パタヘネ輪読: 第一章

Arch B2 Kota

全体の流れ

- はじめに：コンピュータの急速な進歩
- はじめに：この本で学べること
- 「コンピュータ・アーキテクチャにおける7つの主要なアイディア」
- プログラムの裏側：ソフトウェアの階層化と抽象化
- ハードウェアの話：コンピュータの構成要素
- ハードウェアの話：iPhone XS Maxの中身を見る
- 集積回路の進化と生産工程

全体の流れ

- コンピュータの「性能が良い」とは？
- コンピュータの消費電力の限界
- マルチプロセッサへの変遷と並列処理
- Intel Core i7のベンチマークテストを見る
- Pythonの行列乗算プログラムで見る高速化
- コンピュータに関する誤信

コンピュータの急速な進歩

- コンピュータは1940年代後半の出現以来、急速に進歩している

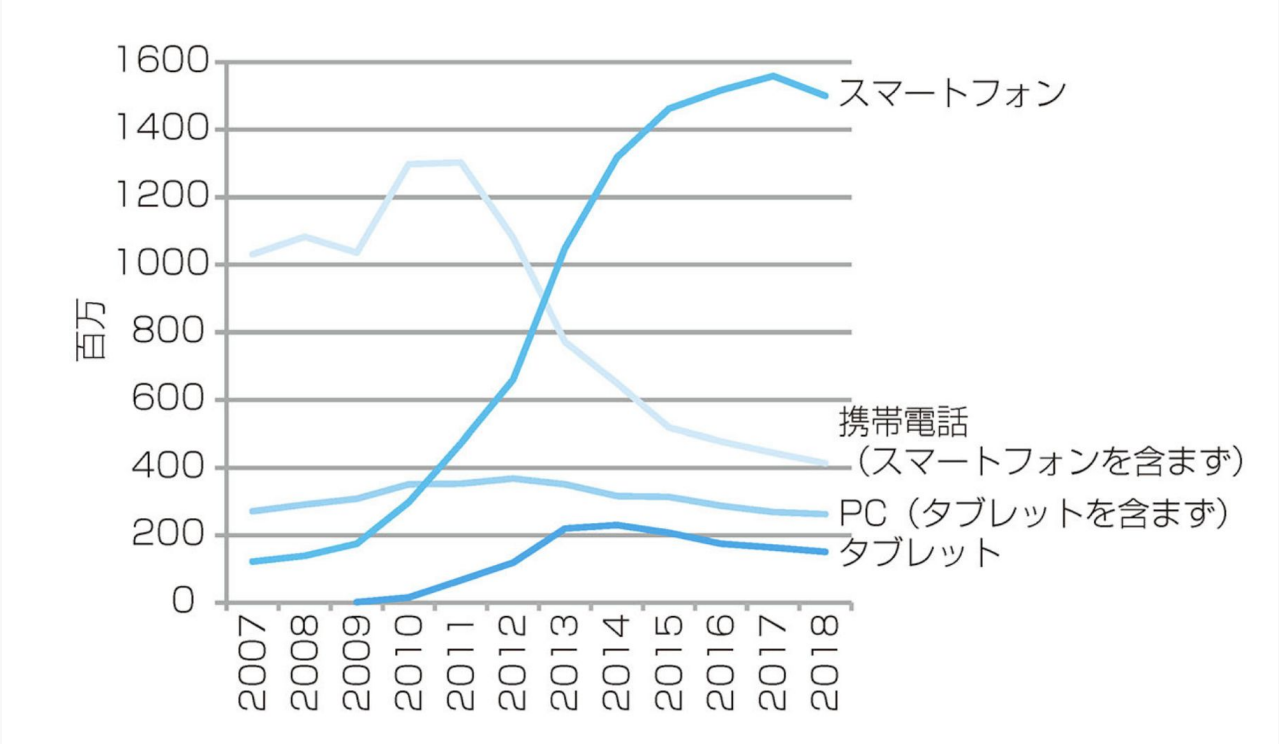
>もし運輸産業がコンピュータ産業と同じ速度で進歩していたとすれば、今日ニューヨークからロンドンまで旅行するのに必要な時間は1秒、料金は1セントになっているはずである

かつては空想のものだった携帯電話、検索エンジン、あるいは車載コンピュータが当たり前の世の中になった

進歩に伴ってコンピュータの利用形態も多様化した

- パーソナルコンピュータ
  - 低コストかつそれなりの性能が求められる
- サーバー
  - 単一の複雑なアプリケーションを動かす場合や小規模なタスクを大量に扱う場合がある
  - 処理能力、容量は用途によって大きく異なる（小さいWebサーバーからスパコンまで）
- 組み込みコンピュータ
  - 自動車やテレビから冷蔵庫まで、あらゆるハードウェアに組み込まれるプロセッサ
  - 比較的低い性能でコストと消費電力を最小限に抑えることが求められる

ポストPCの時代：スマホ、タブレット端末（PMD）の台頭



この本で学べること

- 1960-1970年代のコンピュータの主な制約はメモリ容量だったが、現在ではそれはほとんど問題ではない
  - プロセッサの並列性や記憶容量の階層性に対する理解の方が重要
  - PMDやクラウド上のプログラムについてはエネルギー効率も重要な要因の一つ

昔に比べて、現代のプログラマは知るべき知識が大きく増えている

→だからこそ、その根底になるソフトウェアとコンピュータの内部知識をこの本で学ぼう

この本で学べること

- 高水準言語で書かれたプログラムがハードウェア向けにどう翻訳され、どう実行されるのか
- プログラムを実行するためにソフトウェアはハードウェアにどんな指示を出すのか
- プログラムの性能はどう決まり、どう性能改善できるのか
- ハードウェア設計者は性能・エネルギー効率のためにどのような技法を用いているのか
- 逐次処理から並列処理への転換はなぜ起きたのか、またどういう結果をもたらすのか
- 商用コンピュータが誕生して以来、現代のコンピューティングの基礎を築くためにどのようなアイディアが考案されたのか

>以上の質問に対する答えを理解しないままで、現代のコンピュータにおけるプログラム性能を改善しようとしたり、ここのアプリケーションでコンピュータ間の優劣にどんな特性が影響するかを評価しようとしたりすることは慎んでほしい

過去60年のコンピュータ設計において考案された7つの主要なアイディア

- 抽象化
  - 下位レベルのプログラムの詳細を隠すことによりモデルを単純化する
- 「一般的な場合」の高速化
  - 多くの場合、「一般的な場合」は稀な場合よりも高速化が単純である
  - 「一般的な場合」は様々な指標の測定によって初めて可能になる（ベンチマーク）
- 並列処理
  - この本通して頻繁に登場するアイディア

過去60年のコンピュータ設計において考案された7つの主要なアイディア

- パイプライン処理
  - 非常に多様される並列処理方式
  - 火事場のバケツリレー
- 場合の予測
  - 「先に許可を求めるよりも、後で許しを請す方が良い」
  - 処理の中で、ある程度予測のつく処理を先に実行してしまう
- 記憶の階層化
  - 最高速だがビット当たりのコストが高い最小容量のメモリを最上部、その逆を最下部に配置することでコストを抑えつつ性能を改善した

過去60年のコンピュータ設計において考案された7つの主要なアイディア

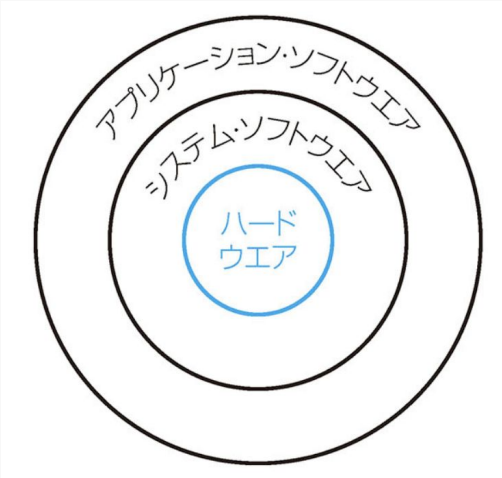
- 冗長化
  - 商用のサーバーなどであれば特に、一つの物理的な装置が故障しても動き続けるような高い信頼性が求められる
  - 故障した時に替えがきくような冗長なコンポーネントを用意する

第5版では「ムーアの法則に従って設計する」も含まれていたらしい

- 集積回路のリソースは毎年倍増するという法則
- 近年ではこの法則の正確性は低下してきた（指数的な成長は続かない）

ソフトウェアの抽象化

- コンピュータのハードウェアが実行できる命令の数はとても少ない
- **ソフトウェアをレイヤーに分け、高水準の処理を単純な命令の組み合わせに変換していく**ことで、複雑で大規模なアプリケーションの実行が可能になる



ソフトウェアの抽象化

- コンピュータのハードウェアが実行できる命令の数はとても少ない
- **ソフトウェアをレイヤーに分け、高水準の処理を単純な命令の組み合わせに変換していく**ことで、複雑で大規模なアプリケーションの実行が可能になる

👨‍🍳: 赤身の寿司を提供したい
→赤身を切る
→シャリを握る
→赤身をシャリの上に乗せる
→客に差し出す

ソフトウェアの抽象化

- コンピュータのハードウェアが実行できる命令の数はとても少ない
- **ソフトウェアをレイヤーに分け、高水準の処理を単純な命令の組み合わせに変換していく**ことで、複雑で大規模なアプリケーションの実行が可能になる

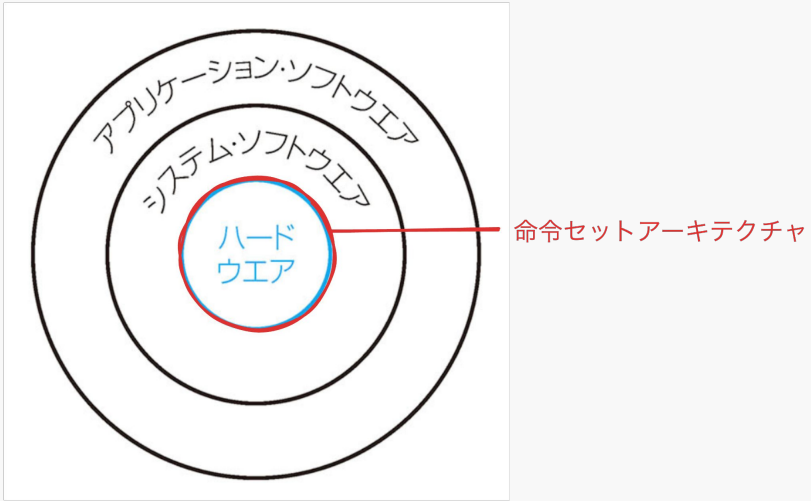
👨‍🍳: 赤身の寿司を提供したい
→赤身を切る
→シャリを握る
→赤身をシャリの上に乗せる
→客に差し出す

より低いレイヤ

👨‍🍳: シャリを握る
→ご飯を炊く
→酢飯にする
→適当な分量掴み取る
→握る

ソフトウェアの抽象化

- ハードウェアとソフトウェアの最低水準のインターフェースとなる言語体系を**命令セットアーキテクチャ**と呼ぶ



高水準言語からハードウェアの言語へ：言語の抽象化

- ハードウェアは電気的な0/1の信号のみを理解する
- アルファベットと同じように、0/1の組み合わせでハードウェアに命令を伝えている（これを**機械語**という）
- 命令を文字で書けるように**アセンブリ言語**が作られた
  - 一命令一行で書いていく言語
  - アセンブリ言語を機械語に翻訳するシステムを**アセンブラ**という
  - しかしこれでは実際のアプリケーションを書くにはコード量が多すぎる。。
    - 例えばどのアセンブリ言語でもHello Worldを出力するのに10-20行は書く必要がある

高水準言語からハードウェアの言語へ：言語の抽象化

- より多くのことを一度に書きたい→**高水準言語**の誕生
    - CとかRustとか、いわゆるプログラミング言語
    - 高水準言語をアセンブリ言語に変換するシステムを**コンパイラ**という

◎ 擬似的な例

高水準言語: A+B → アセンブリ言語: add A, B → 機械語: 1000110010100000

高水準言語を使うことで、プログラムをコンピュータから独立させられる

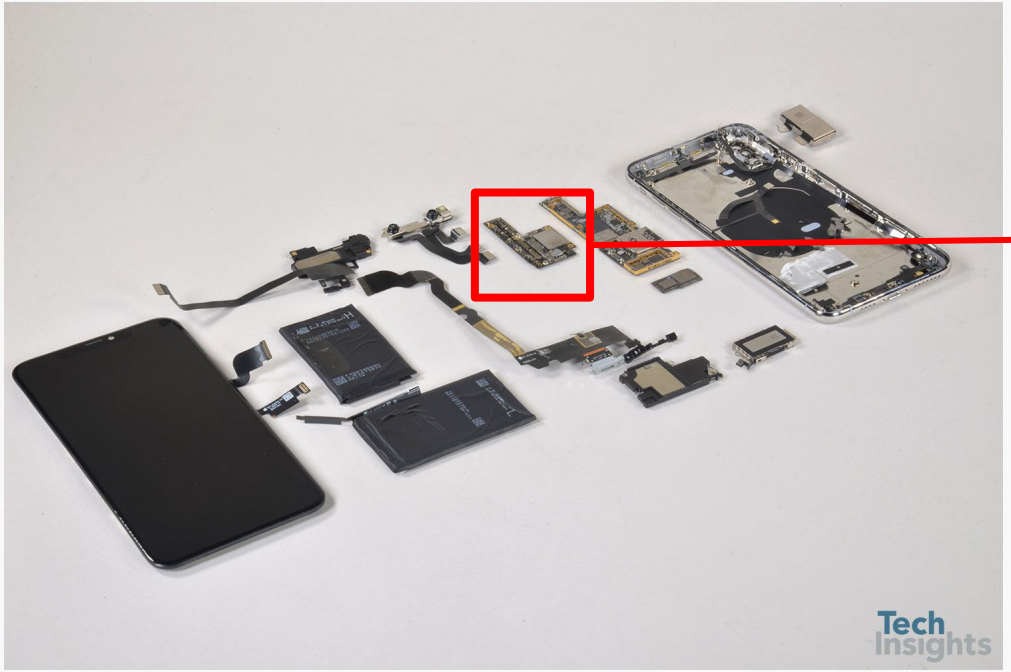
コンピュータ・ハードウェアの役割

- データの入力
  - マイクやキーボード、マウスのような入力装置
- データの出力
  - スピーカーやディスプレイなどの出力装置
- データの記憶
  - HDD、メモリなどの記憶装置
- データの処理
  - プロセッサでの演算、制御など

ハードウェアの話：iPhone XS Maxの中身を見る

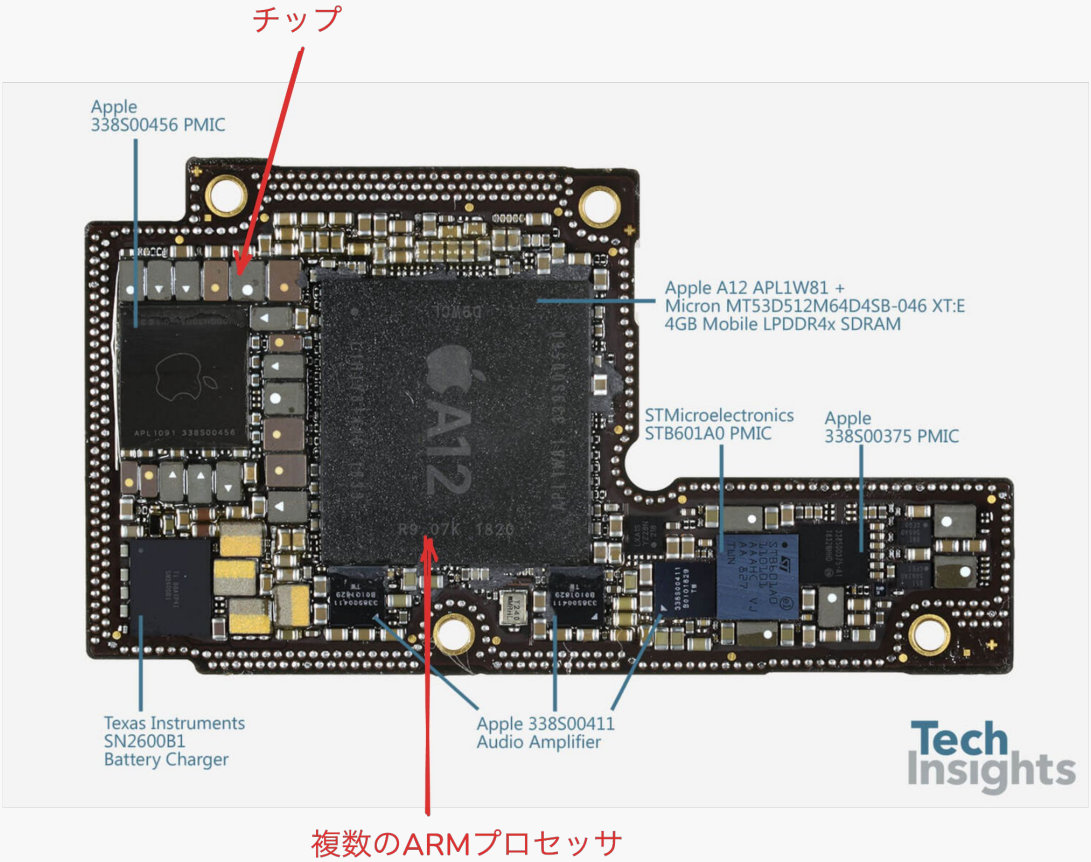
装置の大部分は入出力装置で占められている

- ディスプレイ、カメラ、マイク、スピーカー、加速度計 etc.



これがプロセッサとメモリを格納した基板

ハードウェアの話：iPhone XS Maxの中身を見る



プロセッサ（CPU）

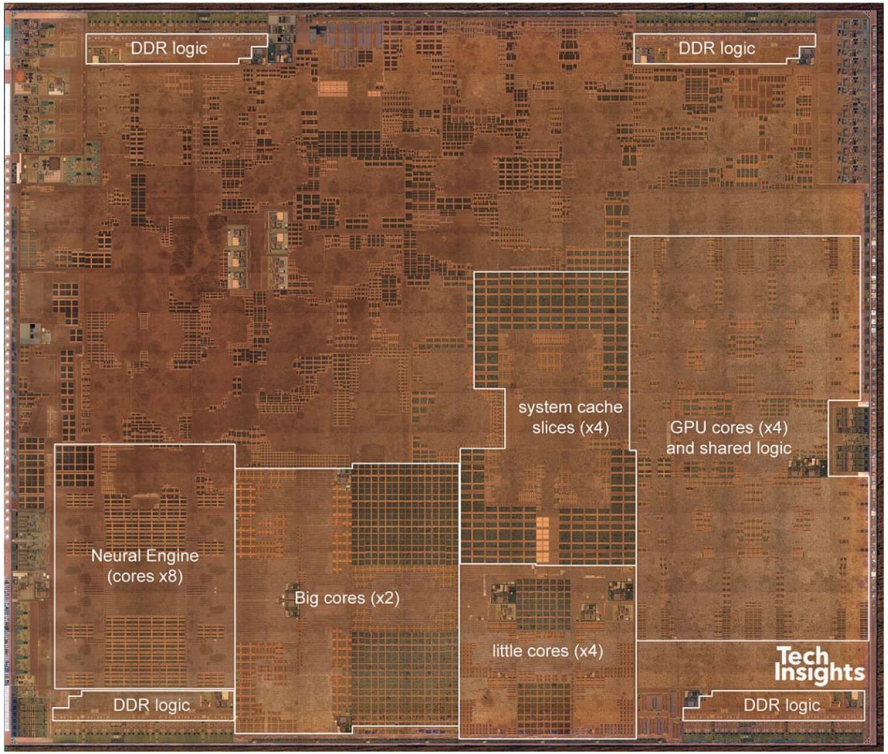
- 数値演算や条件判定などを行い、他のコンポーネントの動作を制御する役割を持つ

PMIC

- 電源管理装置
- 疑問：なんでこんなにあるのか

A12のようにCPU、GPUなど多機能を1チップに集約したものを**SoC** (System on Chip)という

ハードウェアの話：iPhone XS Maxの中身を見る



Apple A12 Bionic Chip APL1W81 Annotated Floorplan Die Photo

キャッシュは**SRAM**を使用している

- 「フリップフロップ回路」なる複雑な回路
- リフレッシュ動作が要らないかつ高速

一方メモリ（1次記憶）は**DRAM**を用いる

- 電荷で情報を保持する
- 定期的にリフレッシュ動作が必要

さらに下位の2次記憶には**フラッシュメモリ**が用いられる

- より低速でより安価&不揮発

記憶の階層化

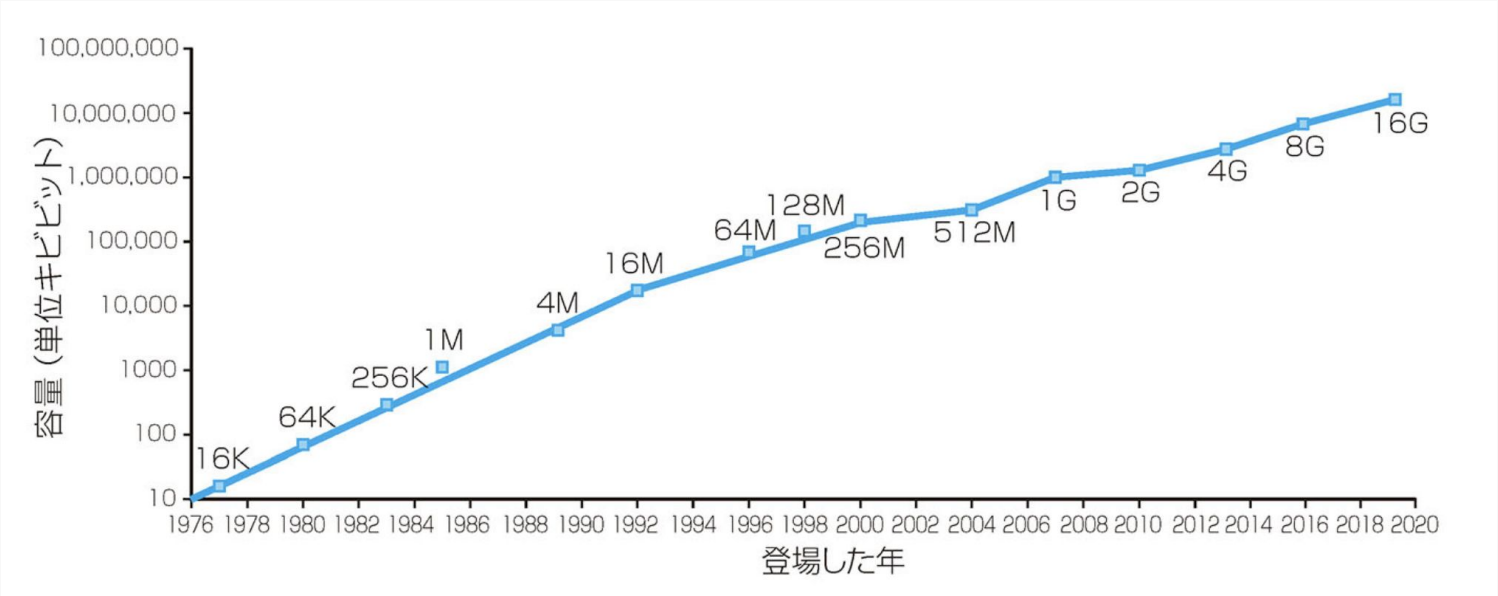
プロセッサ、メモリを構成する技術とその性能の進化

年	コンピュータのテクノロジ	相対コスト性能比
1951	真空管	1
1965	トランジスタ	35
1975	集積回路	900
1995	超大規模集積回路（VLSI）	2,400,000
2020	超々大規模集積回路（ULSI）	500,000,000,000

集積回路はトランジスタを1つのチップにまとめて接続したもの

- プロセッサになる
- **ムーアの法則:** 1チップに載るトランジスタ数は約2年ごとに倍増する

ムーアの法則の低下

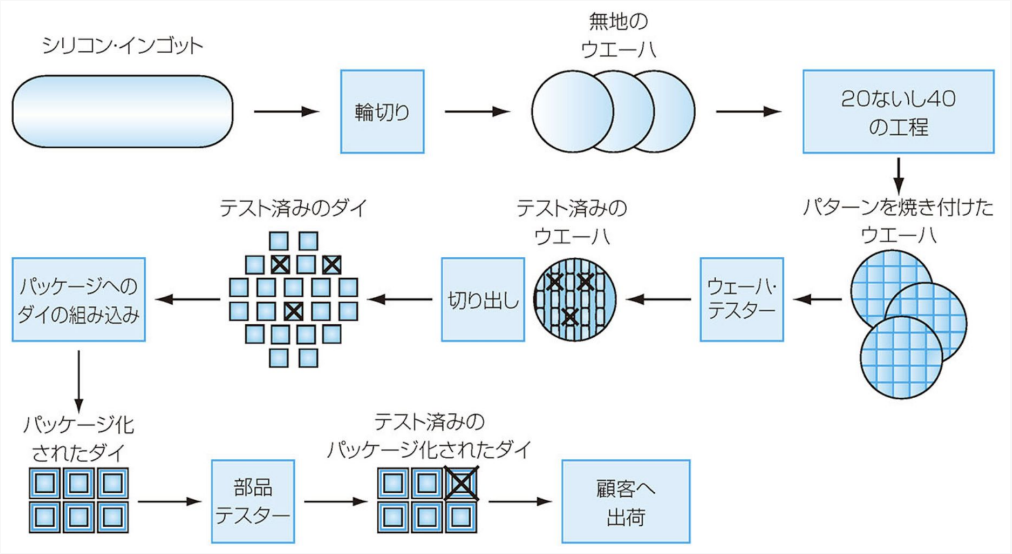


- 近年では大体3年で2倍くらいという感じ

集積回路はシリコンからなる

- シリコンは砂の中に含まれ、電気を中程度通すので**半導体**とも呼ばれる
- 特別な化学処理を施すことで以下の3つの性質のいずれかを持たせる
  - 電気をよく通す導体にする
  - 電気を通さない絶縁体にする
  - 条件に応じで電気を通したり通さなかったりする物体にする
    - （これがトランジスタ）

集積回路の進化と生産工程



- シリコンの結晶、シリコンインゴットを輪切りにして円形のウエーハを作る
- ウエーハに多重な工程でパターンを焼き付ける
- ウエーハをテストし、その後構成要素の**ダイ**（後のチップ）に切り分ける
- ダイはその後パッケージングされ、さらなるテストを経て出荷される

なぜ細かくダイに分けるのか

ウエーハを小さく分割することで、傷がついた場合に破棄する部分を最小限に抑える

- ウエーハ内のダイのテスト成功率を**歩留まり**（yield）という
- ダイのサイズが大きいと歩留まりが下がり、単一ウエーハに載せられるダイの数も減る→コスト増加
- 現在はダイ、つまりチップ内のトランジスタの最小単位が5nmになるレベルまで小型化が進んでいる

コンピュータの「性能が良い」とは？

飛行機の「性能」を定義しにくいのと同様コンピュータの「性能」も難しい

航空機	搭乗人員数	航続距離（マイル）	巡航速度（マイル／時）	輸送能力（人・マイル／時）
ボーイング 737	240	3000	564	135,360
BAC/Sud コンコルド	132	4000	1350	178,200
ボーイング 777-200LR	301	9395	554	166,761
エアバス A380-800	853	8477	587	500,711

搭乗員数だったらエアバス、航続距離だったら777、巡航速度ならコンコルドだけど輸送能力はエアバス🤔…

コンピュータの「性能が良い」とは？

1タスクを速く終わらせれば性能が良いのか？

サーバーなどの大型コンピュータであれば複数のユーザーの複数のタスクを限られたスレッドで処理している（タイムシェアリング方式）

- タイムシェアリングできることもまた性能。。。

個人のコンピュータユーザーは**実行時間の短さ**に関心があるが、データセンターの管理者は一定時間内に終了した作業の総量（**スループット**）、一定時間に送信できるデータ量（**バンド幅**）に関心がある

コンピュータの「性能が良い」とは？

この本の最初数章における性能の定義

主に**タスク（プログラム）の実行時間の短さ**に注目する

- 実行時間が短い＝性能が良い
- XがYよりn倍速ければ、実行時間はYの方がXよりもn倍長い
- ここでいう実行時間とは、プロセッサが実際にタスクの処理に費やした時間（**CPU実行時間**）である
    - 前述のタイムシェアリング方式の場合、応答時間が必ずしもCPU実行時間とは限らない

コンピュータにおける「時間」

- コンピュータは、**クロック**というタイミング単位で処理を行う
  - その周期を**クロックサイクル時間**、その逆数を**クロック周波数**という
  - CPU実行時間 = プログラムのクロックサイクル数 / クロック周波数

コンピュータにおける「時間」

- コンピュータは、**クロック**というタイミング単位で処理を行う
  - その周期を**クロックサイクル時間**、その逆数を**クロック周波数**という
  - CPU実行時間 = プログラムのクロックサイクル数 / クロック周波数

大トロ（1000サイクル）を握る

コンピュータにおける「時間」

- コンピュータは、**クロック**というタイミング単位で処理を行う
  - その周期を**クロックサイクル時間**、その逆数を**クロック周波数**という
  - CPU実行時間 = プログラムのクロックサイクル数 / クロック周波数

大トロ（1000サイクル）を握る

👨‍🍳寿司職人A（1GHz）

1秒間に10億クロック
クロックサイクル時間は1ns
大トロのCPU実行時間は1μs
1秒間に100万大トロ

コンピュータにおける「時間」

- コンピュータは、**クロック**というタイミング単位で処理を行う
  - その周期を**クロックサイクル時間**、その逆数を**クロック周波数**という
  - CPU実行時間 = プログラムのクロックサイクル数 / クロック周波数

大トロ（1000サイクル）を握る

👨‍🍳寿司職人A（1GHz）	👨‍🍳寿司職人B（2GHz）
1秒間に10億クロック クロックサイクル時間は1ns 大トロのCPU実行時間は1μs 1秒間に100万大トロ	1秒間に20億クロック クロックサイクル時間は0.5ns 大トロのCPU実行時間は0.5μs 1秒間に200万大トロ

コンピュータにおける「時間」

- コンピュータは、**クロック**というタイミング単位で処理を行う
  - その周期を**クロックサイクル時間**、その逆数を**クロック周波数**という
  - CPU実行時間 = プログラムのクロックサイクル数 / クロック周波数

プログラムの性能を上げるには、プログラムのクロックサイクル数を減らすか、クロック周波数を上げる必要がある

プログラムのクロックサイクル数はどう算出するのか？

- 実際にコンピュータが実行する命令は、命令ごとに必要なクロックサイクル数が異なる
  - **CPI**（命令あたりの平均クロックサイクル数）とプログラムの命令数をかければそのプログラムのクロックサイクル数が算出できる

$$\text{CPUクロック・サイクル数} = \begin{matrix}\text{プログラムの}\\ \text{実行命令数}\end{matrix} \times \begin{matrix}\text{命令当たりの平均}\\ \text{クロック・サイクル数}\end{matrix}$$

まとめ：古典的なCPU性能方程式

CPU実行時間 = 実行命令数 * CPI * クロックサイクル時間

もしくは

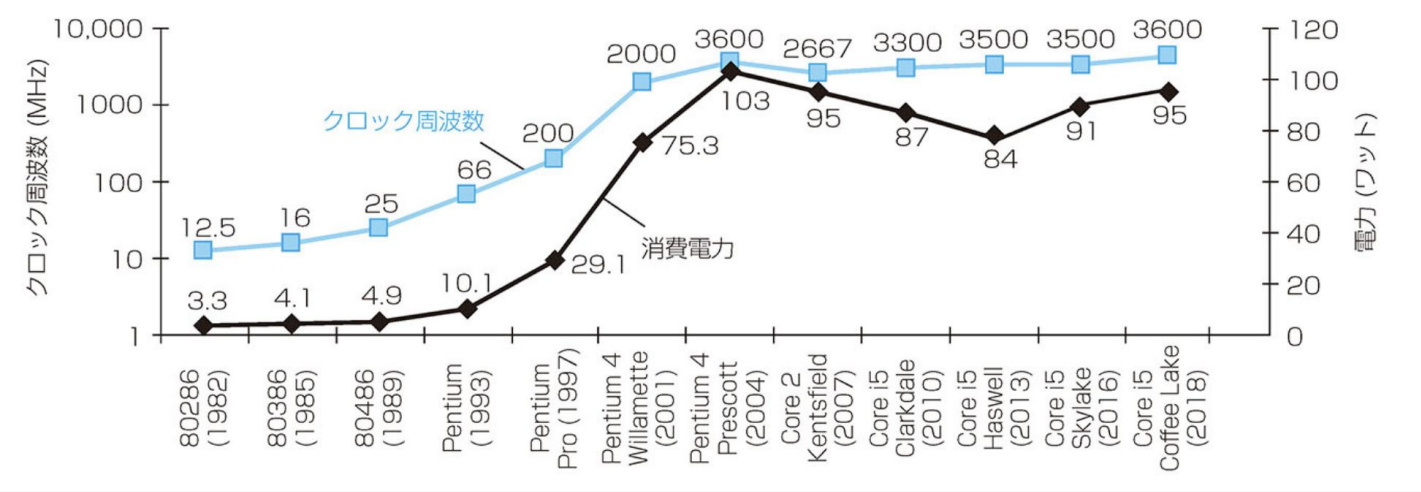
CPU実行時間 = 実行命令数 * CPI / クロック周波数

- - - - - - - - - - - - - -

- クロックサイクル時間は通常コンピュータのスペックとして公開されている
- 実行命令数はシミュレータ、ハードウェアカウンタなどを通して計算する
- CPIはメモリシステムやプロセッサの構造などによるため計算が難しい

コンピュータの消費電力の限界

9世代のIntelマイクロプロセッサに見るクロック周波数と消費電力の変遷



- 2004年あたり以降から両方とも頭打ちになっている
  - 量産品であるマイクロプロセッサの冷却能力に限界が来た

<u>電力にものを言わせて1つのプロセッサの性能を追求する時代の終焉</u>

消費電力の算出方法

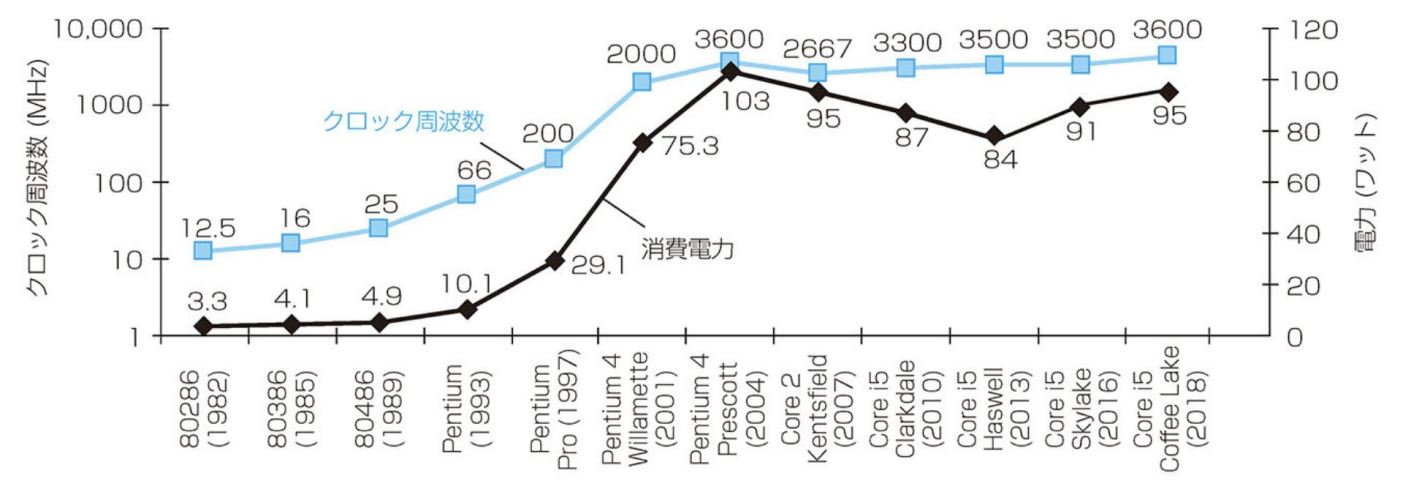
集積回路内の一つのトランジスタに必要な電力は以下の式で算出される

消費電力 = ½ * 容量性負荷 * 電圧^2 * 切り替え周波数

- 集積回路の主要テクノロジである**CMOS**において、エネルギー消費の主要因はトランジスタの0/1の切替の動的エネルギーである
  - これは½ * 容量性負荷 * 電圧^2で求められる
  - 容量性負荷は、コンデンサの電荷が変わる際の抵抗のようなもの？？

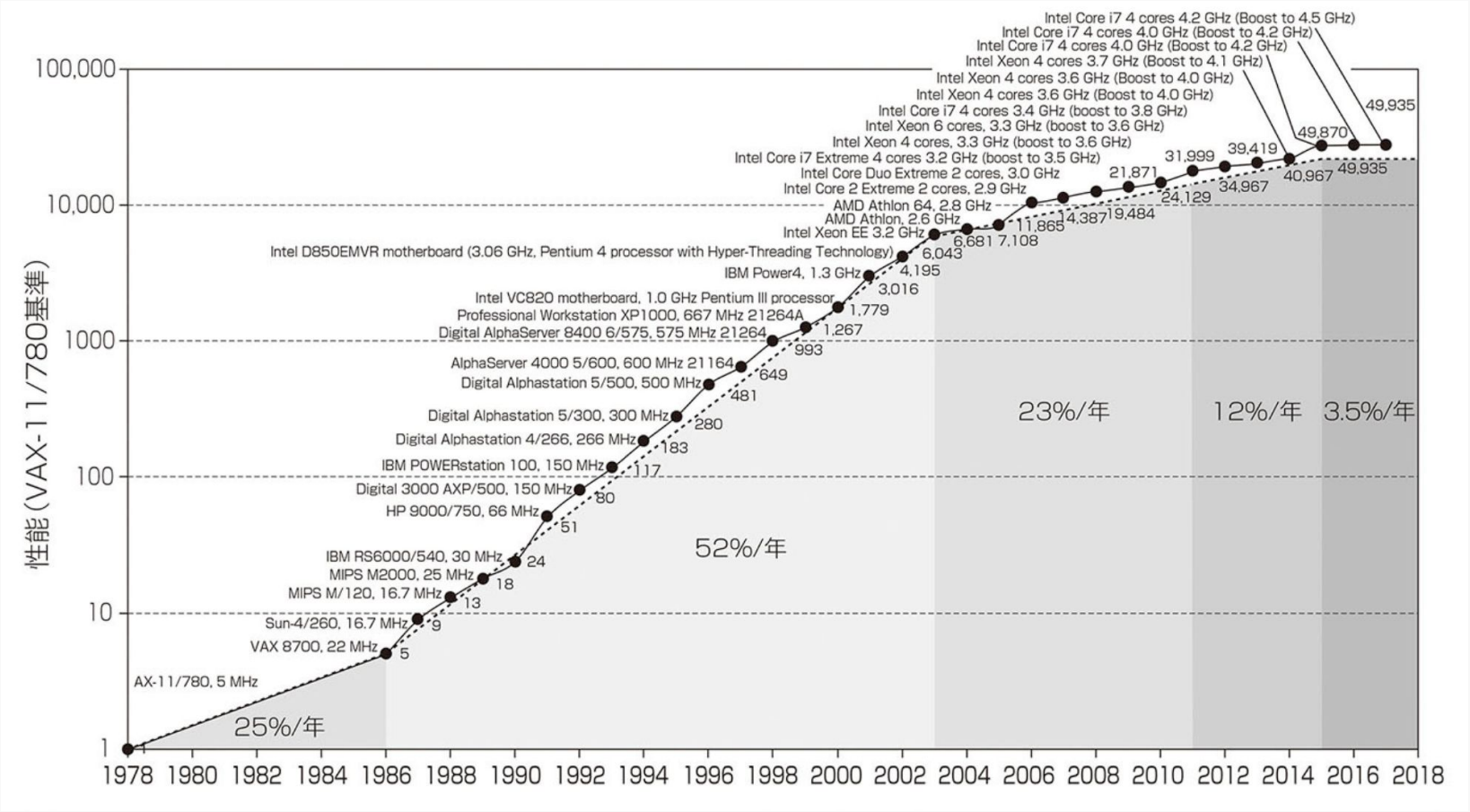
コンピュータの消費電力の限界

なぜ電力が30倍になる間にクロック周波数は1000倍に上昇していたのか？



- 電力は電圧の2乗の関数である
- 世代が進むごとに、プロセッサが要する電圧は約15%下げられてきた
- クロック周波数の成長に比べて消費電力は低く抑えられてきた

マルチプロセッサへの変遷と並列処理



マルチプロセッサへの変遷と並列処理

- 2006年時点で、すべてのメーカーが1つのチップに複数のプロセッサを載せた**マイクロプロセッサ**を出荷するようになった
  - これはスループットの向上に大きく貢献した
  - 「コア」と呼ばれるものは単一のプロセッサで、マイクロプロセッサはnコア・マイクロプロセッサなどと呼ばれるようになった
- これにより、プログラマは並列処理を前提としたプログラミング技術を求められるようになった
  - 古くから存在するパイプライン処理などは**命令レベル並列性**と呼ばれ、プログラマ、コンパイラ共に逐次処理と同じ書き方で何も問題はなかった
  - この頃からプログラマは、並列処理のためのスケジューリング、負荷の平準化、同期のオーバーヘッドの減少に取り組む必要が出てきたのである

※並列処理については各章に説明が頻出する

SPEC CPU

- CPUに対するベンチマークセット
  - 「一般的な場合」の性能を判断するための、汎用的な処理を用いたベンチマーク
- 今回の例ではSPEC CPU 2017を用いる
  - 10本の整数用ベンチマークと13本の浮動小数点用ベンチマーク
  - 対象はIntel Xeon E5-2650L

Intel Core i7のベンチマークテストを見る

説明	名前	命令数 ($\times 10^9$)	CPI	クロックサイクル時間（ナノ秒）	実行時間（秒）	基準時間（秒）	SPECratio
Perl インタープリタ	perlbench	2,684	0.42	0.556	627	1,774	2.83
GNU C コンパイラ	gcc	2,322	0.67	0.556	863	3,976	4.61
経路計画	mcf	1,786	1.22	0.556	1,215	4,721	3.89
離散事業シミュレーション・コンピュータ・ネットワーク	omnetpp	1,107	0.82	0.556	507	1,630	3.21
XSLT を通じての XML から HTML への変換	xalancbmk	1,314	0.75	0.556	549	1,417	2.58
ビデオ圧縮	x264	4,488	0.32	0.556	813	1,763	2.17
人工知能：アルファ・ベータ木構造検索（チェス）	deepsjeng	2,216	0.57	0.556	698	1,432	2.05
人工知能：モンテ・カルロ木構造検索（囲碁）	leela	2,236	0.79	0.556	987	1,703	1.73
人工知能：再帰的解法ジェネレータ（数独）	exchange2	6,683	0.46	0.556	1,718	2,939	1.71
汎用データ圧縮	xz	8,533	1.32	0.556	6,290	6,182	0.98
幾何平均							2.36

CPU実行時間 = 実行命令数 * CPI * クロックサイクル時間

Intel Core i7のベンチマークテストを見る

説明	名前	命令数 ($\times 10^9$)	CPI	クロックサイクル時間（ナノ秒）	実行時間（秒）	基準時間（秒）	SPECratio
Perl インタープリタ	perlbench	2,684	0.42	0.556	627	1,774	2.83
GNU C コンパイラ	gcc	2,322	0.67	0.556	863	3,976	4.61
経路計画	mcf	1,786	1.22	0.556	1,215	4,721	3.89
離散事業シミュレーション・コンピュータ・ネットワーク	omnetpp	1,107	0.82	0.556	507	1,630	3.21
XSLT を通じての XML から HTML への変換	xalancbmk	1,314	0.75	0.556	549	1,417	2.58
ビデオ圧縮	x264	4,488	0.32	0.556	813	1,763	2.17
人工知能：アルファ・ベータ木構造検索（チェス）	deepsjeng	2,216	0.57	0.556	698	1,432	2.05
人工知能：モンテ・カルロ木構造検索（囲碁）	leela	2,236	0.79	0.556	987	1,703	1.73
人工知能：再帰的解法ジェネレータ（数独）	exchange2	6,683	0.46	0.556	1,718	2,939	1.71
汎用データ圧縮	xz	8,533	1.32	0.556	6,290	6,182	0.98
幾何平均							2.36

SPECratio = 実行時間 / 基準プロセッサ上での実行時間

Intel Core i7のベンチマークテストを見る

SPECpower

- SPECが出す、電力測定用のベンチマーク
- ある期間サーバーを稼働させ10%刻みで負荷を変化させる

負荷レベル	性能 (ssj_ops)	平均電力 (ワット)
100%	4,864,136	347
90%	4,389,196	312
80%	3,905,724	278
70%	3,418,737	241
60%	2,925,811	212
50%	2,439,017	183
40%	1,951,394	160
30%	1,461,411	141
20%	974,045	128
10%	485,973	115
0%	0	48
合計	26,815,444	2,165
Σssj_ops /Σ電力=		12,385

図 1.19 デュアル・ソケットで 2.2GHz の Intel Xeon Platinum 8276L 上で，192G バイトの DRAM と 1 台の 80G バイトの SSD ディスクを用いて実行した SPECpower_ssj2008.

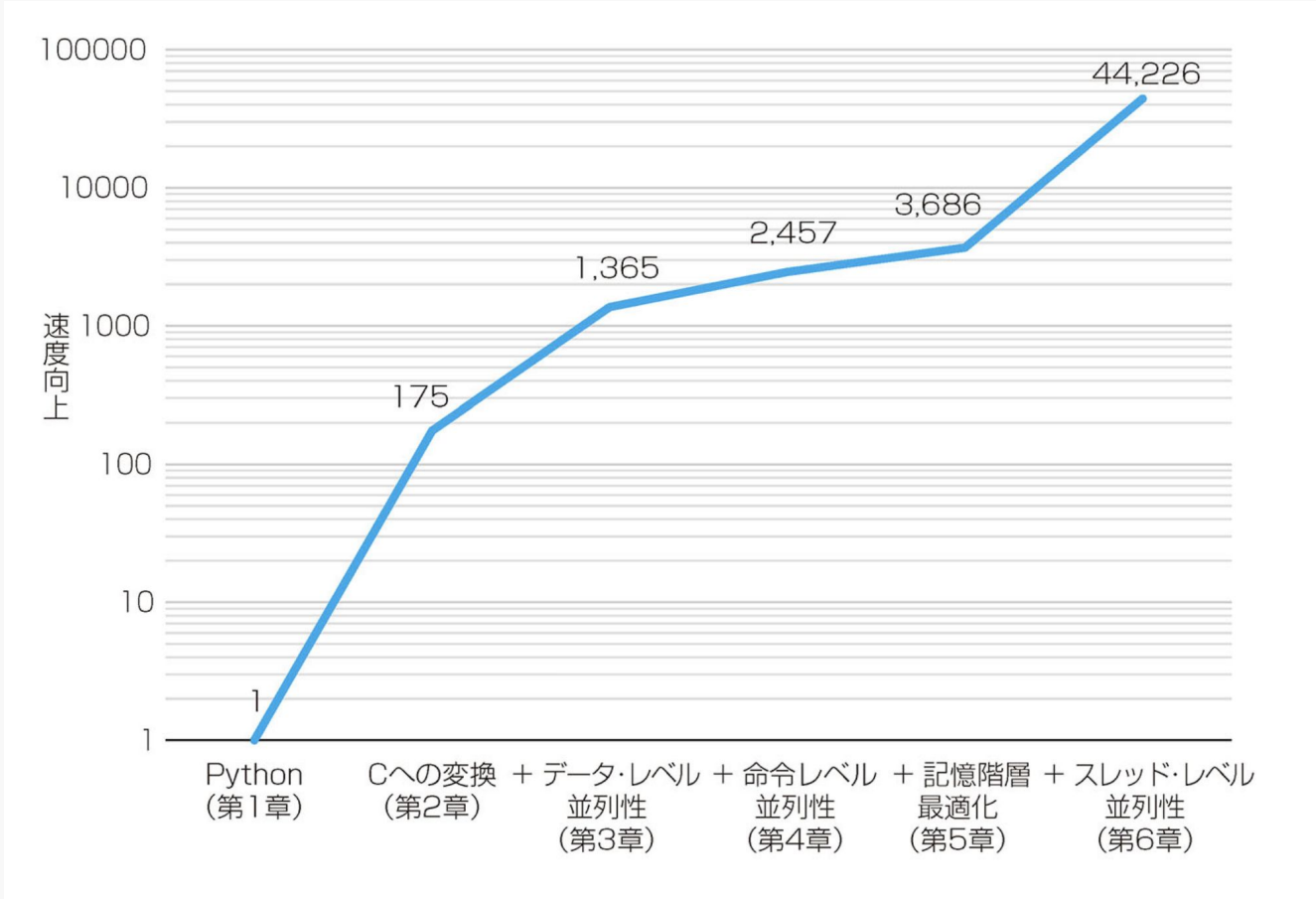
Pythonの行列乗算プログラムで見る高速化

今後の章の高速化に使われるPythonプログラム

```
for i in xrange(n):
    for j in xrange(n):
        for k in xrange(n):
            C[i][j] += A[i][k] * B[k][j]
```

- これをIntel Skylake Xeon上で実行する
  - 行列が960*960の時に約5分、4096*4096になると6時間になる
  - 2章ではこれをc言語版に変換、3章では部分的に並列化を行う、4章では...

Pythonの行列乗算プログラムで見る高速化



🤦‍♀️ コンピュータの一面を改善すればそれに等しい性能向上が得られる

- あるコンピュータ上で実行に100秒かかるプログラムがあり、そのうち80秒が乗算に占められているとする
  - この時乗算をどれだけ効率化しても、プログラム全体の実行速度を5倍以上速くすることは**不可能**である
  - 改善後の実行時間 = 改善後の乗算にかかる時間 + 20秒
  - ある面を改善したことによる性能向上はその改善された機能の割合に制約される
    - →**Amdahlの法則**（経済学においては収穫逓減の法則と呼ばれる）

コンピュータに関する誤信

🤦‍♀️ コンピュータの利用率が低ければ消費電力は少ない

- 2020年において、SPECpower向きに構成されたコンピュータでも、**負荷が10%の時に以前としてピーク電力の33%を使用している**
  - 一般的なシステムであればさらに結果は悪くなる
  - この問題が解決できれば、環境にも良いコンピュータが実現できる

🤦 性能の尺度に性能方程式の一部を用いる

- 前述の通り、クロック周波数、命令数、CPIはそれぞれ別の要素に影響され、どれが欠けても正確な性能評価には至らない
- **MIPS** = 実行命令数 / (実行時間 * 10^6) なる直感的な指標があるが、命令の働きが考慮されておらず、命令セットが異なるとこの指標は役に立たない

おわりに

- コンピュータのあらゆる場面で抽象化の概念が用いられている
- この本では特に並列処理を重要していて、全章にこのトピックが登場する
- コンピュータの性能の尺度として有効なのは実行時間のみである