Guided Equality Saturation

Supplementary Material

This supplementary material contains the Lean proofs and gudies from introduction and Section 4 of the paper, as well as the matrix multiplication programs and sketch guides underlying the program optimization case study from Section 5 of the paper.

The guides underlying the Lean case study from the introduction are in Section 1.1, and the theorem proving case study in Section 1.2.

Section 2.1 contains the C programs generated by rewriting the RISE programs. Section 2.2 contains the handwritten sketches used to guide the rewriting, and enumerated in Table 4 of the paper. Section 2.3 contains the RISE programs that are found using the sketch guides.

1 Guided Equality Saturation for Lean

We first show the example from the introduction:

1.1 Inversion is an Involution

We see the implementation of the sketch guided proof of the lemma that inversion in a group is an involution:

```
theorem inv_inv
  (assocMul: forall (a b c: G), a * (b * c) = (a * b) * c)
  (invLeft: forall (a: G), (inv a) * a = one)
  (oneMul: forall (a: G), one * a = a)
  (mulOne: forall (a: G), a * one = a)
  (invRight: forall (a: G), a * (inv a) = one)
  (x: G)
  : (inv (inv x) = x) := by calc
  inv (inv x) = inv (inv x) * one := by ges
            _ = (inv (inv x))*((inv x) * x) := by ges
            _ = x := by ges
```

1.2 Case Study

Below, we show in the proofs of the knuth-bendix rewrite rules for the theory of groups. The proofs are written using guided equality saturation. We also compare the proofs of the binomial theorem written manually versus written using guided equality saturation.

1.2.1 Knuth Bendix Completion of Group Axioms (Guided)

The complete proofs for the knuth bendix axioms for groups using guided equality saturation:

```
import PowerCalc.Examples.Groups.Basic
open G

theorem inv_inv
```

```
  (assocMul: forall (a b c: G), a * (b * c) = (a * b) * c)
  (invLeft: forall (a: G), (inv a) * a = one)
  (oneMul: forall (a: G), one * a = a)
  (mulOne: forall (a: G), a * one = a)
  (invRight: forall (a: G), a * (inv a) = one)
  (x: G)
  : (inv (inv x) = x) := by calc
  inv (inv x) = inv (inv x) * one := by ges
            _ = (inv (inv x))*((inv x) * x) := by ges
            _ = x := by ges

theorem inv_mul_cancel_left
  (assocMul: forall (a b c: G), a * (b * c) = (a * b) * c)
  (invLeft: forall (a: G), (inv a) * a = one)
  (oneMul: forall (a: G), one * a = a)
  (mulOne: forall (a: G), a * one = a)
  (invRight: forall (a: G), a * (inv a) = one)
  (x: G)
  (x y : G)
  : ((inv x) * (x * y)) = y := by ges

theorem mul_inv_cancel_left
  (assocMul: forall (a b c: G), a * (b * c) = (a * b) * c)
  (invLeft: forall (a: G), (inv a) * a = one)
  (oneMul: forall (a: G), one * a = a)
  (mulOne: forall (a: G), a * one = a)
  (invRight: forall (a: G), a * (inv a) = one)
  (x y : G)
  : x * ((inv x) * y) = y := by ges

theorem inv_mul
  (assocMul: forall (a b c: G), a * (b * c) = (a * b) * c)
  (invLeft: forall (a: G), (inv a) * a = one)
  (oneMul: forall (a: G), one * a = a)
  (mulOne: forall (a: G), a * one = a)
  (invRight: forall (a: G), a * (inv a) = one)
  (x y : G)
  : (inv (x * y)) = (inv y) * (inv x) := by calc
  inv (x * y) = (inv (x * y)) * (x * (inv x)) := by ges
            _ = (inv (x * y)) * (x * (y * inv y) * (inv x)) := by ges
            _ = (inv (x * y)) * (x * y) * (inv y * inv x) := by ges
            _ = one * (inv y * inv x) := by ges
            _ = inv y * inv x := by ges
```

1.2.2 Binomials in Characteristic 2 (Guided)

Here, we provide the proof of the guided proof of the "freshman's dream" theorem $(x+y)^2 = x^2 + y^2$, in characteristic two.

```
open R in
theorem freshmans_dream
  (comm_add : forall a b : R,  a + b = b + a)
  (comm_mul:  forall a b : R, a * b = b * a)
  (add_assoc: forall a b c : R, a + (b + c) = (a + b) + c)
  (mul_assoc: forall a b c : R, a * (b * c) = (a * b) * c)
  (sub_canon: forall a b : R, (a - b)  = a + (R.neg b))
  (neg_add : forall a : R , a + (R.neg a) = zero)
  (div_canon : forall a b : R, (a / b) = a * (R.inv b))
  (zero_add: forall a : R, a + zero = a )
  (zero_mul: forall a : R, a * zero = zero)
  (one_mul: forall a : R,  a * one = a)
  (distribute: forall a b c : R, a * (b + c)  = (a * b) + (a * c))
  (pow_one: forall a : R, a^1 = a )
```

```
  (pow_two: forall a : R, a*a = a^2)
  (char_two : forall a : R, a + a = zero )
  (x y : R)
  : (x + y)^2 = (x^2) + y^(2)   :=
  by calc (x + y)^2 = (x + y) * (x + y) := by ges
                  _ = (x * (x + y) + y * (x + y)) := by ges
                  _ = (x^2 + x * y + y * x + y^2) := by ges
                  _ = (x^2) + (y^2) :=  by ges
```

1.2.3 Binomials in Characteristic 2 (Manual)

The manual proof of the binomial theorem for power 2 in characteristic 2, rewrite-by-rewrite is given below:

```
import PowerCalc.Examples.PolynomialsPerformance.Basic
open R in
theorem freshmans_dream
  (comm_add : forall a b : R,  a + b = b + a)
  (comm_mul:  forall a b : R, a * b = b * a)
  (add_assoc: forall a b c : R, a + (b + c) = (a + b) + c)
  (mul_assoc: forall a b c : R, a * (b * c) = (a * b) * c)
  (sub_canon: forall a b : R, (a - b)  = a + (R.neg b))
  (neg_add : forall a : R , a + (R.neg a) = zero)
  (div_canon : forall a b : R, (a / b) = a * (R.inv b))
  (zero_add: forall a : R, a + zero = a )
  (zero_mul: forall a : R, a * zero = zero)
  (one_mul: forall a : R,  a * one = a)
  (distribute: forall a b c : R, a * (b + c)  = (a * b) + (a * c))
  (pow_one: forall a : R, a^1 = a )
  (pow_2: forall a : R, a*a = a^2)
  (char_2 : forall a : R, a + a = zero )
  (x y : R)
  : (x + y)^2 = (x^2) + y^(2)   := by
  calc (x + y)^2 = (x + y) * (x + y) := by rw [← pow_2]
               _ = ((x + y) * x) + ((x + y) * y) := by rw [distribute]
               _ = (x * (x + y)) + ((x + y) * y) := by rw [comm_mul]
               _ = ((x * x) + (x * y)) + ((x + y) * y) := by rw [distribute]
               _ = ((x^2) + (x * y)) + ((x + y) * y) := by rw [pow_2]
               _ = ((x^2) + (x * y)) + (y * (x + y)) := by rw [comm_mul y _]
               _ = ((x^2) + (x * y)) + ((y * x) + (y * y)) := by rw [distribute]
               _ = ((x^2) + (x * y)) + ((y * x) + (y^2)) := by rw [pow_2 y]
               _ = ((x^2) + (x * y)) + ((x * y) + (y^2)) := by rw [comm_mul y x]
               _ = (x^2) + ((x * y) + ((x * y) + (y^2))) := by rw [add_assoc (x^2)]
               _ = (x^2) + (((x * y) + (x * y)) + (y^2)) := by rw [add_assoc (x * y)]
               _ = (x^2) + (zero + (y^2)) := by rw [char_2]
               _ = (x^2) + ((y^2) + zero) := by rw [comm_add zero]
               _ = (x^2) + (y^2) := by rw [zero_add]
```

2 Matrix Multiplication Programs and Guides

2.1 Generated C Code

This subsection contains the C programs generated from the RISE programs rewritten using the 7 reference ELEVATE strategies. To keep the code readable and compact, we may apply cosmetic changes to the code such as renaming variables and removing unnecessary parentheses, brackets, or space. The same programs are generated using guided equality saturation, modulo variable names.

```
void baseline(float* output, float* x0, float* x1) {
  for (int i0 = 0;(i0 < 1024);i0 = (1 + i0)) {
    for (int i1 = 0;(i1 < 1024);i1 = (1 + i1)) {
      float t1 = 0.0f;
      for (int i2 = 0;(i2 < 1024);i2 = (1 + i2)) {
        t1 += x0[(i2 + (1024 * i0))] *
              x1[(i1 + (1024 * i2))];
      }

      output[(i1 + (1024 * i0))] = t1;
    }
  }
}
```

Listing 1: C code generated for the *baseline* matrix multiplication.

```
void blocking(float* output, float* x0, float* x1) {
  for (int i0 = 0;(i0 < 32);i0 = (1 + i0)) {
    for (int i1 = 0;(i1 < 32);i1 = (1 + i1)) {
      float t1[1024];
      for (int i2 = 0;(i2 < 32);i2 = (1 + i2)) {
        for (int i3 = 0;(i3 < 32);i3 = (1 + i3)) {
          t1[(i3 + (32 * i2))] = 0.0f;
        }
      }

      for (int i4 = 0;(i4 < 256);i4 = (1 + i4)) {
        float t2[1024];
        for (int i5 = 0;(i5 < 32);i5 = (1 + i5)) {
          for (int i6 = 0;(i6 < 32);i6 = (1 + i6)) {
            t2[(i6 + (32 * i5))] = t1[(i6 + (32 * i5))];
          }
        }

        for (int i7 = 0;(i7 < 4);i7 = (1 + i7)) {
          for (int i8 = 0;(i8 < 32);i8 = (1 + i8)) {
            for (int i9 = 0;(i9 < 32);i9 = (1 + i9)) {
              t2[i9 + 32*i8] += x0[i7 + 4*i4 + 1024*i8 + 32768*i0] *
                                x1[i9 + 32*i1 + 1024*i7 + 4096*i4];
            }
          }
        }

        for (int i10 = 0;(i10 < 32);i10 = (1 + i10)) {
          for (int i11 = 0;(i11 < 32);i11 = (1 + i11)) {
            t1[(i11 + (32 * i10))] = t2[(i11 + (32 * i10))];
          }
        }
      }

      for (int i12 = 0;(i12 < 32);i12 = (1 + i12)) {
        for (int i13 = 0;(i13 < 32);i13 = (1 + i13)) {
          output[((((i13 + (32 * i1)) + (1024 * i12)) + (32768 * i0))] = t1[(i13 + (32 * i12))];
        }
      }
    }
  }
}
```

Listing 2: C code generated for the *blocking* matrix multiplication.

```
void vectorization(float* output, float* x0, float* x1) {
  for (int i0 = 0;(i0 < 32);i0 = (1 + i0)) {
    for (int i1 = 0;(i1 < 32);i1 = (1 + i1)) {
      float t1[1024];
      for (int i2 = 0;(i2 < 32);i2 = (1 + i2)) {
        for (int i3 = 0;(i3 < 32);i3 = (1 + i3)) {
          t1[(i3 + (32 * i2))] = 0.0f;
        }
      }

      for (int i4 = 0;(i4 < 256);i4 = (1 + i4)) {
        float t2[1024];
        for (int i5 = 0;(i5 < 32);i5 = (1 + i5)) {
          for (int i6 = 0;(i6 < 32);i6 = (1 + i6)) {
            t2[(i6 + (32 * i5))] = t1[(i6 + (32 * i5))];
          }
        }

        for (int i7 = 0;(i7 < 4);i7 = (1 + i7)) {
          for (int i8 = 0;(i8 < 32);i8 = (1 + i8)) {
            #pragma omp simd
            for (int i9 = 0;(i9 < 32);i9 = (1 + i9)) {
              t2[i9 + 32*i8] += x0[i7 + 4*i4 + 1024*i8 + 32768*i0] *
                                x1[i9 + 32*i1 + 1024*i7 + 4096*i4];
            }
          }
        }

        for (int i10 = 0;(i10 < 32);i10 = (1 + i10)) {
          for (int i11 = 0;(i11 < 32);i11 = (1 + i11)) {
            t1[(i11 + (32 * i10))] = t2[(i11 + (32 * i10))];
          }
        }
      }

      for (int i12 = 0;(i12 < 32);i12 = (1 + i12)) {
        for (int i13 = 0;(i13 < 32);i13 = (1 + i13)) {
          output[((((i13 + (32 * i1)) + (1024 * i12)) + (32768 * i0))] = t1[(i13 + (32 * i12))];
        }
      }
    }
  }
}
```

Listing 3: C code generated for the *vectorization* matrix multiplication.

```
void loop_permutation(float* output, float* x0, float* x1) {
  for (int i0 = 0;(i0 < 32);i0 = (1 + i0)) {
    for (int i1 = 0;(i1 < 32);i1 = (1 + i1)) {
      float t1[1024];
      for (int i2 = 0;(i2 < 32);i2 = (1 + i2)) {
        for (int i3 = 0;(i3 < 32);i3 = (1 + i3)) {
          t1[(i3 + (32 * i2))] = 0.0f;
        }
      }

      for (int i4 = 0;(i4 < 256);i4 = (1 + i4)) {
        for (int i5 = 0;(i5 < 32);i5 = (1 + i5)) {
          float t2[32];
          for (int i6 = 0;(i6 < 32);i6 = (1 + i6)) {
            t2[i6] = t1[(i6 + (32 * i5))];
          }

          for (int i7 = 0;(i7 < 4);i7 = (1 + i7)) {
            #pragma omp simd
            for (int i8 = 0;(i8 < 32);i8 = (1 + i8)) {
              t2[i8] += x0[i7 + 4*i4 + 1024*i5 + 32768*i0] *
                        x1[i8 + 32*i1 + 1024*i7 + 4096*i4];
            }
          }

          for (int i9 = 0;(i9 < 32);i9 = (1 + i9)) {
            t1[(i9 + (32 * i5))] = t2[i9];
          }
        }
      }

      for (int i10 = 0;(i10 < 32);i10 = (1 + i10)) {
        for (int i11 = 0;(i11 < 32);i11 = (1 + i11)) {
          output[((((i11 + (32 * i1)) + (1024 * i10)) + (32768 * i0))] = t1[(i11 + (32 * i10))];
        }
      }
    }
  }
}
```

Listing 4: C code generated for the *loop-perm* matrix multiplication.

```
void array_packing(float* output, float* x0, float* x1) {
  float t1[1048576];
  #pragma omp parallel for
  for (int i0 = 0;(i0 < 32);i0 = (1 + i0)) {
    for (int i1 = 0;(i1 < 1024);i1 = (1 + i1)) {
      #pragma omp simd
      for (int i2 = 0;(i2 < 32);i2 = (1 + i2)) {
        t1[((i2 + (32 * i1)) + (32768 * i0))] = x1[((i2 + (32 * i0)) + (1024 * i1))];
      }
    }
  }

  for (int i3 = 0;(i3 < 32);i3 = (1 + i3)) {
    for (int i4 = 0;(i4 < 32);i4 = (1 + i4)) {
      float t2[1024];
      for (int i5 = 0;(i5 < 32);i5 = (1 + i5)) {
        for (int i6 = 0;(i6 < 32);i6 = (1 + i6)) {
          t2[(i6 + (32 * i5))] = 0.0f;
        }
      }

      for (int i7 = 0;(i7 < 256);i7 = (1 + i7)) {
        for (int i8 = 0;(i8 < 32);i8 = (1 + i8)) {
          float t3[32];
          for (int i9 = 0;(i9 < 32);i9 = (1 + i9)) {
            t3[i9] = t2[(i9 + (32 * i8))];
          }

          for (int i10 = 0;(i10 < 4);i10 = (1 + i10)) {
            #pragma omp simd
            for (int i11 = 0;(i11 < 32);i11 = (1 + i11)) {
              t3[i11] += x0[i10 + 4*i7 + 1024*i8 + 32768*i3] *
                         t1[i11 + 32*i10 + 128*i7 + 32768*i4];
            }
          }

          for (int i12 = 0;(i12 < 32);i12 = (1 + i12)) {
            t2[(i12 + (32 * i8))] = t3[i12];
          }
        }
      }

      for (int i13 = 0;(i13 < 32);i13 = (1 + i13)) {
        for (int i14 = 0;(i14 < 32);i14 = (1 + i14)) {
          output[((((i14 + (32 * i4)) + (1024 * i13)) + (32768 * i3)))] = t2[(i14 + (32 * i13))];
        }
      }
    }
  }
}
```

Listing 5: C code generated for the *array-packing* matrix multiplication.

```
void cache_blocks(float* output, float* x0, float* x1) {
  float t1[1048576];
  #pragma omp parallel for
  for (int i0 = 0;(i0 < 32);i0 = (1 + i0)) {
    for (int i1 = 0;(i1 < 1024);i1 = (1 + i1)) {
      #pragma omp simd
      for (int i2 = 0;(i2 < 32);i2 = (1 + i2)) {
        t1[((i2 + (32 * i1)) + (32768 * i0))] = x1[((i2 + (32 * i0)) + (1024 * i1))];
      }
    }
  }
  for (int i3 = 0;(i3 < 32);i3 = (1 + i3)) {
    for (int i4 = 0;(i4 < 32);i4 = (1 + i4)) {
      float t2[1024];
      for (int i5 = 0;(i5 < 32);i5 = (1 + i5)) {
        for (int i6 = 0;(i6 < 32);i6 = (1 + i6)) {
          t2[(i6 + (32 * i5))] = 0.0f;
        }
      }
      for (int i7 = 0;(i7 < 256);i7 = (1 + i7)) {
        for (int i8 = 0;(i8 < 32);i8 = (1 + i8)) {
          float t3[32];
          for (int i9 = 0;(i9 < 32);i9 = (1 + i9)) {
            t3[i9] = t2[(i9 + (32 * i8))];
          }

          #pragma omp simd
          for (int i11 = 0;(i11 < 32);i11 = (1 + i11)) {
            t3[i11] += x0[4*i7 + 1024*i8 + 32768*i3] * t1[i11 + 128*i7 + 32768*i4];
          }
          #pragma omp simd
          for (int i11 = 0;(i11 < 32);i11 = (1 + i11)) {
            t3[i11] += x0[1 + 4*i7 + 1024*i8 + 32768*i3] * t1[32 + i11 + 128*i7 + 32768*i4];
          }
          #pragma omp simd
          for (int i11 = 0;(i11 < 32);i11 = (1 + i11)) {
            t3[i11] += x0[2 + 4*i7 + 1024*i8 + 32768*i3] * t1[64 + i11 + 128*i7 + 32768*i4];
          }
          #pragma omp simd
          for (int i11 = 0;(i11 < 32);i11 = (1 + i11)) {
            t3[i11] += x0[3 + 4*i7 + 1024*i8 + 32768*i3] * t1[96 + i11 + 128*i7 + 32768*i4];
          }

          for (int i12 = 0;(i12 < 32);i12 = (1 + i12)) {
            t2[(i12 + (32 * i8))] = t3[i12];
          }
        }
      }
      for (int i13 = 0;(i13 < 32);i13 = (1 + i13)) {
        for (int i14 = 0;(i14 < 32);i14 = (1 + i14)) {
          output[((((i14 + (32 * i4)) + (1024 * i13)) + (32768 * i3)))] = t2[(i14 + (32 * i13))];
        }
      }
} } }
```

Listing 6: C code generated for the *cache-blocks* matrix multiplication.

```
void parallel(float* output, float* x0, float* x1) {
  float t1[1048576];
  #pragma omp parallel for
  for (int i0 = 0;(i0 < 32);i0 = (1 + i0)) {
    for (int i1 = 0;(i1 < 1024);i1 = (1 + i1)) {
      #pragma omp simd
      for (int i3 = 0;(i3 < 32);i3 = (1 + i3)) {
        t1[((i3 + (32 * i1)) + (32768 * i0))] = x1[((i3 + (32 * i0)) + (1024 * i1))];
      }
    }
  }
  #pragma omp parallel for
  for (int i4 = 0;(i4 < 32);i4 = (1 + i4)) {
    for (int i5 = 0;(i5 < 32);i5 = (1 + i5)) {
      float t2[1024];
      for (int i6 = 0;(i6 < 32);i6 = (1 + i6)) {
        for (int i_50146 = 0;(i_50146 < 32);i_50146 = (1 + i_50146)) {
          t2[(i_50146 + (32 * i6))] = 0.0f;
        }
      }
      for (int i7 = 0;(i7 < 256);i7 = (1 + i7)) {
        for (int i8 = 0;(i8 < 32);i8 = (1 + i8)) {
          float t3[32];
          for (int i9 = 0;(i9 < 32);i9 = (1 + i9)) {
            t3[i9] = t2[(i9 + (32 * i8))];
          }

          #pragma omp simd
          for (int i11 = 0;(i11 < 32);i11 = (1 + i11)) {
            t3[i11] += x0[4*i7 + 1024*i8 + 32768*i4] * t1[i11 + 128*i7 + 32768*i5];
          }
          #pragma omp simd
          for (int i11 = 0;(i11 < 32);i11 = (1 + i11)) {
            t3[i11] += x0[1 + 4*i7 + 1024*i8 + 32768*i4] * t1[32 + i11 + 128*i7 + 32768*i5];
          }
          #pragma omp simd
          for (int i11 = 0;(i11 < 32);i11 = (1 + i11)) {
            t3[i11] += x0[2 + 4*i7 + 1024*i8 + 32768*i4] * t1[64 + i11 + 128*i7 + 32768*i5];
          }
          #pragma omp simd
          for (int i11 = 0;(i11 < 32);i11 = (1 + i11)) {
            t3[i11] += x0[3 + 4*i7 + 1024*i8 + 32768*i4] * t1[96 + i11 + 128*i7 + 32768*i5];
          }

          for (int i12 = 0;(i12 < 32);i12 = (1 + i12)) {
            t2[(i12 + (32 * i8))] = t3[i12];
          }
        }
    , }
      for (int i13 = 0;(i13 < 32);i13 = (1 + i13)) {
        for (int i14 = 0;(i14 < 32);i14 = (1 + i14)) {
          output[((((i14 + (32 * i5)) + (1024 * i13)) + (32768 * i4)))] = t2[(i14 + (32 * i13))];
        }
      }
} } }
```

Listing 7: C code generated for the *parallel* matrix multiplication.

2.2 Handwritten Sketches

This section contains the handwritten sketches used to guide the rewriting of the matrix multiplication program. These sketches are enumerated in Table 4 of the paper.

The sketches use sketch abstractions defined by combining generic constructs from SketchBasic. Listing 8 shows some of the sketch abstractions used. → is a function type, and `n.dt` is an array type of `n` elements of domain type `dt`. The type annotations restrict the iteration domains of `map` and `reduceSeq`: the input arrays must have type `n.?dt`, and therefore such patterns will iterate over `n` elements.

```
def containsMap(n: NatSketch, f: Sketch): Sketch =
  contains((map :: ?t → n.?dt → ?t) f)
def containsReduceSeq(n: NatSketch, f: Sketch): Sketch =
  contains((reduceSeq :: ?t → ?t → n.?dt → ?t) f)
def containsAddMul: Sketch =
  contains(? + contains(×))
```

Listing 8: Some sketch abstractions used below.

```
containsMap(m,
 containsMap(n,
  containsReduceSeq(k,
   containsAddMul)))
```

Listing 9: A sketch for the *baseline* goal.

```
containsMap(m / 32,
 containsMap(32,
  containsMap(n / 32,
   containsMap(32,
    containsReduceSeq(k / 4,
     containsReduceSeq(4,
      containsAddMul))))))
```

Listing 10: *split* sketch specifying how to split loops for all 7 goals.

```
containsMap(m / 32,
 containsMap(n / 32,
  containsReduceSeq(k / 4,
   containsReduceSeq(4,
    containsMap(32,
     containsMap(32,
      containsAddMul))))))
```

Listing 11: $reorder_1$ sketch for the *blocking* and *vectorization* goals.

```
containsMap(m / 32,
 containsMap(n / 32,
  containsReduceSeq(k / 4,
   containsMap(32,
    containsReduceSeq(4,
     containsMap(32,
      containsAddMul))))))
```

Listing 12: $reorder_2$ sketch for the *loop-perm*, *array-packing*, *cache-blocks*, and *parallel* goals.

```
containsMap(m / 32,
 containsMap(n / 32,
  containsReduceSeq(k / 4,
   containsReduceSeq(4,
    containsMap(32,
     containsMap(1,
      containsAddMulVec))))))
```

Listing 13: $lower_1$ sketch goal for the *vectorization* goal.

```
containsMap(m / 32,
 containsMap(n / 32,
  containsReduceSeq(k / 4,
   containsMap(32,
    containsReduceSeq(4,
     containsMap(1,
      containsAddMulVec))))))
```

Listing 14: $lower_2$ sketch goal for the *loop-perm* goal.

```
containsMap(m / 32,
 containsMap(n / 32,
  containsReduceSeq(k / 4,
   containsMap(32,
    containsReduceSeq(4,
     containsMap(32,
      containsAddMul))))),
 containsToMem(n.k.f32,
  containsMap(n / 32,
   containsMap(k,
    containsMap(32.f32, ?)))))
```

Listing 15: *store* sketch for the *array-packing*, *cache-blocks*, and *parallel* goals.

```
containsMap(m / 32,
 containsMap(n / 32,
  containsReduceSeq(k / 4,
   containsMap(32,
    containsReduceSeq(4,
     containsMap(1,
      containsAddMulVec))))),
 containsToMem(n.k.f32,
  containsMap(n / 32,
   containsMap(k,
    containsMap(1.<32>f32, ?)))))
```

Listing 16: $lower_3$ sketch goal for the *array-packing* goal.

```
containsMap(m / 32,
 containsMap(n / 32,
  containsReduceSeq(k / 4,
   containsMap(32,
    containsReduceSeqUnroll(4,
     containsMap(1,
      containsAddMulVec))))),
 containsToMem(n.k.f32,
  containsMapPar(n / 32,
   containsMap(k,
    containsMap(1.<32>f32, ?)))))
```

Listing 17: $lower_4$ sketch goal for the *cache-blocks* goal.

```
containsMapPar(m / 32,
 containsMap(n / 32,
  containsReduceSeq(k / 4,
   containsMap(32,
    containsReduceSeqUnroll(4,
     containsMap(1,
      containsAddMulVec))))),
 containsToMem(n.k.f32,
  containsMapPar(n / 32,
   containsMap(k,
    containsMap(1.<32>f32, ?)))))
```

Listing 18: $lower_5$ sketch goal for the *parallel* goal.

2.3 Discovered RISE Programs for the parallel Optimization Goal

This section contains the RISE programs that are found using guided equality saturation for the *parallel* matrix multiplication optimization goal.

The initial program is shown in listing 19. The intermediate programs are shown in listings 20 to 22, and each satisfy a corresponding sketch guide. The final program satisfying the sketch goal is shown in listing 23.

Additionally, a final automatic transformation is applied as a final cleanup phase before generating code with the RISE compiler. Sequential loops and memory copies are inserted where required, and let expressions are hoisted as much as possible, resulting in listing 24.

```
Λn0:nat. Λn1:nat. Λn2:nat. λx0. λx1.
map (λx2.
 map (λx3.
  reduce add 0 (map (λx4. (mul (fst x4) (snd x4))) (zip x2 x3)))
  (transpose x1))
 x0
```

Listing 19: Initial RISE program for matrix multiplication.

```
Λn0:nat. Λn1:nat. Λn2:nat. λx0. λx1.
join (
 map (
  map (λx2.
   join (
    map (
     map (λx3.
      reduceSeq (λx4. λx5.
       add x4 (reduceSeq (λx6. λx7. add x6 (mul (fst x7) (snd x7))) 0 x5))
       0
       (split 4 (zip x2 x3))))
     (split 32 (transpose x1)))))
 (split 32 x0))
```

Listing 20: RISE program satisfying the *split* sketch guide.

```
Λn0:nat. Λn1:nat. Λn2:nat. λx0. λx1.
join (map (map join) (map transpose (
 map (
  map (λx2.
   reduceSeq (λx3. λx4.
    map (λx5.
     (λx6. reduceSeq (λx7. λx8.
      map (λx9. add (fst x9) (mul (fst (snd x9)) (snd (snd x9))))
       (zip x7 x8))
      (fst x6)
      (transpose (snd x6)))
     (unzip (zip (fst x5) (snd x5))))
     (zip x3 x4))
    (generate (λx3. generate (λx4. 0)))
    (transpose x2)))
 (map transpose (map (map (λx2.
  map transpose (map (map (λx3.
   split 4 (zip x2 x3))) (split 32 (transpose x1)))))
  (split 32 x0))))))
```

Listing 21: RISE program satisfying the $reorder_2$ sketch guide.

```
Λn0:nat. Λn1:nat. Λn2:nat. λx0. λx1.
join (map (map join) (map transpose (
 map (
  map (λx2.
   reduceSeq (λx3. λx4.
    map (λx5.
     reduceSeq (λx6. λx7.
      map (λx8.
       add (fst x8) (mul (fst (snd x8)) (snd (snd x8))))
       (zip x6 x7))
     (fst (unzip (zip (fst x5) (snd x5))))
     (transpose (snd (unzip (zip (fst x5) (snd x5))))))
    (zip x3 x4))
   (generate (λx3. generate (λx4. 0)))
   (transpose x2)))
 (map transpose (map (map (λx2.
   map transpose (map (map (λx3. split 4 (zip x2 x3)))
    (split 32 (let (toMem (
      join (map transpose (map (map (map (λx3. x3)))
       (map transpose (split 32 (transpose x1)))))))
      (λx3. x3))))))
  (split 32 x0))))))
```

Listing 22: RISE program satisfying the *store* sketch guide.

```
Λn0:nat. Λn1:nat. Λn2:nat. λx0. λx1.
join (mapPar (λx2.
  map join (transpose (map (λx3.
   reduceSeq (λx4. λx5.
    map (λx6.
     reduceSeqUnroll (λx7. λx8.
      (λx9. asScalar (map (λx10.
       add (fst x10) (mul (fst (snd x10)) (snd (snd x10))))
       (zip (asVector 32 (fst (unzip x9)))
        (zip (asVector 32 (fst (unzip (snd (unzip x9)))))
         (asVector 32 (snd (unzip (snd (unzip x9))))))))))
      (zip x7 x8))
      (fst (unzip (zip (fst x6) (snd x6))))
      (transpose (snd (unzip (zip (fst x6) (snd x6))))))
     (zip x4 x5))
    (generate (λx4. generate (λx5. 0)))
    (transpose x3))
   (transpose x2))))
  (map
   (map (λx2. map transpose
      (map (map (λx3. split 4 (zip x2 x3)))
        (split 32 (let (toMem (join (
          mapPar (λx3.
           transpose (map (λx4.
            asScalar (map (λx5. x5) (asVector 32 x4)))
            (transpose x3)))
          (split 32 (transpose x1)))))
         (λx3. x3))))))
   (split 32 x0)))
```

Listing 23: RISE program satisfying the $lower_4$ sketch goal.

```
Λn0:nat. n1:nat. Λn2:nat. λx0. λx1.
let (toMem (
 join (mapPar (λx18.
  transpose (mapSeq (λx19.
   asScalar (mapSeq (λx20. x20) (asVector 32 x19)))
   (transpose x18)))
  (split 32 (transpose x1)))))
 (λx21. join (
  mapPar (λx2.
   map join (transpose (mapSeq (λx3.
    mapSeq (mapSeq (λx4. x4)) (reduceSeq (λx5. λx6.
     mapSeq (λx7. mapSeq (λx8. x8) (
      reduceSeqUnroll (λx9. λx10.
       asScalar (mapSeq (λx11.
        add (fst x11) (mul (fst (snd x11)) (snd (snd x11))))
        (zip (asVector 32 (fst (unzip (zip x9 x10))))
         (zip (asVector 32 (fst (unzip (snd (unzip (zip x9 x10))))))
          (asVector 32 (snd (unzip (snd (unzip (zip x9 x10)))))))))
      (mapSeq (λx12. x12) (fst (unzip (zip (fst x7) (snd x7)))))
      (transpose (snd (unzip (zip (fst x7) (snd x7))))))
     (zip x5 x6))
    (mapSeq (mapSeq (λx13. x13)) (generate (λx14. generate (λx15. 0))))
    (transpose x3)))
   (transpose x2))))
  (map (λe3839.
   map (λx16.
    map transpose (map (map (λx17. split 4 (zip x16 x17))) (split 32 x21)))
    e3839)
   (split 32 x0))))
```

Listing 24: RISE program after final lowering.