Lecture 4: Performance Metrics

Spring 2024
Jason Tang

Topics

- Different performance metrics
- Performance comparisons
- Effects of software on hardware benchmarks

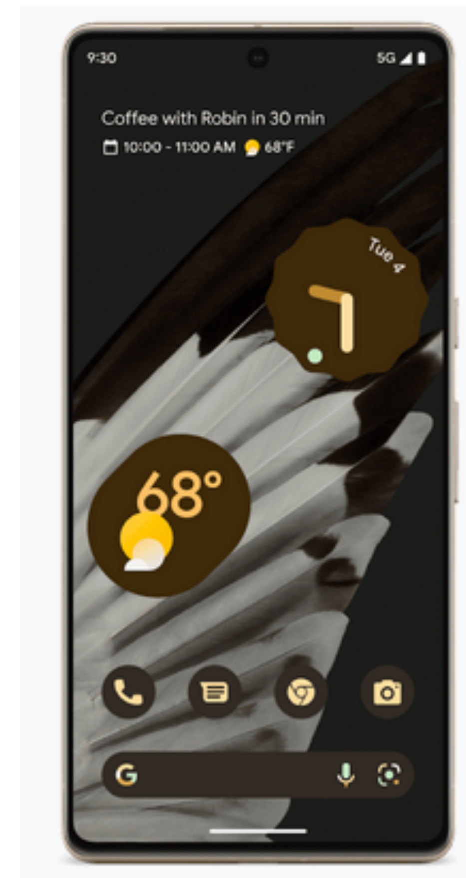
Hardware Performance

- Key to effectiveness of entire system
- Different performance metrics need to be measured and compared to evaluate system design
- Depending upon system requirements, different metrics may be appropriate
- Factors that may affect performance: instructions use, instruction implementation, memory hierarchy, I/O handling

Which is Better?

Samsung
Galaxy Z Fold5

Apple
iPhone 14 Plus



Google
Pixel 7 Pro

Criteria of performance evaluation
differs among users and designers

Common Performance Metrics

- Response time: time between the start of a task and its first output
  - Measures user perception of system speed
  - Common in time-critical (real-time) systems
- Throughput: total amount of completed “work” done per unit time
  - Depends upon what a unit of “work” is: credit card processing, mining a Bitcoin, etc.

Response-Time Metric

- Maximizing "performance" often means minimizing response time
  - $performance = \dfrac{1}{execution\ time}$
  - Thus **P_1** > **P_2** when **$E(P_1, L)$** < **$E(P_2, L)$** for some time period **L**
- Thus relative performance of $\dfrac{CPU_2}{CPU_1} = \dfrac{E(P_1, L)}{E(P_2, L)}$

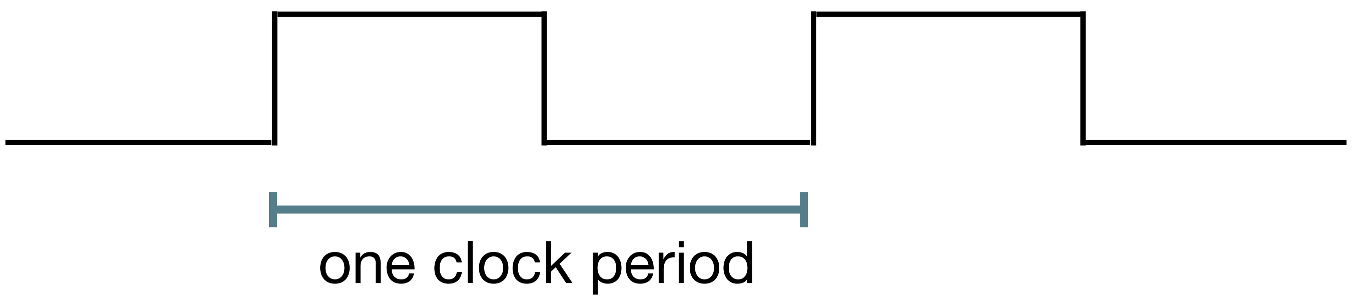
Measuring Performance

- Different definitions of execution time:
  - Elapsed (wall-clock) time: total time spent on task, including I/O activities, OS overhead, memory access
  - CPU time: time consumed by CPU
    - User CPU time: time spent processing the task itself
    - System CPU time: time consumed by operating system overhead
- Unix `time` utility can report the above values

Machine Clock Rate

- Clock rate is inverse of clock cycle time (clock period)

10 ns	100 MHz
1 ns	1 GHz
500 ps	2 GHz
250 ps	4 GHz



- CPU execution time = CPU clock cycles for program × clock cycle time

- $$CPU\ execution\ time = \frac{CPU\ clock\ cycles\ for\ program}{clock\ rate}$$

- To decrease CPU execution time, either decrease number of CPU clock cycles and/or decrease clock cycle time
  - Often, these are conflicting goals

CPU Time Example

- A program **P** runs in 10 seconds on computer **A** that has a 400 MHz clock. That same program needs to run in 6 seconds on computer **B**. However, running **P** on **B** would require 1.2 times more clock cycles than **A**. What is the minimum clock rate for **B**?

- CPU time = number of instructions × cycles per instruction (CPI) × clock cycle time

Component of Performance	Units of Measure
CPU execution time for a program	Seconds for the program
Instruction count	Instructions executed
Clock cycles per instruction (CPI)	Average number of clock cycles / instruction
Clock cycle time	Seconds per clock cycle

CPI Example

- Let there be two implementations for the same instruction set architecture. Machine **A** has a clock cycle time of 1 ns and a CPI of 2.0 for some program **P**. Machine **B** has a clock cycle time of 2 ns and a CPI of 1.2 for that same **P**. Which machine is faster for **P** and by how much?

- CPU time(**A**) = CPU clock cycles(**A**) × clock cycle time(**A**)
  CPU time(**B**) = CPU clock cycles(**B**) × clock cycle time(**B**)

- CPU time(**A**) = **I** × 2.0 × 1 ns = **I** × 2 ns
  CPU time(**B**) = **I** × 1.2 × 2 ns = **I** × 2.4 ns

- Therefore, **A** is 16.66% faster than **B**

Measuring CPI

- While clock cycle time is easily obtainable by CPU manufacturer, CPI and instruction counts are not trivial
- Instruction count can be measured by software profiling, architecture simulator, or using hardware counters on some architectures
- CPI depends upon processor structure, memory system, implementation of instructions, and which instructions are executed
- **Average CPI** = Σ **CPI_i** $\times$ **C_i**, for each different instruction classes

CPI Example

- A compiler designer is trying to decide which instruction sequence to use for a particular machine. The hardware designer provides a table of CPI for each instruction class. For a particular high-level language statement, the compiler could generate either of the following instruction sequence. Which is faster? What is the CPI for each sequence?

Instruction Class	CPI for This Instruction Class
A	1
B	2
C	3

Code Sequence	Instruction Count for Instruction Class		
	A	B	C
1	2	1	2
2	4	1	1

Factors Affecting Performance

	Instruction Count	CPI	Clock Cycles
Algorithm	Yes	Somewhat	
Programming Language	Yes	Somewhat	
Compiler	Yes	Yes	
Instruction Set Architecture	Yes	Yes	
Processor Organization		Yes	Yes
Technology / Manufacturing			Yes

Instruction Selection Example

- How much faster would system be if a better data cache reduced load time to 2 cycles?
- How does this compare when an improved branch implementation takes only 1 cycle?
- What if two ALU instructions could be executed simultaneously?

Op	Freq	CPI
ALU	50%	1
Load	20%	5
Store	10%	3
Branch	20%	2

Compiler Choices

- Difficult to compare performance across different architectures
  - Differences in compilers
  - Differences in optimization strategies

ARMv8-A / gcc Optimization Example

```c
extern unsigned int label1, label2;

int main(int argc, char *argv[]) {
    asm("label1:");
    ptrdiff_t len = &label2 - &label1;
    printf("This function is %td bytes long\n", len);
    asm("label2:");
    return 0;
}
```

-O0

```
label1:
    adrp    x0, 400000 <_init-0x3f0>
    add     x1, x0, #0x5fc
    adrp    x0, 400000 <_init-0x3f0>
    add     x0, x0, #0x5d0
    sub     x0, x1, x0
    asr     x0, x0, #2
    str     x0, [x29,#40]
    adrp    x0, 400000 <_init-0x3f0>
    add     x0, x0, #0x6a0
    ldr     x1, [x29,#40]
    bl      400460 <printf@plt>
label2:
    mov     w0, #0x0
```

-O2

```
label1:
    adrp    x2, 400000 <_init-0x3f8>
    adrp    x0, 400000 <_init-0x3f8>
    add     x0, x0, #0x478
    add     x2, x2, #0x4a0
    sub     x2, x2, x0
    adrp    x1, 400000 <_init-0x3f8>
    add     x1, x1, #0x698
    mov     w0, #0x1
    asr     x2, x2, #2
    bl      400440 <__printf_chk@plt>
label2:
    mov     w0, #0x0
```

Performance Benchmarks

- Many widely-used benchmarks are small programs that have significant locality of instruction and data references (caching effects)
- Universal benchmarks can be misleading because hardware and compiler vendors may optimize their design for **only** those programs
- Architectures might perform well for some software and poorly for other software
- Compilers can boost performance by taking advantage of architecture-specific features

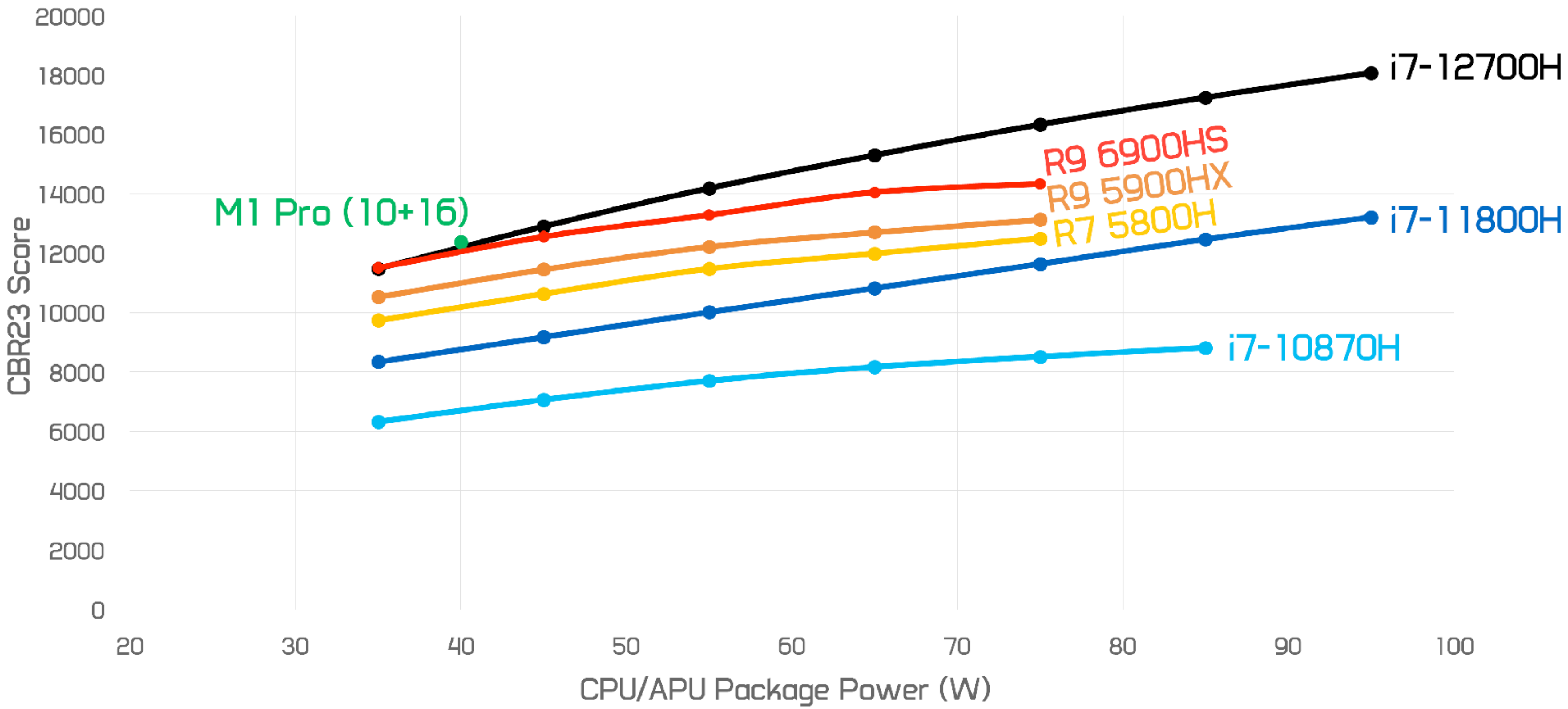
Real applications are often the best benchmarks since they reflect end-user interest

SPEC Benchmarks

- SPEC (System Performance Evaluation Cooperative) is a suite of benchmarks created by several companies to simplify reporting of performance
- SPEC CPU2006 consists of 12 integer and 17 floating-point benchmarks: running gcc, running a chess game, video compression, etc.
  - Tests are unweighted
  - As that tests are complex, test measures memory and other system components in addition to CPU

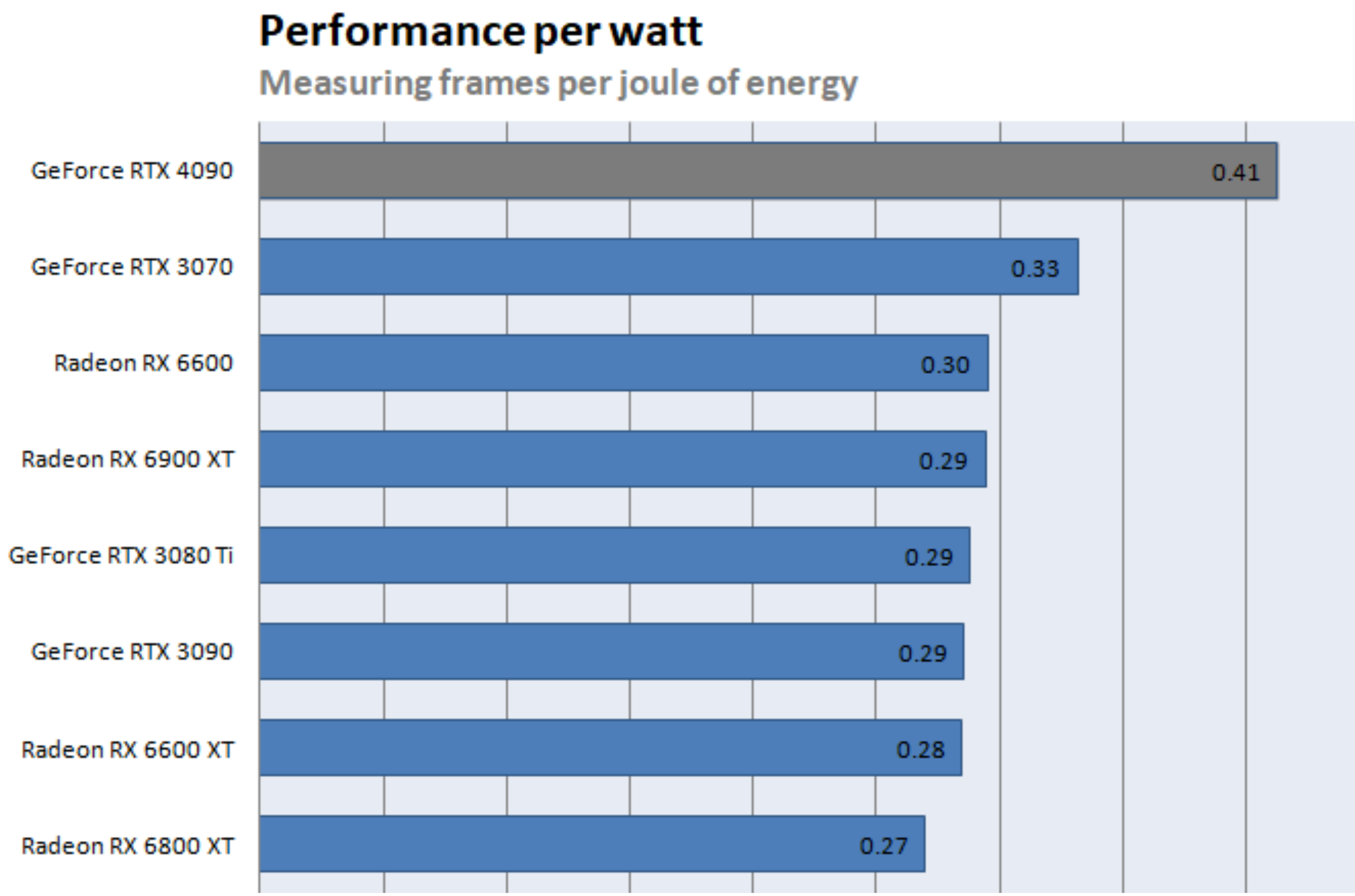
https://www.spec.org/cpu2006/results/

Performance Per Watt Comparison



https://www.techspot.com/review/2419-amd-ryzen-9-6900hs/

Performance Per Watt Comparison



https://www.guru3d.com/articles_pages/geforce_rtx_4090_founder_edition_review,30.html

Other Metrics

- FLOPS: floating point operations per second
    - Used when measuring scientific computations
- MIPS: million instructions per second
    - Useful when comparing CPUs with same instruction set
    - Not comparable between instruction sets as that the same high-level code will result in different instruction counts
- BogoMIPS: Linux's unscientific measurement based upon how long a busy-loop takes to complete

Amdahl's Law

- Performance enhancement possible with a given improvement is limited by amount that the improved feature is used
    - Therefore, make the common case fast
- $$T_{new} = \frac{T_{affected}}{Improvement} + T_{unaffected}$$
- Example: Floating point instructions are improved to run twice as fast, but only 10% of actual instructions are floating point
    - **T_{new}** = 0.1 / 2 + 0.9 = 0.95
    - **Speedup** = **T_{old}** / **T_{new}** = 1 / 0.95 = 1.053