Gen-AI in a Bottle

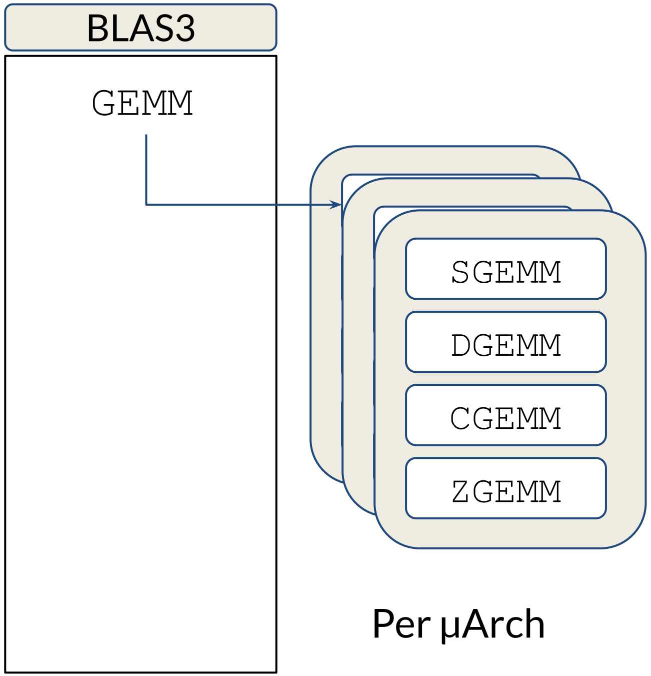
Experiments with LLMs to generate HPC kernels

Upasana Sridhar, Elliott Binder, Tze Meng Low
Carnegie Mellon University

Electrical & Computer ENGINEERING

Writing Libraries is a Recurring Task

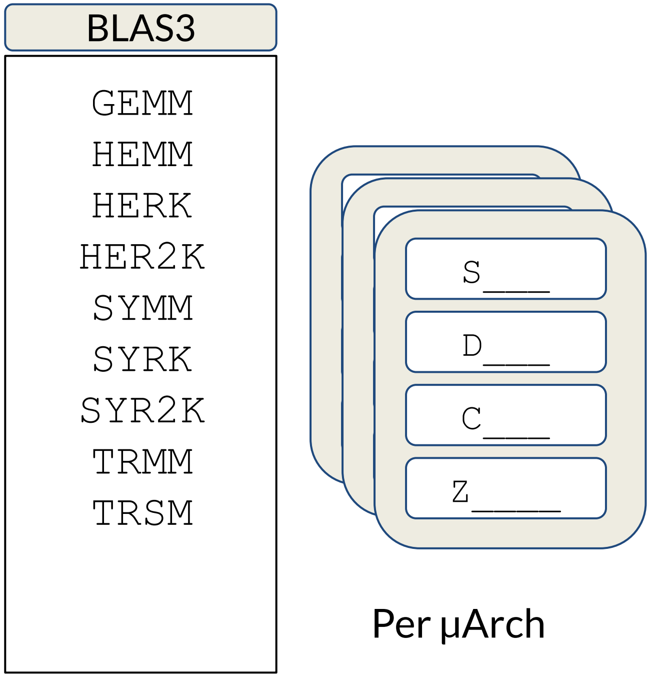
- Library implementations must be rewritten for different:
  - Datatypes - float, double, int8, etc
  - ISAs - x86 AVX, AVX2, ARM NEON, etc
  - Memory hierarchies: cach size, associativity, levels



Electrical & Computer ENGINEERING

Writing Libraries is a Recurring Task

- Library implementations must be rewritten for different:
  - Datatype precisions - float, double, int8, etc
  - ISAs - x86 AVX, AVX2, ARM NEON, etc
  - Memory hierarchies: cach size, associativity, levels
- Repeated for each API in the library



How can we make library-writing more productive?

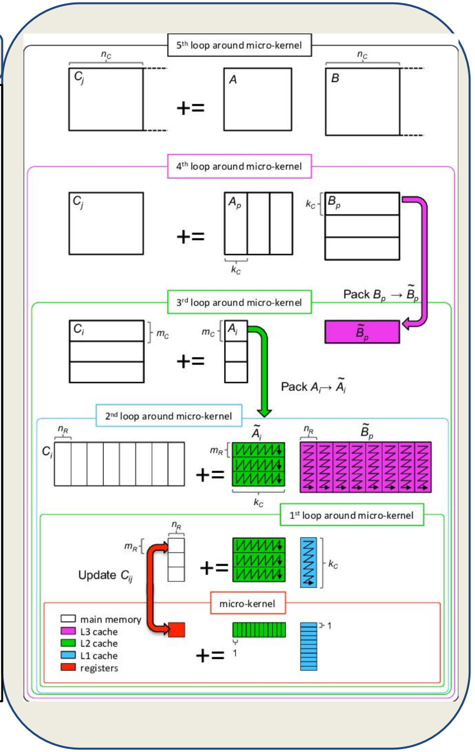
Electrical & Computer ENGINEERING

Specialization can be Simplified

- Portable library frameworks such as BLIS[1] reduce specialization effort
- Construct an abstraction that is shared across:
  - API calls,
  - µArchs, and
  - datatypes

BLAS3

GEMM
HEMM
HERK
HER2K
SYMM
SYRK
SYR2K
TRMM
TRSM



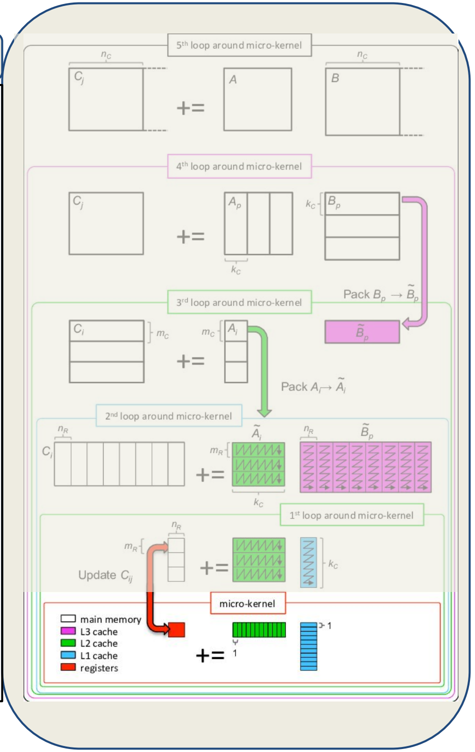
[1] Field G. Van Zee, Tyler M. Smith, Bryan Marker, Tze Meng Low, Robert A. Van De Geijn, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John A. Gunnels, and Lee Killough. 2016. The BLIS Framework: Experiments in Portability. ACM Trans. Math. Softw. 42, 2, Article 12 (June 2016), 19 pages. https://doi.org/10.1145/2755561

Specialization can be Simplified

- Portable library frameworks such as BLIS reduce specialization effort
- Construct an abstraction that is shared across:
  - API calls,
  - µArchs, and
  - datatypes

BLAS3

GEMM
HEMM
HERK
HER2K
SYMM
SYRK
SYR2K
TRMM
TRSM



ISA specialization is still necessary

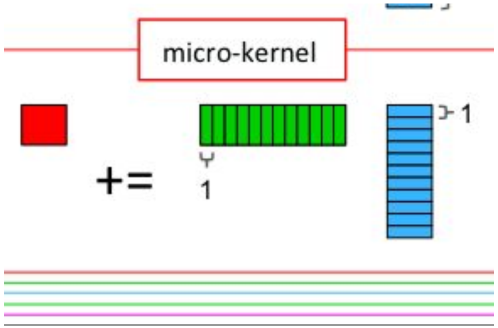
Can we use LLMs to specialize libraries?

Using LLMs for HPC Code-Generation

1. LLMs for Code-Generation: Tips and Tricks
2. Designing High-Performance GEMM Microkernels
3. Encoding Microkernel Design Principles with Prompt Engineering

Microkernels present a solvable problem

- Restricted scope has benefits:
  - Fixed problem size
  - Microkernel code does not have to handle many edge cases
  - Commonly used example
- Complexity still scales with datatype precision and ISA
  - LLM needs to retrieve the correct "form" for specialized instruction



Goal: Generate specialized microkernels using LLMs

Performance Expertise is Latent

Prompt:

Write a high-performance C-function to implement double-precision matrix multiplication with
A: 6x256, B:256x8 and C: 6x8
Function signature:
`dgemm(double const * A, const double * B, double * C)`

```
void dgemm(const double *A, const double *B,
           double *C) {
 int i, j, k;
 for (i = 0; i < 6; i++) {
  for (j = 0; j < 8; j++) {
   C[i * 8 + j] = 0.0;
   for (k = 0; k < 256; k++) {
    C[i * 8 + j] += A[i * 256 + k] * B[k * 8 + j];
   }
  }
 }
}
```

Does not automatically use any ISA-specific code

Providing ISA is helpful but insufficient

Prompt:

Write a high-performance C-function to implement double-precision matrix multiplication with
A: 6x256, B:256x8 and C: 6x8
Function signature:
`dgemm(double const * A, const double * B, double * C)`

Context
Architecture: AVX2 vector instructions
Data type: double

```
void dgemm(double const * A, const double * B,
double * C) {
int i, j, k;
__m256d a0, b0, c0, … ,a6, b6, c6;
__m256d t0, …, t14, t15;
// Loop over the rows of C
for (i = 0; i < 6; i += 8) {
 for (j = 0; j < 8; j += 16) {
  // Initialize the accumulators
  a0 = _mm256_setzero_pd();
  b0 = _mm256_loadu_pd(B + j);
  c0 = _mm256_setzero_pd();
}
```

Providing ISA is helpful but insufficient

Prompt:

Write a high-performance C-function to implement double-precision matrix multiplication with
A: 6x256, B:256x8 and C: 6x8
Function signature:
`dgemm(double const * A, const double * B, double * C)`

Context
Architecture: AVX2 vector instructions
Data type: double

```
void dgemm(double const * A, const double * B,
double * C) {
int i, j, k;
__m256d a0, b0, c0, … ,a6, b6, c6;
__m256d t0, …, t14, t15;
// Loop over the rows of C
for (i = 0; i < 6; i += 8) {
 for (j = 0; j < 8; j += 16) {
  // Initialize the accumulators
  a0 = _mm256_setzero_pd();
  b0 = _mm256_loadu_pd(B + j);
  c0 = _mm256_setzero_pd();
}
```

Using specialized datatypes

Providing ISA is helpful but insufficient

Prompt:

Write a high-performance C-function to implement double-precision matrix multiplication with
A: 6x256, B:256x8 and C: 6x8
Function signature:
`dgemm(double const * A, const double * B, double * C)`

Context
Architecture: AVX2 vector instructions
Data type: double

```
void dgemm(double const * A, const double * B,
double * C) {
int i, j, k;
__m256d a0, b0, c0, … ,a6, b6, c6;
__m256d t0, …, t14, t15;
// Loop over the rows of C
for (i = 0; i < 6; i += 8) {
 for (j = 0; j < 8; j += 16) {
  // Initialize the accumulators
  a0 = _mm256_setzero_pd();
  b0 = _mm256_loadu_pd(B + j);
  c0 = _mm256_setzero_pd();
}
```

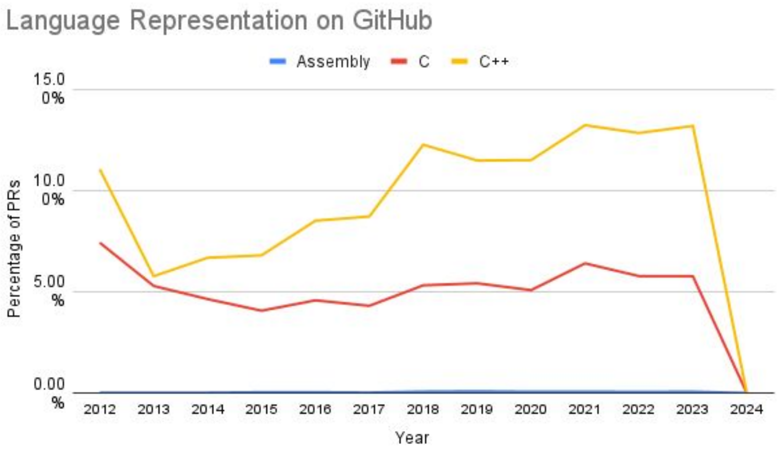
Too many vector registers

Incorrect indexing

Model needs more guidance for how to write code

A Primer on Priming LLMs

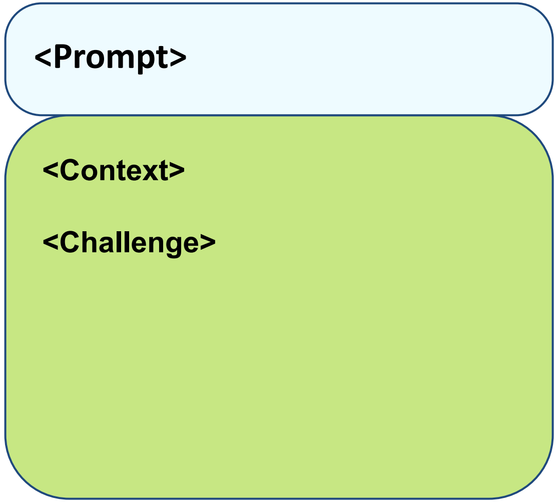
- HPC-code is a tiny fraction of training data
  - Hard to retrieve without context



Electrical & Computer ENGINEERING

Scaffolded Learning

- Prompt can be used to “prime” LLM to use the right subset of its knowledge
- Context could contain
  - text-based information
  - “Worked examples”



What should be encoded in the microkernel prompt?

Designing GEMM Microkernels

```
#define FLOAT_CONV_TILE_C(step, a, b, W_ob, C_ob)          \
    float const * a_ptr = a;                                \
    a_reg = _mm256_broadcast_ss(a_ptr + 0 * step);          \
    b0 = _mm256_load_ps(b);                                 \
    b1 = _mm256_load_ps(b + FLOAT_SIMD);                    \
    c12 = _mm256_broadcast_ss(a_ptr + 1 * step);            \
    c0 = _mm256_fmadd_ps(a_reg, b0, c0);                    \
    c1 = _mm256_fmadd_ps(a_reg, b1, c1);                    \
    a_reg = _mm256_broadcast_ss(a_ptr + 2 * step);          \
    a_ptr += 3*step;                                        \
    c2 = _mm256_fmadd_ps(c12, b0, c2);                      \
    c3 = _mm256_fmadd_ps(c12, b1, c3);                      \
    c12 = _mm256_broadcast_ss(a_ptr + 0 * step);            \
                                                            \
    c4 = _mm256_fmadd_ps(a_reg, b0, c4);                    \
    c5 = _mm256_fmadd_ps(a_reg, b1, c5);                    \
    a_reg = _mm256_broadcast_ss(a_ptr + 1 * step);          \
```

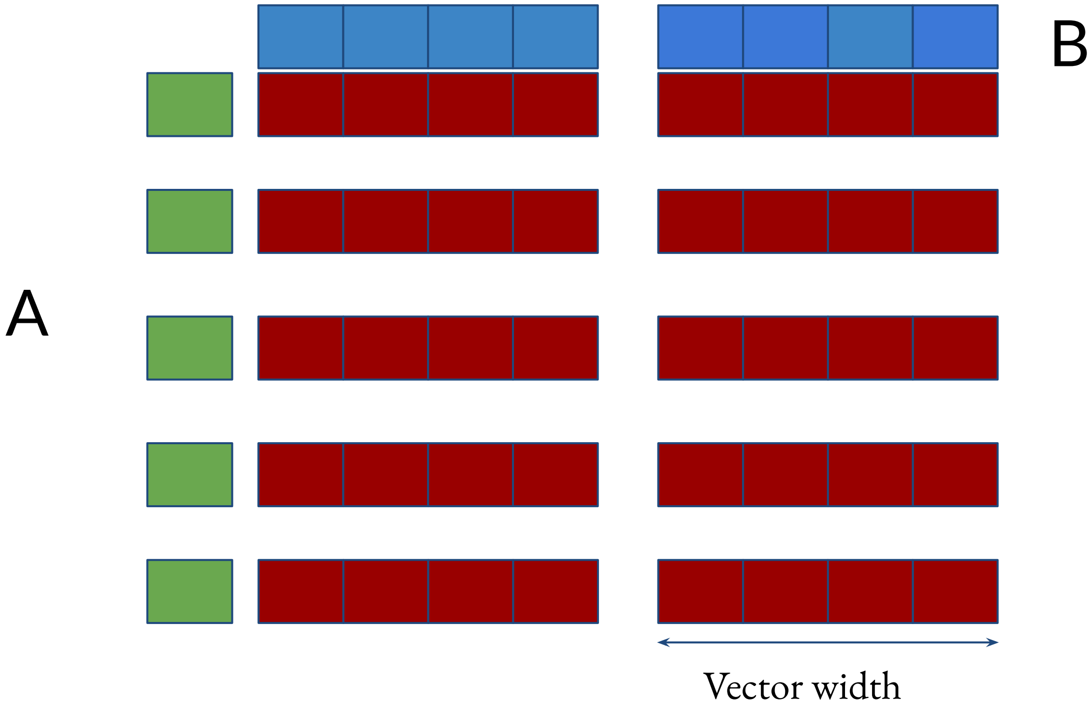
x86 AVX2 float Kernel

```
a_0 = vld1q_dup_f32(a + 0 * step + 0 * FLOAT_SIMD);\
b_0 = vld1q_f32(bb + 0*C_ob + (0 * 4 + 0)*FLOAT_SIMD);\
__asm__ volatile ("fmla %0.4s, %1.4s, %2.s[0]" : "+w"(c_0_0) : "w"(b_
b_1 = vld1q_f32(bb + 0*C_ob + (0 * 4 + 1)*FLOAT_SIMD);\
__asm__ volatile ("fmla %0.4s, %1.4s, %2.s[0]" : "+w"(c_0_1) : "w"(b_
b_2 = vld1q_f32(bb + 0*C_ob + (0 * 4 + 2)*FLOAT_SIMD);\
__asm__ volatile ("fmla %0.4s, %1.4s, %2.s[0]" : "+w"(c_0_2) : "w"(b_
b_3 = vld1q_f32(bb + 0*C_ob + (0 * 4 + 3)*FLOAT_SIMD);\
__asm__ volatile ("fmla %0.4s, %1.4s, %2.s[0]" : "+w"(c_0_3) : "w"(b_
a_1 = vld1q_dup_f32(a + 1 * step + 0 * FLOAT_SIMD);\
__asm__ volatile ("fmla %0.4s, %1.4s, %2.s[0]" : "+w"(c_1_0) : "w"(b_
__asm__ volatile ("fmla %0.4s, %1.4s, %2.s[0]" : "+w"(c_1_1) : "w"(b_
__asm__ volatile ("fmla %0.4s, %1.4s, %2.s[0]" : "+w"(c_1_2) : "w"(b_
__asm__ volatile ("fmla %0.4s, %1.4s, %2.s[0]" : "+w"(c_1_3) : "w"(b_
a_2 = vld1q_dup_f32(a + 2 * step + 0 * FLOAT_SIMD);\
__asm__ volatile ("fmla %0.4s, %1.4s, %2.s[0]" : "+w"(c_2_0) : "w"(b_
__asm__ volatile ("fmla %0.4s, %1.4s, %2.s[0]" : "+w"(c_2_1) : "w"(b_
```

ARM NEON Float Kernel

There is a shared underlying structure

Both are a Vectorized Outer Product



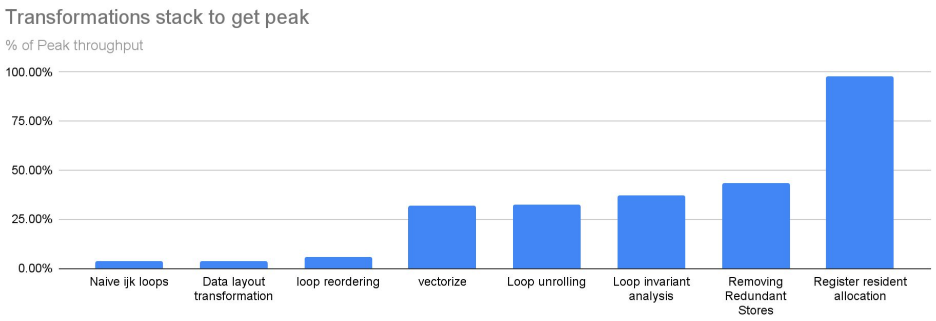
We need encode this pattern into the prompt

Electrical & Computer ENGINEERING

Steps to write a Microkernel

Transformation Class	Transformation Required
Indexing	column-major A
Loop Order	kij
Vectorization	Vectorize j-loop use bcast, vload and fma
Loop Unrolling	Unroll i and j loops
Loop Invariant Analysis	Hoist B-loads
Redundant Store Removal	Delay C stores
Register Allocation	Reuse registers for A broadcast

```
// naive loops
for (i = 0; i < 6; i++) {
 for (j = 0; j < 8; j++) {
  C[i * 8 + j] = 0.0;
  for (k = 0; k < 256; k++) {
   C[i * 8 + j] += A[i * 256 + k] * B[k * 8 + j];
  }
 }
}
```



Create prompts with these steps in the context

Electrical & Computer ENGINEERING

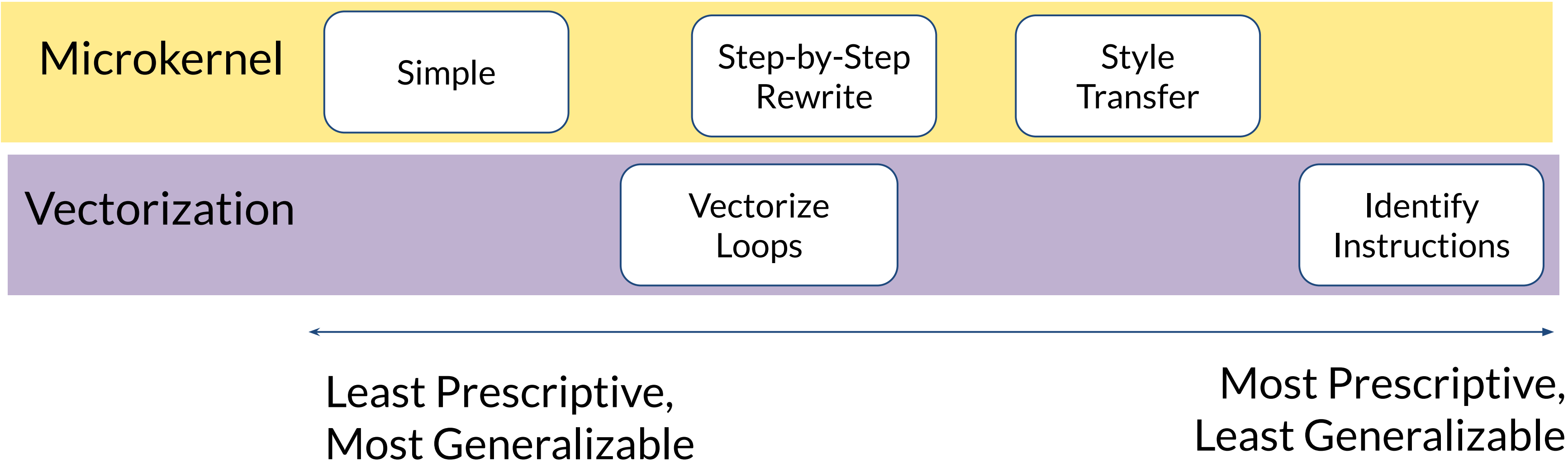
CodeLLaMA as a Kernel Generator

- Small model, runs locally on a mini-pc
- Trained specifically to generate code
- Weights are available to refine/finetune

Name	codellama/CodeLlama-7b-Instruct-hf
architecture	LlamaForCausalLM
Embedding Size	4096
Max Seq. Length	16384
model_type	llama
Heads	32
Layers	32
DataType	bfloat16
Vocabulary Size	32016

The Prompt Types

- We tried 5 different types



Focus on generating entire microkernel

Microkernel

Simple

Step-by-Step Rewrite

Style Transfer

Least Prescriptive, Most Generalizable

Most Prescriptive, Least Generalizable

Context-Encoding: Entire Microkernel

1. Explicitly encode kernel pattern as transformations from naive loops.
2. Implicitly encode the pattern with a related example

Step-by-Step:

Prompt

Kernel Writing Steps
Follow these steps in order
1. Loop Order

<Steps from Table 1>

Challenge
Use the above steps to fill out the dgemm function

Style Transfer:

Prompt

Kernel
This is an example of using AVX2 for floating-point numbers

```
void sgemm(...)
{
// example kernel
}
```

Challenge
Use the above example to fill out the dgemm function

Electrical & Computer ENGINEERING

Generating the entire microkernel is hard

Prompt Strategy	Indexing	Loop Order	Vectorize	Unroll	Loop Invari-ant	Redundant Store	Register Alloc.	Errors	%Peak
Simple Prompt (Scalar)	X	X (ikj)	X	X	X	X	n/a	None	7%
Simple Prompt (Vector)	X	X (ijk)	✓	X	✓	X	✓	**No reduction within vector register**	-
Step-by-step Transformations	✓	X (ijki)	✓	X (not j)	✓	✓	✓	Undeclared variables. **Extra i-loop. Wrong store indexing**	-
Style Transfer	✓	✓	✓	✓	✓	✓	X (20)	Undeclared variables	99%

Create prompts with these steps in the context

Electrical & Computer ENGINEERING

What Worked

- CodeLLaMA can reproduce patterns very well
  - Patterns shown in the context are applied even if they are not often seen in training data
- Breaking down each step in a complex reasoning process improved performance
  - When asked to emit a precise piece of code, Code-LLaMA was the most successful
- Style-transfer works well, but is surface level
  - General structure is copied – which is the intent in this case. But requires at least one "golden kernel" to be used as a seed

What Didn't

- When an arithmetic calculation was left implicit, CodeLLaMA often guessed incorrectly
  - e.g. the stride of a vectorized loop is reproduced exactly from the example
- Inter-connected constraints leads to confused code-generation
  - When a loop is involved in unrolling as well as vectorization, the generated code muddles the transformations

Towards Fully Automatic Kernel Generation

- Preliminary success with using off-the-shelf LLMs for writing specialized GEMM code.
  - Good reasons to do a hybrid approach with traditional code-gen and LLMs
- A good prompt shows success with a small model
  - could be run natively on the target device for the library
- Systematic steps to write microkernels is impactful
  - Serves as a metric to evaluate the performance of other models
  - Can be used in finetuning
  - Integrated into guardrails for provably correct code-generation

Thank you!

https://linktr.ee/upasanasridhar

Prompt Strategies: Only Vectorization

Single Step:

Objective
Vectorize the following code snippet

Vectorization Principles

Vectorization Examples
Original code
Vectorized code
Analysis (text)

Prompt
Vectorize the following code snippet

Table-filling

You are a vectorization expert…

Example
Objective
Identify the vector instructions to fill out the table below. Use the AVX2 vector ISA with float scalars

```
{
 "Vector":
  {"load": "_mm256_load_ps"
   <a list of vector instructions>
  }
 }
```

Prompt
Identify the vector instructions to fill out the table below. Use the AVX2 vector ISA with ***double*** scalars