線形代数演算ライブラリBLASとLAPACKの基礎と実践 (II)
BLAS, LAPACK実践編

中田 真秀
理化学研究所 情報システム本部
2019/5/30 計算科学技術特論A

13:00

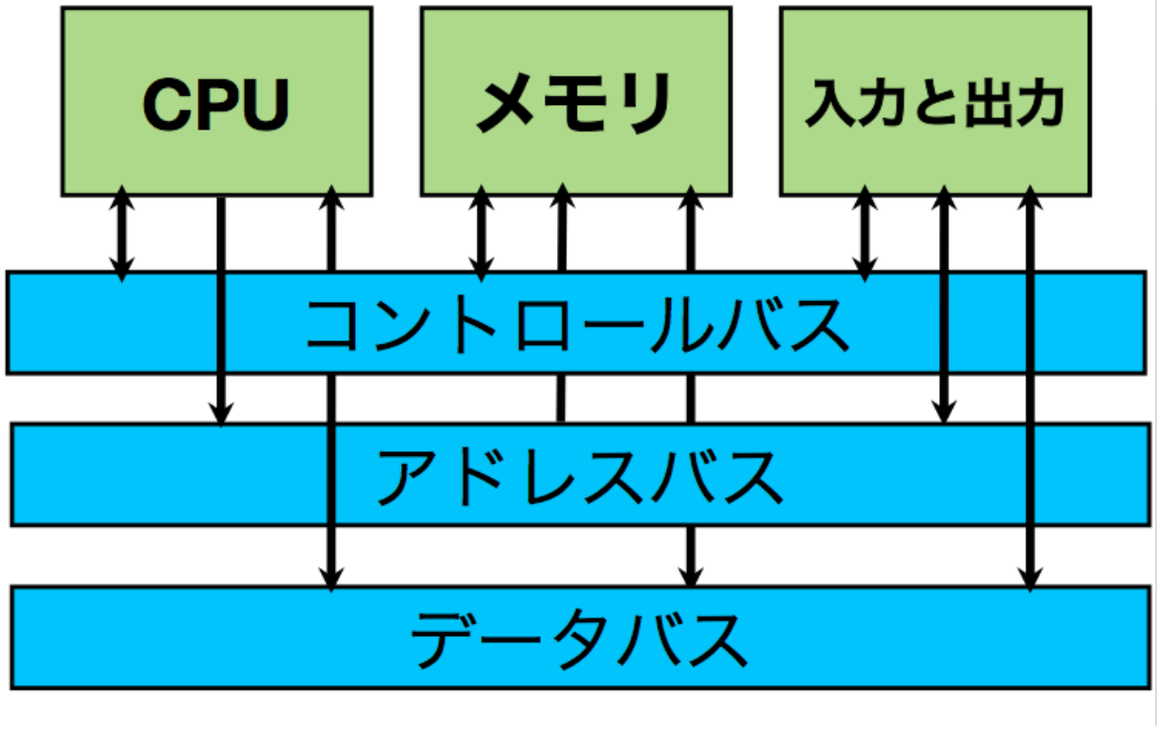
BLAS, LAPACK実践編 講義内容

- コンピュータの簡単な仕組みとボトルネック
  - フォン・ノイマン型コンピュータ
  - フォン・ノイマンボトルネック
  - 演算バンド幅のトレンド
  - メモリバンド幅のトレンド
  - 演算バンド幅の理論性能
  - メモリバンド幅の理論性能
  - GPUについて
- 最適なBLAS/LAPACKとリンクする
- DGEMM: 演算バンド幅がボトルネックになる
- DGEMV: メモリバンド幅がボトルネックになる
- DGEMM高速化手法の紹介：ブロック化 (メモリの階層構造)
- GPUでBLAS/LAPACKを使う+DGEMMベンチマーク

コンピュータの簡単な仕組みについて

コンピュータの仕組み:ノイマン型コンピュータ

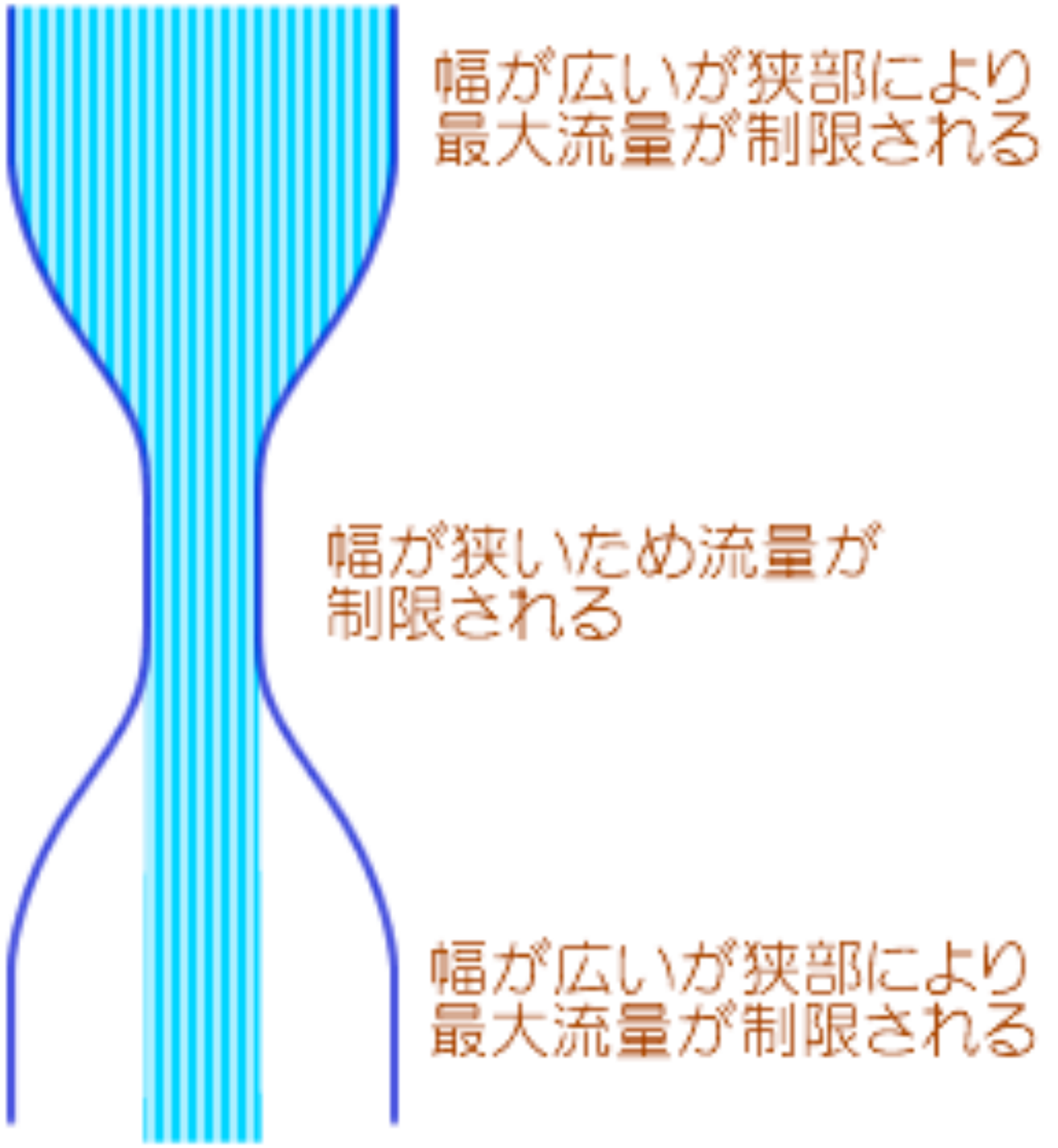
- コンピュータを最も簡単にあらわした図
- フォン・ノイマン型コンピュータ
- メモリにプログラムが内蔵されている。
- CPUがそれを処理
- バスがそれぞれの装置を結んでいて、データのやり取りをする



プログラム全体の処理スピードを決める箇所

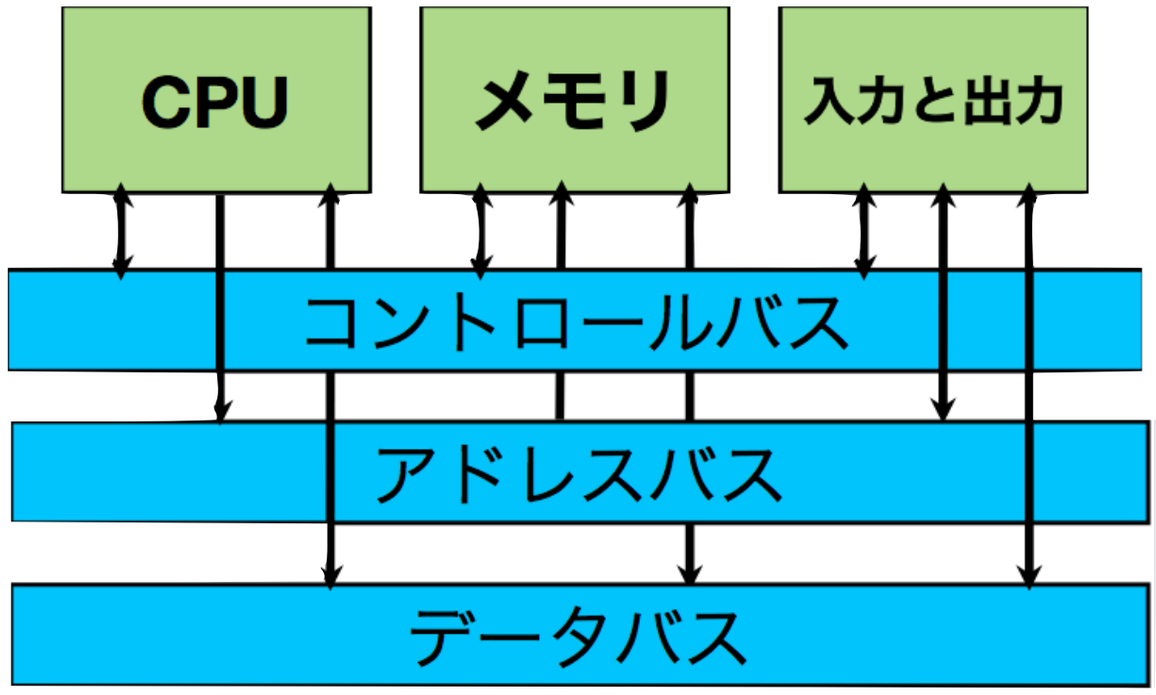
=ボトルネック

ボトルネック概念図


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

フォン・ノイマンボトルネック

- CPUが高速だとプログラムは高速になる
- メモリが高速だとプログラムは高速になる
- 入出力が高速だとプログラムは高速になる



それだけではなくて
バスのスピードが低速だと、それに全体の処理速度が引っ張られる場合がある

フォン・ノイマンボトルネック

主な２つの
コンピュータのボトルネック

- 計算能力が高い/低い CPUの速度が速い/遅い
    - 計算バンド幅
- メモリの転送能力が高い/低い
    - メモリバンド幅 (フォン・ノイマン型ボトルネック)
- 他にも
    - HDD/SSDの読み書き速度、SATA転送速度、レイテンシ(遅れ)=1bit欲しいと命令してからバスにのるまでの時間などなどなど…あらゆるところに存在する

2大コンピュータのボトルネックと高速化戦略

- **計算バンド幅 < メモリバンド幅**
  - **計算能力が低く、メモリ転送が速い**
    - **計算結果をなるべくメモリに残す**
- **メモリバンド幅 < 計算バンド幅**
  - **計算能力が高く、メモリ転送は遅い**
    - **少量のデータをなるべく使い回す**

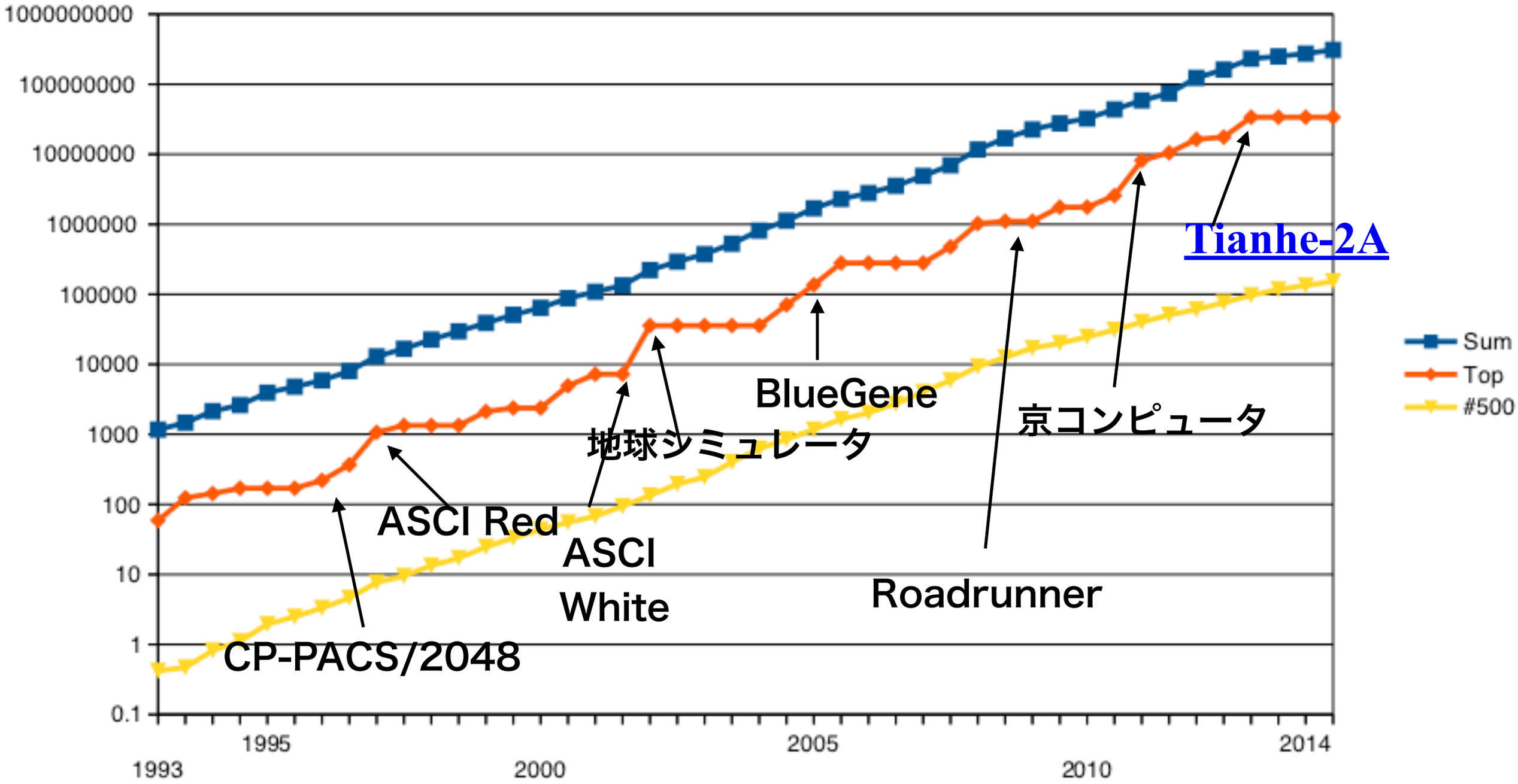
ボトルネック:
計算バンド幅について

FLOPS:演算バンド幅性能の指標

- FLOPS：マシンやプログラム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。
  - ただし、その通りの値は実際の計算では出ない。
- 人間は多分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
- 2018/11時点の世界一はSummit 143PFlops
  - 143,000,000,000,000,000 Flops

計算バンド幅(CPUの速度)は年々上がっている

Top500での
合計パフォーマンス、最速、最も遅いコンピュータのパフォーマンス (Flops)

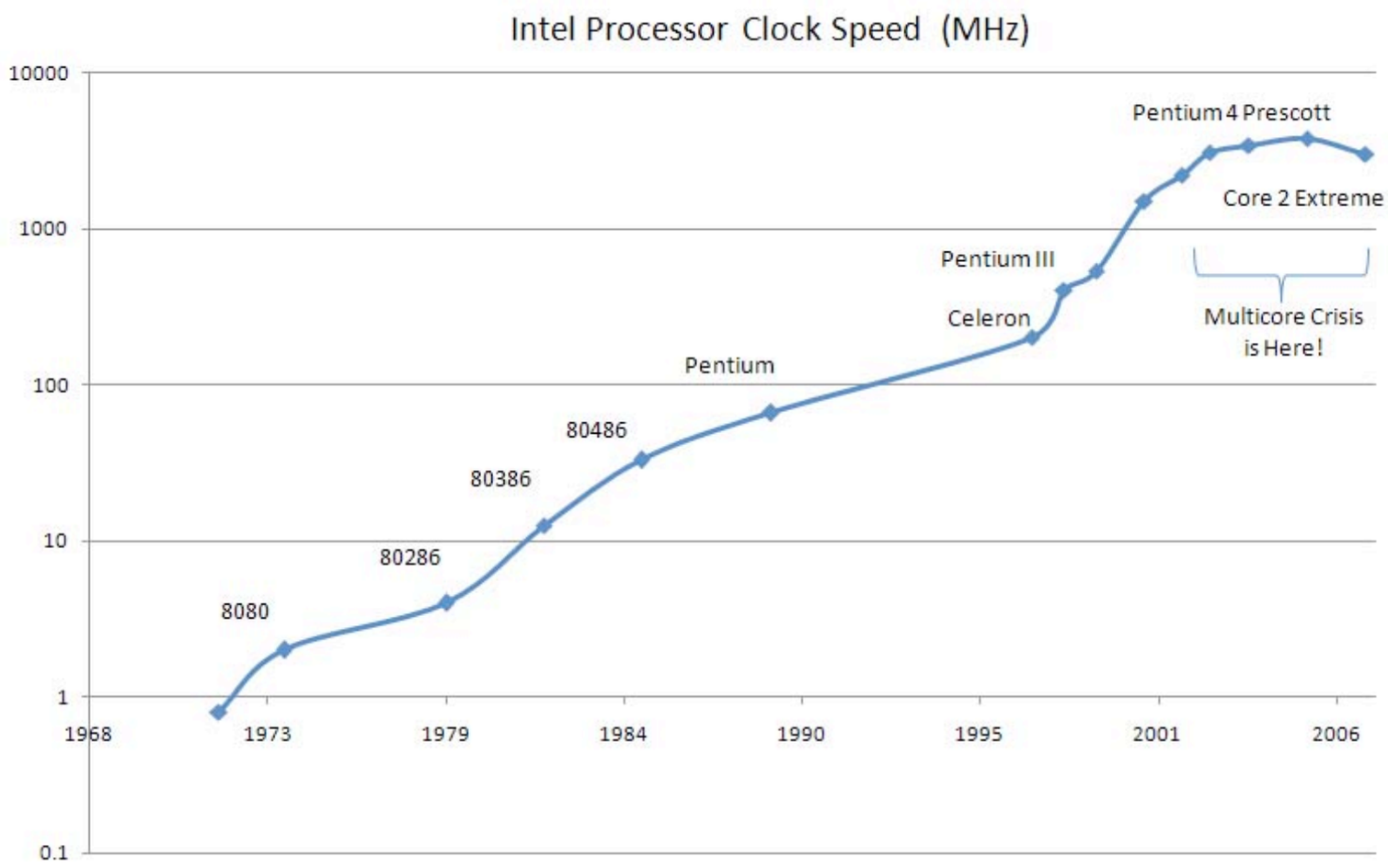


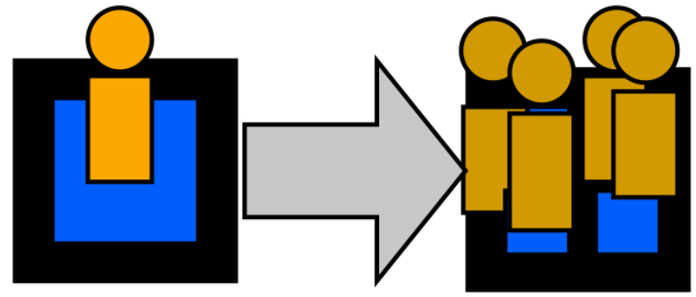
計算バンド幅(CPUの速度)のトレンド

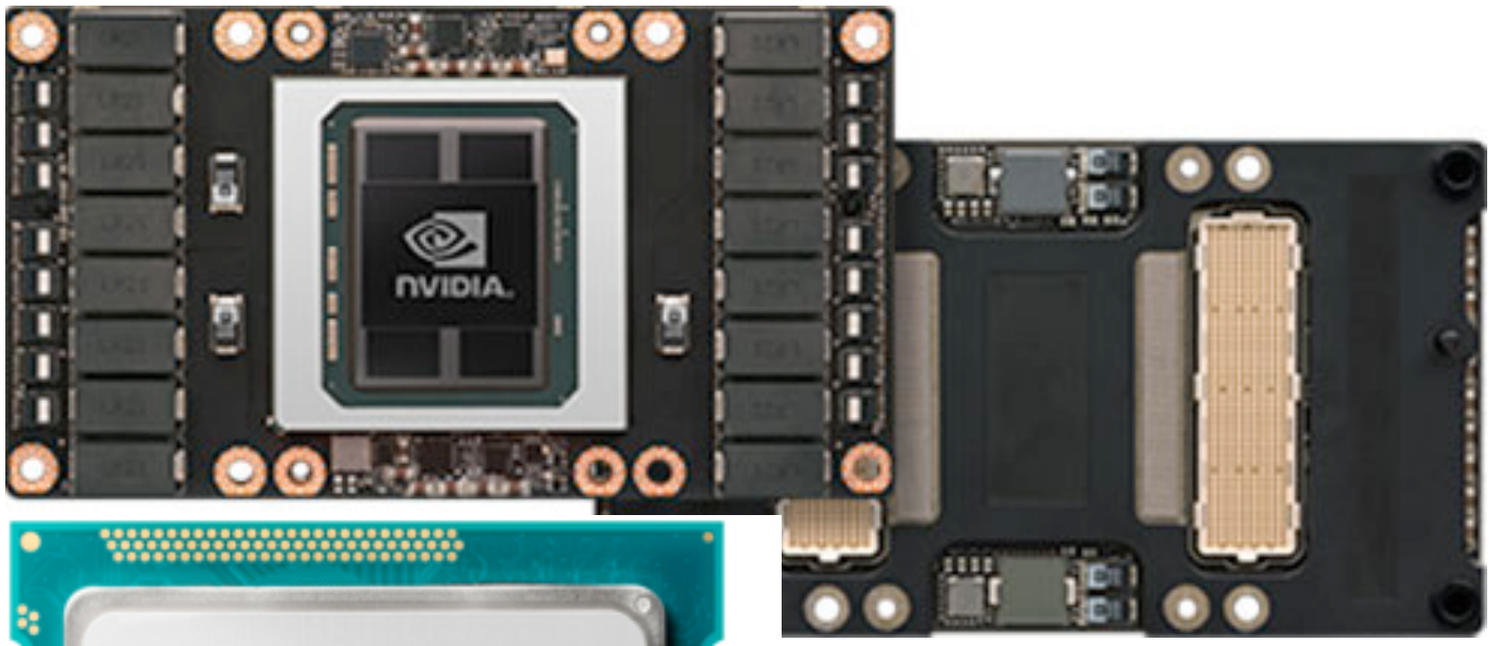
- 近年コア一個単位処理能力は落ちてきており、2000年からマルチコア化をしてきている
  - 様々な物理的な限界：微細加工の限界(量子的ノイズ)、熱の発生…

•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。






NVIDIA


summit

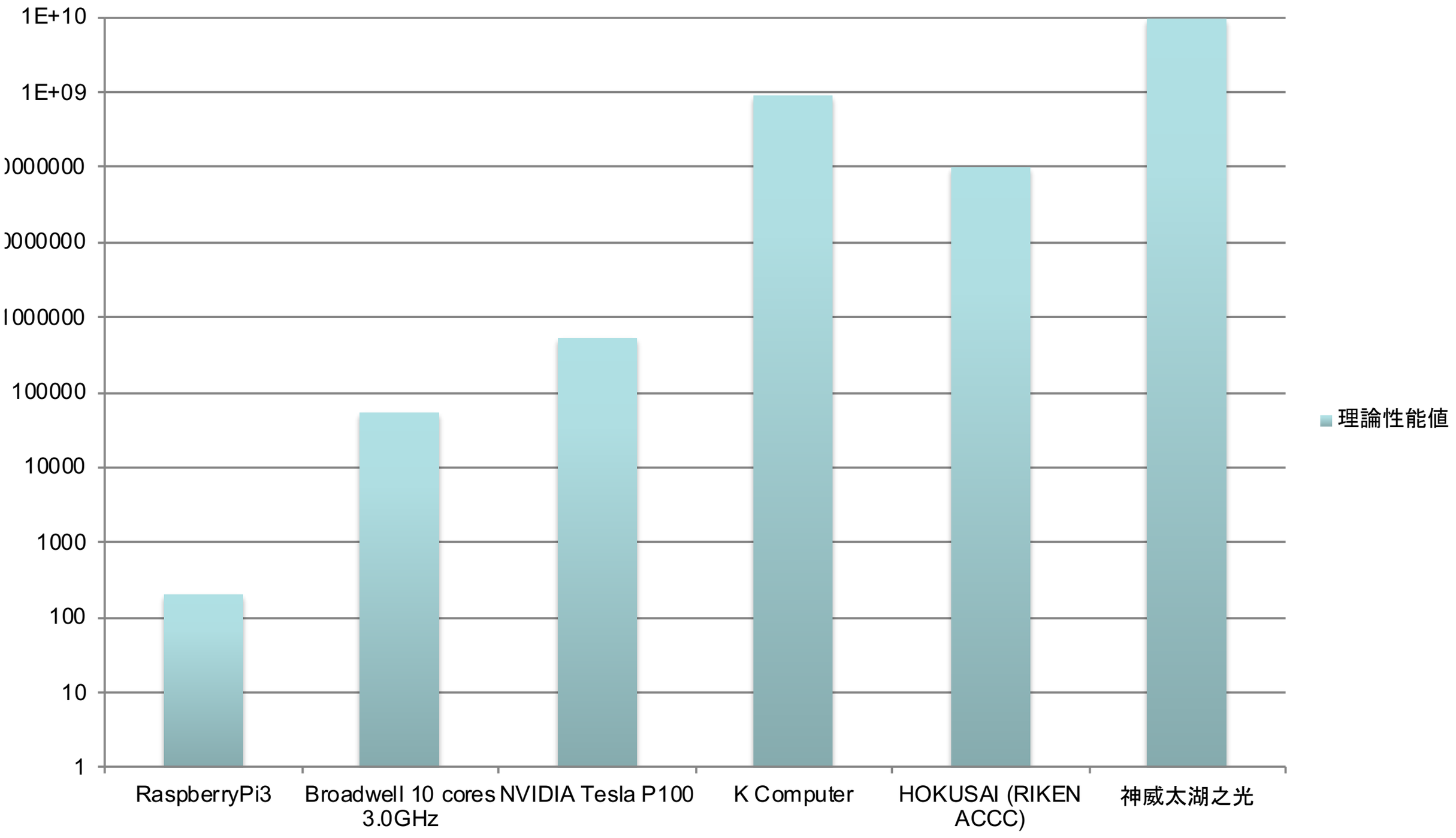

intel
Broadwell


京
京
©RIKEN


FUJITSU


神威

様々なマシンの性能



13:15

理論性能値
or 理論計算バンド幅の調べ方

- 通常理論性能値、ピーク性能値、カタログ性能値などと呼ばれる
    - CPUなどの処理装置が一斉に演算をした場合、倍精度演算で何FLOPSでるか。
        - 演算に意味はなくて良い。
            - 結果を外に出す必要さえない。
        - 物理コア数が重要 (これらが演算を実際に行うから)
            - hyper threadingは理論性能値を上げない
- パフォーマンスチェックすることは、本当に正しいCPUを買ったかどうかを確認するためにも重要

理論性能値

or 理論計算バンド幅の調べ方

- 物理CPU/物理コア/論理CPUの数の調べ方 on Linux

```
$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    2
Core(s) per socket:    6
Socket(s):             2
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 44
Model name:            Intel(R) Xeon(R) CPU           X5680  @ 3.33GHz
Stepping:              2
CPU MHz:               3326.000
CPU max MHz:           3326.0000
CPU min MHz:           1596.0000
BogoMIPS:              6666.50
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              12288K
NUMA node0 CPU(s):     0-23
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi
mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl
xtopology nonstop_tsc aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid dca
sse4_1 sse4_2 popcnt aes lahf_lm epb kaiser tpr_shadow vnmi flexpriority ept vpid dtherm ida arat
```

理論性能値計算式

- 周波数 x (コア数 / ソケット) x ソケット数 x 4 (SSE4.2)
- **3.33 * 6 * 2 * 4 = 159.84GFlops**

次スライド

理論性能値
or 理論計算バンド幅の調べ方

- CPU毎の1cycleでの倍精度浮動小数点演算数
- 4 FLOPS/clock : SSE4.2 (Nehalem 以降)
- 8 FLOPS/clock : AVX (128bit wide; Sandy Bridge 以降)
- 16 FLOPS/clock : AVX2 (FMA 256bit wide; Haswell以降)
- 4 FLOPS/clock : AMD K10
- 8 FLOPS/clock : AMD Bulldozer
- 8 FLOPS/clock : AMD Ryzen

https://stackoverflow.com/questions/15655835/flops-per-cycle-for-sandy-bridge-and-haswell-sse2-avx-avx2
にまとまった表があり参考になる。正しくはIntelやAMDなどベンダのサイトで仕様を調べる必要がある。

理論性能値
or 理論計算バンド幅の調べ方

- 意外とまとまった情報はなく、断片的な情報から計算する必要がある事が多い。

- 最近はTurboBoostなどで周波数が変わったりするので状況に応じて変化する。

- https://www.arcbrain.jp/support/Intel/Xeon_Scalable_Processor/list/all/ にまとまっている

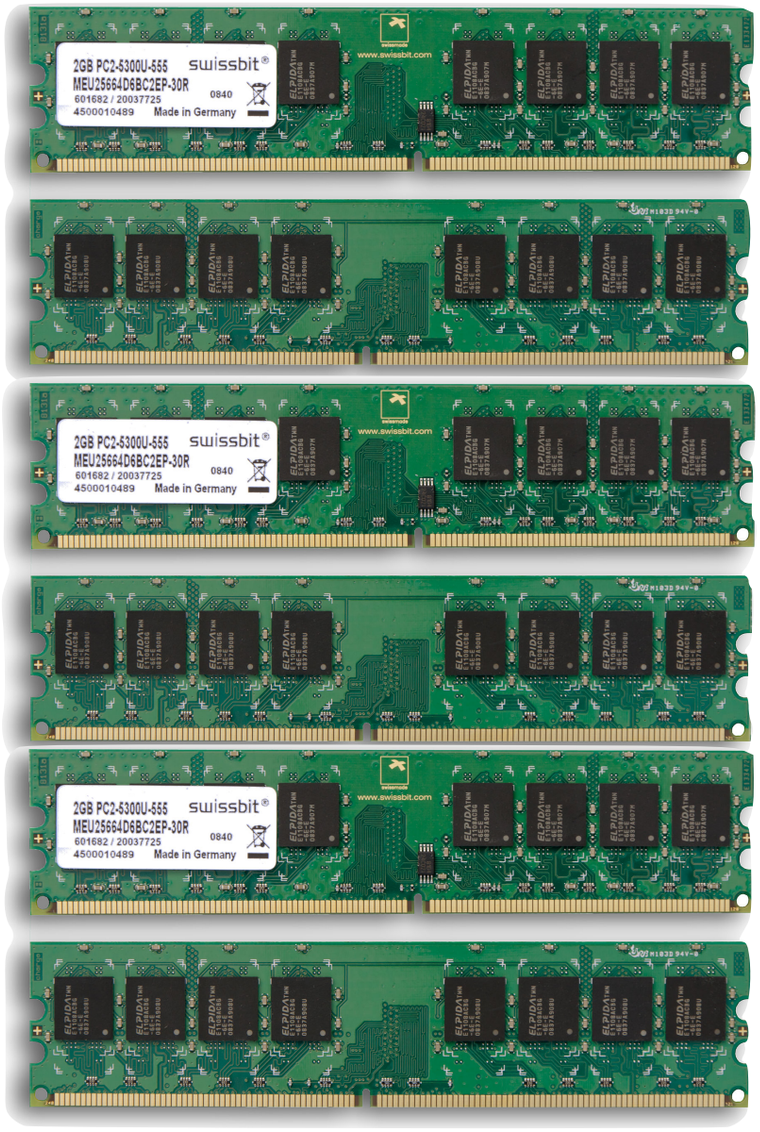
- https://web.archive.org/web/20150823174551/http://www.intel.com/support/jp/processors/xeon/sb/cs-020863.htm Intelのサイトにもまとまっていたが、今は消えている。archive.orgより。

ボトルネック:
メモリバンド幅について

13:20

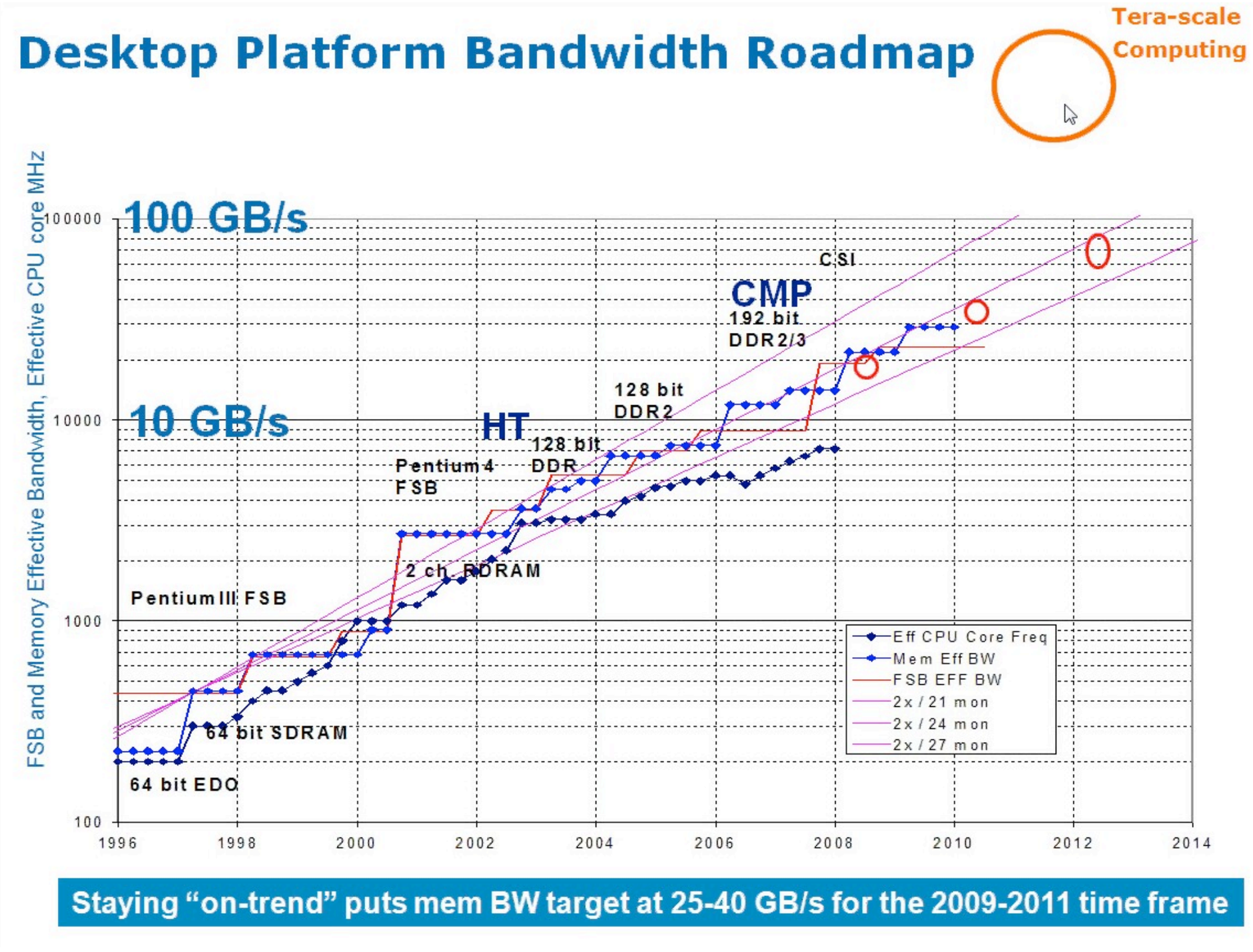
メモリバンド幅

CPUは一秒間に何バイトメモリを読み書きできるか? = メモリバンド幅



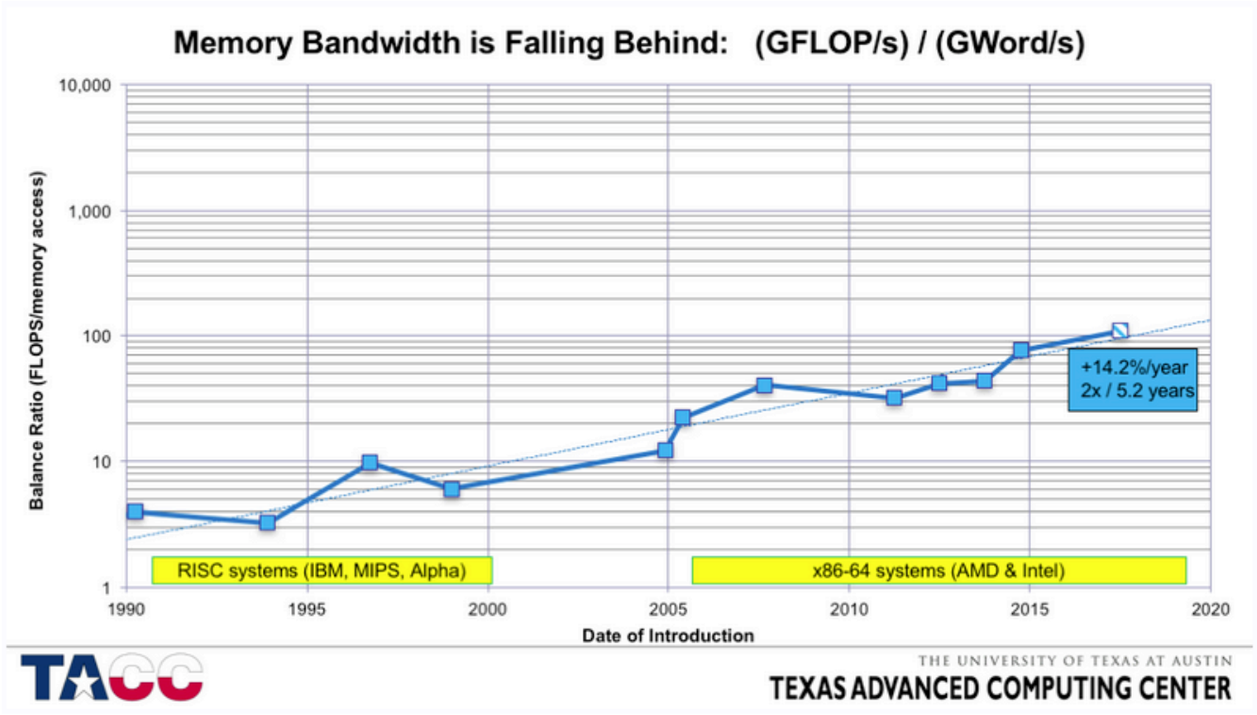
何MB/GB/TB/s

メモリバンド幅(メモリのスピード)も年々高速になってきている



メモリのバンド幅のトレンドについて

- 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

メモリバンド幅の理論幅を調べる

- 一般にメモリの動作周波数 x チャネル数 x 8
  - チャネル数はDIMMの枚数, DIMMの刺し方、CPU, ボードなどによって変わる
  - Xeon Gold 6244の例:
  - 2933MHz (DDR4) x 6 ch x 8 / 1024 = 137GB/s
  - Xeon(R) Gold 6148のHOKUSAI BWMPC@理研の場合
    - 2666 MHz x 6 ch x 8 / 1024 = 125GB/s
- プロセッサの仕様書にもある
  - X5680の場合 32GB/s
  - https://ark.intel.com/content/www/jp/ja/ark/products/47916/intel-xeon-processor-x5680-12m-cache-3-33-ghz-6-40-gt-s-intel-qpi.html
  - 1333 MHz x 3 ch x 8 / 1024 = 31GB/sとだいたい同じ値になった

https://www.arcbrain.jp/support/Intel/Xeon_Scalable_Processor/Common_Specification/

メモリバンド幅の測定

- 実マシンでメモリバンド幅を測定するにはstreamを使うのが良い

http://www.cs.virginia.edu/stream/ref.html

  - Xeon(R) Gold 6148のHOKUSAI BWMPCの場合

    - 2666 MHz x 6 ch x 8 / 1024 = 125GB/s

116GB/sと大体ピーク性能が出ている

```
$ wget https://www.cs.virginia.edu/stream/FTP/Code/stream.c
...
$ gcc -O2 -fopenmp -DSTREAM_ARRAY_SIZE=100000000 stream.c -o stream.100M
$ ./stream.100M
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 100000000 (elements), Offset = 0 (elements)
Memory per array = 762.9 MiB (= 0.7 GiB).
Total memory required = 2288.8 MiB (= 2.2 GiB).
Each kernel will be executed 10 times.
The *best* time for each kernel (excluding the first iteration)
will be used to compute the reported bandwidth.
-------------------------------------------------------------
Number of Threads requested = 40
Number of Threads counted = 40
-------------------------------------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 11111 microseconds.
(= 11111 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:          116312.6     0.013786     0.013756     0.013835
Scale:         113888.6     0.014084     0.014049     0.014110
Add:           131608.4     0.018261     0.018236     0.018292
Triad:         131326.8     0.018290     0.018275     0.018311
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-------------------------------------------------------------
```

タイマーがしょぼいのと
マシンの状況で若干上下する(?)

Bytes per FLOP, B/F

- 1回演算するのに何バイトメモリ転送ができますか?
- マシンの性能を測る指標
  - Bytes per FLOPが低いマシンだと、演算バンド幅が広い (=計算が得意)
    - 例: Xeonなど最近の普通のCPU, GPU一般
- Bytes per FLOPが高いマシンだと、メモリバンド幅が広い (=計算は不得意なので、結果はメモリに書いておくとよい)
  - 例: かつてのベクトル型スパコン SX-9, SX-8など

ハードのBytes per FLOPの例

- 京コンピュータ
  - 演算バンド幅 (CPUの速度): 128GFlops
  - メモリバンド幅: 64GB/s
  - Bytes per FLOP = 64/128 = 0.5 B/F
- Intel® Xeon® Processor E5-2699 v3 (18 cores, 2.3GHz)
  - 演算バンド幅 (CPUの速度) ：16 FLOPS/Clock × 2.3 GHz × 18コア =662.4GFlops
  - メモリのバンド幅68GB/s
  - Bytes per flop = 68 / 662.4 = 0.103 B/F
- NVIDIA P100 for NVLink-Optimized Servers
  - 演算バンド幅 (GPUの速度) : 5.3 TFlops
  - メモリバンド幅 732 GB/s
  - Bytes per flop = 732 / 5300 = 0.138 B/F　<—- 意外とB/F値は小さい
- NVIDIA V100 NVLINK B/F = 900 / 7800 = 0.115 B/F <—- 意外とB/F値は小さい

Bytes per FLOP, B/F

- 一回計算するのに何バイト読み書き必要?
  - 科学技術ソフトウェアの特性を測る指標
  - Bytes per FLOPが小さいソフトウェア：演算バンド幅を要求する (=計算を主に行うソフトウェア)
    - 例: 分子動力学、重力多体問題, LINPACK…
- Bytes per FLOPが大きなソフトウェア：メモリバンド幅を要求する
  - 例: 流体解析、連続体解析、格子QCDなど連立一次方程式を解くプログラム一般

BLASのB/Fの例：DAXPY

- daxpyについて:x, yは n 次元ベクトル、a はスカラー。これから以下の演算をする

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

- このとき、bytes per flopはいくつになるか。
  - 2n回の浮動小数点演算
  - 3n回のデータの読み書き
    - x, yを読んで、y[i]に書く。
  - 倍精度一つで8bytesなので、24bytes / 2 Flops = 12 bytes/ flop

    daxpy(ベクトルの和)のbytes per flopは12となる

BLASのB/F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。
  - y ← αA x +β y
  - $2n^2+2n$回の浮動小数点演算
  - A x : 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
  - 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が大きいところで
  - $(n^2+3n) * 8 / (2n^2+2n) = 4$

dgemv(行列ベクトル積)のbytes per flopはn->∞で4

BLASのB/Fの例：DGEMM

- A, B, Cは ｎ x ｎ の行列として、積を計算する。
  - C ← A * B + C
- $2n^3$回の浮動小数点演算
  - A * B : n^3回積 ,$n^2(n-1)$回和
  - (A * B) + C : n^2回和
- $4n^2$回のデータの読み書き
  - A, B, C：$3n^2$読み
  - (A * B) + C : n^2書き
- 倍精度一つで8bytesなので nが大きいところで
  - $4n^2 / (2n^3)$ * 8 -> 16/n

dgemm(行列の積)のbytes per flopsはn->∞ で 0

BLAS/LAPACKのLevelとB/F

- Level 1：大きい　メモリバンド幅を要求する
- Level 2：大きい　メモリバンド幅を要求する
- Level 3：小さい (漸近的に0)　演算バンド幅を要求する
- (LAPACK) 一概に言えない

高いB/Fマシンがほしい…

- CPUの性能向上 = 速いコンピュータ、ではない
- アプリケーション、マシンの両方のB/Fが近いとコスパがよい。
  - CPUだけ速くてもメモリバンド幅が小さい場合:
    LINPACKは速くとも流体解析は遅くなる。
- しかしながら、メモリの性能は上昇しなさそうなので、アルゴリズムの変更が望ましい
- ハードやさんのよくあるセリフ: 高いB/Fは甘え

GPUについて

GPUとは？GPGPU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。
- 主にPCI expressバスにつなげる形で存在。
  - PCI express バス 32GB/s の転送速度がボトルネック
  - NVLink : 160GB/s

13:35

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

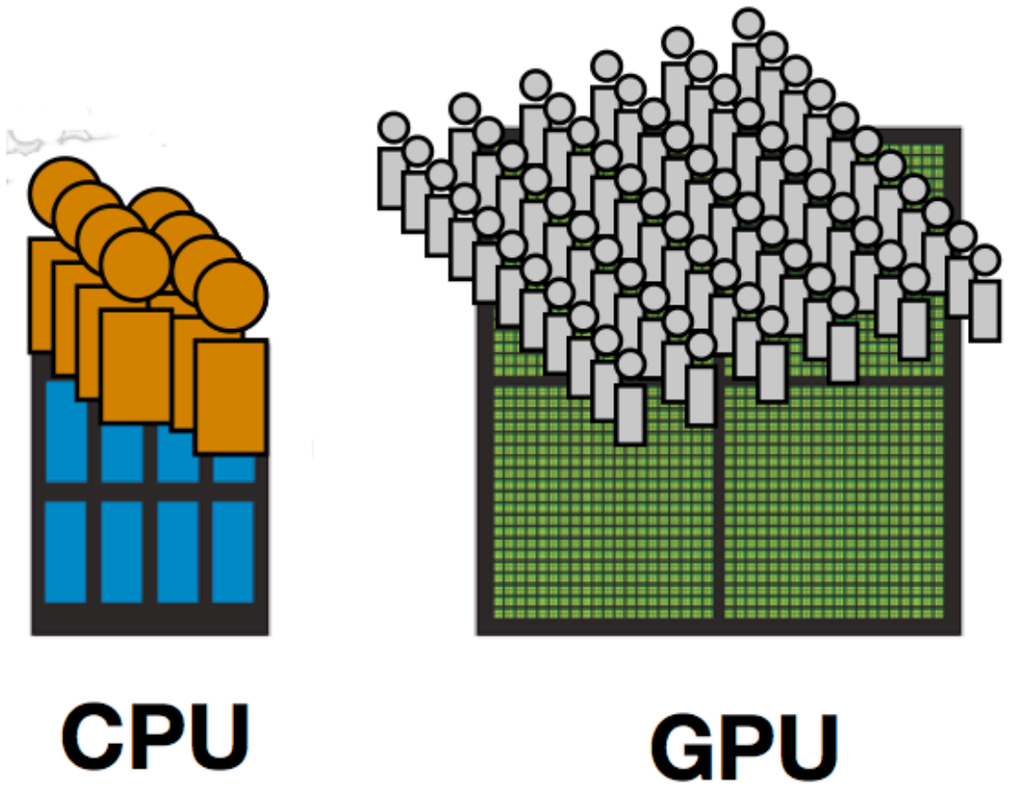
1.データを送る

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。

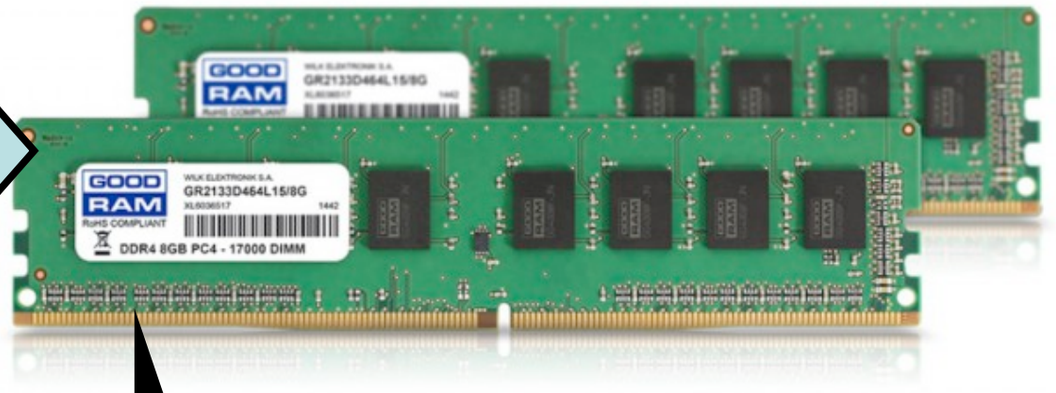


- 画像処理だと沢山独立した点に対して似たような処理をする
- CPUみたいには複雑な処理はできないが、工夫次第で色々可能

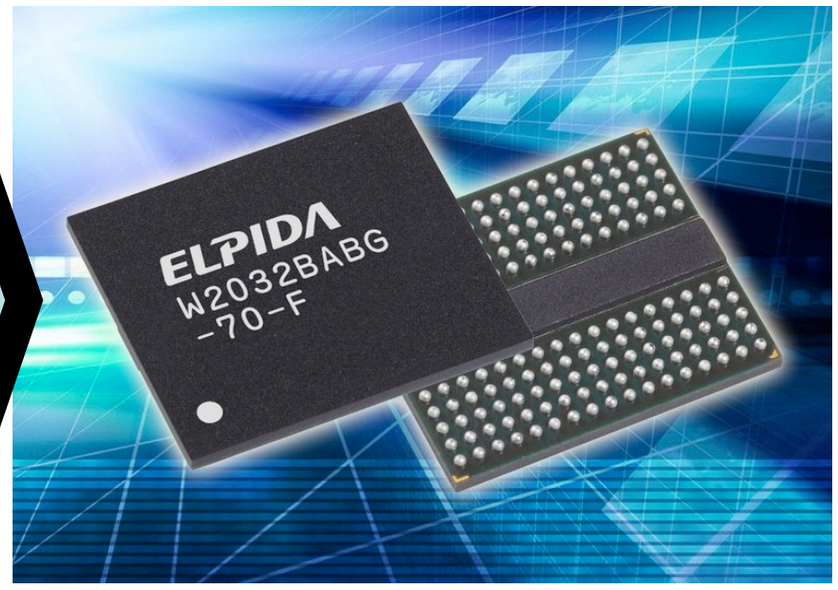
GPUはどうして高速か? Part II

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
- Intel 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。

GPUの使い方?

- B/Fを計算すると意外と低く、最近のマルチコアCPUとそんなに変わらない。しかしながら…
  - メモリバンド幅の絶対値はXeonなど普通のCPUと比べて広い。
    - 理論ピークバンド幅: Intel Xeon(R) Gold 6148 : 125GB/s, NVIDIA Tesla V100 900GB/s
- 高性能メモリマシンとして使うのが良い??
  - GPUの演算性能を無視して、B/Fの大きなアプリケーションを動かすべきなのか(?)
- 電力性能マシンとして使うのがよい?

13:40

ここまでのまとめ

- ボトルネックの概念：一番遅い(弱い)ところで全体のパフォーマンスが決まる
- 2大ボトルネック：演算バンド幅 (CPUの速度) + メモリバンド幅 (メモリの速度)
- 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
  - メモリも高速になってきているが、CPUに比べて鈍い。
- Byte per flop or B/F 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? 計算するソフトか?
  - マシンのB/F、アプリケーションのB/Fの計算の仕方
  - トレンド: B/Fの小さいマシンに移行、アルゴリズムの書き直しが重要。
- 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。
    - ただしB/Fでみるとあまり CPUと違いがなくなってきた…
  - 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

高速なBLAS LAPACKを使うには

dgemvとdgemmを例にとって

高速なBLAS、LAPACKを使う

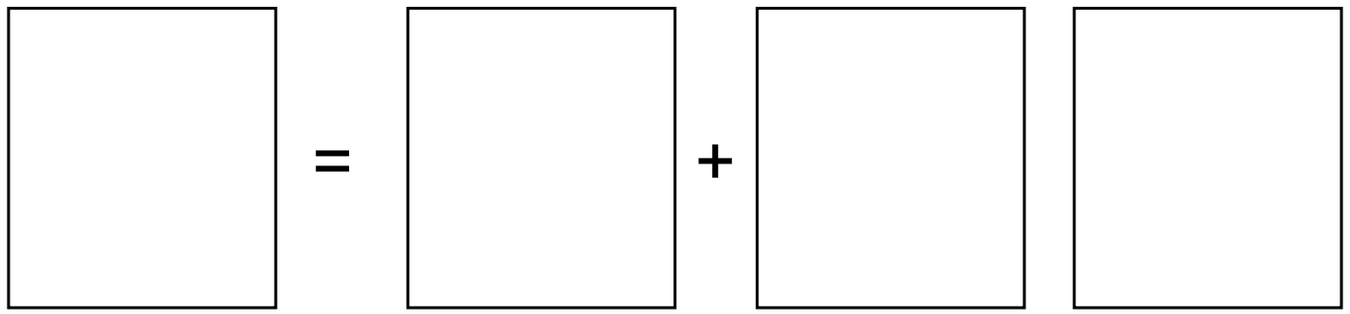
- 二つの典型的な例
  - DGEMM (行列-行列積)　　演算バンド幅が　ボトルネック
  - DGEMV (行列-ベクトル積)　メモリバンド幅が　ボトルネック
- 高速なBLAS, LAPACK : OpenBLAS
  - Octaveでreference BLASとatlasとOpenBLASをベンチマーク on Ubuntu
- どうして高速なのか?
- GPUでcuBLASを使うDGEMM

最近は特に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

DGEMM 行列-行列積

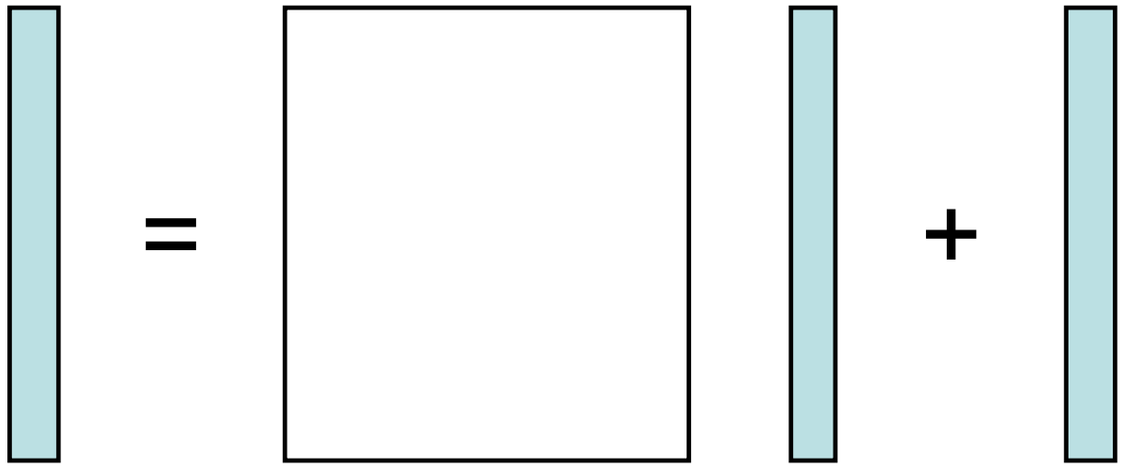
- DGEMM (行列-行列積)　　演算バンド幅が　ボトルネック
- $C \leftarrow \alpha AB + \beta C$



- データの量は$O(n^2)$: n は行列のサイズ、(今回はn=m=k)
- 演算量は$O(n^3)$
- Bytes / flop は O(1/n)
  - 大きなサイズのDGEMMは演算スピードが律速となる。

DGEMV : 行列ベクトル積

- DGEMV (行列-ベクトル積) メモリバンド幅が ボトルネック
- $y \leftarrow \alpha Ax + \beta y$



- 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。

13:45

高速なBLAS、LAPACKの力を知る

- 行列-行列積 DGEMM, 行列-ベクトル積 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：お手本となる実装。これを元に高速なBLASが作られる
  - Ubuntu 標準 ATLAS
  - OpenBLAS (現時点で最高速の部類)
- Octave
  - Matlabのフリーのクローン
- 内部でBLAS, LAPACKを呼び、計算もしやすい。
- 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)

- Ubuntu 16.04を使う。端末から
- Octave + reference BLASのインストール

```
$ 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

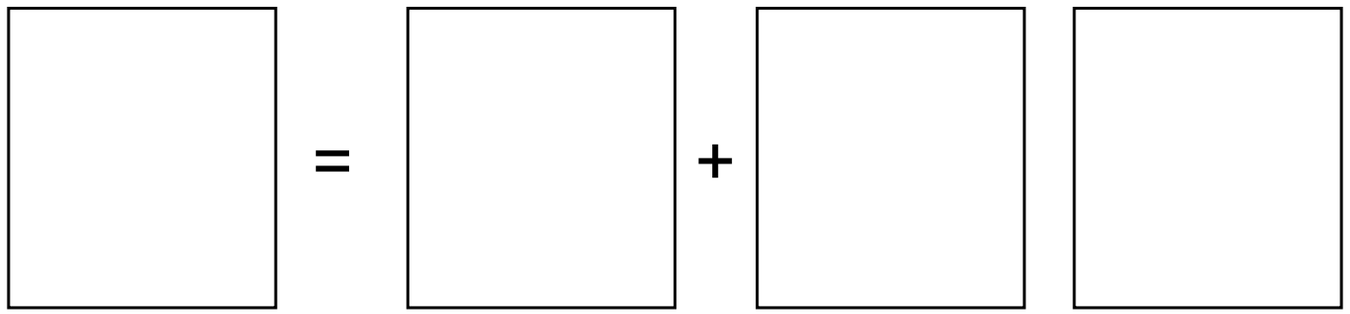
```
$ sudo update-alternatives --config libblas.so.3
There are 3 choices for the alternative libblas.so.3 (providing /usr/lib/
libblas.so.3).
Selection    Path                                            Priority   Status
------------------------------------------------------------
* 0            /usr/lib/openblas-base/libblas.so.3             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

- DGEMM (行列-行列積)　　演算バンド幅が　ボトルネック
- $C \leftarrow \alpha AB + \beta C$



- データの量は$O(n^2)$: n は行列のサイズ、(今回はn=m=k)
- 演算量は$O(n^3)$
- Bytes / flop は O(1/n)
  - 大きなサイズのDGEMMは演算スピードが律速となる。

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)

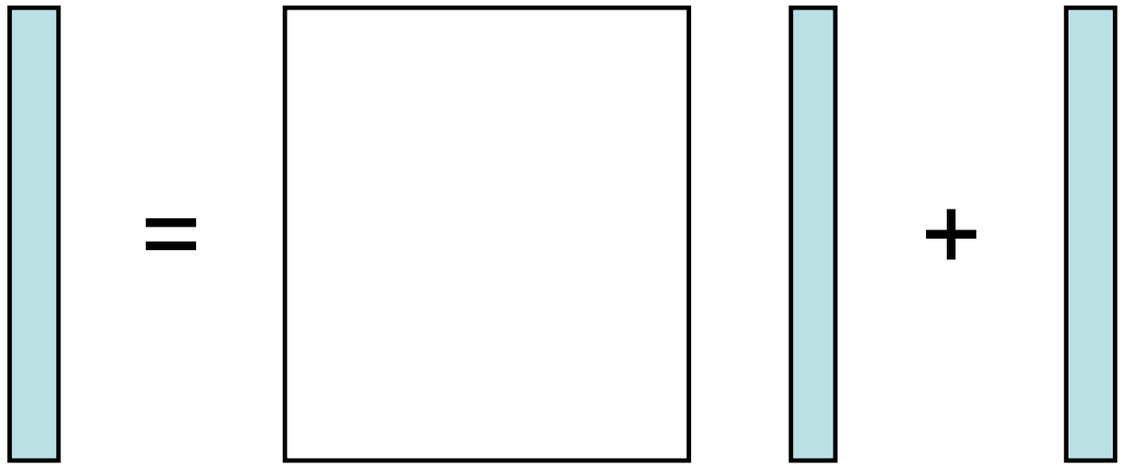
GFLOPS = 4.3942 (Cortex A53 (ARMv8))

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 ボトルネック
- $y \leftarrow \alpha Ax + \beta y$



- 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
- 34.1GB/s
  - 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
- ARM53 (ARMv8, ODROID C2) のメモリバンド幅推算
- 3.56GB/s
  - 2GB 32bit DDR3 912MHzSamsung K4B4G1646Dより推算する
  - 恐らく912MHz x 4 / 1024 = 3.56GB/s だと思われる。
  - Stream実測値 triad 3.6GB/s copy 4.2GB/s

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))
- Xeon E5-2603での理論性能値 13.8%
  - 34.1 (GB/s) / 4 (B/F) = 8.53 GFlops
  - メモリ性能比 13.8 % (=1.1751/8.53 * 100)
  - CPUの演算器としての性能は115GFlops、それと比較すると1%
- ODROID C2での理論性能値 21.0%
  - 3.56 (GB/s) / 4 (B/F) = 0.89 GFlops
  - メモリ性能比 20.98 % (= 0.18676 / 0.89 * 100)
  - CPUの演算器としての性能は6GFlops、それと比較すると14.8%

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
  - 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 = 5.8006 (Xeon E5-2603)
  - GFLOPS = 0.681 (Cortex A53 (ARMv8))
- Xeon E5-2603での理論性能値 68.0%
  - 34.1 (GB/s) / 4 (B/F) = 8.53 GFlops
  - メモリ性能比 68.0 % (=5.8006/8.53 * 100)
  - CPUの演算器としての性能は115GFlops、それと比較すると5.04%
- ODROID C2での理論性能値 76.5%
  - 3.56 (GB/s) / 4 (B/F) = 0.89 GFlops
  - メモリ性能比 76.5 % (= 0.681 / 0.89 * 100)
  - CPUの演算器としての性能は6GFlops、それと比較すると11.4%

13:55

ここまでのまとめ

- Reference BLASと最適化されたBLAS (OpenBLAS)を使ったベンチマークを行った。
  - DGEMM (行列-行列積)で、演算バンド幅の理論性能と比較
    - Reference BLASだと演算バンド幅の5%未満だったが、最適化されたBLASだと8-9割近く出た
  - DGEMV (行列-ベクトル積)で、メモリバンド幅の理論性能と比較
    - Reference BLASだとメモリバンド幅の1-2割が出たが、最適化されたBLASだと7-8割近く出た
    - 演算バンド幅は5-10%程度利用していた (=CPUはほとんど休んでる)

- 大きな次元での結果であることに注意
  - 小さいサイズの行列-行列積、行列-ベクトル積を大量にやる場合は話が大きく変わる…

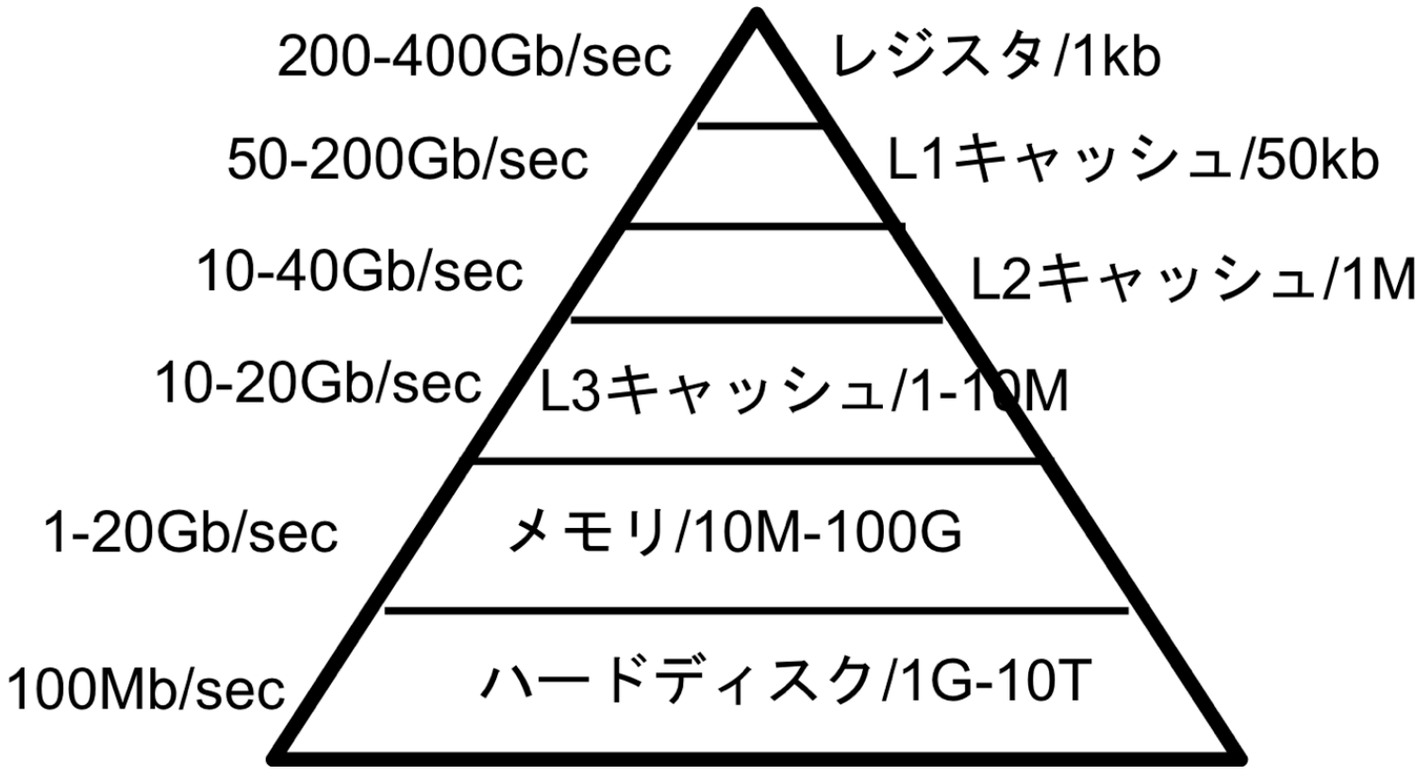
高速化の手法：DGEMMを例にして

プログラムを高速化する一般的な手法

- プログラムのボトルネックを考える
  - アルゴリズムのB/F
  - 要求演算バンド幅と、 要求メモリバンド幅
- 要求メモリバンド幅＜要求演算バンド幅
  - BF < 1
  - データの使い回しを行なっているアルゴリズム
  - データをメモリまで読み書きせずに、キャッシュにとどめておく
    - 比較的簡単、かつわかりやすい
    - 例:行列-行列積
- 要求メモリバンド幅 >= 要求演算バンド幅
  - BF >= 1
  - メモリバンド幅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、ハードウェアに頼る必要あり
    - 高速なメモリを搭載しているマシンが少なく、限界もすぐ見える
    - マルチスレッドは必須。1スレッドがメモリバンド幅すべて使うことは普通できない。
    - マシンを複数台用意して一斉に読み書きすることでメモリバンド幅を増やす

遅いメモリを効率的に使うための工夫: メモリの階層構造

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



キャッシュを利用した
データの再利用の例

14:00

典型的なチューニング: 行列のブロック化

- 要求メモリバンド幅＜要求演算バンド幅
  - データの使い回しを行なっているアルゴリズム
  - データをメモリまで読み書きせずに、キャッシュにとどめておく
- 行列-行列積のチューニング技法について
- 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B/F=0
- 別の表現:B/F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せるのではないか。

行列-行列積はピーク性能が出しやすい。
行列-行列積のチューニングを行う

行列-行列積のブロック化アルゴリズム

- 普通 C= AB という 行列-行列積は

$$C_{ij} = \sum A_{ik} B_{kj}$$

- で行うが、これそのままでは実行が遅くなる。
- 行列-行列積の、入力と結果は変えずに、アルゴリズムを変える。
- 行列のブロック化を行って積を行う

典型的なチューニング: 行列のブロック化

次の事実を使う。

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

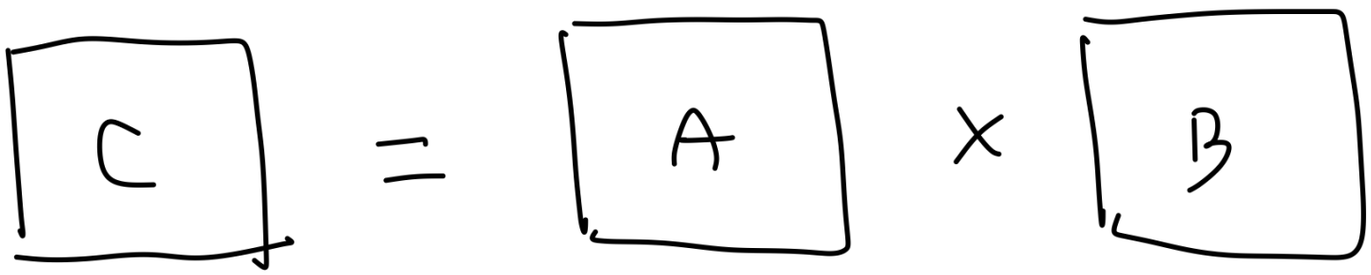
$$AB = \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1r} \\ C_{21} & C_{22} & \cdots & C_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ C_{p1} & C_{p2} & \cdots & C_{pr} \end{pmatrix}$$

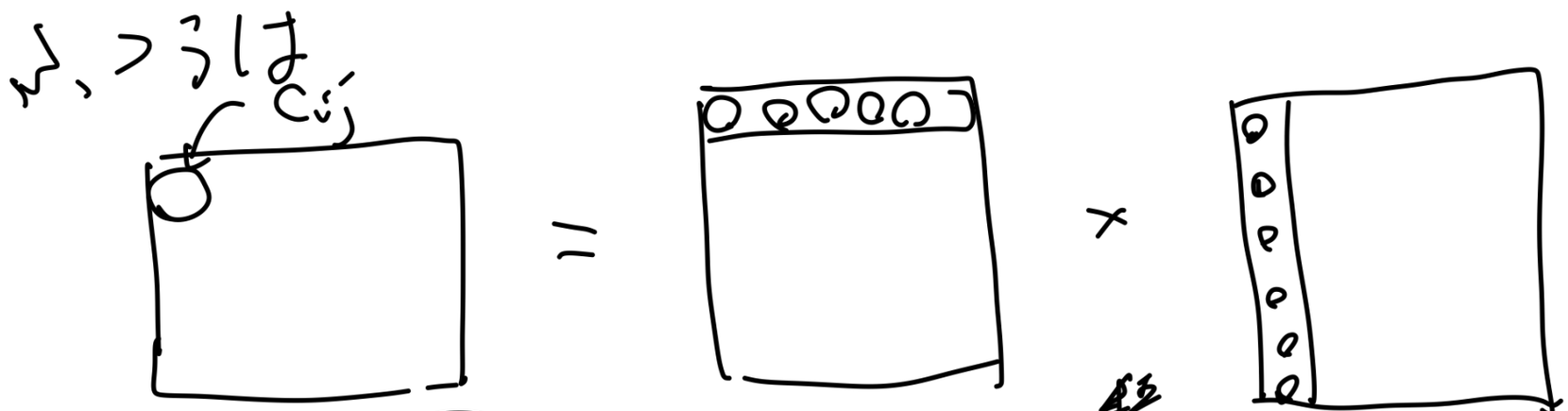
の各ブロックは

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

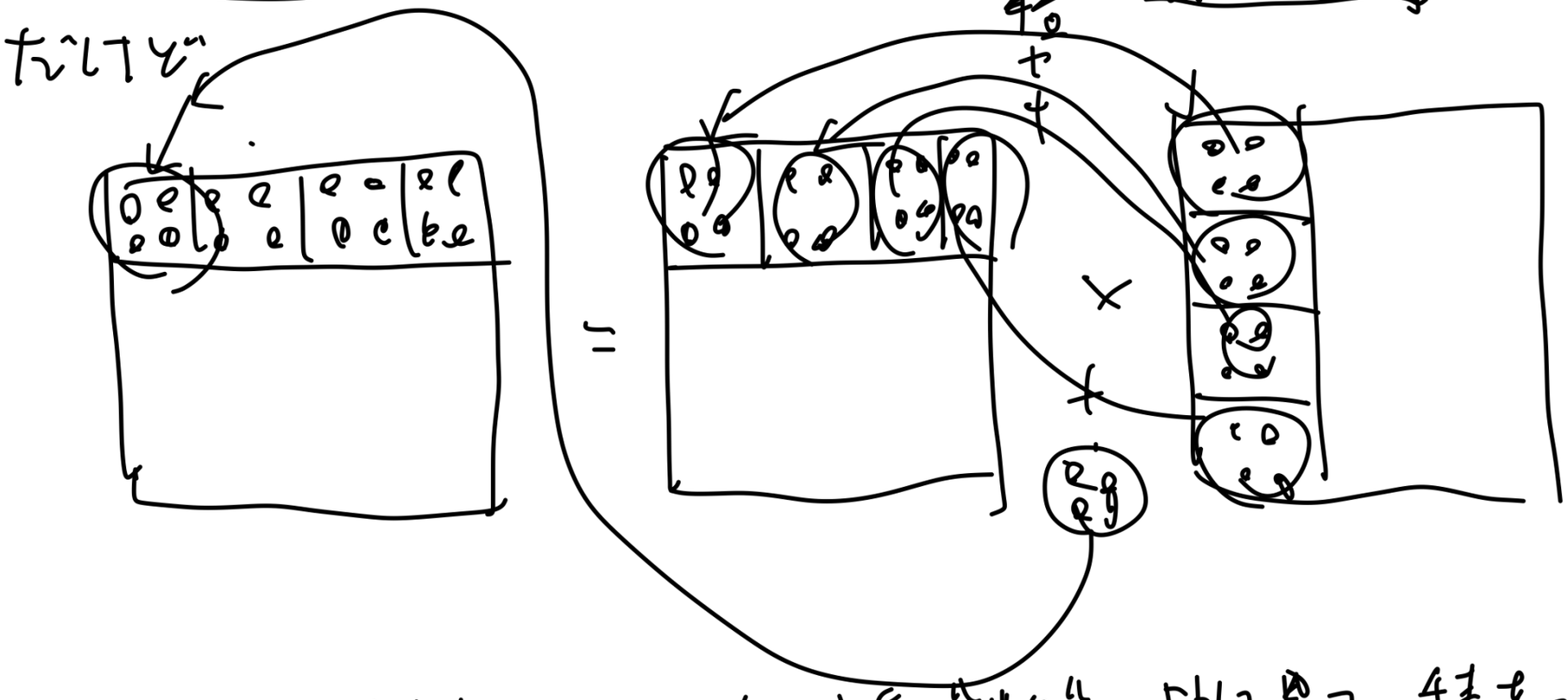
で与えられる。すなわち、区分行列の積は（適切に区分けされていれば）各ブロックをあたかも行列の成分のように見なして計算できる。

典型的なチューニング: 行列のブロック化





○は行列要素

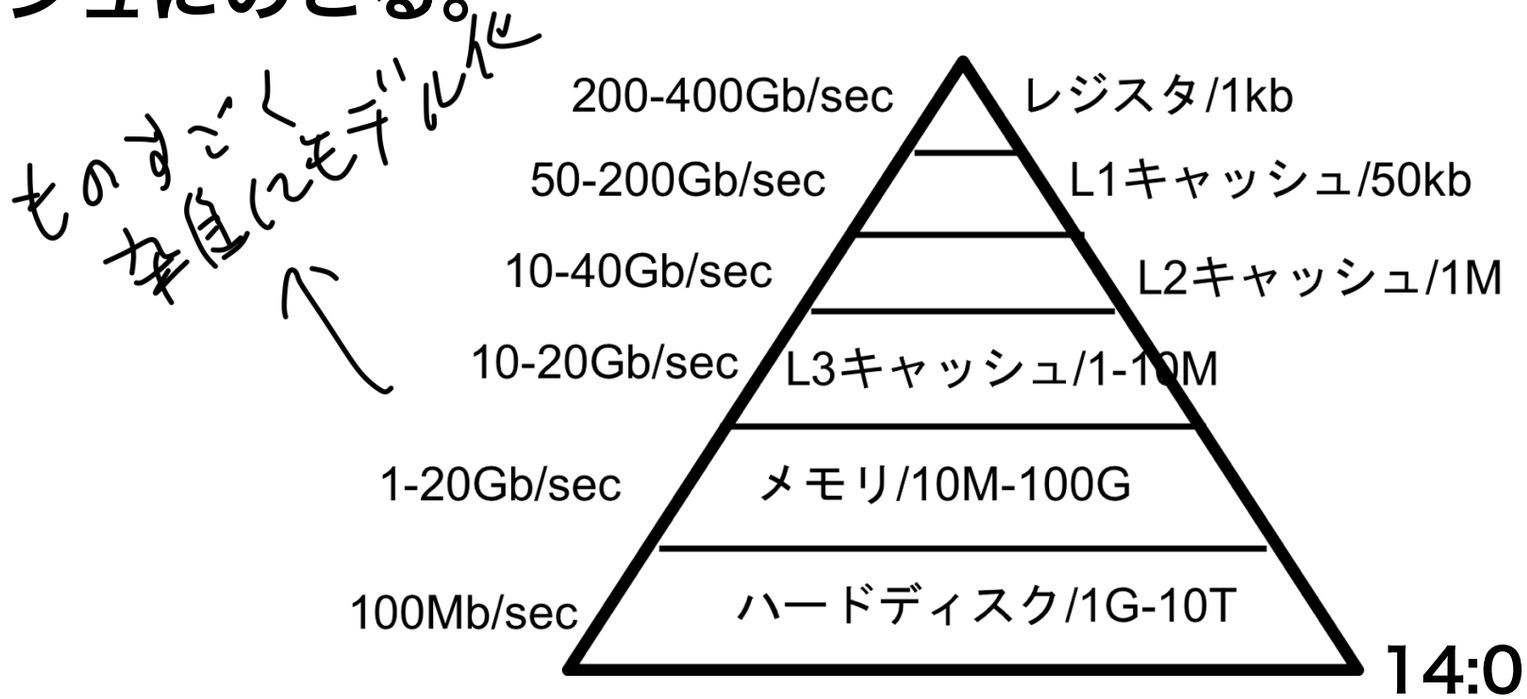


は block 行列

のようにブロック行列を"数"のように思って 積をつくってもよい

今のコンピュータ:メモリから倍精度1個とってくる間に約100回計算できる

- (さんざんこの授業でやったが)現在メモリアクセスはCPUからみると、とっても遅い。
- 大体、メインメモリから倍精度一個 (=8 bytes) とってくる間に、100回計算できる。
  - イメージ：一日分の食料をとってくるのに、100日 (=三ヵ月かかる)
- CPUの近くに、速いけど容量の少ないキャッシュが装備されている。
  - イメージ：冷蔵庫がある。冷蔵庫あけたら食料が手に入る
  - 一回手に取ったらキャッシュにのこる。



14:05

ブロック行列化とキャッシュ

- 簡単なモデルを作って、高速化の理論的裏付けを行う。

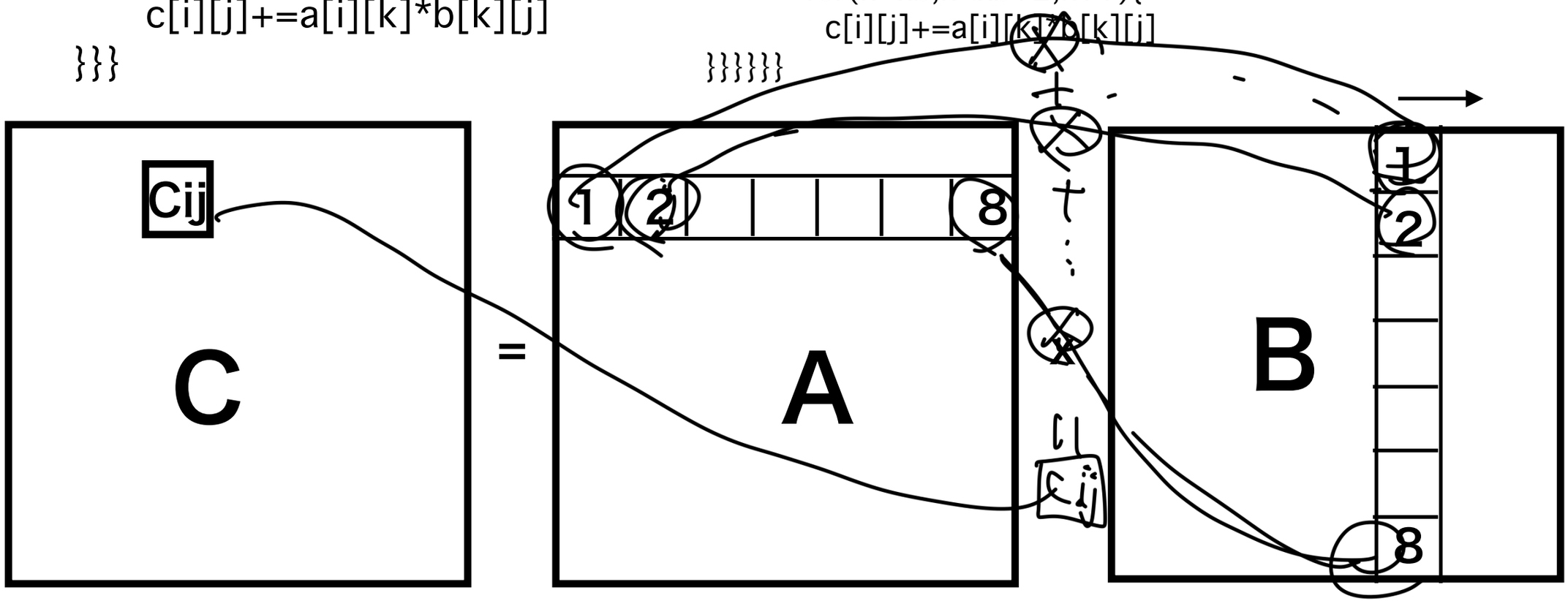
設定

- 16 x 16 の倍精度行列の掛け算を行う。C=A*B+C
- 倍精度一個、メモリ参照するには100クロックかかる。(CPU 1GHz -> 0.08GB/s)
- 倍精度一個、キャッシュを参照するのは0クロック ← ありえんけど 趣旨はかわらん
- 倍精度一演算を1クロックで処理 (CPU 1GHz -> 8GB/s)
- つまりB/F = 0.08 程度コンピュータを考えよう; 最近はB/F 0.1位が主流
- キャッシュは十分あり、(Aのブロックx 行くらい入る) 最も使ってないものから消えてゆく。
- 2x2のブロッキングを行う

```
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]
}}}
```

2x2ブロック化の実装

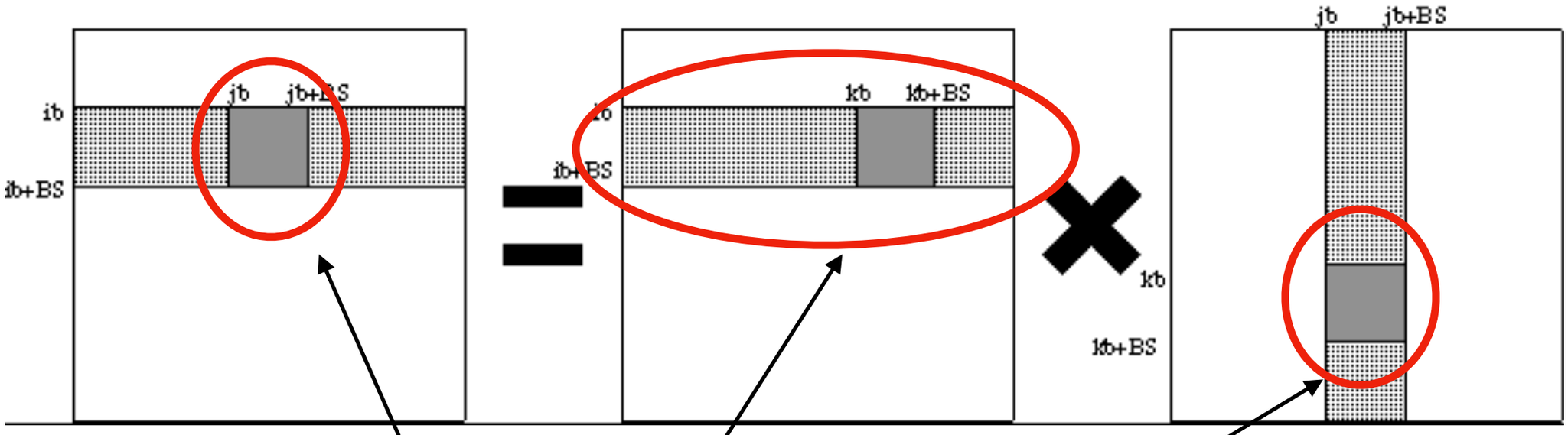
```
for(ib=0;ib<8;ib+=2){
 for(jb=0;jb<8;jb+=2){
  for(kb=0;kb<8;kb+=2){
   for(i=ib;i<ib+2;i++){
    for(j=jb;j<jb+2;j++){
     for(k=kb;k<kb+2;k++){
      c[i][j]+=a[i][k]*b[k][j]
}}}}}}
```



典型的なチューニング: 行列のブロック化

どうやったらメモリキャッシュに載るか？

- メモリアクセスすれば自動的に載る。
- キャッシュに乗ったら、アクセスは0クロックで可能
- あまりアクセスされなければキャッシュから消される -> 乗せる部分は消されない ようにプログラムを組む



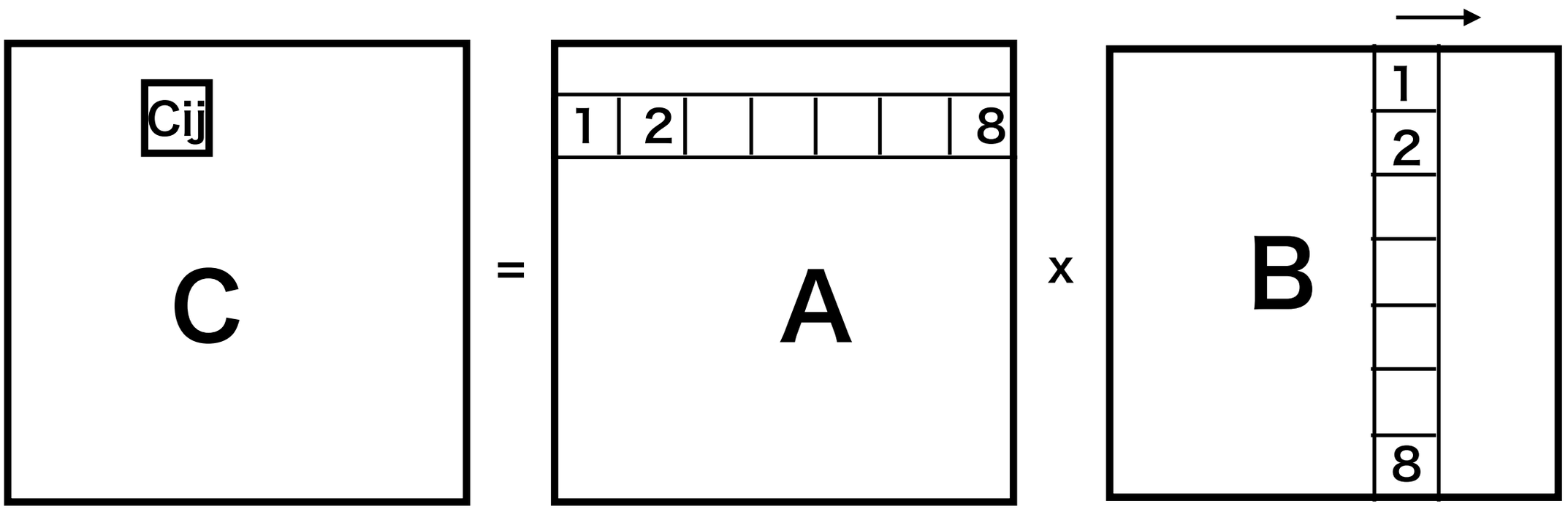
キャッシュに載せる部分

ブロック行列化とキャッシュ:一般化

1. キャッシュは空だとする
2. Aのブロック行列1, 2, …, 8 とBのブロック行列 1, 2, 3, …, 8 を設定する。
3. Ci,j ブロックは Cij=A1xB1 + A2xB2 + … + A8xB8のように行う。
4. A1, A2, ... A8とCijのブロック行列をキャッシュに残る
5. 読み出しには (2*2)* 16*100 = 6400 クロックかかる。
6. 演算は2*(2*2*2) * 8 = 128 クロック (＝一個ブロックあたり$2n^3$)

ここまでは全然スピードが出せない。

7. Bの次の列にうつる。A,Cのブロックはキャッシュに存在。Bのブロック再読み出し。
8. ブロックの演算は16クロック Bのブロックの読み出しは 400(2*2*100)クロック
9. Cijブロックあたり(400-16)*8クロック＝3072クロック、約2倍高速化される

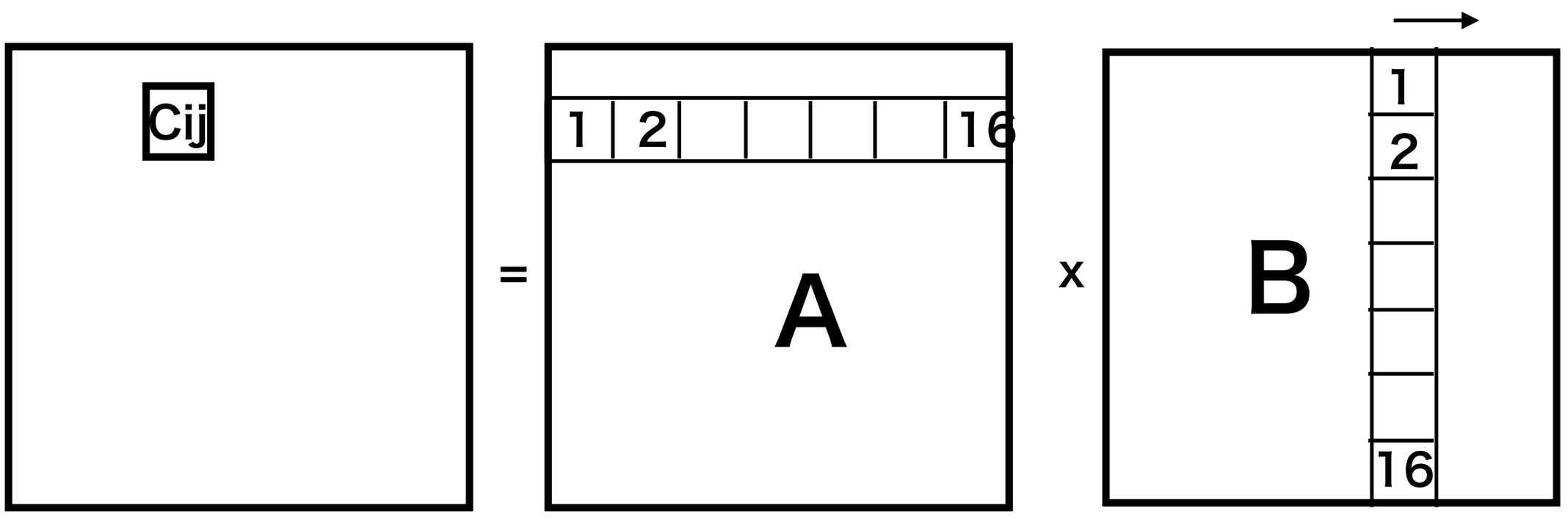


1024x1024の倍精度行列、ブロックサイズ64で行い、キャッシュは十分あるとして もう一度

1. キャッシュは空だとする
2. Aのブロック行列1, 2, …, 16 とBのブロック行列 1, 2, 3, …, 16 を設定する。
3. Ci,j ブロックは Cij=A1xB1 + A2xB2 + … + A16xB16のように行う。
4. A1,A2,A3...A16,Cijはキャッシュに残る
5. 読み出しには (64*64)*16*2*100 =13107200クロックかかる。
6. 演算は2*(64*64*64)*16 = 8388608 クロック (＝一個ブロックあたり$2n^3$)

ここまでは全然スピードが出せない(1.56倍メモリよみだしにかかる)。

7. Bの次の列にうつる。A,Cのブロックはキャッシュに存在。Bのブロック再読み出し。
8. Ai x Bj ブロックの演算は524Kクロック Bjのブロックの読み出しは409K(64*64*100)
9. ブロックの演算量が読み出しスピードを上回った -> 演算性能ピークがでる!!



行列行列積でのブロック行列化と

キャッシュサイズ最適化

- ブロック行列の演算量: $2n^3$ (Clock)
- ブロック行列読み込み時間 : 8 * n^2 B / 0.08 (B/F) = $100n^2$ (CLOCK)
  - 倍精度1個= 8 bytes, 0.08 = 8 / 100 B/F (= 一個倍精度とるときに100回演算できる)
- $2n^3 > 100 * n^2$ならば、演算量が読み込みより時間かかるようになる。
- n>50 でブロックサイズ50以上なら演算時間がメモリアクセス時間を上回る
  - なので先程はn=64のブロックサイズでokだった
- 基本DGEMMの高速実装のアイディアはキャッシュを有効活用するためのブロック化。

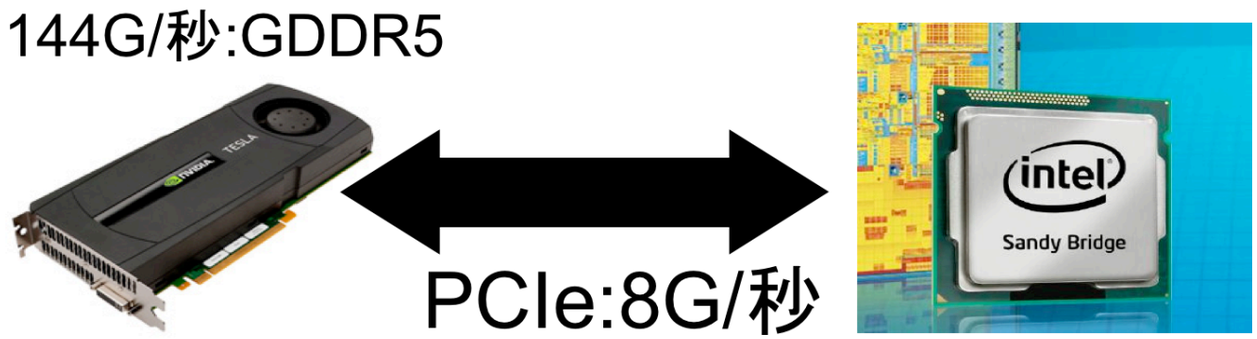
14:10

GPUでのBLASの使い方
+DGEMMベンチマーク

GPUでのBLASの使い方

- GPUはPCとは別のコンピュータ
- PCIeというバスでつながっているが、この転送速度が遅い

- 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とは何か/BLASとの違いと特徴

- CUDAで書かれた、NVIDIA社製GPU向けに加速されたBLAS
- ソースコードには変更が必要。
  - 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
  - 難しくはないが機械的にはできないので、コストはかかる
  - すべてをGPUで処理すると非常に速くなる
- 変更なしでも一応使えるが、大きな行列を扱わないとむしろ遅くなる
  - 例: 行列ベクトル積はむしろ遅くなる
    - メモリバンド幅 > PCIe-CPUバス幅より、行列からGPUへ転送するのがボトルネックになる

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
- A, B, Cを行列とし、α, βをスカラーとして

C←αAB＋βC を行う。

- 前回と同じように3x3の行列A,B,Cを下のように設定した
- スカラは、α=3.0, β=-2.0とした。

$$A=\begin{pmatrix}1 & 8 & 3\\ 2 & 10 & 8\\ 9 & -5 & -1\end{pmatrix}\quad B=\begin{pmatrix}9 & 8 & 3\\ 3 & 11 & 2.3\\ -8 & 6 & 1\end{pmatrix}\quad C=\begin{pmatrix}3 & 3 & 1.2\\ 8 & 4 & 8\\ 6 & 1 & 2\end{pmatrix}\qquad \alpha AB+\beta C=\begin{pmatrix}21 & 336 & 70.8\\ -64 & 514 & 95\\ 210 & 31 & 47.5\end{pmatrix}$$

14:10

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

14:15

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);
```

GPUで行列-行列積!!

```
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;
}
```

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
```

コンパイルにはaccelに入る必要がある

```
[maho@upc0000 ~]$ gcc dgemm_demo.cpp  -I/usr/local/cuda/include -L/usr/local/cuda/lib64  -lcudart -lcublas
[maho@upc0000 ~]$ cat test.sh
```

コンパイル

```
#!/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 ~]$
```

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まで変えた場合のベンチマークをとった。
- 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の通信も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

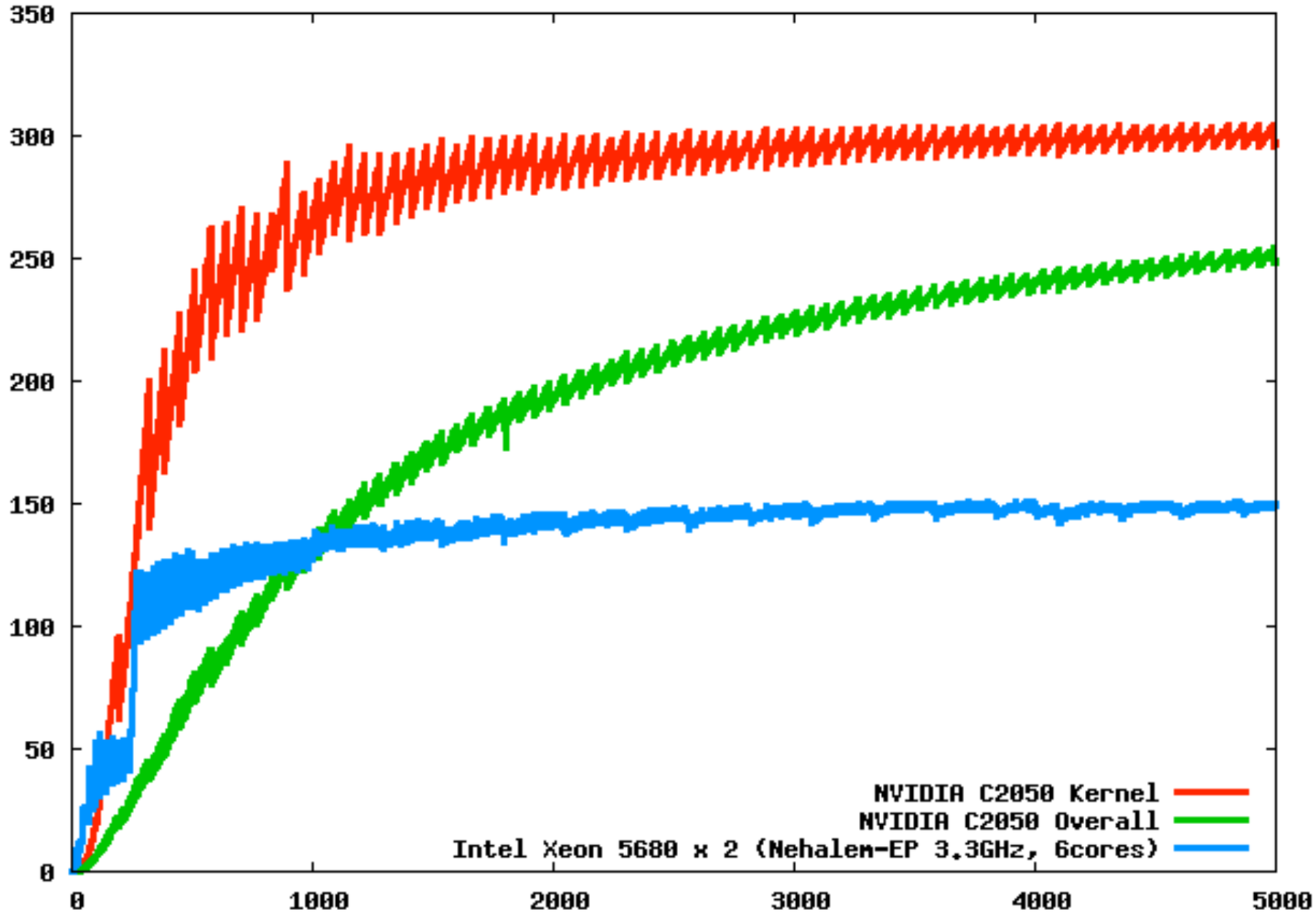
cuBLASでの行列-行列積 (VII)

横軸: 行列の大きさ n (m=k=n)

縦軸: 行列-行列積のGFLOPS

詳細は前スライド

どうして下図のようなグラフになるか、考察してみよう。

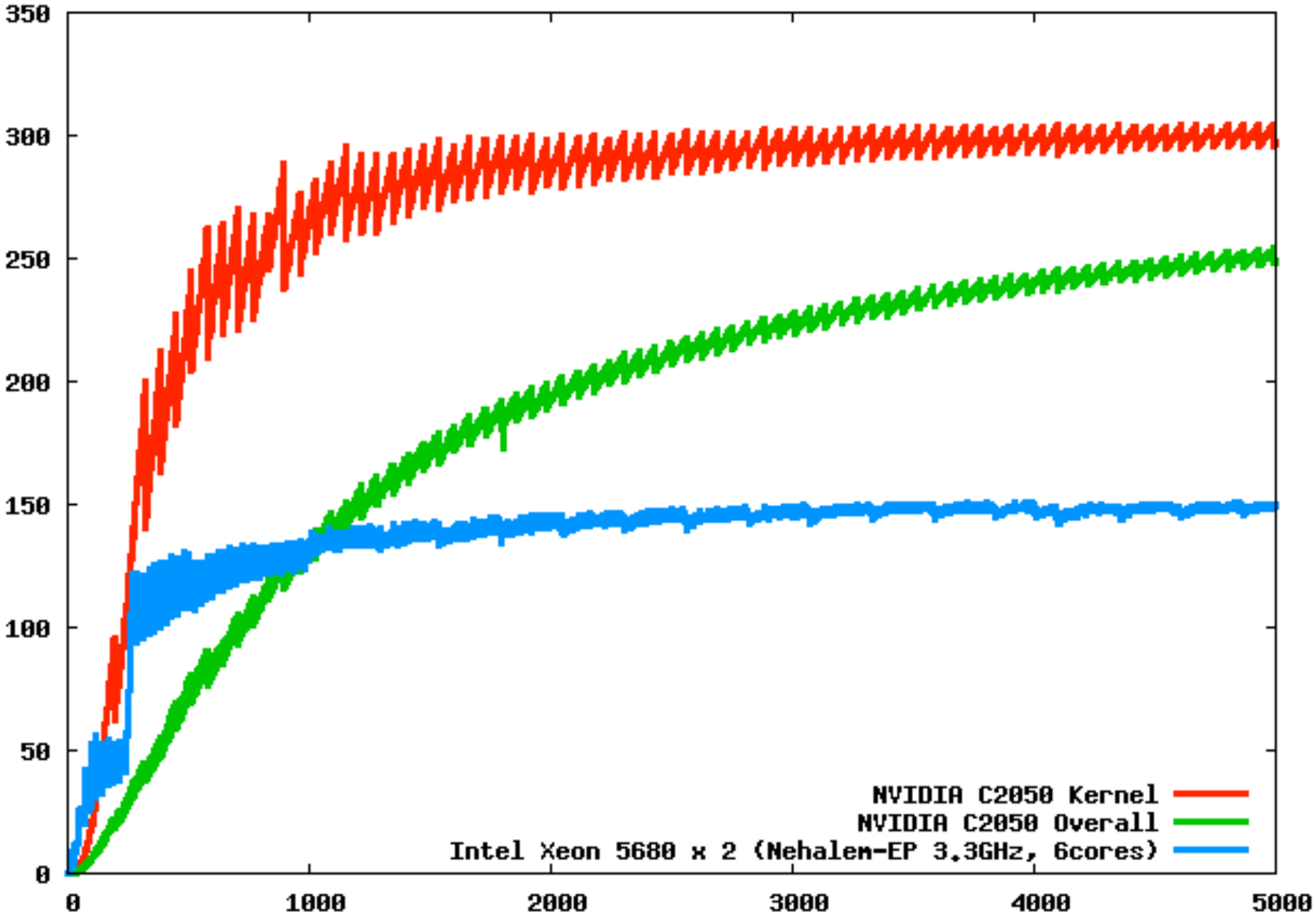


14:15

cuBLASでの行列-行列積 (VII)

ポイント

- 行列のサイズが大きくなると、行列積の性能が上がる
  - B/Fはn→∞で0だが、O(1/n)になっている。
- パフォーマンスに「ギザギザ」がでるのはなぜだろうか。
- Intelマシンのほうが安定的にパフォーマンスが出ている



まとめ

- コンピュータの簡単な仕組みとボトルネック
  - フォン・ノイマン型コンピュータ
  - フォン・ノイマンボトルネック
  - 演算バンド幅のトレンド
  - メモリバンド幅のトレンド
  - 演算バンド幅の理論性能と調べ方
  - メモリバンド幅の理論性能と調べ方
  - GPUについて
- 最適なBLAS/LAPACKとリンクする
- DGEMM: 演算バンド幅がボトルネックになる
- DGEMV: メモリバンド幅がボトルネックになる
- DGEMM高速化手法の紹介：ブロック化 (メモリの階層構造)
- GPUでBLAS/LAPACKを使う+DGEMMベンチマーク