Carnegie Mellon University

OAK RIDGE National Laboratory

Towards Semantics Lifting for Scientific Computing: A Case Study on FFT

Naifeng Zhang, Sanil Rao, Mike Franusich, Franz Franchetti

ORNL AI4Science Workshop

Does this code implement the fast Fourier transform?

```c
#include <math.h>
#include <stdlib.h>
#define M_PI 3.14159265358979323846
void fft_recursive(double* data, int n) {
    if (n <= 1) return;
    // Allocate temporary storage for half-size FFTs.
    double* even = (double*)malloc(n * sizeof(double));
    double* odd = (double*)malloc(n * sizeof(double));
    for (int i = 0; i < n / 2; ++i) {
        even[2 * i] = data[4 * i];
        even[2 * i + 1] = data[4 * i + 1];
        odd[2 * i] = data[4 * i + 2];
        odd[2 * i + 1] = data[4 * i + 3];
    }
    // Recursively compute FFTs.
    fft_recursive(even, n / 2);
    fft_recursive(odd, n / 2);
    for (int i = 0; i < n / 2; ++i) {
        double theta = -2.0 * M_PI * i / n;
        double wr = cos(theta);
        double wi = sin(theta);
        // Twiddle factor multiplication.
        double real = odd[2 * i] * wr - odd[2 * i + 1] * wi;
        double imag = odd[2 * i] * wi + odd[2 * i + 1] * wr;
        data[2 * i] = even[2 * i] + real;
        data[2 * i + 1] = even[2 * i + 1] + imag;
        data[2 * (i + n / 2)] = even[2 * i] - real;
        data[2 * (i + n / 2) + 1] = even[2 * i + 1] - imag;
    }
    // Cleanup.
    free(even);
    free(odd);
}
```

Complex numbers

Recursion

Sine & cosine

Hey GPT, can you provide a C implementation of the fast Fourier transform (FFT)?

Does this code implement the fast Fourier transform?

```c
#include <math.h>
#include <stdlib.h>
#define M_PI 3.14159265358979323846
void fft_recursive(double* data, int n) {
    if (n <= 1) return;
    // Allocate temporary storage for half-size FFTs.
    double* even = (double*)malloc(n * sizeof(double));
    double* odd = (double*)malloc(n * sizeof(double));
    for (int i = 0; i < n / 2; ++i) {
        even[2 * i] = data[4 * i];
        even[2 * i + 1] = data[4 * i + 1];
        odd[2 * i] = data[4 * i + 2];
        odd[2 * i + 1] = data[4 * i + 3];
    }
    // Recursive FFTs.
    fft_recursive(even, n / 2);
    fft_recursive(odd, n / 2);
    for (int i = 0; i < n / 2; ++i) {
        double theta = -2.0 * M_PI * i / n;
        double wr = cos(theta);
        double wi = sin(theta);
        // Twiddle factor multiplication.
        double real = odd[2 * i] * wr - odd[2 * i + 1] * wi;
        double imag = odd[2 * i] * wi + odd[2 * i + 1] * wr;
        data[2 * i] = even[2 * i] + real;
        data[2 * i + 1] = even[2 * i + 1] + imag;
        data[2 * (i + n / 2)] = even[2 * i] - real;
        data[2 * (i + n / 2) + 1] = even[2 * i + 1] - imag;
    }
    // Cleanup.
    free(even);
    free(odd);
}
```

Complex numbers

Recursion

Sine & cosine

How to verify the *semantic correctness* for LLM-generated code?

Hey GPT, can you provide a C implementation of the fast Fourier transform (FFT)?

We can solve for small sizes using symbolic execution

```c
#include <math.h>
#include <stdlib.h>
#define M_PI 3.14159265358979323846
void fft_recursive(double* data, int n) {
    if (n <= 1) return;
    // Allocate temporary storage for half-size FFTs.
    double* even = (double*)malloc(n * sizeof(double));
    double* odd = (double*)malloc(n * sizeof(double));
    for (int i = 0; i < n / 2; ++i) {
        even[2 * i] = data[4 * i];
        even[2 * i + 1] = data[4 * i + 1];
        odd[2 * i] = data[4 * i + 2];
        odd[2 * i + 1] = data[4 * i + 3];
    }
    // Recursively compute FFTs.
    fft_recursive(even, n / 2);
    fft_recursive(odd, n / 2);
    for (int i = 0; i < n / 2; ++i) {
        double theta = -2.0 * M_PI * i / n;
        double wr = cos(theta);
        double wi = sin(theta);
        // Twiddle factor multiplication.
        double real = odd[2 * i] * wr - odd[2 * i + 1] * wi;
        double imag = odd[2 * i] * wi + odd[2 * i + 1] * wr;
        data[2 * i] = even[2 * i] + real;
        data[2 * i + 1] = even[2 * i + 1] + imag;
        data[2 * (i + n / 2)] = even[2 * i] - real;
        data[2 * (i + n / 2) + 1] = even[2 * i + 1] - imag;
    }
    // Cleanup.
    free(even);
    free(odd);
}
```

FFT is a linear transform: $y = T \cdot x$

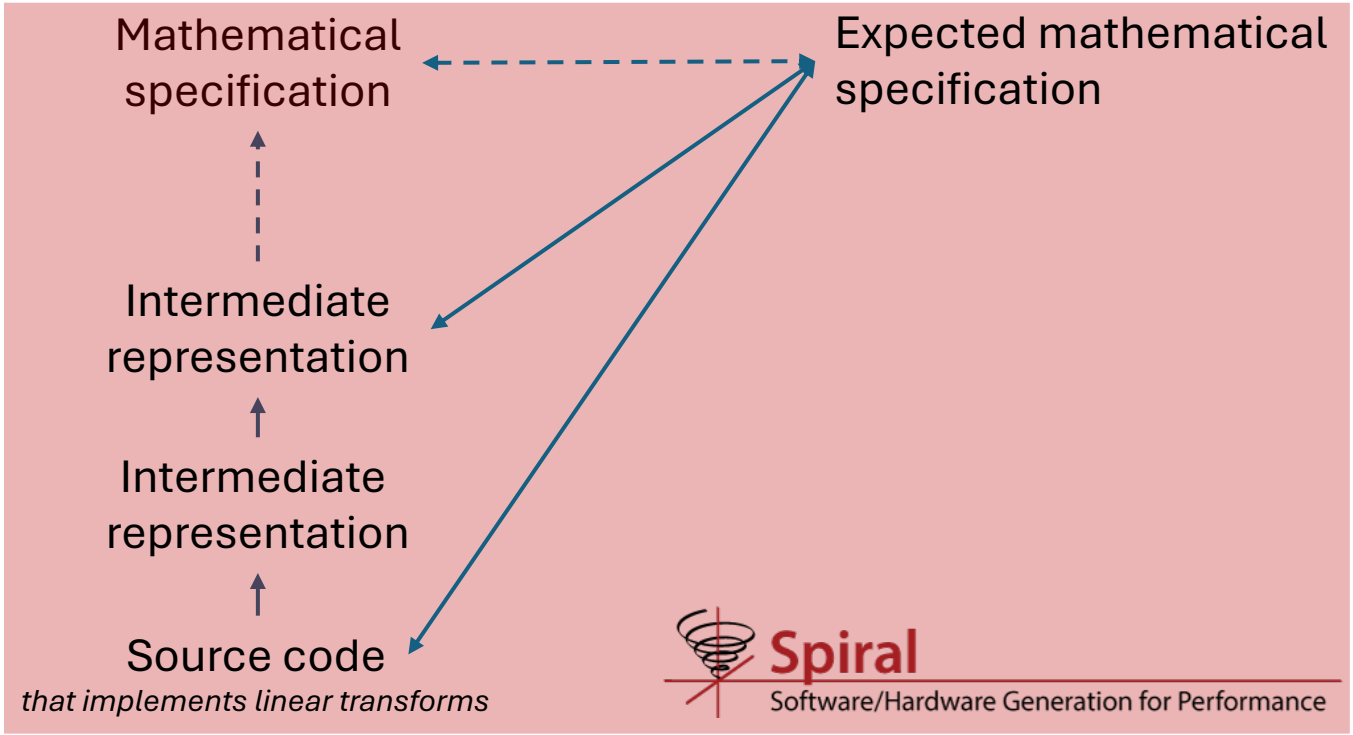
When $n = 4$,

$$T = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}$$

We can derive this matrix via symbolic execution!

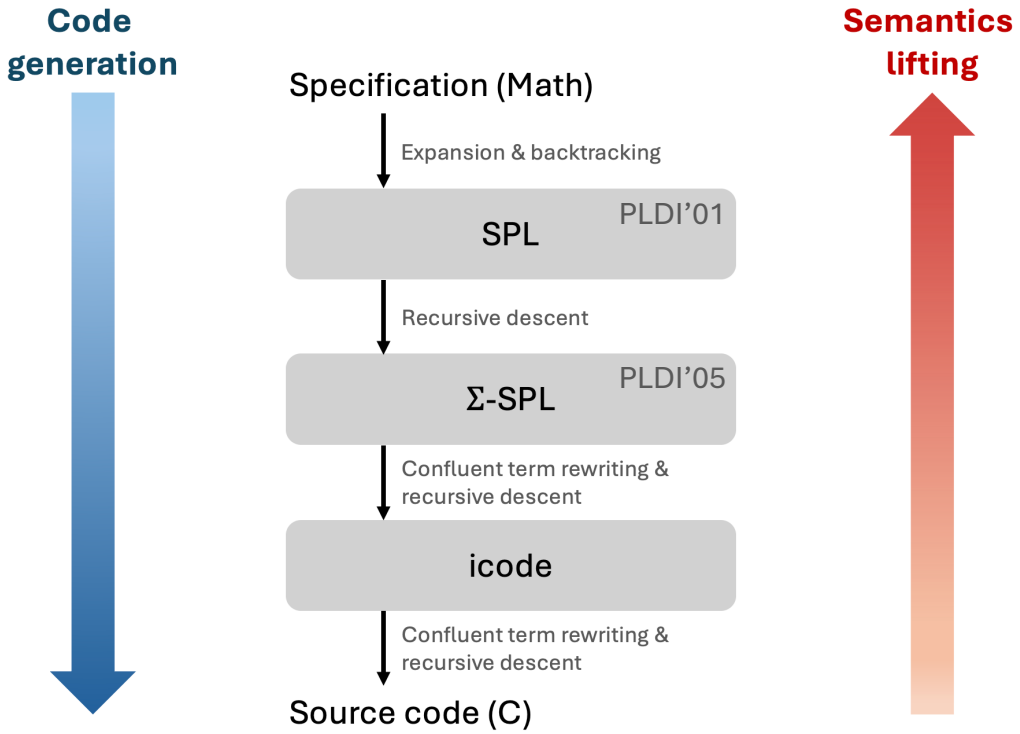
What about larger sizes?

Proposed solution: stepwise semantics lifting



Semantics lifting via Spiral

Software/Hardware Generation for Performance



Lifting GPT-generated C FFT

LLM-generated C code

```c
#include <math.h>
#include <stdlib.h>
#define M_PI 3.14159265358979323846
void fft_recursive(double* data, int n) {
    if (n <= 1) return;
    // Allocate temporary storage for half-size FFTs.
    double* even = (double*)malloc(n * sizeof(double));
    double* odd = (double*)malloc(n * sizeof(double));
    for (int i = 0; i < n / 2; ++i) {
        even[2 * i] = data[4 * i];
        even[2 * i + 1] = data[4 * i + 1];
        odd[2 * i] = data[4 * i + 2];
        odd[2 * i + 1] = data[4 * i + 3];
    }
    // Recursively compute FFTs.
    fft_recursive(even, n / 2);
    fft_recursive(odd, n / 2);
    for (int i = 0; i < n / 2; ++i) {
        double theta = -2.0 * M_PI * i / n;
        double wr = cos(theta);
        double wi = sin(theta);
        // Twiddle factor multiplication.
        double real = odd[2 * i] * wr - odd[2 * i + 1] * wi;
        double imag = odd[2 * i] * wi + odd[2 * i + 1] * wr;
        data[2 * i] = even[2 * i] + real;
        data[2 * i + 1] = even[2 * i + 1] + imag;
        data[2 * (i + n / 2)] = even[2 * i] - real;
        data[2 * (i + n / 2) + 1] = even[2 * i + 1] - imag;
    }
    // Cleanup.
    free(even);
    free(odd);
}
```

Symbolic expression

$$M_n = \overline{(\mathrm{DFT}_2 \otimes \mathrm{I}_{n/2})\, \mathrm{T}^n_{n/2}}\, (\mathrm{I}_2 \otimes M_{n/2})\, \overline{\mathrm{L}^n_2}$$

Symbolic execution on small n & Induction

High-level semantics

$$\overline{\mathrm{DFT}_n} = \overline{(\mathrm{DFT}_2 \otimes \mathrm{I}_{n/2})\, \mathrm{T}^n_{n/2}}\, (\mathrm{I}_2 \otimes \mathrm{DFT}_{n/2})\, \overline{\mathrm{L}^n_2}$$

Equivalence checking ✅

FFT specification in SPIRAL

$$\mathrm{DFT}_n = (\mathrm{DFT}_m \otimes \mathrm{I}_k)\, \mathrm{T}^n_k (\mathrm{I}_m \otimes \mathrm{DFT}_k)\, \mathrm{L}^n_m, \quad n = mk$$

Lifting GPT-generated C FFT

LLM-generated C code

```c
#include <math.h>
#include <stdlib.h>
#define M_PI 3.14159265358979323846
void fft_recursive(double* data, int n) {
    if (n <= 1) return;
    // Allocate temporary storage for half-size FFTs.
    double* even = (double*)malloc(n * sizeof(double));
    double* odd = (double*)malloc(n * sizeof(double));
    for (int i = 0; i < n / 2; ++i) {
        even[2 * i] = data[4 * i];
        even[2 * i + 1] = data[4 * i + 1];
        odd[2 * i] = data[4 * i + 2];
        odd[2 * i + 1] = data[4 * i + 3];
    }
    // Recursively compute FFTs.
    fft_recursive(even, n / 2);
    fft_recursive(odd, n / 2);
    // Combine results.
    for (int i = 0; i < n / 2; ++i) {
        double theta = -2.0 * M_PI * i / n;
        double wr = cos(theta);
        double wi = sin(theta);
        // Twiddle factor multiplication.
        double real = odd[2 * i] * wr - odd[2 * i + 1] * wi;
        double imag = odd[2 * i] * wi + odd[2 * i + 1] * wr;
        data[2 * i] = even[2 * i] + real;
        data[2 * i + 1] = even[2 * i + 1] + imag;
        data[2 * (i + n / 2)] = even[2 * i] - real;
        data[2 * (i + n / 2) + 1] = even[2 * i + 1] - imag;
    }
    // Cleanup.
    free(even);
    free(odd);
}
```

Symbolic expression

$$M_n = \overline{(\mathrm{DFT}_2 \otimes \mathrm{I}_{n/2})\, \mathrm{T}^n_{n/2} (\mathrm{I}_2 \otimes M_{n/2}) \overline{\mathrm{L}^n_2}}$$

Symbolic execution on small n & Induction

High-level semantics

$$\overline{\mathrm{DFT}_n} = \overline{(\mathrm{DFT}_2 \otimes \mathrm{I}_{n/2})\, \mathrm{T}^n_{n/2} (\mathrm{I}_2 \otimes \mathrm{DFT}_{n/2}) \mathrm{L}^n_2}$$

Equivalence checking ✅

FFT specification in SPIRAL

$$\mathrm{DFT}_n = (\mathrm{DFT}_m \otimes \mathrm{I}_k)\, \mathrm{T}^n_k (\mathrm{I}_m \otimes \mathrm{DFT}_k)\, \mathrm{L}^n_m, \quad n = mk$$

Safeguarding LLMs: Neural-symbolic AI in code generation

naifengz@cmu.edu