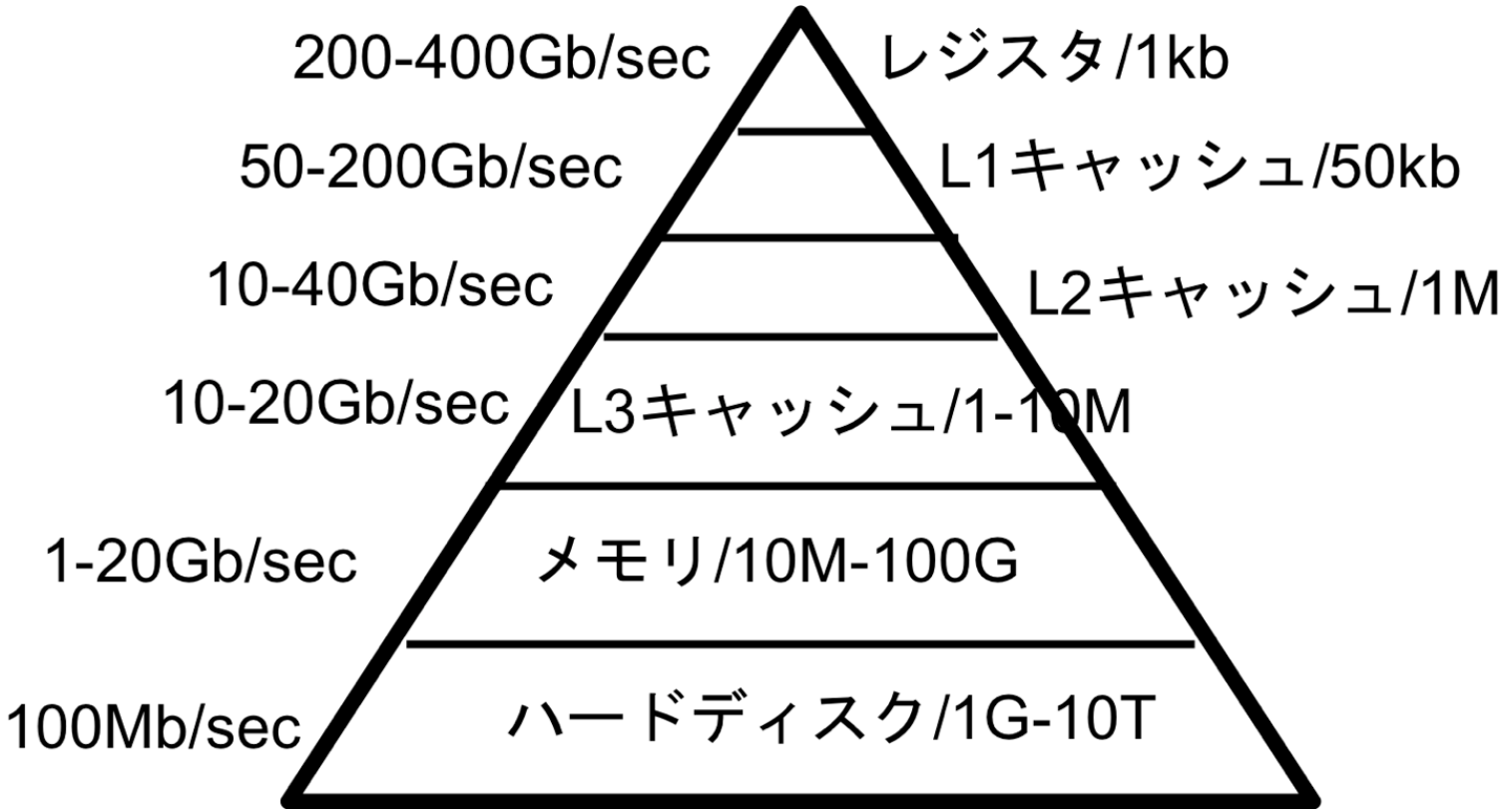
CPUとメモリだと、CPUが一方的に高速になりつつある



上図は64bitデータのロードにかかる時間/倍精度計算演算にかかる時間

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



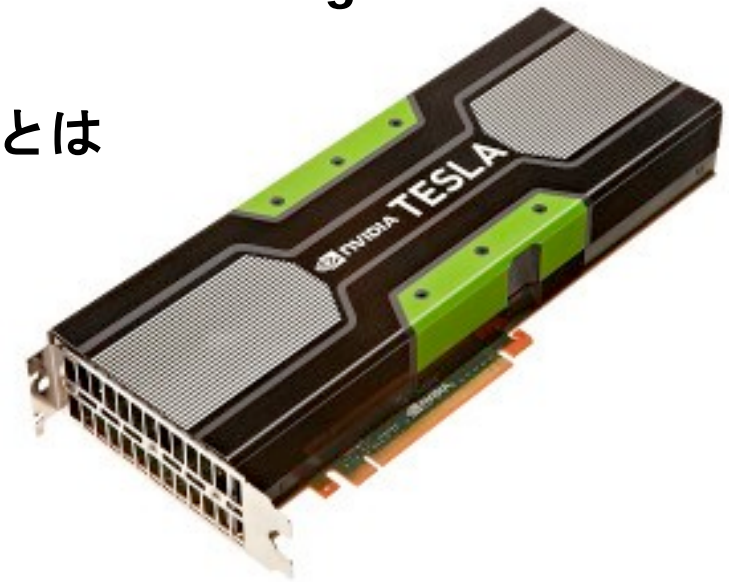
プログラムを高速化する一般的な手法

- ノイマン型のコンピュータのボトルネック
  - 演算量? データ転送量(=メモリのバンド幅)?
- 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。

GPUについても紹介

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。

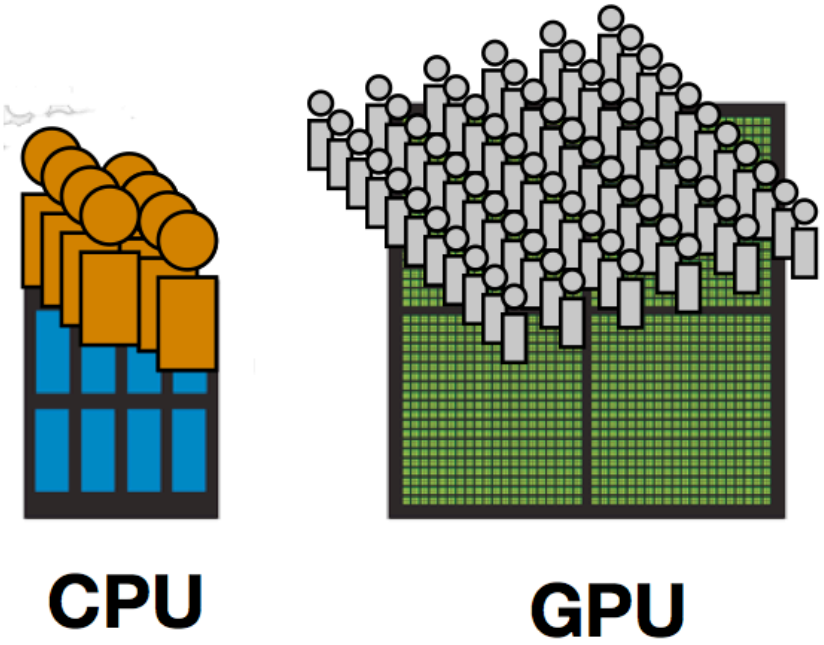
1.データを送る

3.計算結果を返す

2.計算をする
(ゲームの場合は3D画像処理など)

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。



- 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

76.8GB/s

732GB/s

GPUはどうして高速か? Part III

- コア一個の性能は高くないが頭数でこなす
- メモリアクセスは複雑になると遅くなるが順番に沢山アクセスする場合は非常に高速

- 性能の指標:1Wの消費電力で何Mflops計算できるか(green500より)?
  - NVIDIA DGX-1 : 9462.1MFlops/W
  - Xeon 2698v4 20C : 2595.9M Flops/W

  CPUに比べて3.6倍GPUは電力当たりの性能が良い。

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

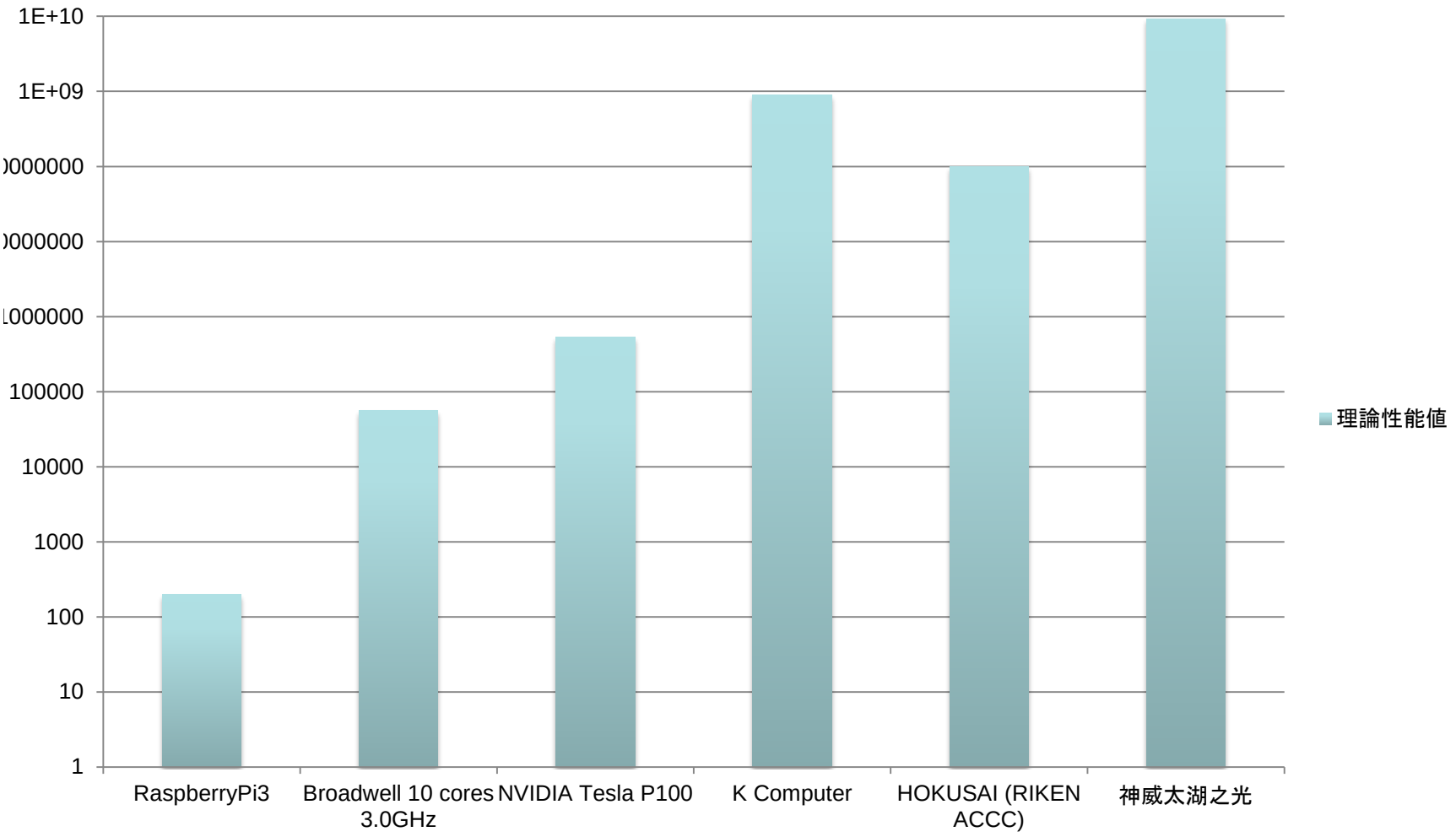
- Floating point operations per second
- 一秒間に何回浮動小数点演算ができるか。
- カタログ値ではこのピーク性能をFLOPSで出すことが多い
  - ただし、その通りの値は実際の計算では出ない。

- 中田は多分0.0001Flops程度
  - そろばんやってる人は、0.1Flopsくらいあるかもしれない。
  - 間違いは無いとする(実際は間違う)

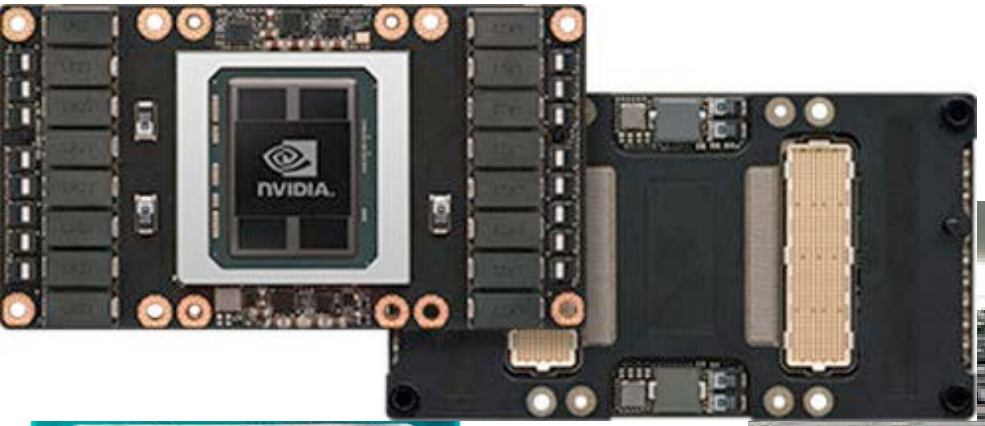
- GPUは1TFlops = 1,000,000,000,000Flpps

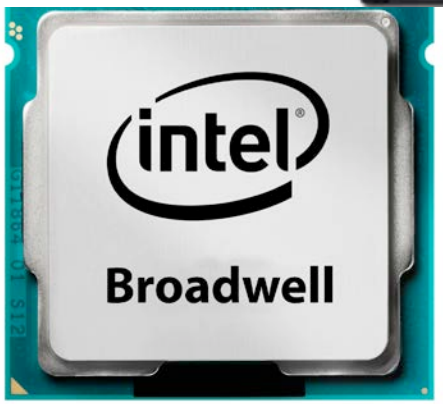
- 京コンピュータは10PFlops
  - 10,000,000,000,000,000 Flops

様々なマシンの性能:logスケールで



理化学研究所　情報基盤センター





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)

Bytes per FLOPS の例:daxpy

- daxpyについて:x, yは n 次元ベクトル、a はスカラー以下の演算をするとbyes per flopsはいくつになるか。

$$y \leftarrow y + a\,x$$

- 2n回の浮動小数点演算
  - ax : n回積, y+ax :n回和
  - 3n回のデータの読み書きが必要
  - ax: n回読み、y:n回読み、y: n回書き。
- 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。

daxpy(ベクトルの和)のbytes per flopsは12となる

Bytes per FLOPS の例:dgemv

- Aは n x n の行列として、x,y はn次元ベクトルとして積を計算する。α、βはスカラ

$$y \leftarrow \alpha A x + \beta y$$

- $2n^2+2n$回の浮動小数点演算
  - Ax: n^2回積 +n(n-1)回和
  - βy :n回積 : α(Ax) n回積
  - α(Ax)+βy n回和
- n^2+3n回のデータの読み書きが必要
  - A：n^2回読み、x: n回読み y: n回読み
  - y: n回書き
- 倍精度一つで8bytesなので $n \rightarrow \infty$ で

$(n^2+3n) * 8 / (2n^2+2n) = 4$

dgemv(行列ベクトル積)のbytes per flopsは
nが大きいところで4に近づく

Bytes per FLOPS の例:dgemm

- A, B, Cは n x n の行列として、積を計算する。α、βをスカラーとする

$$C \leftarrow \alpha AB + \beta C$$

- $2n^3+2n^2$回の浮動小数点演算
  - AB : n^3回積 ,$n^2(n-1)$回和、αAB : n^2回積
  - βC : n^2回積、$(\alpha AB) + (\beta C)$: n^2回積
- $4n^2$回のデータの読み書きが必要
  - A, B, C : $3n^2$読み
  - $\alpha(AB) + \beta C$: n^2書き
- 倍精度一つで8bytesなので $n \rightarrow \infty$ で

  $4n^2 / (2n^3+2n^2) * 8 \rightarrow 16/n$

dgemm(行列の積)のbytes per flopsはnが大きいところで、ゼロに近づく

Bytes per FLOPS の例

- **このマシンのbytes per flopは?という言い方もよくされる。**
  - **メモリバンド幅(Bytes/s)**
  - **CPUの速度 (Flop/s) の比**
  - **B/F値などと略される。**

京の Bytes/Flopは?

メモリバンド幅: 64GB/s

CPUの速度: 128GFlops

従って64/128 = 0.5 B/F

- **Intel® Xeon® Processor E5-2699 v3 (**18 cores, **2.3GHz)**

CPU : 16 FLOPS/Clock × 2.3 GHz × 18コア =662.4GFlops

メモリのバンド幅68GB/s

Bytes per flops = 0.103

ここまでのまとめ

- プログラムのボトルネックを同定するのが重要。
  - CPU以外にバス幅がボトルネックになる場合がある
- コンピュータは年々速くなってきている。
  - CPUは高速になってきているが、マルチコア化
  - メモリは相対的に遅くなってきている
    - さらに速度と容量により、階層化している
- GPUはなぜ高速か/どう高速か?
  - コアを沢山積んでいる。メモリも高速。
  - 電力1W当たりのパフォーマンスが3.6倍程度高い!
  - 分岐等複雑なことをさせると大変遅くなる。プログラムも大変
- Byte per flopの概念
  - メモリを読むのにかかる時間に何回計算できるか?

どういう風にプログラムすればいいの?

- 残念ながらバスのボトルネック、メモリ階層を明示的にプログラムするのは難しい
  - マシンやプログラム言語、環境などに強く依存している
  - 常にある処理にどの程度時間取っているか把握しておこう
- メモリ階層を意識するプログラム
  - メモリを読んだらキャッシュに書き込まれる。
  - 容量あふれ、に注意しつつそのデータはもう一度使えるかを考える
- メモリバンド幅を意識するプログラム
  - なるべく塊としてまとまって読むようにする。
  - マルチコアのCPUでは、なるべく多くのコアを使ってマルチスレッドで読む

高速なBLAS LAPACKを使うには

daxpyとdgemmを例にとって

高速なBLAS、LAPACKを使う

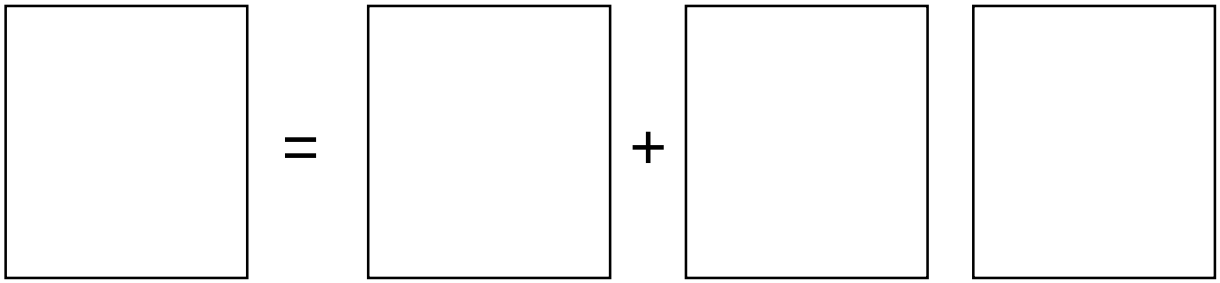
- DGEMM (行列-行列積), DGEMV (行列-ベクトル積)
  - 二つの典型的な例 CPU演算、メモリバンド幅がボトルネック
  - 超重要な二つの基礎的なボトルネック!!

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

- 最近はx86系では、BLAS LAPACK周りはかなり環境が整備されてきて、だれでも簡単に使えるようになってきたよ!!
  - GotoBLAS2のオープンソース化と、OpenBLASの開発開始、それがLinuxのディストリビューションに含まれるようになってきたため。

DGEMM 行列-行列積

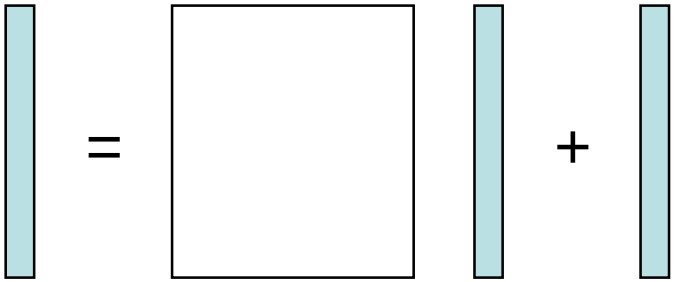
- マシンのパワーをみるには、DGEMM (行列-行列積)と、DGEMV (行列ベクトル積)をみればよい。
- DGEMM (行列-行列積)
  - CPUのパワーがどの程度あるかの良い目安。
  - C←αAB+βC



  - 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) で、n→∞でゼロとなる。

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)=4
    - 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
  - OpenBLAS (現時点で最高速の部類)
- 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 Xeon E5-2603 @ 1.80GHz 8コア(理論性能値 115.2GFlops)
  - CortexA53 (ARMv8, ODROID C2) 1.5GHz 4コア(理論性能値 6GFlops)

環境を整える

- Ubuntu 16.04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 octave
- OpenBLASのインストール
  - $ sudo apt-get install libopenblas-base libopenblas-dev

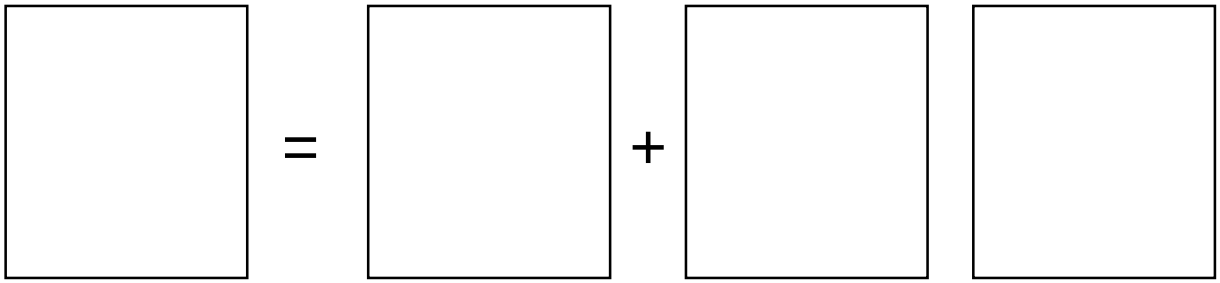
UbuntuでのBLAS, LAPACKの選択

```
$ sudo update-alternatives --config libblas.so.3
There are 3 choices for the alternative libblas.so.3 (providing /usr/lib/libblas.so.3).

  Selection    Path                                Priority   Status
------------------------------------------------------------
* 0            /usr/lib/openblas-base/libblas.so.3      40        auto mode
  1            /usr/lib/atlas-base/atlas/libblas.so.3   35        manual mode
  2            /usr/lib/libblas/libblas.so.3            10        manual mode
  3            /usr/lib/openblas-base/libblas.so.3      40        manual mode

Press <enter> to keep the current choice[*], or type selection number:
```

2 : リファレンス (低速、お手本)
3 : OpenBLAS (一番高速)
1 : ATLAS (スピードは2, 3の中間)

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。

Reference BLASの行列-行列積

- Reference BLASの場合の設定

  $ sudo update-alternatives --config libblas.so.3で, 2 (/usr/lib/libblas/libblas.so.3) を選択
- Octaveで行列-行列積
- 1000x1000の正方行列の積。値はランダム

  $ octave

  ... 途中略...

  octave:1> n=1000; A=rand(n); B=rand(n);

  octave:2> tic(); C=A*B; t=toc();

  octave:3> GFLOPS=2*n^3/t*1e-9

  GFLOPS = 1.6514 (Xeon E5-2603)

  GFLOPS = 0.22823 (Cortex A53 (ARMv8))

  1.6865GFLops =>理論性能値のたった 1.4%

  0.22823GFlops =>理論性能値のたった 3.8%

OpenBLASの行列-行列積

- OpenBLASの場合の設定

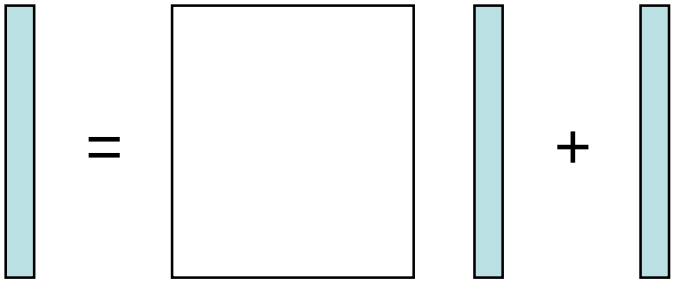
  $ sudo update-alternatives --config libblas.so.3で0(/usr/lib/openblas-base/libblas.so.3) を選択

- Octaveで行列-行列積
- 1000x1000 (ARM) 4000x4000(Intel)の正方行列の積。値はランダム

$ octave
... 途中略...
octave:1> n=1000; A=rand(n); B=rand(n);
octave:2> tic(); C=A*B; t=toc();
octave:3> GFLOPS=2*n^3/t*1e-9
GFLOPS = 101.22 (Xeon E5-2603)
GFLOPS = 4.3942 (Cortex A53 (ARMv8))
101.22GFlops =>理論性能値の87.9%
4.3942GFlops =>理論性能値の73.2%

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)=4
    - 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。

メモリバンド幅理論性能値

- Intel Xeon E5-2603マシンメモリバンド幅の理論性能値
  - 34.1 GB/s理論性能値(1024 = K?)
  - https://ark.intel.com/ja/products/64592/Intel-Xeon-Processor-E5-2603-10M-Cache-1_80-GHz-6_40-GTs-Intel-QPI
  - stream実測値 triad 35.6GB/s (1000=K)
- ARM53 (ARMv8, ODROID C2) のメモリバンド幅推算
  - 2GB 32bit DDR3 912MHzSamsung K4B4G1646Dより推算するが合わない
  - **恐らく1066MHz x 4 = 4.26GB/s だと思われる。**
  - stream実測値 triad 3.6GB/s copy 4.2GB/s

考察

- Byte per flop (Xeon) 34.1 / 8 (bytes/倍精度1個)= 4.26GB/Flop
- Byte per flop (Cortex A53) 4.2/8 = 0.53GB/Flop
- これはCPUの速度に比べて27倍くらい遅い。
- メモリバンド幅は今後CPUと相対的に上がる見込みが少ない…

Reference BLASの行列-ベクトル積

- Reference BLASの場合の設定

  $ sudo update-alternatives --config libblas.so.3で, 2 (/usr/lib/libblas/libblas.so.3) を選択

- 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 = 1.1751 (Xeon E5-2603)

GFLOPS = 0.18676 (Cortex A53 (ARMv8))

1.1751 Gflops / (0.25 flop/byte) / 4.26 B/F=>理論性能値の6.9%

※CPUの演算器としての性能は115GFlops、それと比較すると1%

0.18676*(0.25) / 0.53 GFlops =>理論性能値の8.8%

※CPUの演算器としての性能は6GFlops、それと比較すると3.1%

OpenBLASの行列-ベクトル積

- OpenBLASの場合の設定

  $ sudo update-alternatives --config libblas.so.3で0(/usr/lib/openblas-base/libblas.so.3) を選択

- Octaveで行列-ベクトル積　n=10000 or 30000

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 = 6.3251 (Xeon E5-2603)

GFLOPS = 0.681 (Cortex A53 (ARMv8))

6.3251*(0.25) / 4.26 GB/Flop =>理論性能値の37%

※CPUの演算器としての性能は115GFlops、それと比較すると5.5%

0.681*(0.25) / 0.53 GBFlop =>理論性能値の32%

※CPUの演算器としての性能は6GFlops それと比較すると11%

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

おまけ:streamでメモリバンド幅測定

- Intel Xeon E5-2603マシンメモリバンド幅の理論性能値
  - 34.1 GB/s理論性能値(1024 = K?)
  - https://ark.intel.com/ja/products/64592/Intel-Xeon-Processor-E5-2603-10M-Cache-1_80-GHz-6_40-GTs-Intel-QPI
  - stream実測値 triad 35.6GB/s (1000=K)
- ARM53 (ARMv8, ODROID C2) のメモリバンド幅推算
  - 2GB 32bit DDR3 912MHzSamsung K4B4G1646Dより推算するが合わない
  - **恐らく1066MHz x 4 = 4.26GB/s だと思われる。**
  - stream実測値 triad 3.6GB/s copy 4.2GB/s

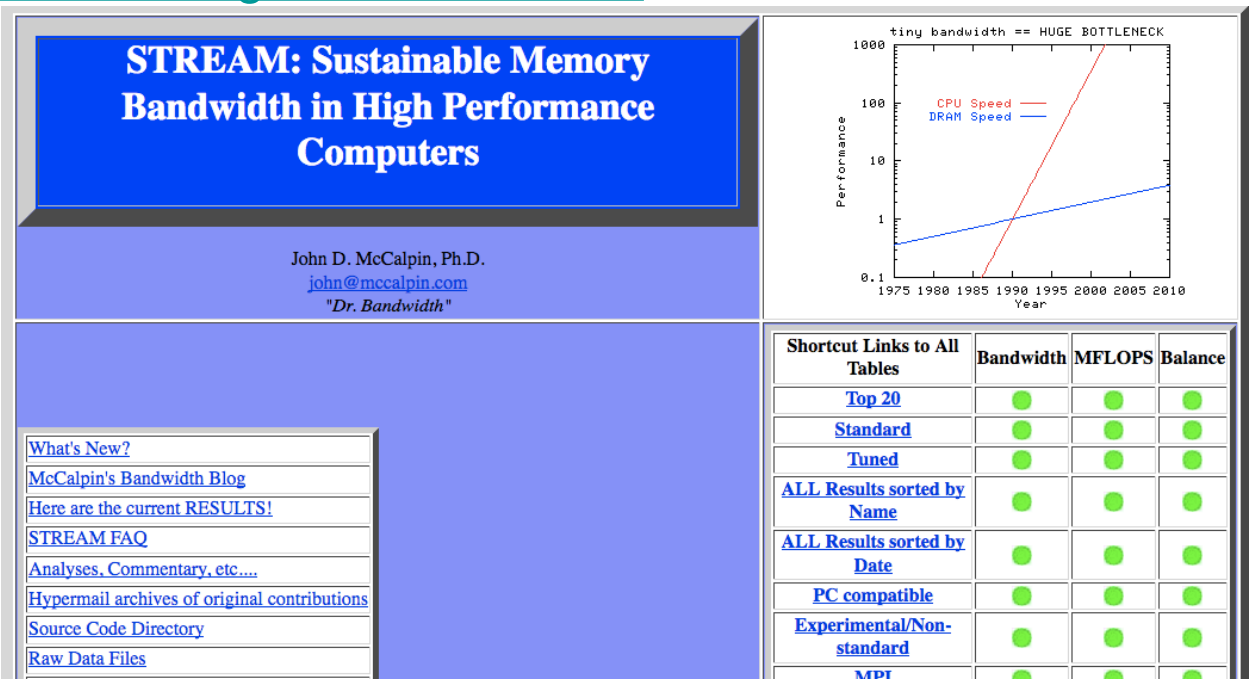
計算はめんどくさい、よくわからないときが多い。

メモリバンド幅の実効性能はstreamで測定しておくのが楽

(理論性能値やdaxpy, dgemvと比較しておくとよい)

おまけ:streamでメモリバンド幅測定

- https://www.cs.virginia.edu/stream/ からダウンロード



- wget https://www.cs.virginia.edu/stream/FTP/Code/stream.c

おまけ:streamでメモリバンド幅測定

$ wget https://www.cs.virginia.edu/stream/FTP/Code/stream.c

…

$ gcc -O -fopenmp stream.c -o stream_openmp

$ ./stream_openmp

…

Function	Best Rate MB/s	Avg time	Min time	Max time
Copy:	34579.7	0.004676	0.004627	0.004705
Scale:	33166.4	0.004852	0.004824	0.004880
Add:	36675.5	0.006557	0.006544	0.006588
Triad:	36737.1	0.006545	0.006533	0.006563

みたいになるので、これを使います。

!ここまでのQuiz!

1. 手持ちのUbuntuマシンでここでやったようにoctaveでパフォーマンス測定して比較してみよう(dgemm, dgemv)
2. dgemm, dgemv, daxpyのbyte per flopを求めよう。
3. ddot, dsymmのbyte per flopを求めよう。
4. スレッド数を変えてみるとstreamの結果はどうなるか。各自のマシンで最も良い値を求めてみよう。

   $ OMP_NUM_THREADS=8 ./stream_openmp

5. OMP_NUM_THREADSを変化させると、なぜstreamのパフォーマンスが変化するか考察してみよう。
6. マニア向け: 自分が持っているマシンのマザーボードのマニュアルを読んで、triple/quadrupleチャネルに対応するようにメモリを差し替えてstreamを測定してみよう。どのように変化するか?
7. マニア向け: BIOS (UEFI)のメモリ関連の設定を変更して変化するか?

!ここまでのQuiz!

8. 手持ちのマシンのCPUの型番を調べてみよう。
9. その型番の倍精度の理論性能値を求めよう。
10. 手持ちのマシンのメモリの型番を調べよう。
11. CPU/マザーボード/メモリの構成でのメモリバンド幅の理論性能値を調べよう。
12. Streamのページから、他のマシンの実行メモリバンド幅と比較してみよう。妥当な値がでているか。

高速化の手法

おさらい: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? データ転送量(=メモリのバンド幅)?
- 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。

プログラムを高速化する一般的な手法

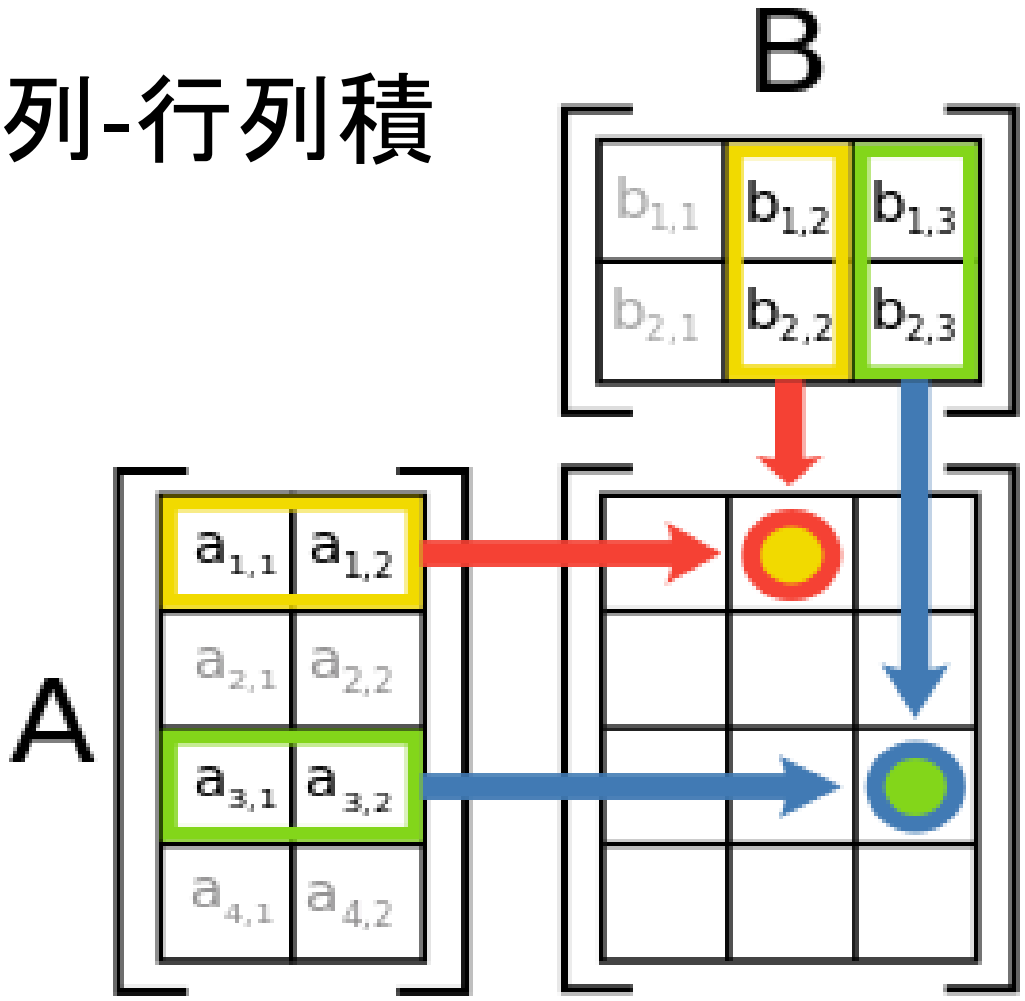
- 特にメモリバンド幅を出そうとしたらどうしたらよいの？
  - 基本的にはバッドノウハウの塊☹
  - メモリの階層構造(スピードと容量の関係)を理解したプログラム言語はほとんど無い
    - CPU依存、OS依存が激しい…
    - CUDAくらい?
- マルチコア、マルチノードの場合
    - OpenMP, MPIなど結構ある。

典型的なチューニング: 行列のブロック化

- 行列-行列積のチューニング技法について
- 昨今のコンピュータのbytes per flopは非常に小さくなってきている B/F << 1
- 行列-行列積のbytes per flopはO(1/n)なので、サイズが大きい極限では0
  - 別の表現:BFの小さなアプリケーションはデータをメモリから一回読んだら、最後までメモリを読み書きはせず、レジスタやキャッシュ内で使いまわす。

行列-行列積はピーク性能が出やすいはず。

典型的なチューニング: 行列のブロック化



典型的なチューニング: 行列のブロック化

そのままだと性能がでないので、アルゴリズムを変更
行列の積は、区分行列の積と等しい。

区分行列の積 [編集]

ふたつの区分行列

$$A = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1q} \\ A_{21} & A_{22} & \cdots & A_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ A_{p1} & A_{p2} & \cdots & A_{pq} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1r} \\ B_{21} & B_{22} & \cdots & B_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ B_{q1} & B_{q2} & \cdots & B_{qr} \end{pmatrix}$$

の区分けがそれぞれ $(l_1, \cdots, l_p; m_1, \cdots, m_q)$ 型、$(m_1, \cdots, m_q; n_1, \cdots, n_r)$ 型であるとき、その積 AB の $(l_1, \cdots, l_p; n_1, \cdots, n_r)$ 型の区分け

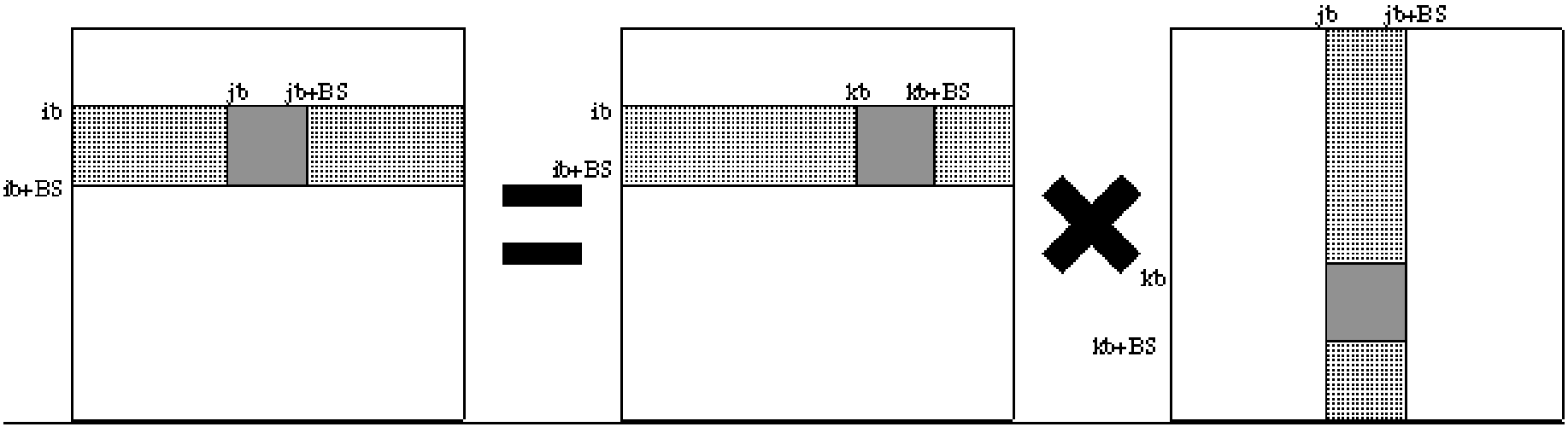
$$AB = \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1r} \\ C_{21} & C_{22} & \cdots & C_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ C_{p1} & C_{p2} & \cdots & C_{pr} \end{pmatrix}$$

の各ブロックは

$$C_{ij} = \sum_{k=1}^{q} A_{ik} B_{kj}$$

で与えられる。すなわち、区分行列の積は（適切に区分けされていれば）各ブロックをあたかも行列の成分のように見なして計算できる。

典型的なチューニング: 行列のブロック化



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

- サイズによって急にパフォーマンスが落ちる場合、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]
```

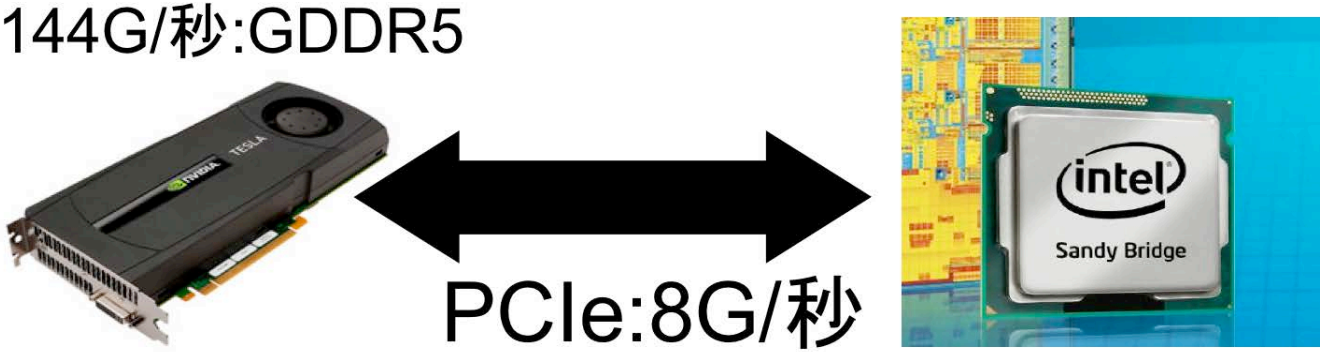
下線を引いた所でキャッシュミス
c[0][0]の計算で11回のキャッシュミス
4要素の計算で11x4=44回のキャッシュミス

```
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

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)
    - OpenBLAS
    - GotoBLAS2
    - 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

前回のBLASの例と比較して随分長くなった…

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

```
[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 ~]$
```

コンパイルにはaccelに入る必要がある

コンパイル

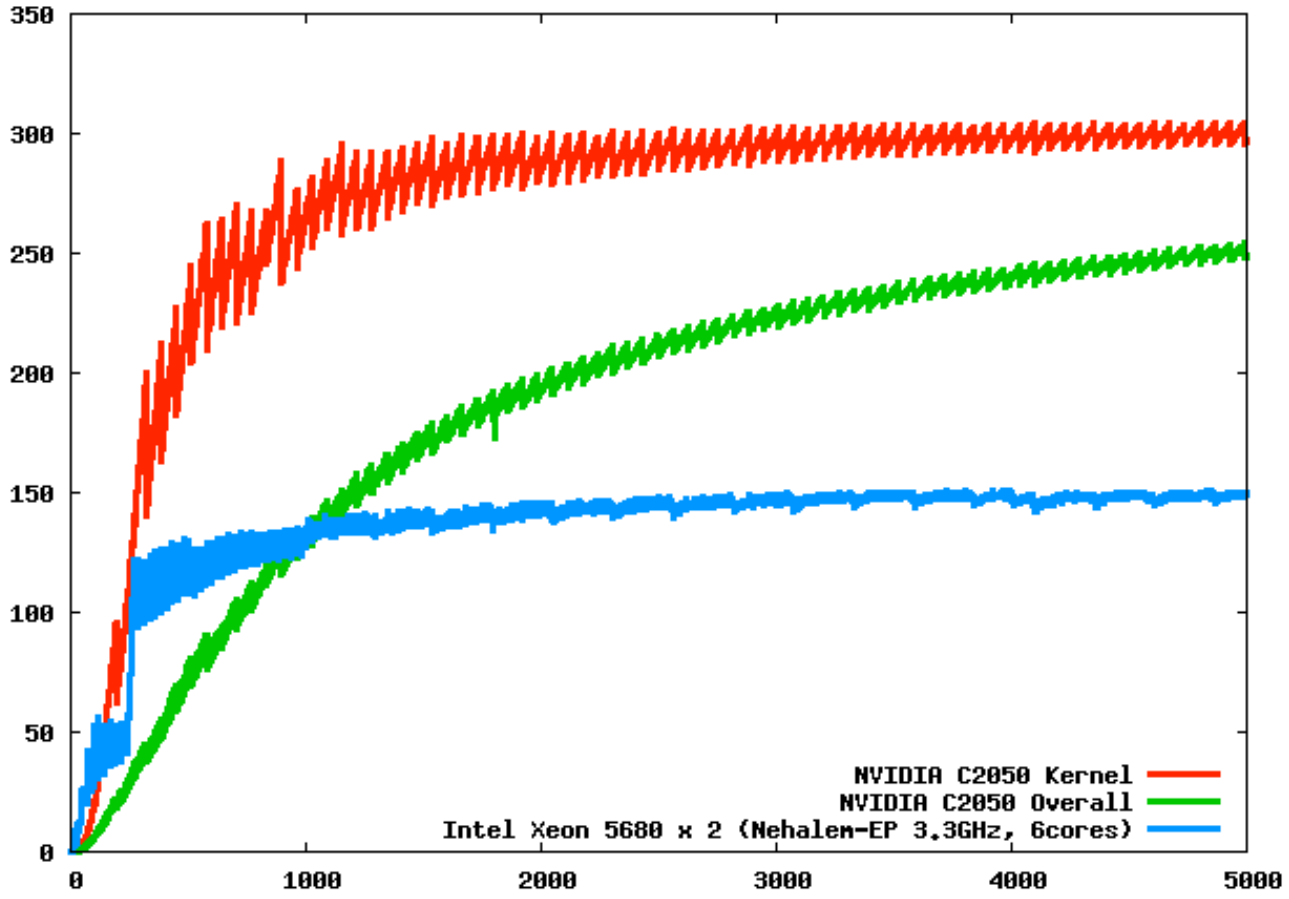
サブミット用スクリプト

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)

- どうして下図のようなグラフになるか、考察してみよう。



まとめ

- コンピュータの簡単な概念図:フォンノイマン図
- ボトルネック
- メモリバンド幅とCPUの高速化、メモリのスピードは追いつかない。
- Bytes per flopの概念
  - アプリケーションによってBF値が異なる.
  - マシンによってBF値が異なるが、傾向はBFが小さくなる方向

- BLAS, LAPACKの高速版、OpenBLASを使ってそのパフォーマンスの違いを見る。
- 行列-行列積は、CPUの性能が出る。行列-ベクトル積はメモリのバンド幅の性能が出る。

- 行列-行列積の高速化の手法: ブロック化
  - アルゴリズムをブロック行列に分けたものの積をとると変更
  - キャッシュのヒット率が上がる。
- GPUで行列行列積を行う例