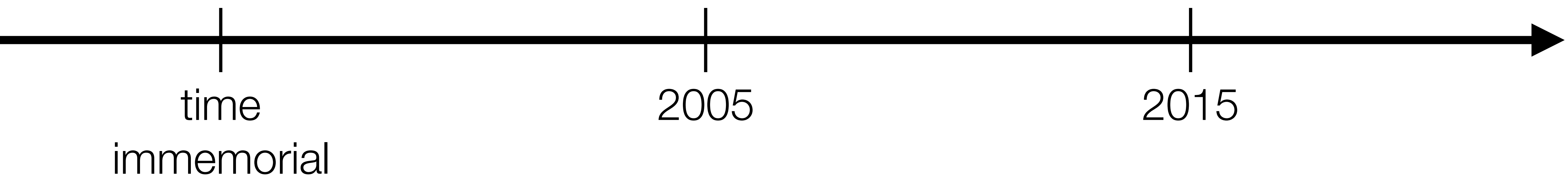
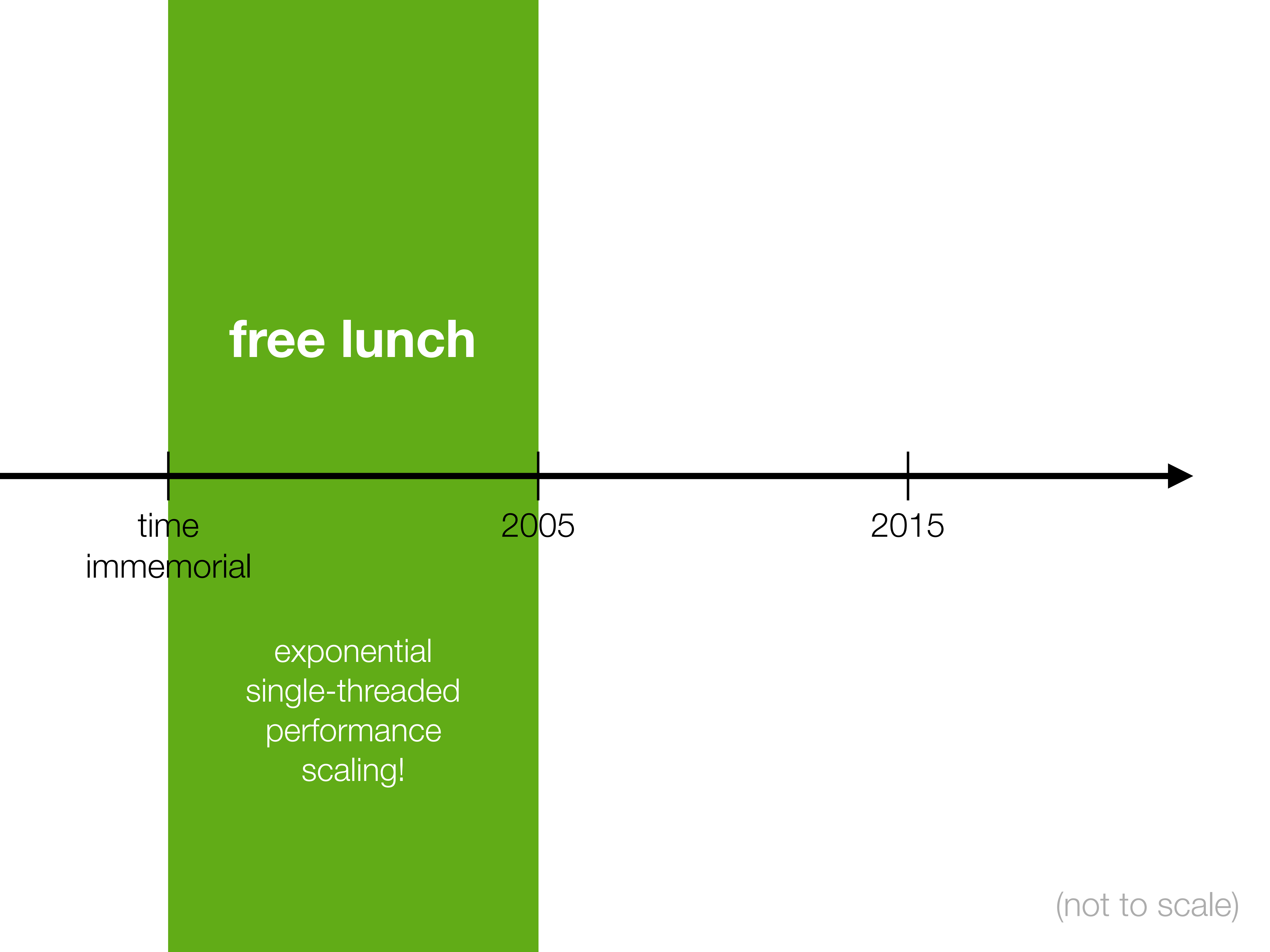
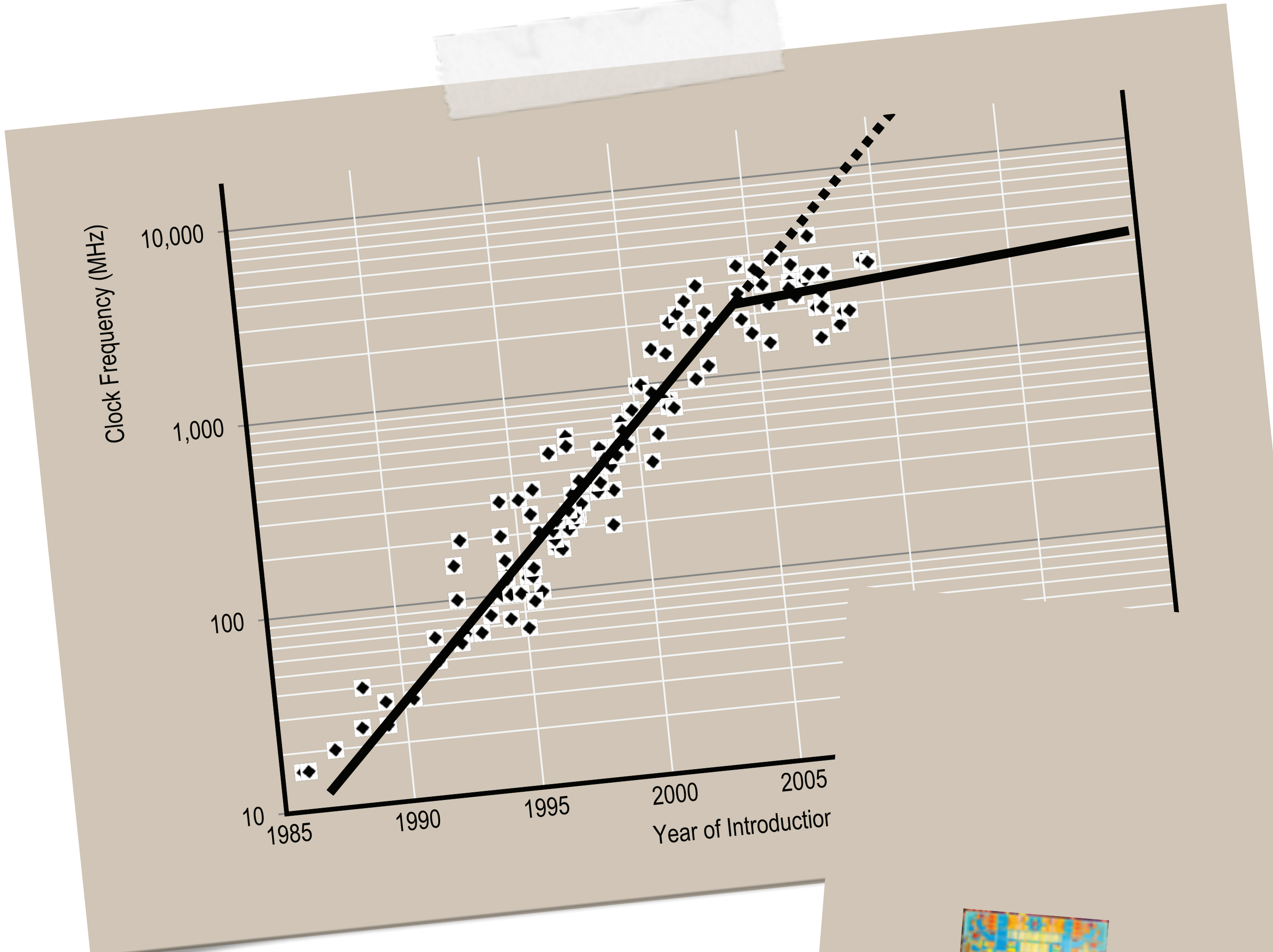
CS 4110

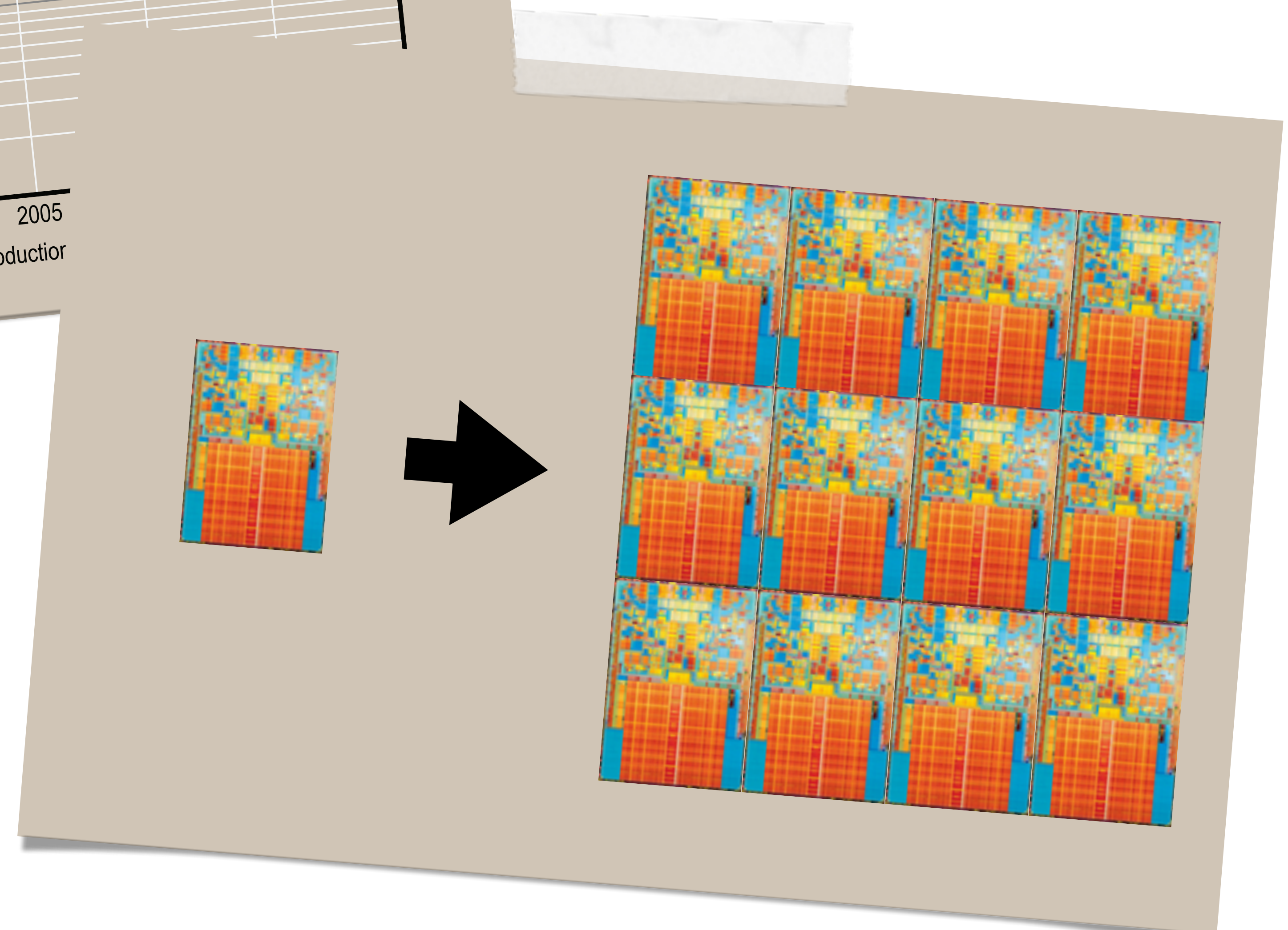
Approximate Computing

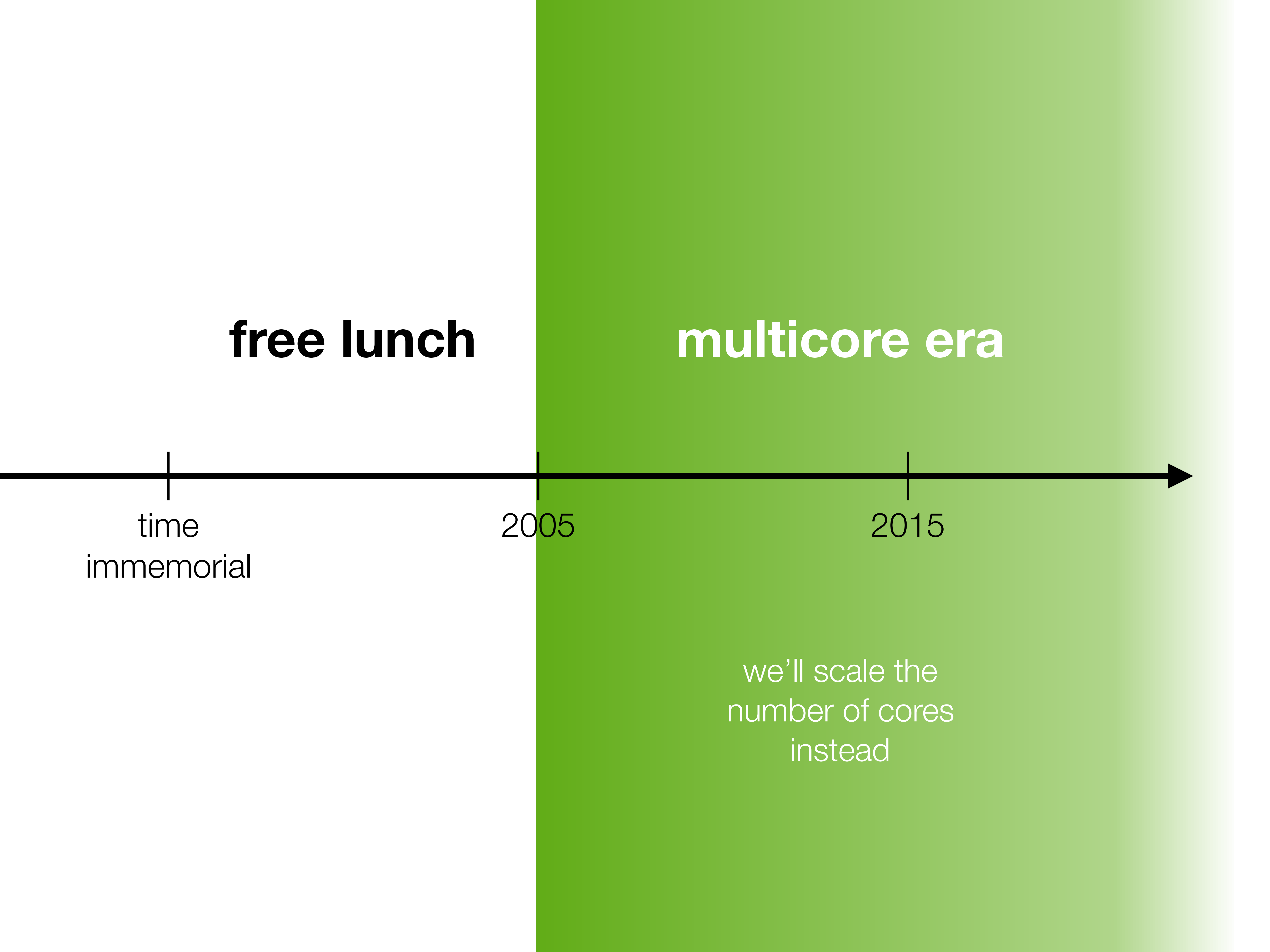


(not to scale)

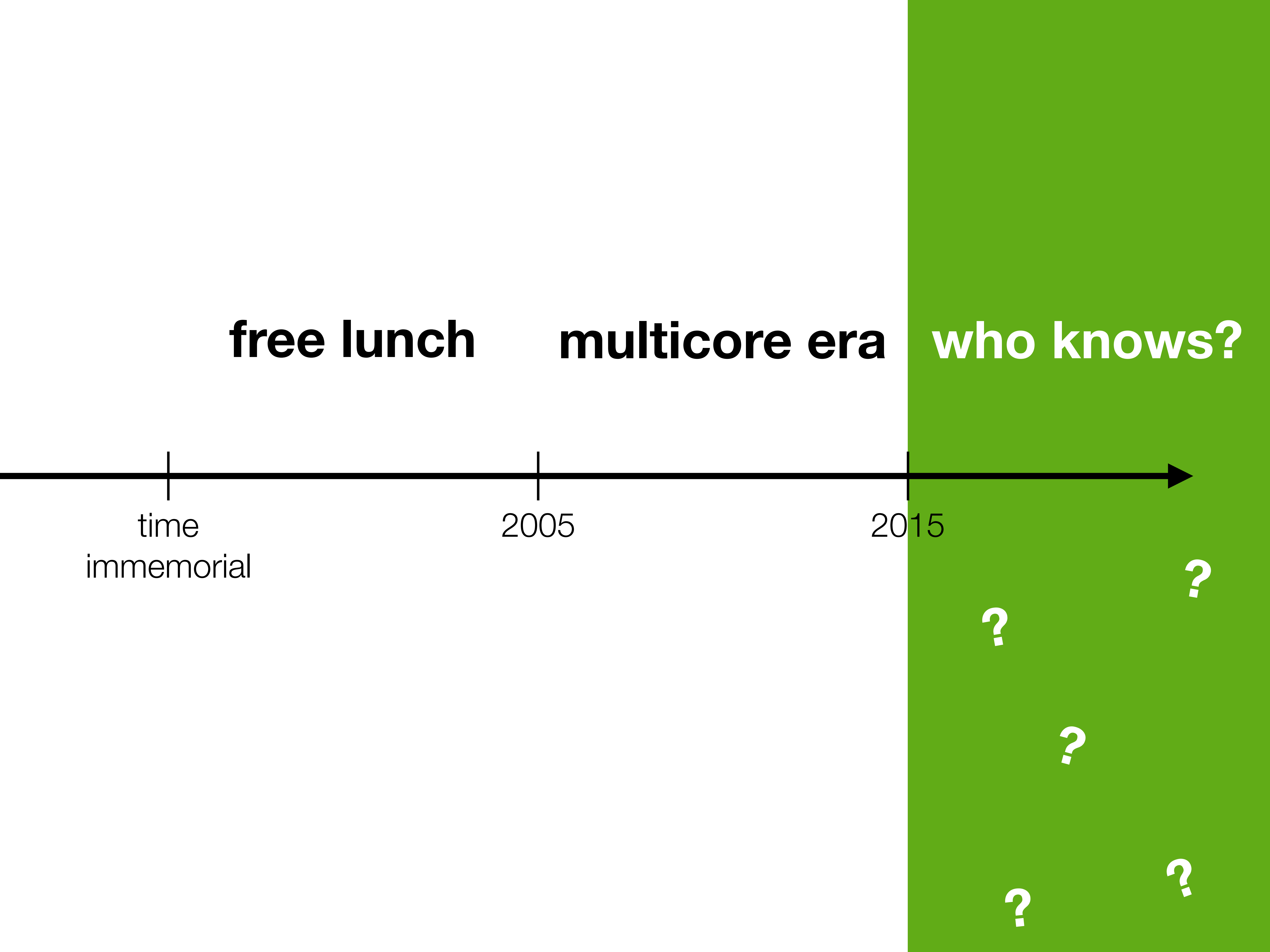

free lunch
time immemorial
2005
2015
exponential single-threaded performance scaling!
(not to scale)

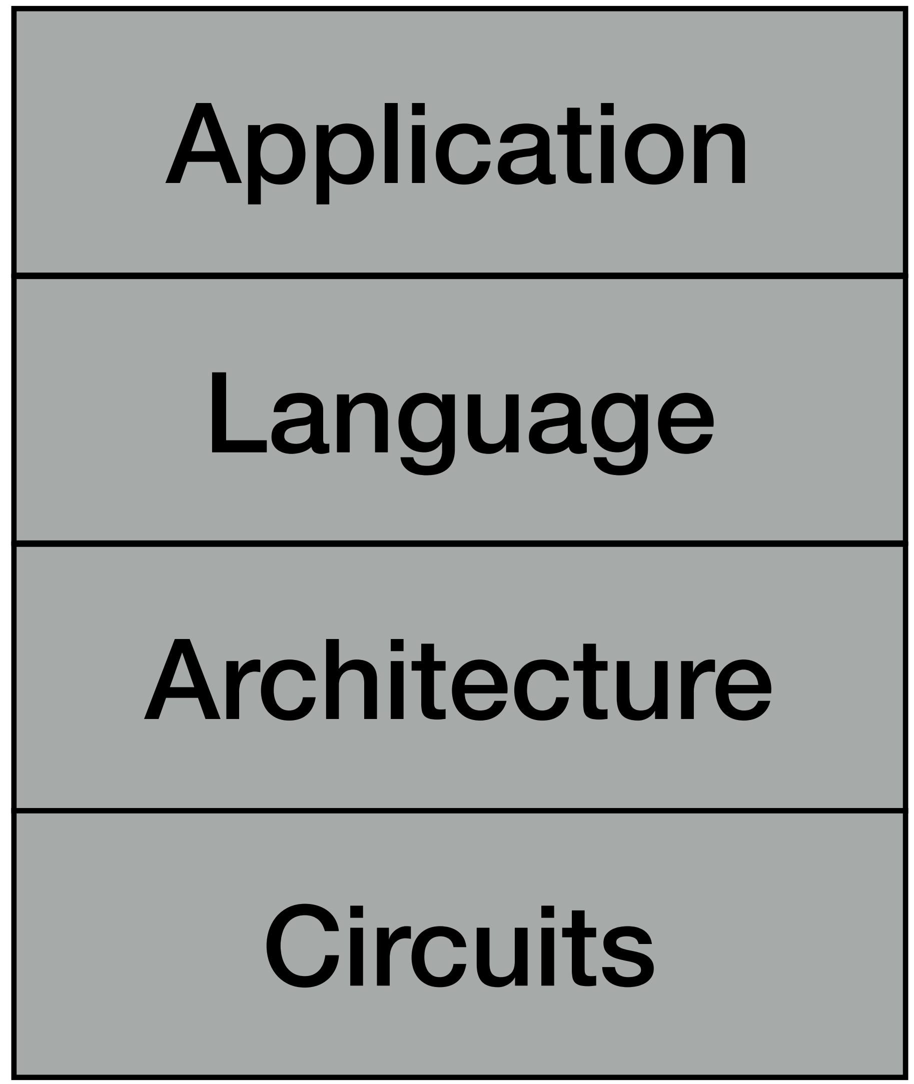

Clock Frequency (MHz)
10,000
1,000
100
10
1985
1990
1995
2000
2005
Year of Introductior

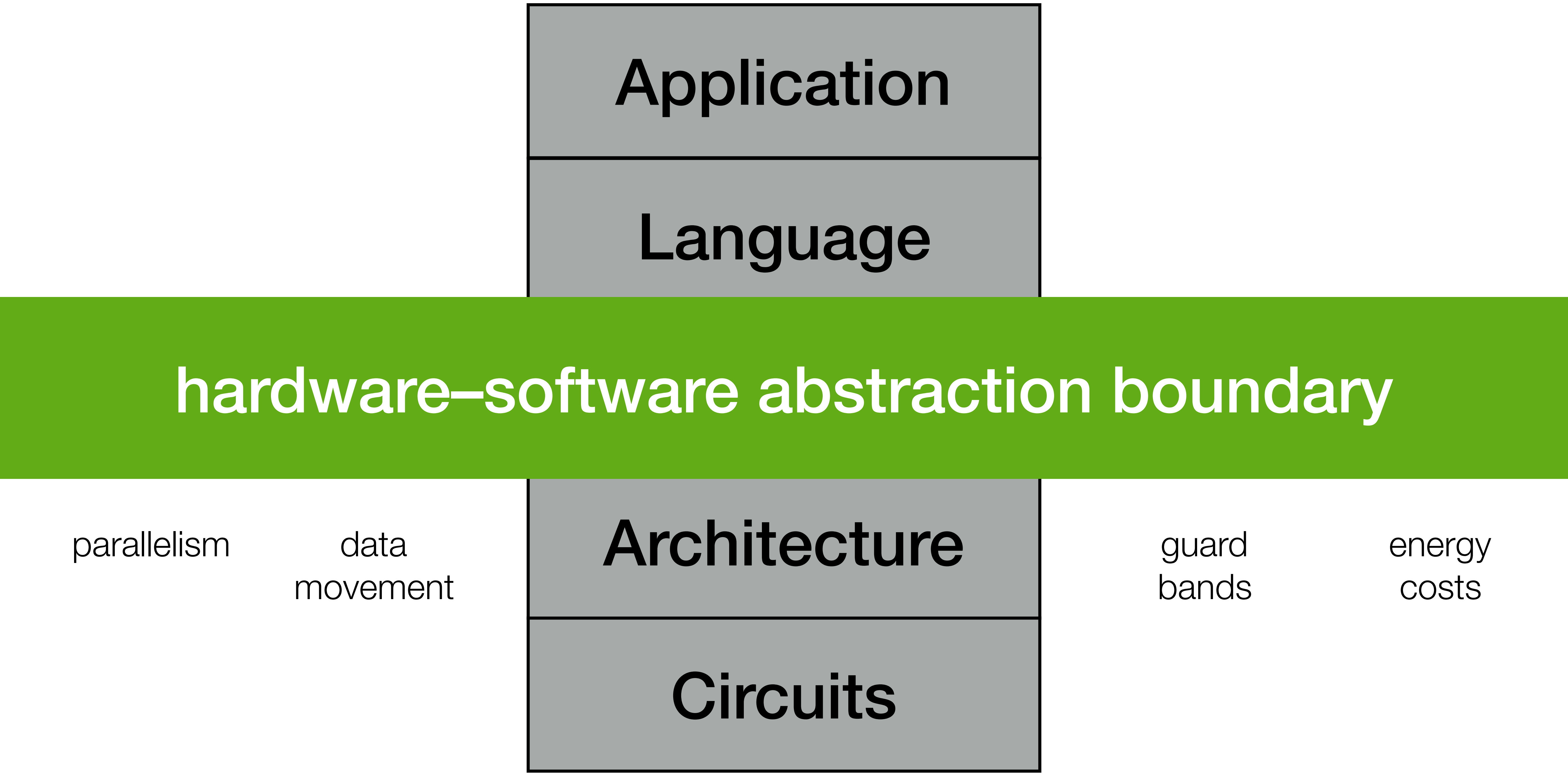


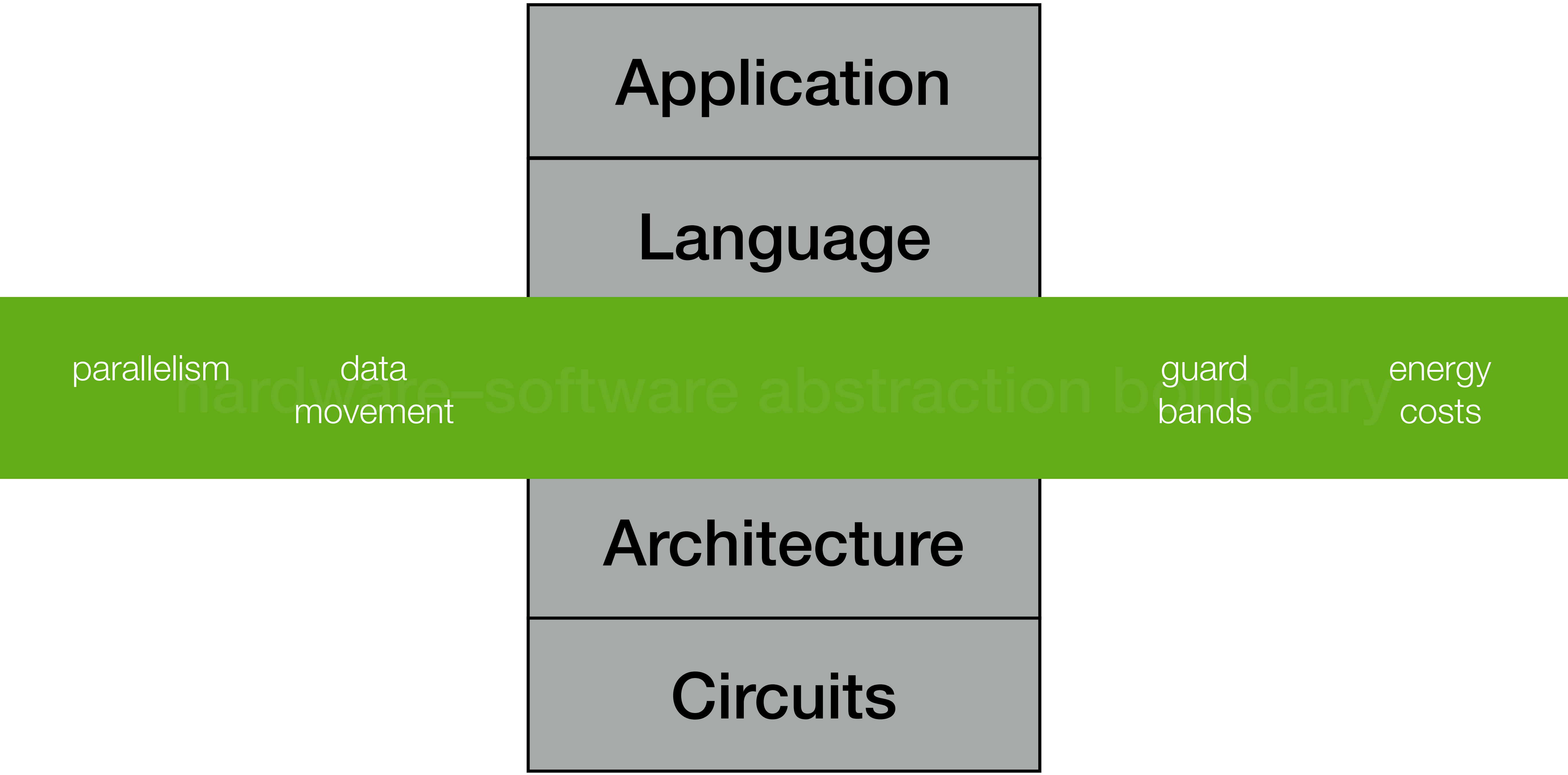

free lunch
multicore era
time
immemorial
2005
2015
we'll scale the
number of cores
instead

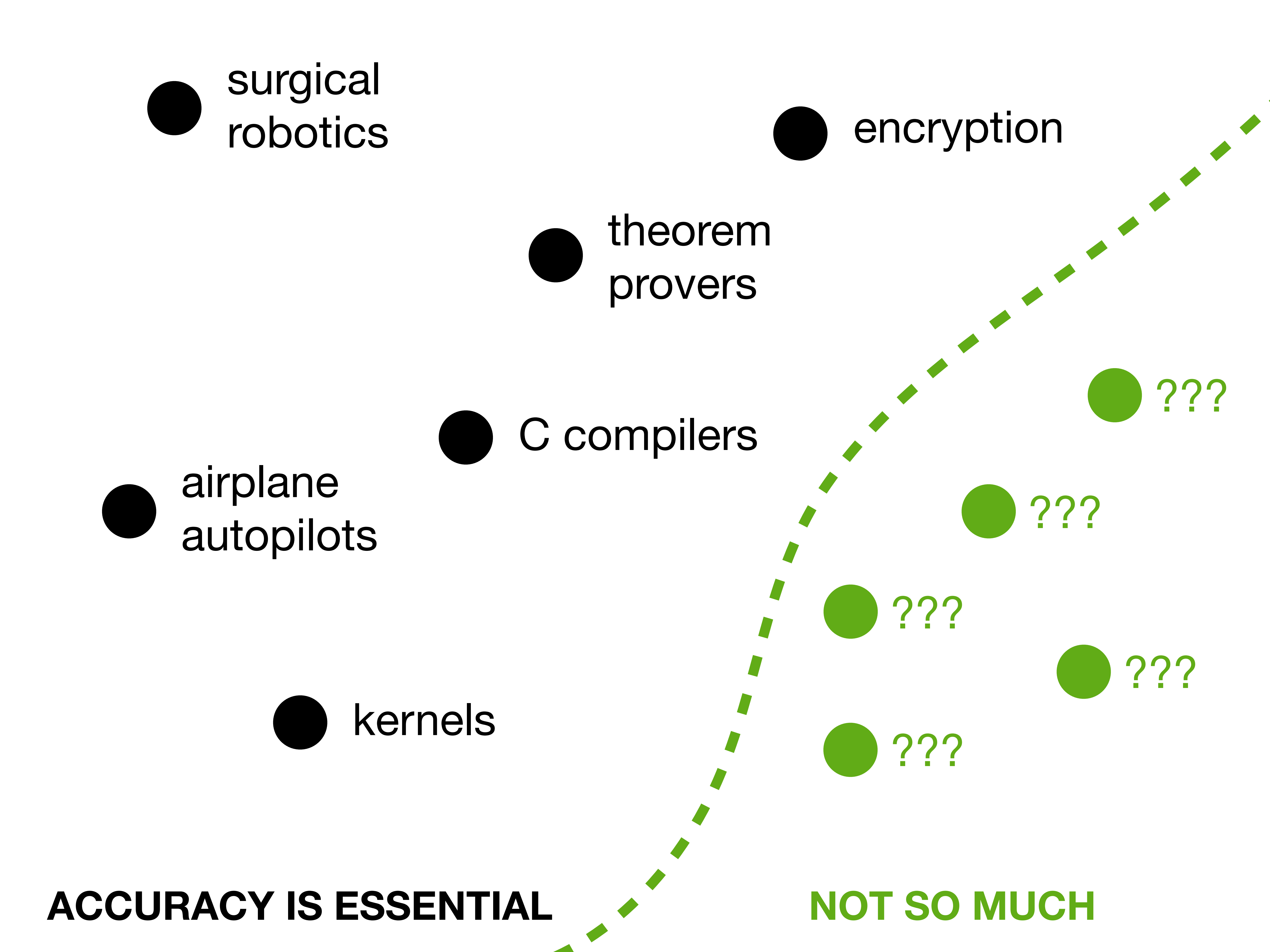
The multicore transition was a **stopgap,** not a panacea.


free lunch
multicore era
who knows?
time immemorial
2005
2015
?
?
?
?
?

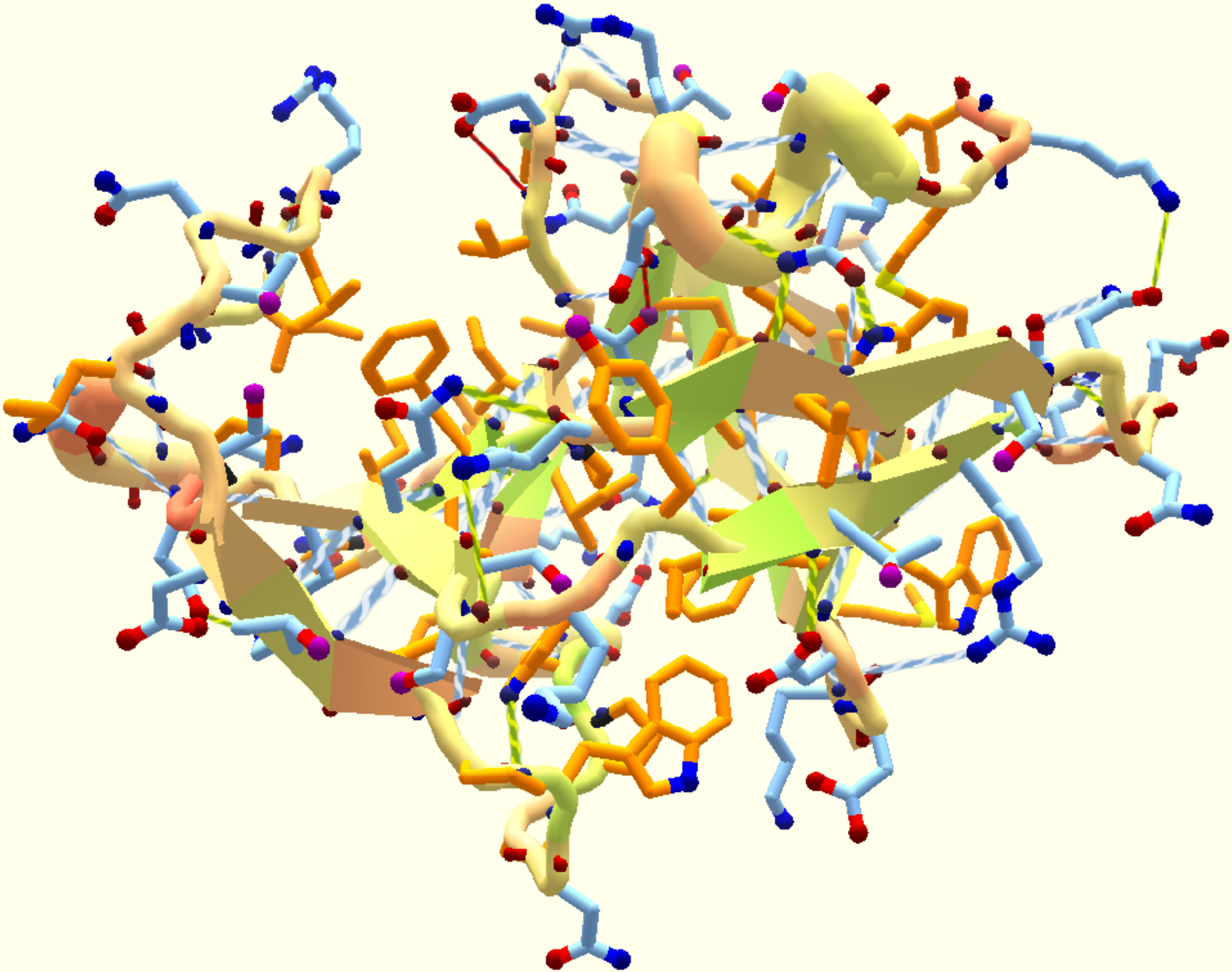

Application
Language
Architecture
Circuits

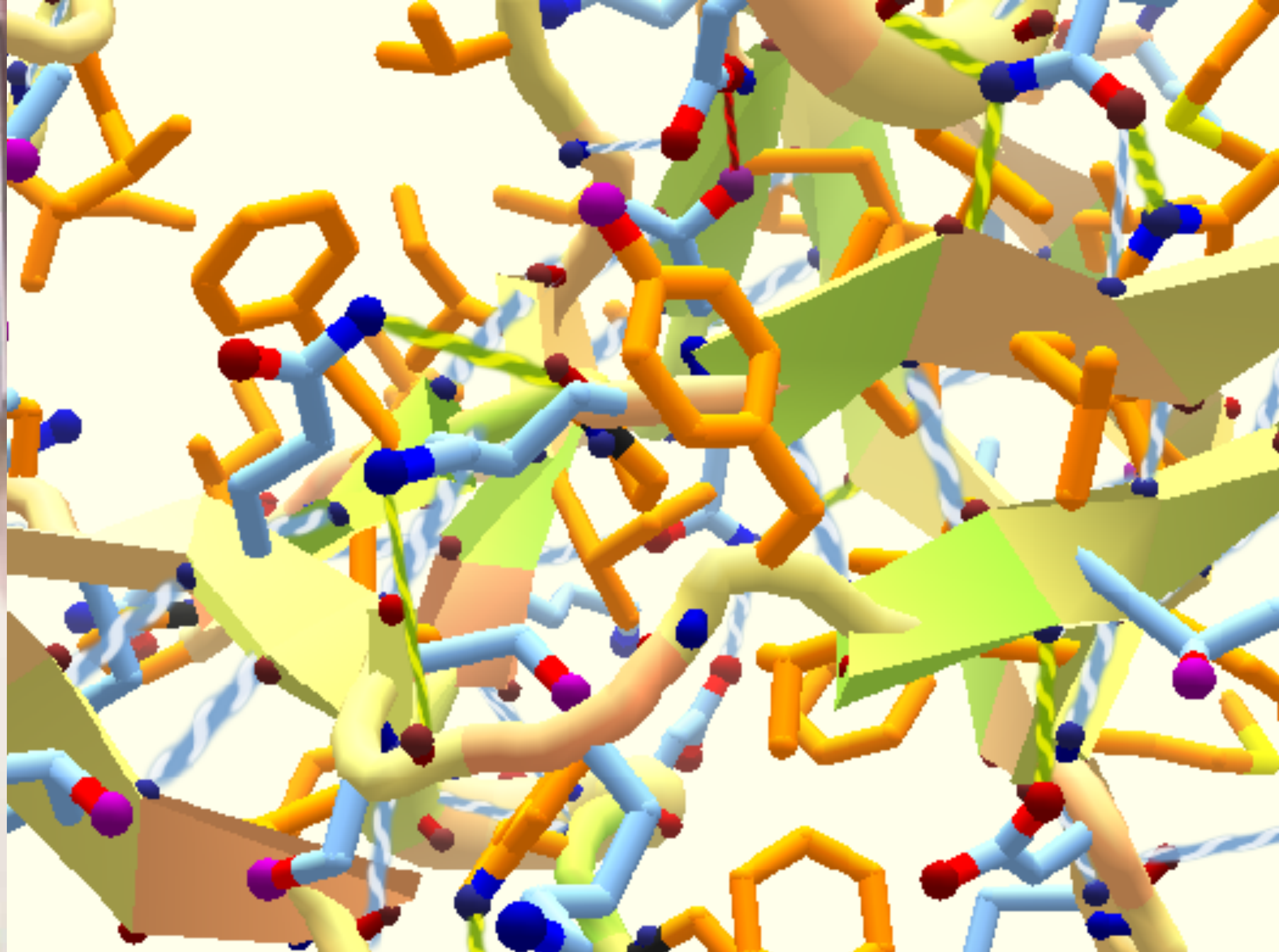

Application
Language
hardware–software abstraction boundary
Architecture
Circuits
parallelism
data movement
guard bands
energy costs

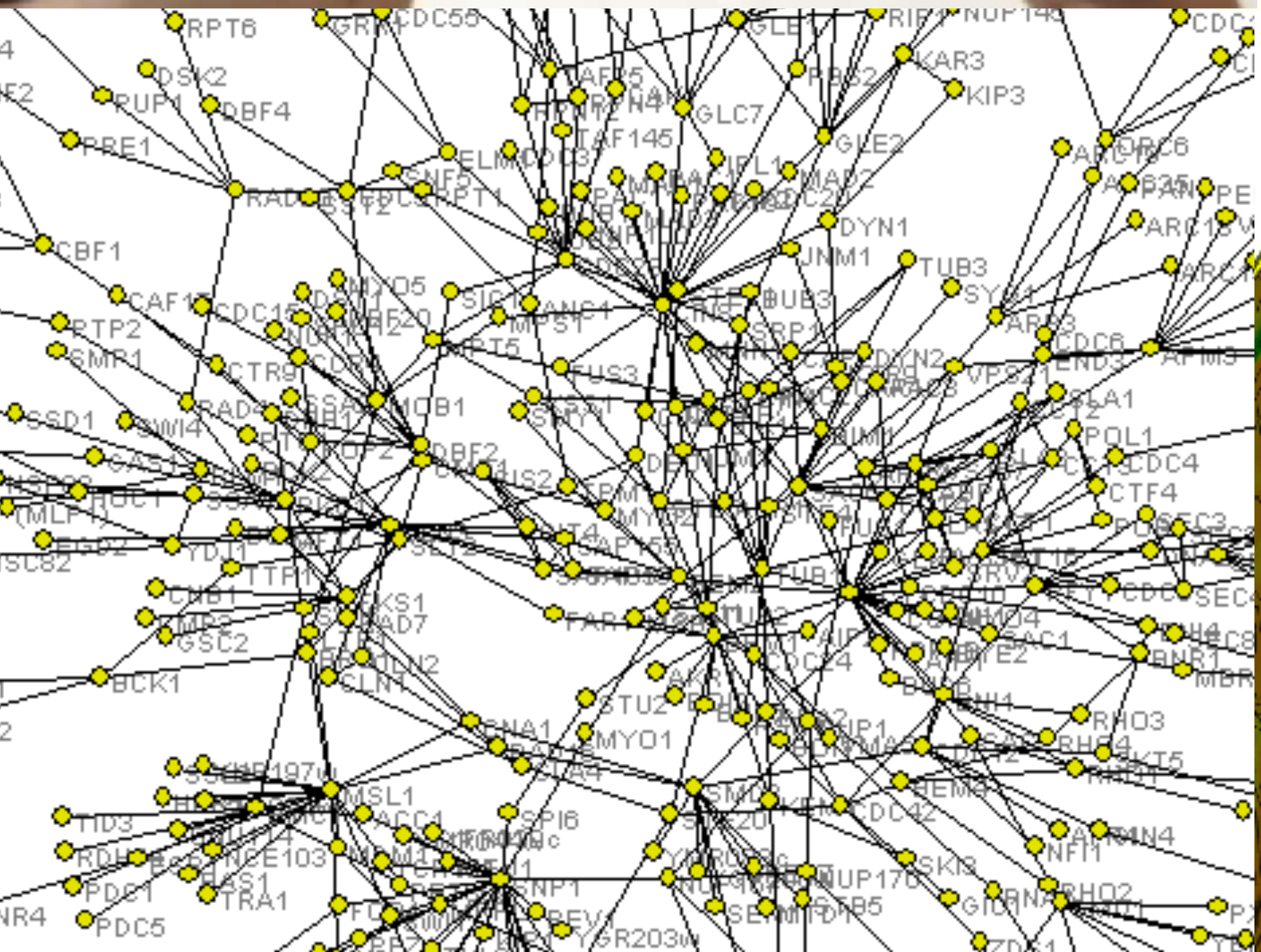

Application
Language
parallelism
data
movement
hardware–software abstraction boundary
guard
bands
energy
costs
Architecture
Circuits

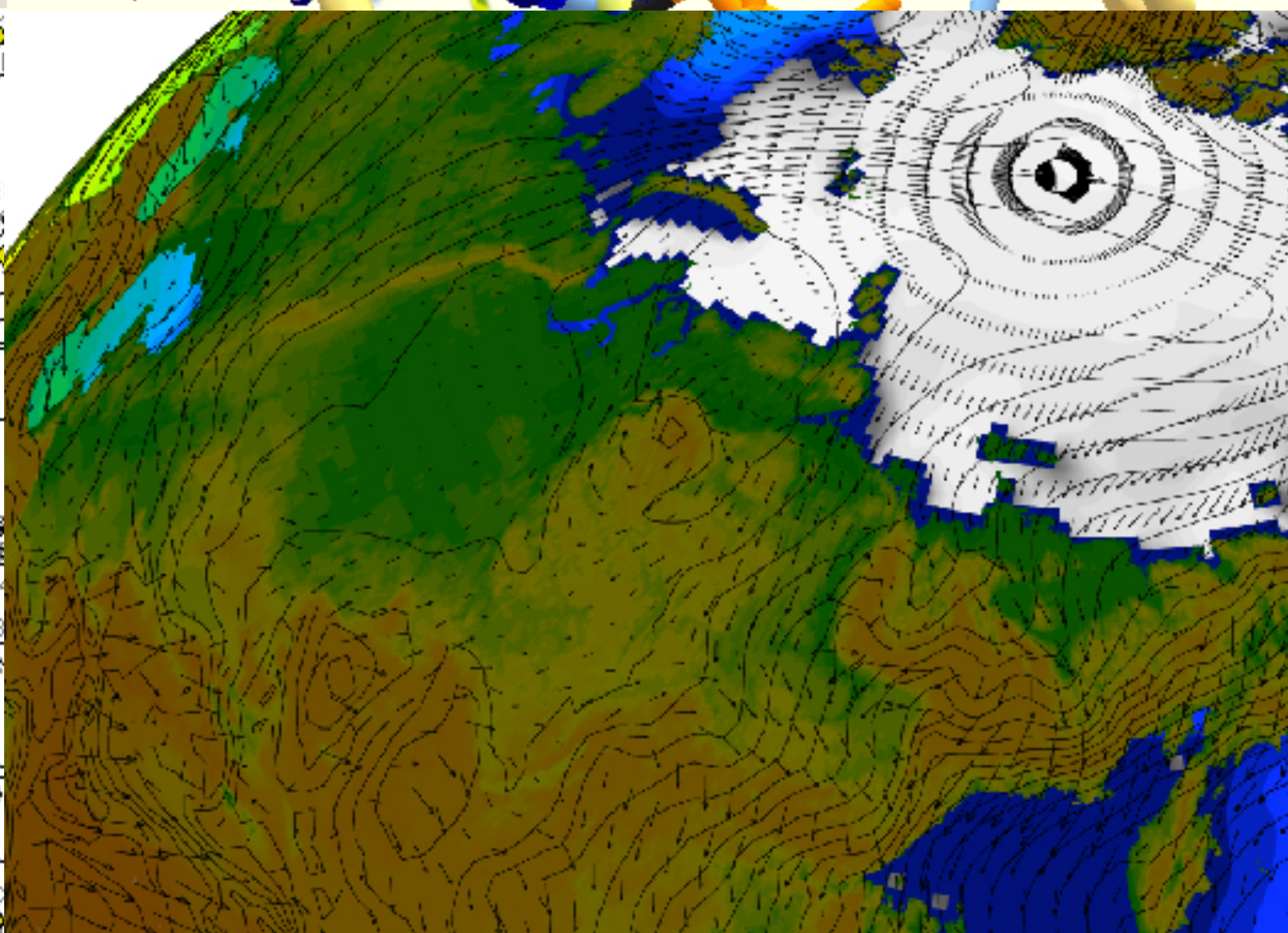

surgical robotics
encryption
theorem provers
C compilers
airplane autopilots
kernels
???
???
???
???
???
ACCURACY IS ESSENTIAL
NOT SO MUCH


SANPELLEGRINO
Italian
English
SAINT SAPWOOD
An Italian Tradition
LIMONATA

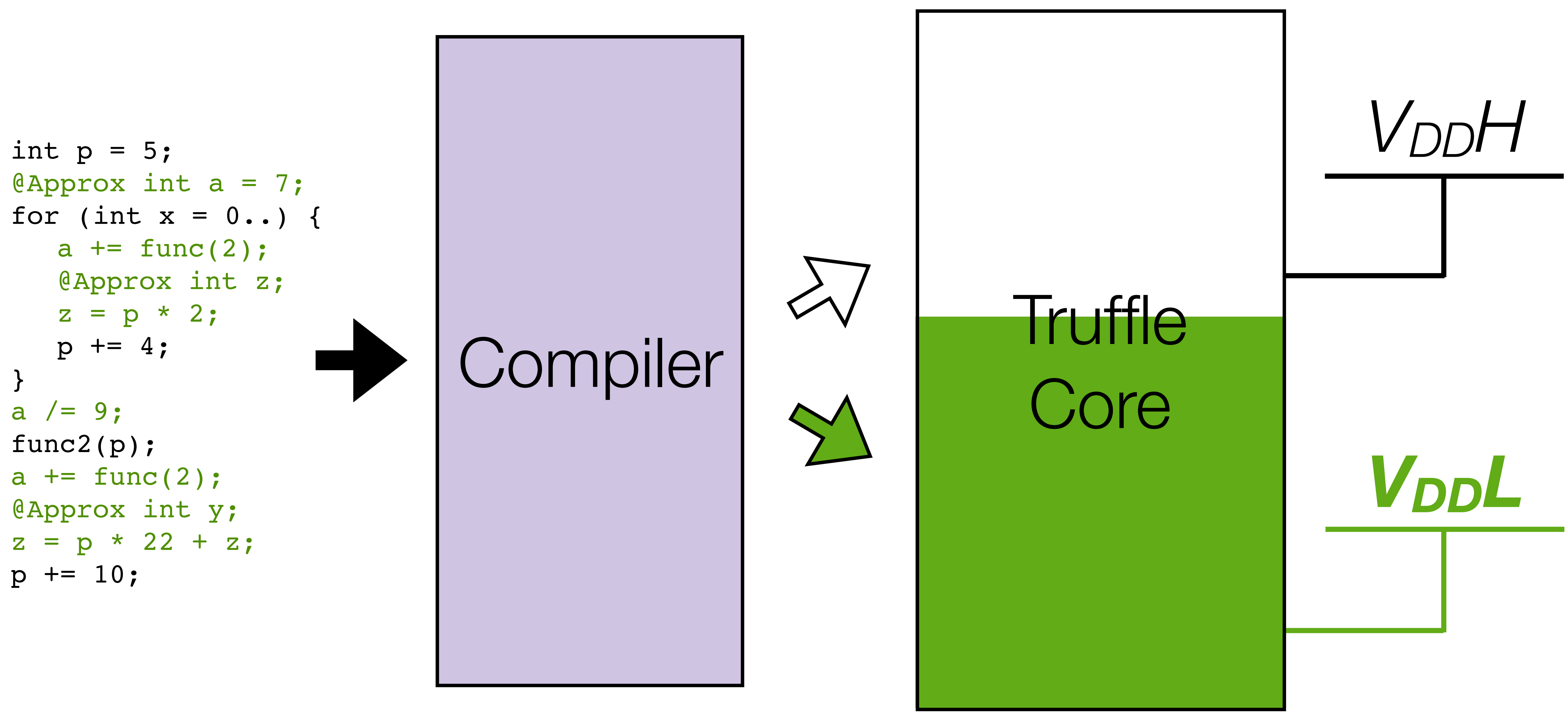





RPT6
DSK2
PUP1
DBF4
PRE1
CBF1
KAR3
KIP3
GLC7
GLE2
DYN1
JNM1
TUB3
CTF4
POL1
CDC4
BCK1
STU2
MYO1
RHO3
CDC42
RHO2
TRA1
PDC1
PDC5
CLN1
SPI6
NFI1
YGR203w



Hardware support for **disciplined approximate programming**

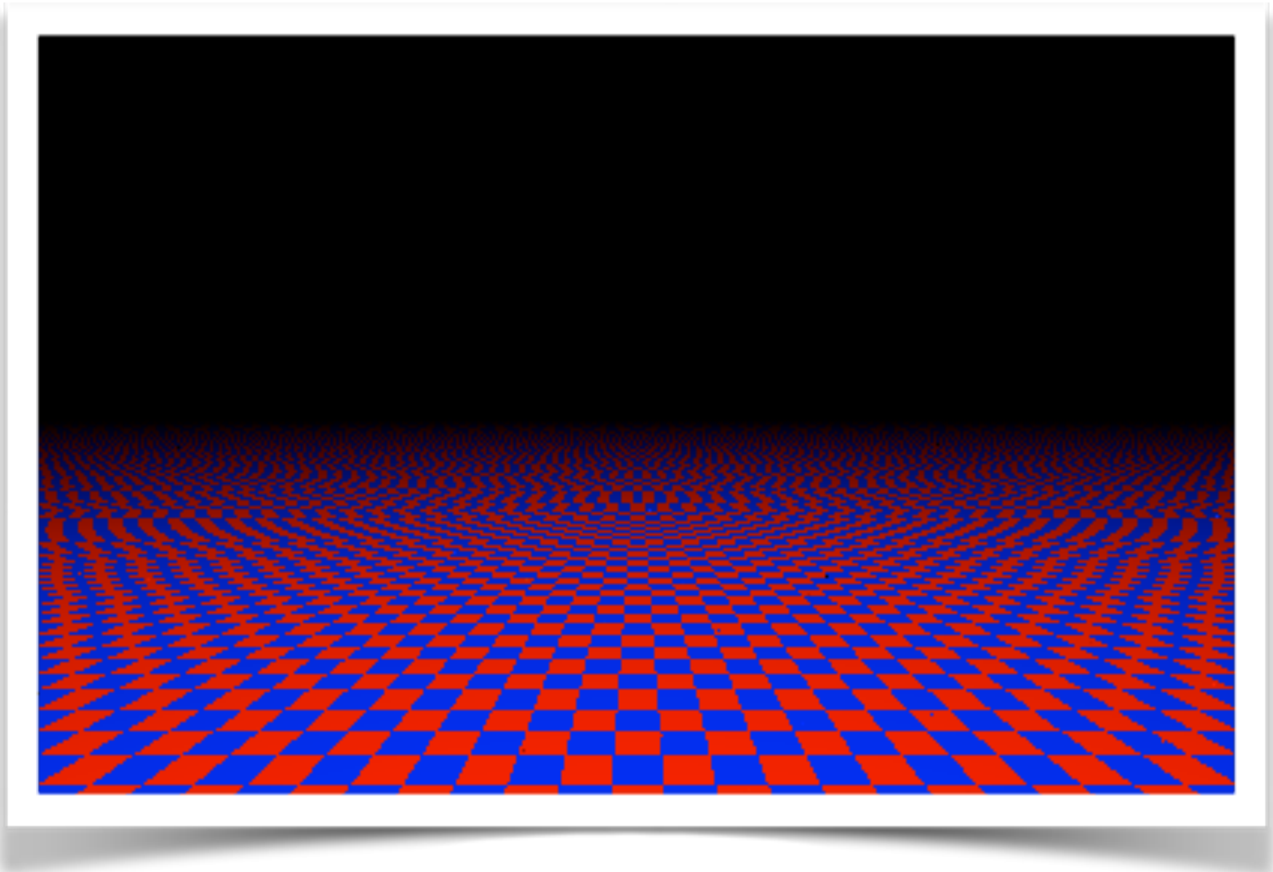
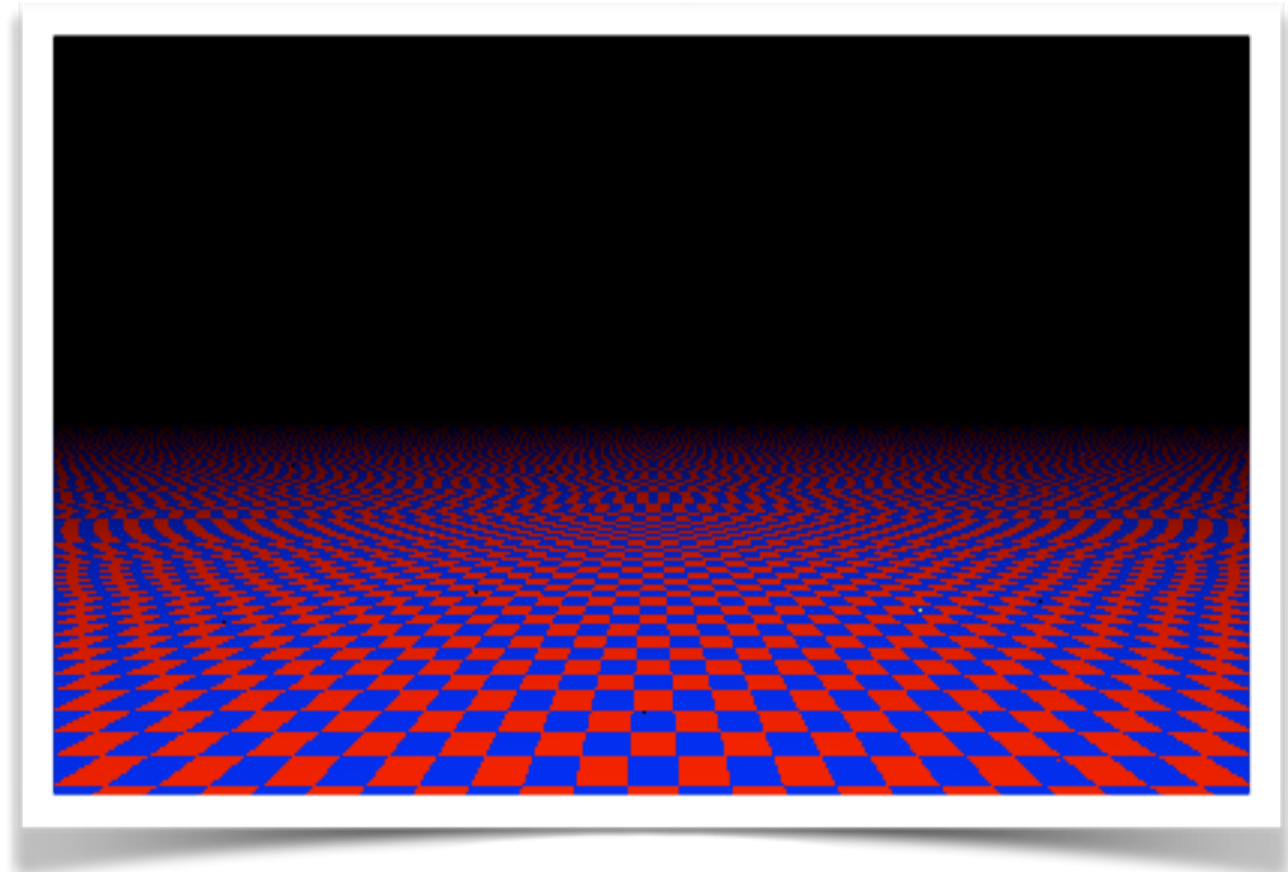
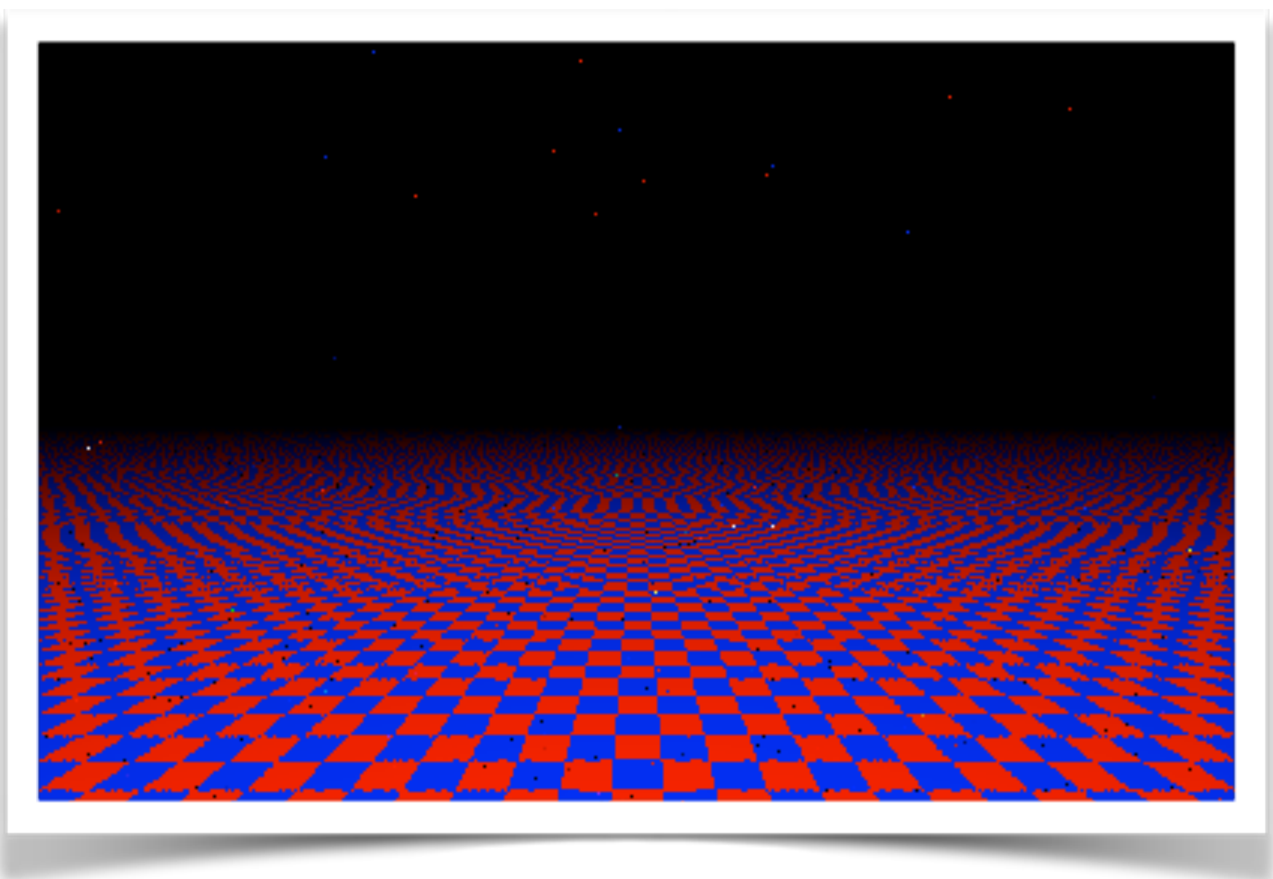
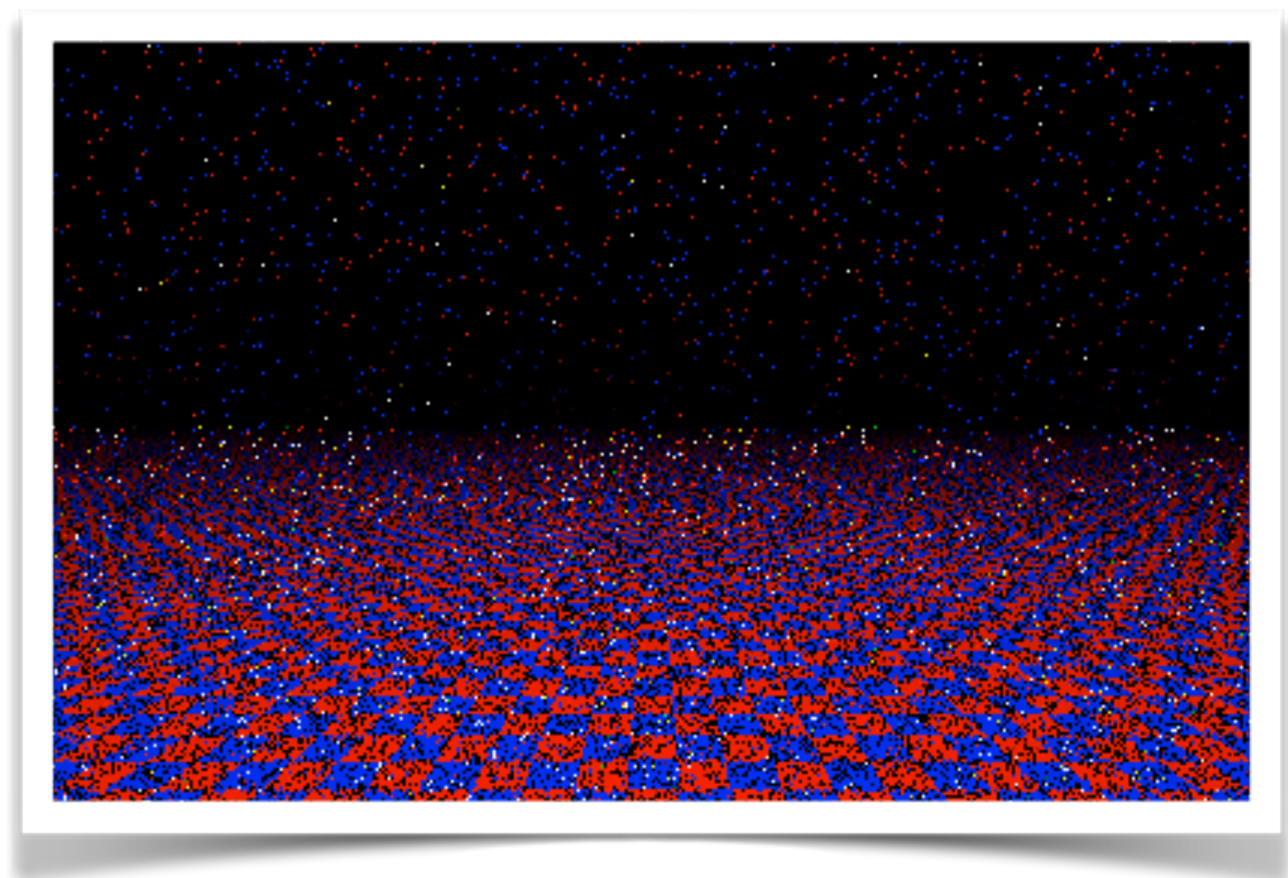


Approximation-aware ISA

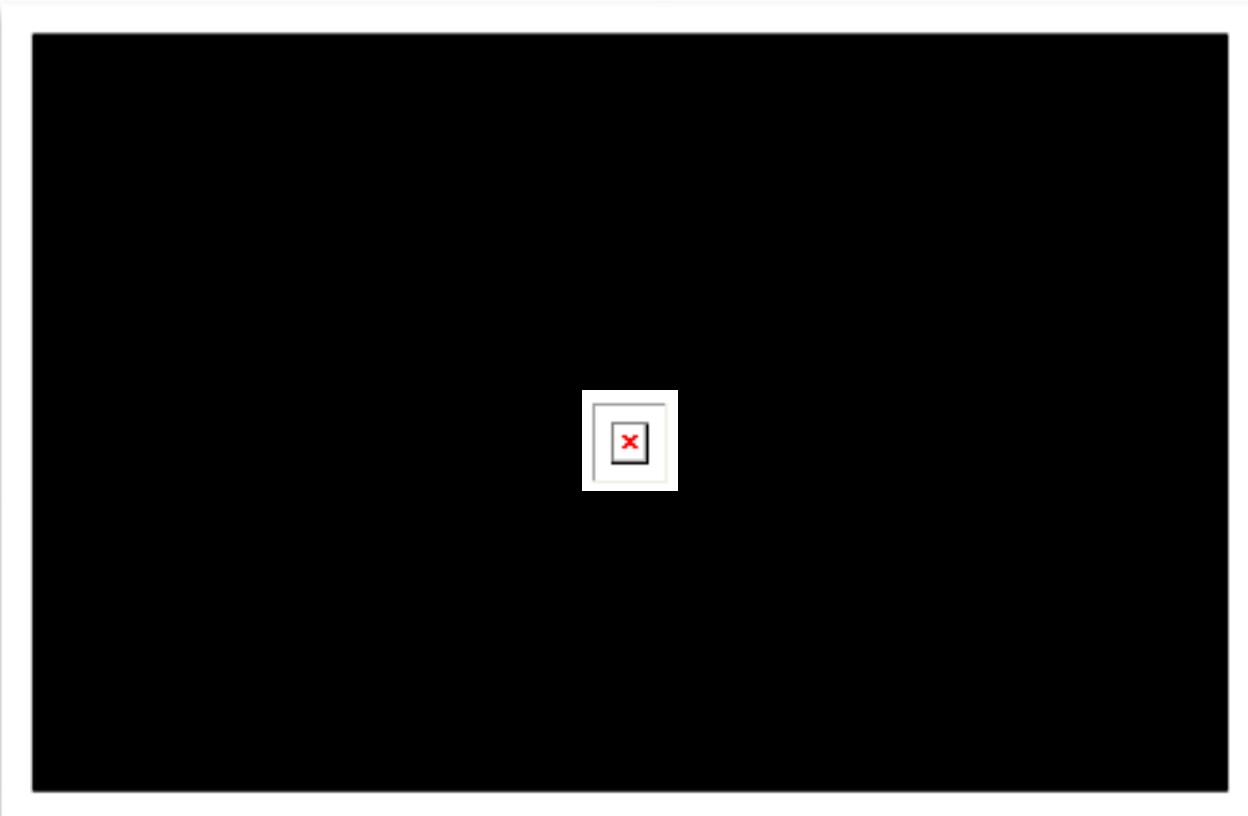
```
ld     0x04 r1
ld     0x08 r2
add    r1   r2   r3
st     0x0c r3
```

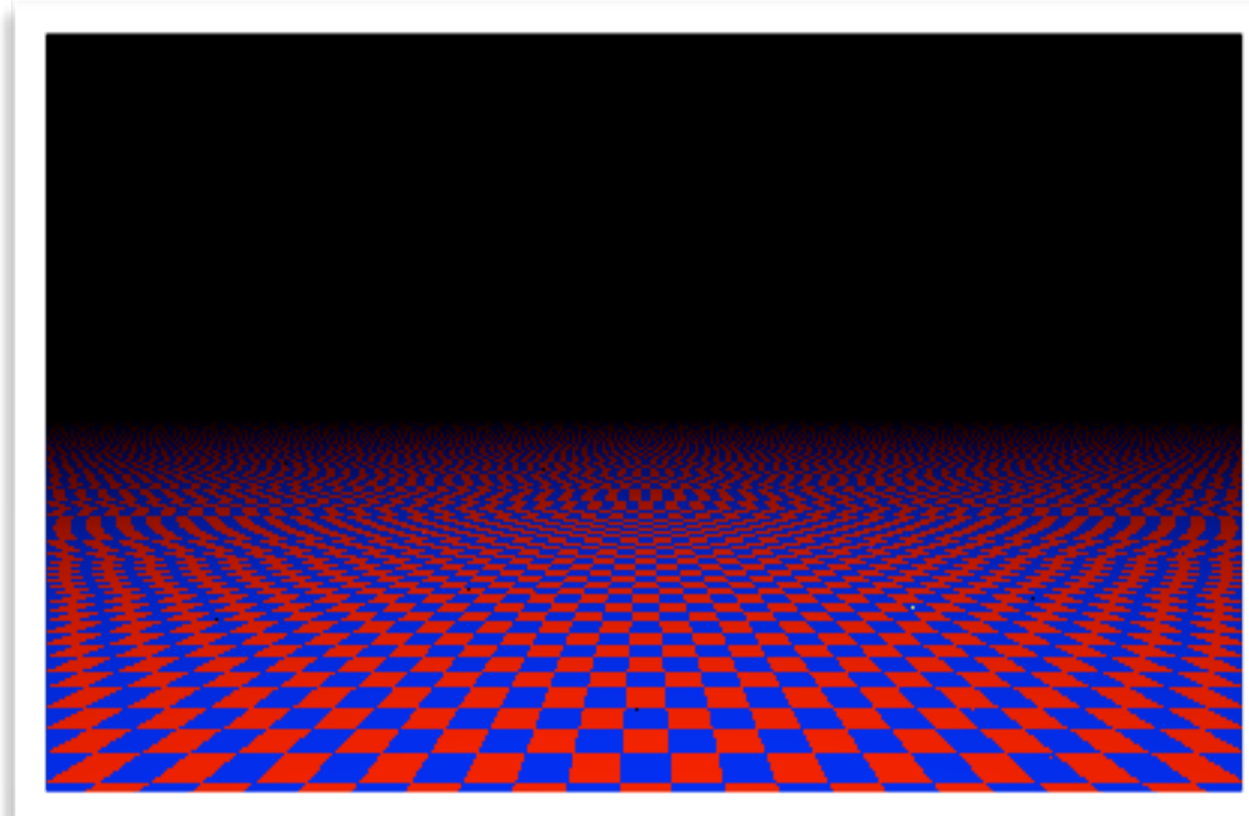
Approximation-aware ISA

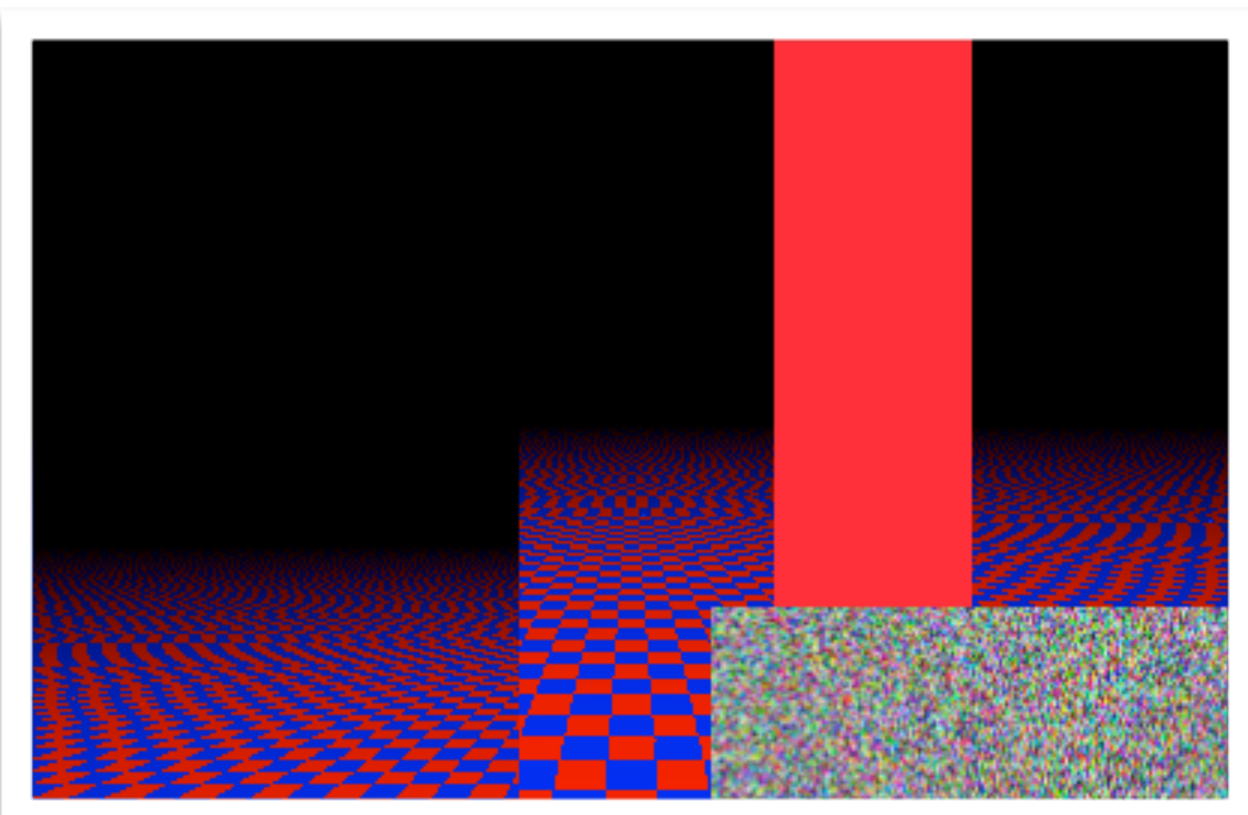
```
ld     0x04 r1
ld     0x08 r2
add.a r1    r2    r3
st.a  0x0c r3
```

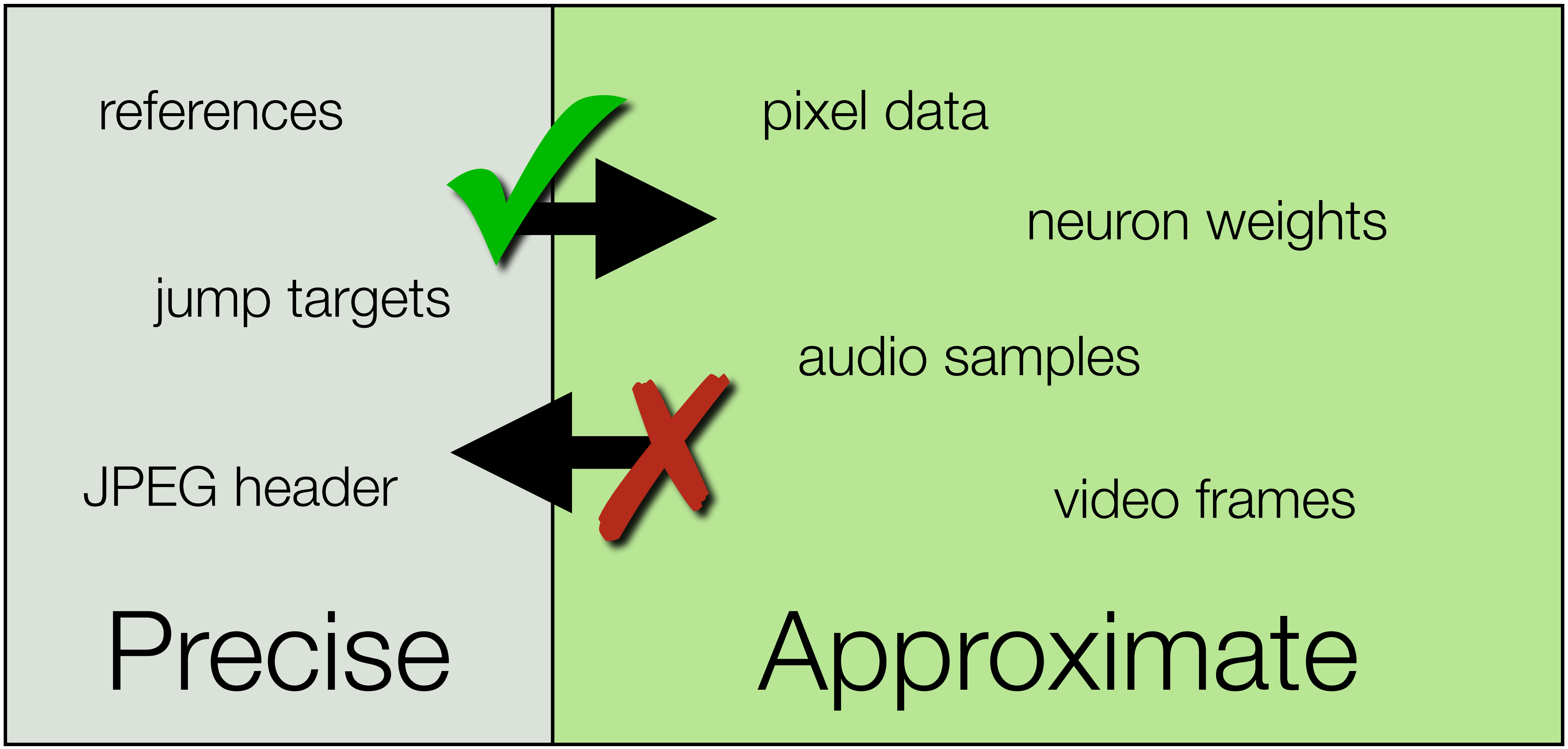

Errors
Energy
Errors
Energy
Errors
Energy
Errors
Energy







Safety by isolation



Type qualifiers

```
@Approx int a = ...;
@Precise int p = ...;
p = a;
a = p;
```

Type qualifiers

```
@Approx int a = ...;
@Precise int p = ...;
p = a;
a = p;
```

Endorsement: escape hatch

```
@Approx int a = expensive();
@Precise int p;
         p = a;
         quickChecksum(p);
         output(p);
```

Endorsement: escape hatch

```
@Approx int a = expensive();
@Precise int p;
         p = endorse(a);
         quickChecksum(p);
         output(p);
```

Logic approximation: overloading

```
  @Approx int a = ...;
 @Precise int p = ...;

          p + p;


          p + a;
          a + a;
```

Control flow: implicit flows

```
@Approx int a = ...;

@Precise int p = ...;

if (a == 10) {
    p = 2;
}
```

Control flow: implicit flows

```
@Approx int a = ...;
@Precise int p = ...;
        if (endorse(a == 10)) {
            p = 2;
        }
```

Objects

```
class FloatSet {
    float[] nums = ...;
    float mean() {
        calculate mean
    }
}
new @Approx FloatSet()
new @Precise FloatSet()
```

Objects

```
class FloatSet {
    @Context float[] nums = ...;
    float mean() {
        calculate mean
    }
}
```

```
class FloatSet {
    @Context float[] nums = ...;
    float mean() {
        calculate mean
    }
    @Approx float mean_APPROX()
    {
        take mean of first ½
    }
}
@Approx FloatSet someSet = ...;
someSet.mean();
```

EnerJ type system

$$
\begin{array}{rcl}
P & ::= & \texttt{int} \mid \texttt{float} \\
q & ::= & \texttt{precise} \mid \texttt{approx} \\
T & ::= & q\,C \mid q\,P
\end{array}
$$

$$\frac{}{\texttt{precise}\ P \leq \texttt{approx}\ P}$$ ← subtyping

“Havoc” rule

small-step operational semantics

“precise equivalence”

$$\frac{{}^{r}\Gamma \vdash h, e \;\rightsquigarrow\; h', v \qquad h' \cong \tilde{h}' \qquad v \cong \tilde{v}}{{}^{r}\Gamma \vdash h, e \;\rightsquigarrow\; \tilde{h}', \tilde{v}}$$

Noninterference theorem

program type checks

$$\left.\begin{array}{l} \vdash Prg\ \texttt{OK} \ \wedge\ \vdash h, {}^r\Gamma : {}^s\Gamma \\ {}^s\Gamma \vdash e : T \\ {}^r\Gamma \vdash h, e \rightsquigarrow h', v \\ h \cong \tilde{h} \ \wedge\ {}^r\Gamma \cong {}^r\tilde{\Gamma} \\ \vdash \tilde{h}, {}^r\tilde{\Gamma} : {}^s\Gamma \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} {}^r\tilde{\Gamma} \vdash \tilde{h}, e \rightarrow \tilde{h}', \tilde{v} \\ h' \cong \tilde{h}' \\ v \cong \tilde{v} \end{array}\right.$$

…and ending heap & value

a new precise-equivalent starting heap

steps to a heap (store) & value