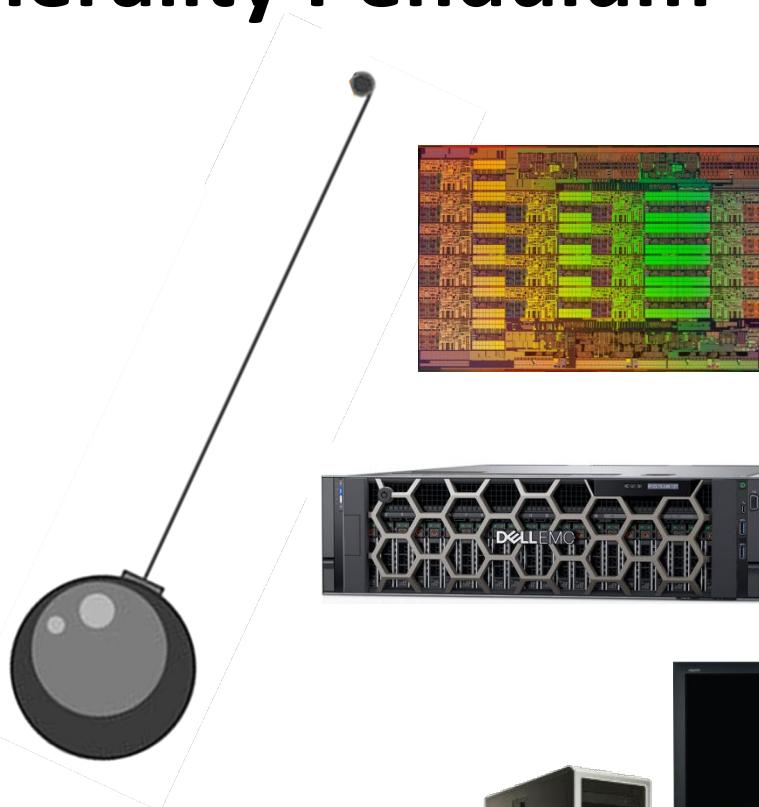
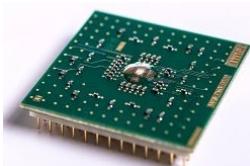
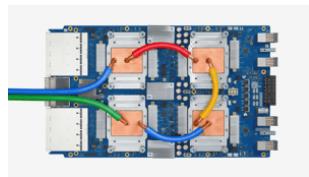
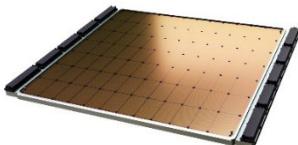


Specialization/Generality Pendulum



Power requires the pendulum to swing towards specialization

Programming Time Scale

Programming languages

- 1953: Fortran
- 1973: C
- 1985: C++
- 1997: OpenMP
- 2007: CUDA
- 2009: OpenCL

Performance libraries

- 1979: BLAS
- 1992: LAPACK
- 1994: MPI
- 1995: ScaLAPACK
- 1995: PETSc
- 1997: FFTW

Productivity/scripting languages

- 1987: Perl
- 1989: Python
- 1993: Ruby
- 1995: Java
- 2000: C#

Big Data and ML Frameworks

- 2004: MapReduce
- 2005: Hadoop
- 2006: TensorFlow
- 2007: PyTorch

Rewrite my software for
yet another
new accelerator?!

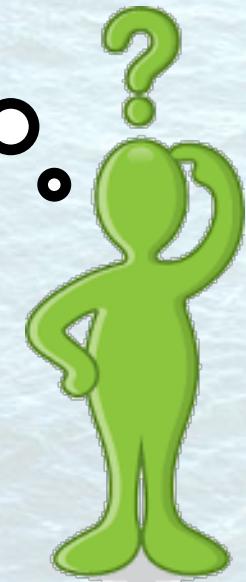
Sy

- 1970: APL
- 1972: Prolog
- 1984: Matlab
- 1982: Maple
- 1988: Mathematica
- 1990: Haskell
- 1993: R

NOPE

Numerical mathematics libraries

- 1970: IMSL
- 1971: NAG
- 2001: GNU Scientific Library
- 2003: Intel Math Kernel Library

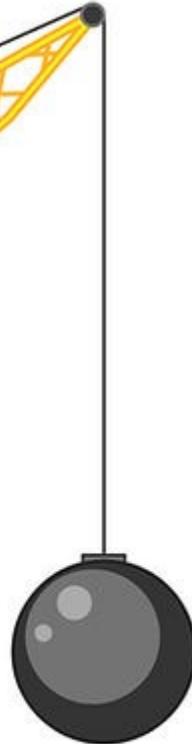


Software + applications force the pendulum towards generality

The Pendulum is a Wrecking Ball...



It just got harder...



...and more pressing



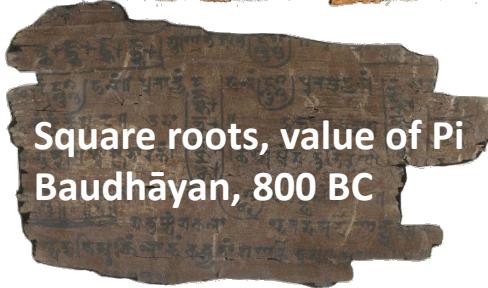
Our software investment

(Same old) urgent need:

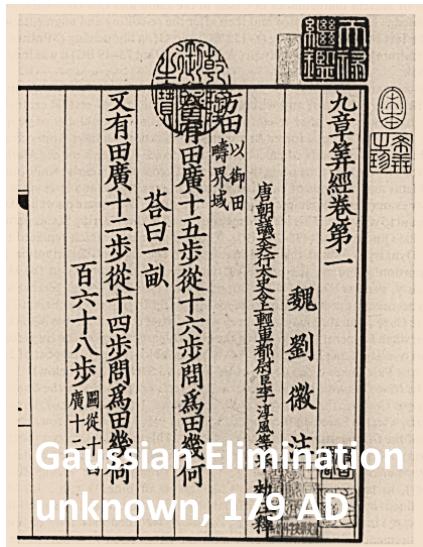
Write general purpose application code, execute efficiently on special hardware

True Longevity: Math over 2,500+ Years

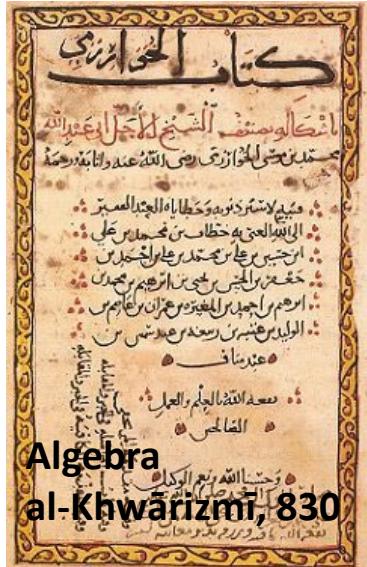
Geometry
Euclid, 300 BC



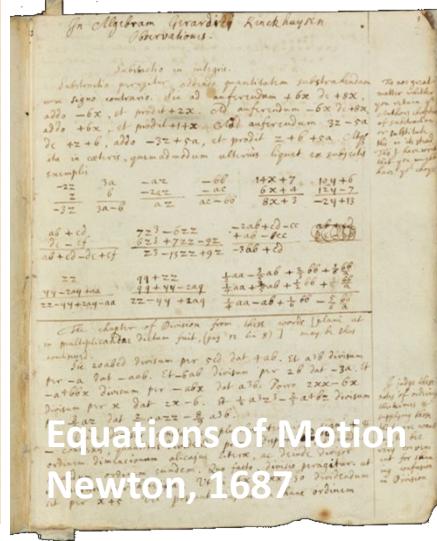
Square roots, value of Pi
Baudhāyan, 800 BC



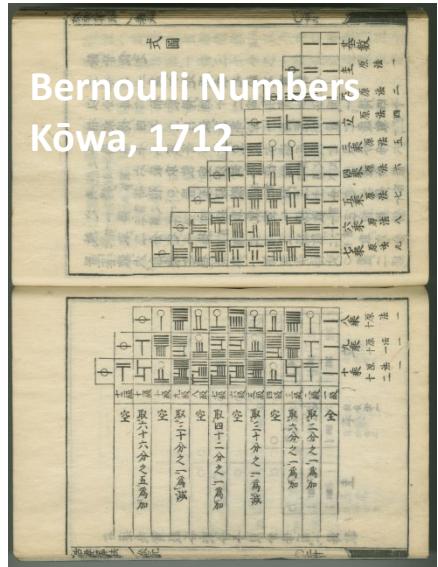
Gaussian Elimination
unknown, 179 AD



Algebra
al-Khwārizmī, 830

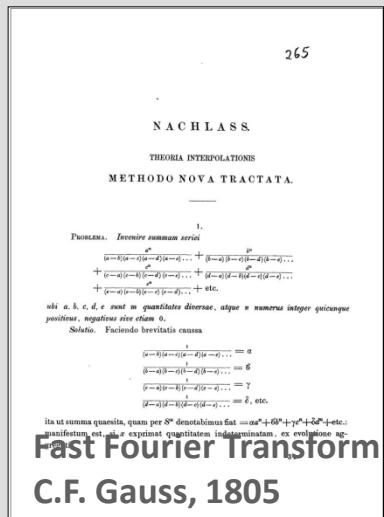


Equations of Motion
Newton, 1687



Bernoulli Numbers
Kōwa, 1712

Fast Fourier Transform: Happy 220th Birthday!



Fast Fourier Transform
C.F. Gauss, 1805

An Algorithm for the Machine Calculation of Complex Fourier Series

By James W. Cooley and John W. Tukey

An efficient method for the calculation of the interaction of a 2^n factorial experiment was introduced by Yates and is widely known by his name. The generalization to N was given by Box et al. [1]. Good [2] generalized these methods and gave simpler proofs. In this note we present a method for calculating complex Fourier series. In their full generality, Good's methods are applicable to certain problems in which one must multiply an N -vector by an $N \times N$ matrix which can be factored into two vectors of length N . This note is concerned with the case where N is a power requiring a number of operations proportional to $N \log N$ rather than N^2 . These methods are applied here to the calculation of complex Fourier series. They are used in the calculation of the interaction of a 2^n factorial experiment when N is a highly composite number. The algorithm is here derived and presented in a rather different form. Attention is given to the choice of N . It is shown how special advantages are gained by using a highly composite number as $N = 2^k$ and how the earlier calculation can be performed within the array of N data storage locations used for the given Fourier coefficients.

Consider the problem of calculating the complex Fourier series

$$(1) \quad X(j) = \sum_k A(k) e^{j \frac{2\pi}{N} k j}, \quad j = 0, 1, \dots, N-1,$$

where the given Fourier coefficients $A(k)$ are complex and N is the principal N th root of unity.

$$(2) \quad Y = Z^{1/N}$$

A straightforward calculation using (1) would require N^2 operations where "operation" means, as it will throughout this note, a complex multiplication followed by a complex addition.

The algorithm described here iterates on the array of given complex Fourier amplitudes and yields the result in less than $2N \log N$ operations without requiring more data storage than is required for the given array A . To derive the algorithm, suppose N is composite, i.e., $N = r_1 r_2 \dots$. Then let the indices in (1) be expressed

$$(3) \quad j = j_1 + j_2, \quad j_2 = 0, 1, \dots, r_1 - 1, \quad j_1 = 0, 1, \dots, r_2 - 1,$$

$$k = k_1 + k_2, \quad k_2 = 0, 1, \dots, r_1 - 1, \quad k_1 = 0, 1, \dots, r_2 - 1.$$

Then, one can write

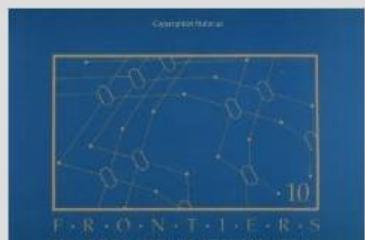
$$(4) \quad X(j_1, j_2) = \sum_{k_1=0}^{r_1-1} \sum_{k_2=0}^{r_2-1} A(k_1, k_2) Y^{j_1 k_1} e^{j \frac{2\pi}{N} k_2 j_2}.$$

Received October 1, 1965; revised January 1, 1966. This research was supported by the Air Force Research Office (Durham). The authors wish to thank Richard Garwin for his comments on a preliminary version and encouragement.

© 1966 by the American Mathematical Society

0002-9939(66)0001-0001-0

Cooley & Tukey, 1965



Computational Frameworks
for the Fast Fourier Transform
Charles Van Loan

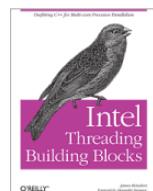
FFT in Matrix Form
Van Loan, 1992

Math To The Rescue (?)

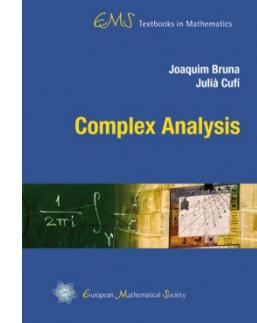
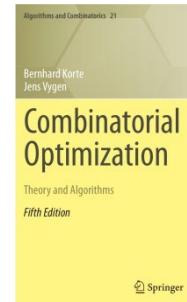
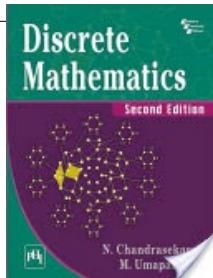
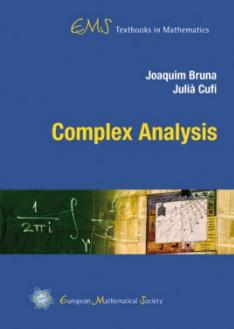
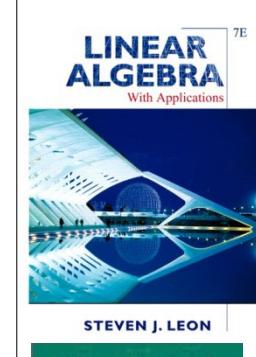
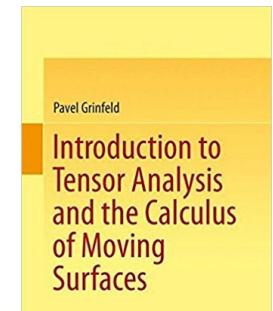
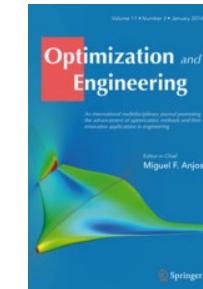
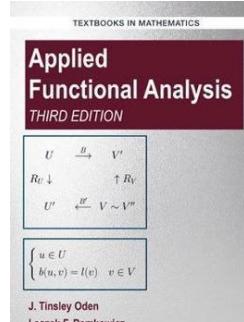
The Problem:
How to program an accelerator



Caffe

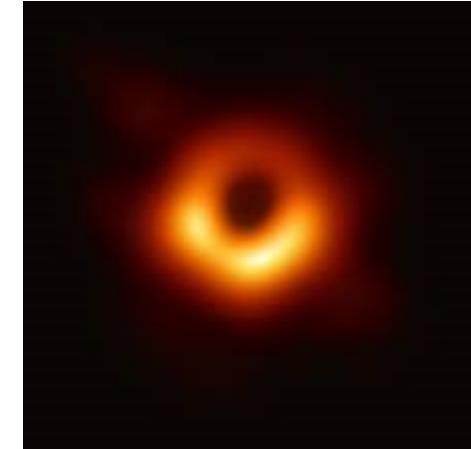
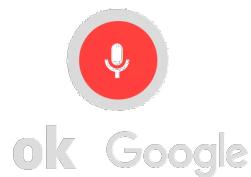
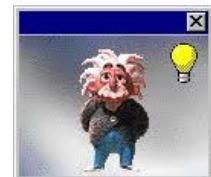


Maybe a Solution:
Higher level abstractions



Higher than what we are used to

Leverage Symbolic AI (not just ML/GenAI)



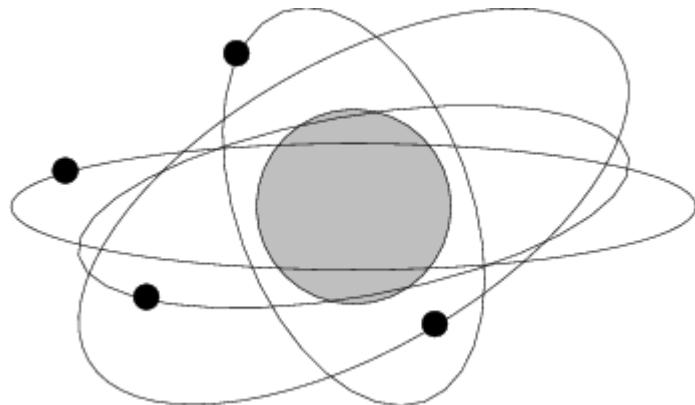
AI in pop culture

AI in CS applications

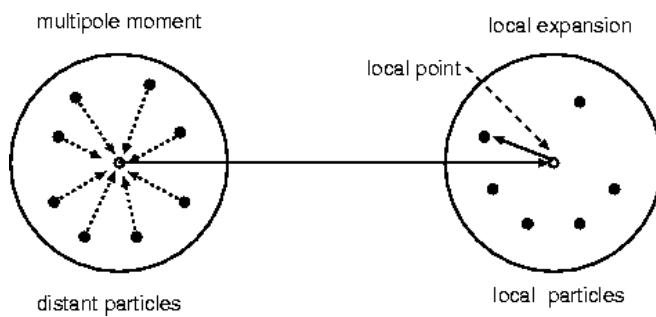
AI in compilers HPC tools

Algorithm HW Co-Design for Energy Efficiency

Physics: n -body problem



Fast multipole: $O(n \log n)$



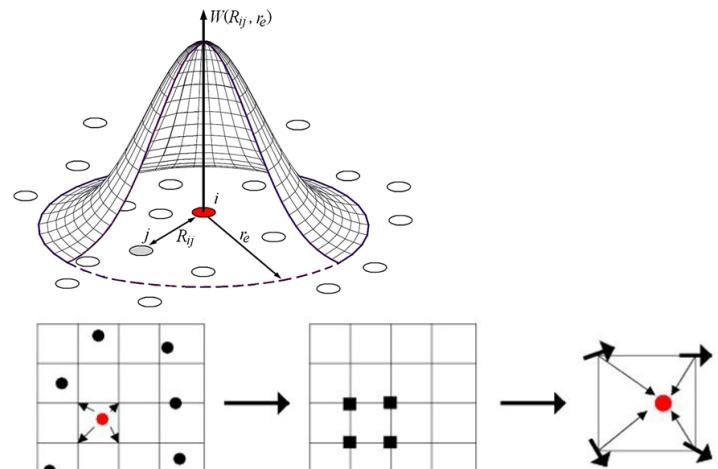
*It's quite complicated...
...but could be our*



Direct solver: $O(n^2)$

$$U_\varepsilon = \sum_{1 \leq i < j \leq n} \frac{Gm_i m_j}{\sqrt{\|\mathbf{q}_j - \mathbf{q}_i\|^2 + \varepsilon^2}}$$

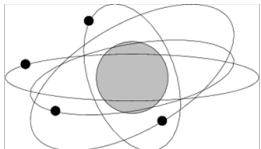
Particle/mesh: $O(k \log k)$



So: AI Must Understand The Math & Physics

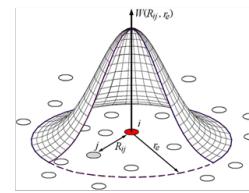
P³M: Particle-Particle + Particle Mesh

Particle-Particle: O(n²)

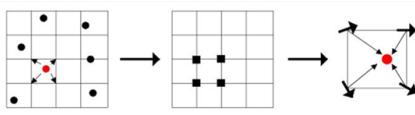


$$U_\varepsilon = \sum_{1 \leq i < j \leq n} \frac{Gm_i m_j}{\sqrt{\|\mathbf{q}_j - \mathbf{q}_i\|^2 + \varepsilon^2}}$$

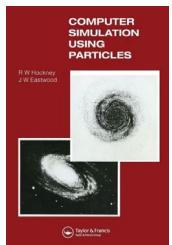
Particle/mesh: O(k log k)



Coupling: O(n) filtering

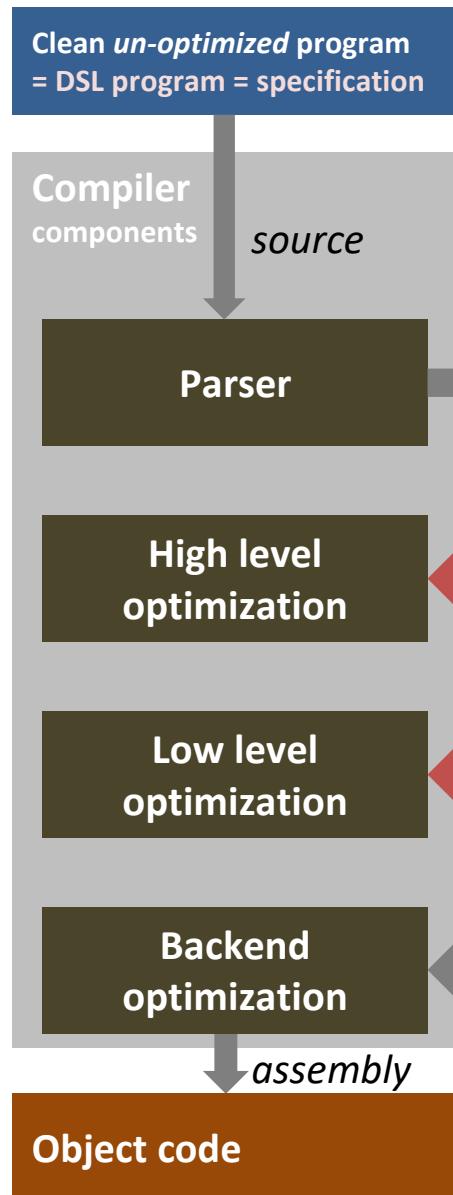
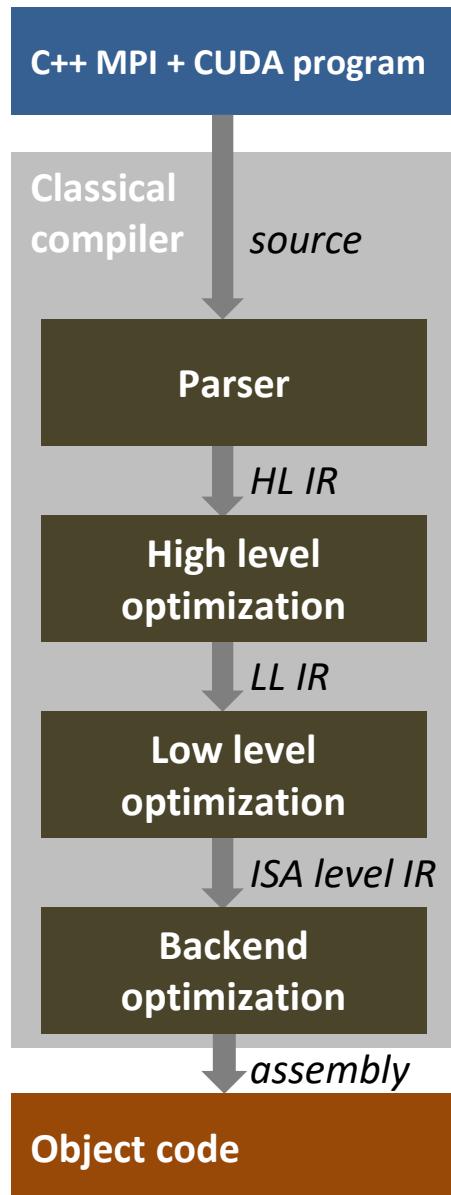


Algorithmic trade-off: memory vs. compute, accuracy vs. speed

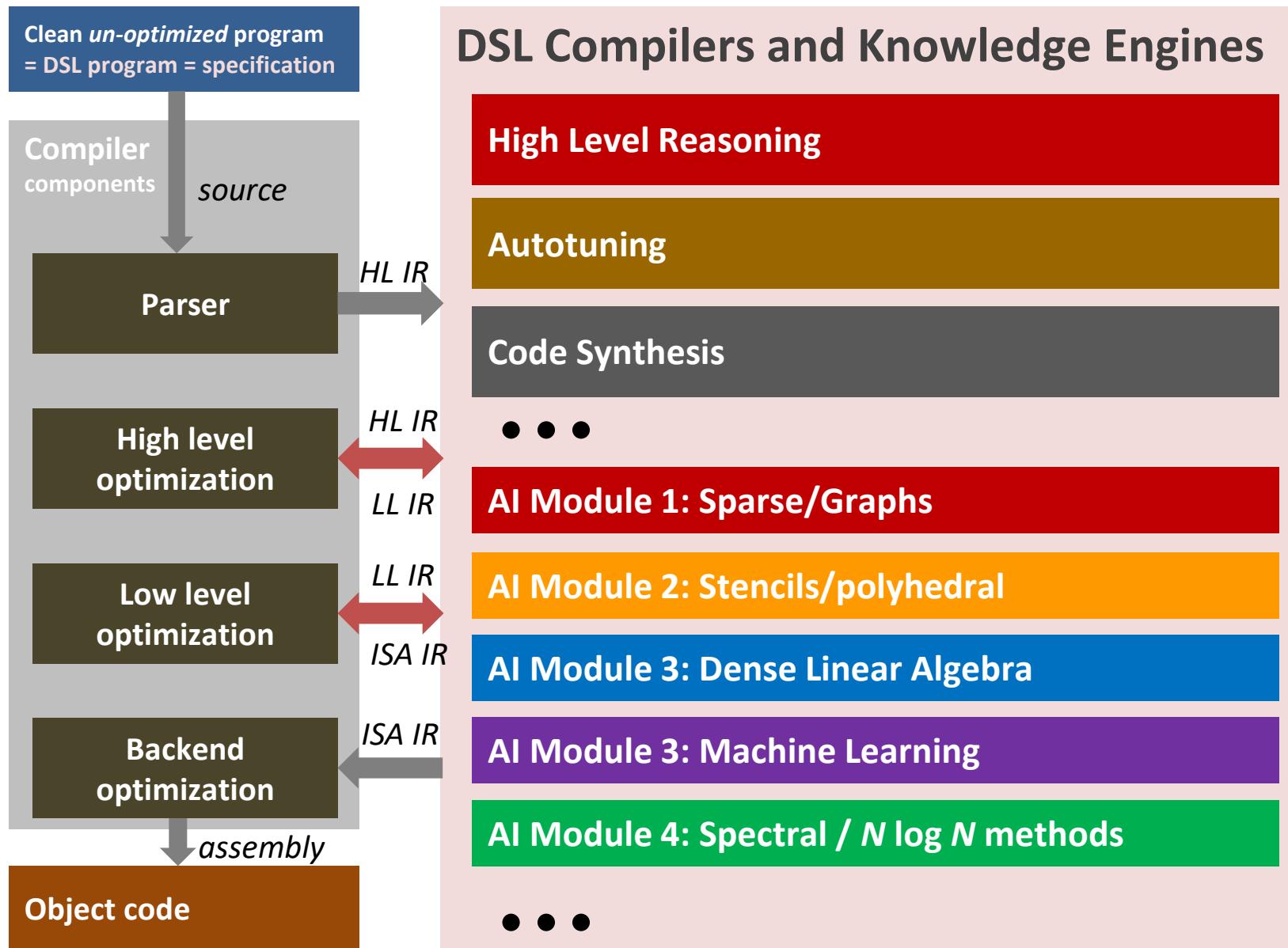


Goal: Automate algorithm/hardware co-design for power efficiency

Let's Try: High Level Reasoning in Compilers

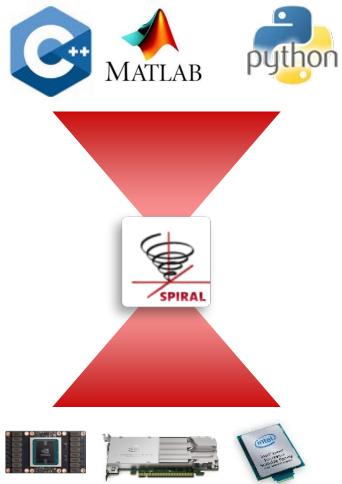


Idea: Turn Domain Knowledge into AI



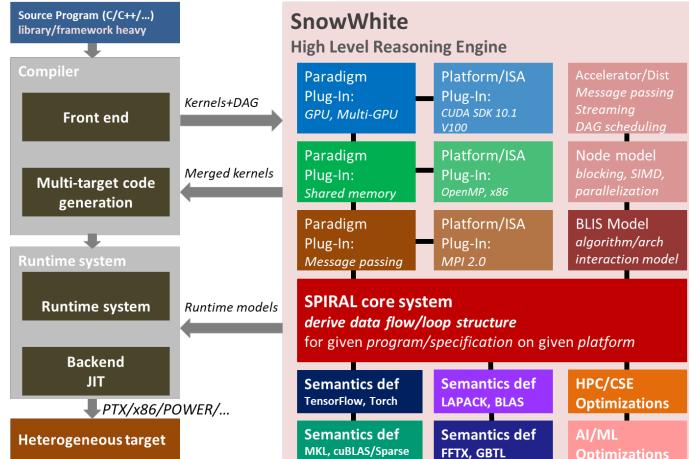
Proof of Concept: FortranX and LibraryX

Multi-language, Multi target Cross-motif optimization

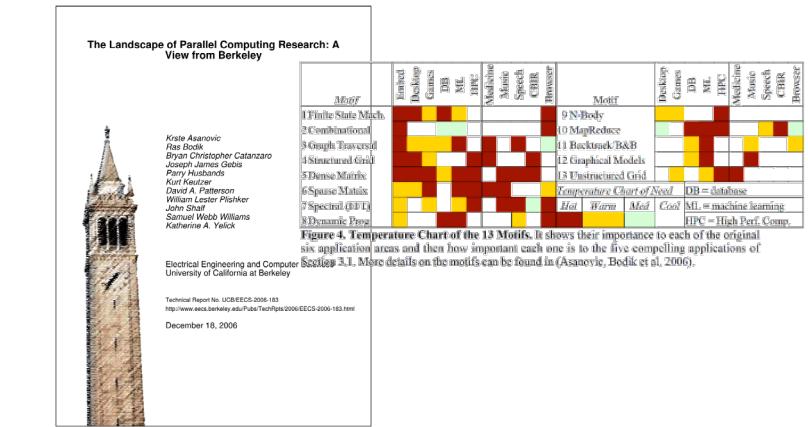


*Powered by SPIRAL
code generation/synthesis,
target multiple runtime systems*

FortranX: SPIRAL inside compilers

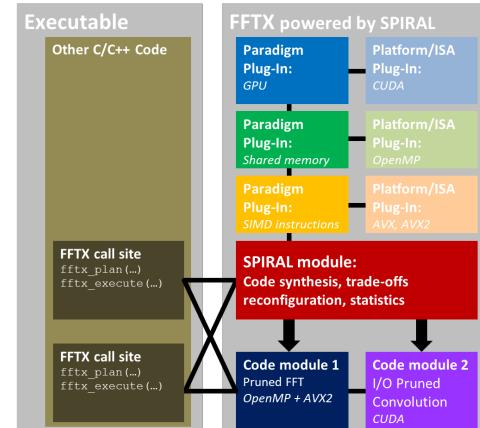


DARPA BRASS, PAPPA, X-Stack Bluestone, SciDAC, ECP



Cross-call, cross-library, cross-motif

LibraryX, powered by SPIRAL



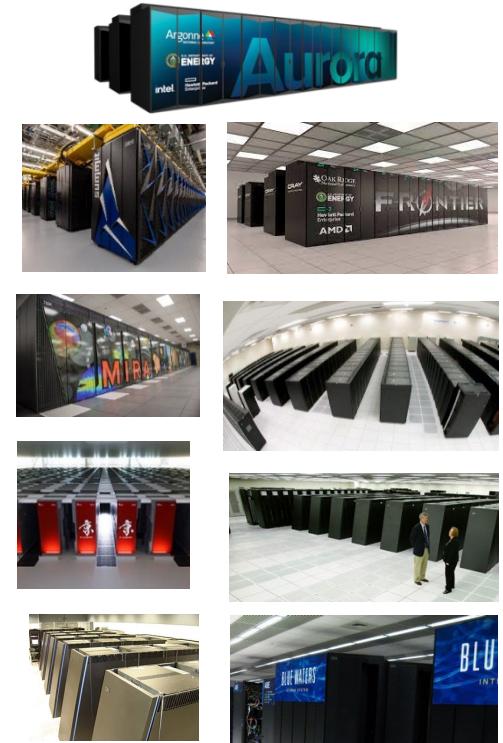
GBTX NTTX



FFTX (ECP)

Multiple active libraries, one infrastructure

FFT: It Can Be Done: For (*at least*) One Motif



Architecture



SC generations

Time

(computational and electric) power



FFT: It Can Be Done: For (*at least*) One Motif

Dynamic range per node:

- <1W – 10 kW
- 1 to 500/41k cores (CPU/GPU)
- 1 MB – 6 TB RAM
- <1 Gflop/s – 200+ Tflop/s
- 25+ years of release dates

From single node to leadership class

- <10,000 – 100,000+ nodes
- 10+ generations of #1 supercomputers

www.spiral.net

SC generations

(computational and electric) power

...one source code, one tool, always highest performance...

How do we do this for multiple (all?!?) motifs?

Brave New *Energy-Constrained* World

