Automatic Generation of LQCD Correlator Evaluation Code from a High Level Specification (WIP)

Presented by Teo Collin

Working with:

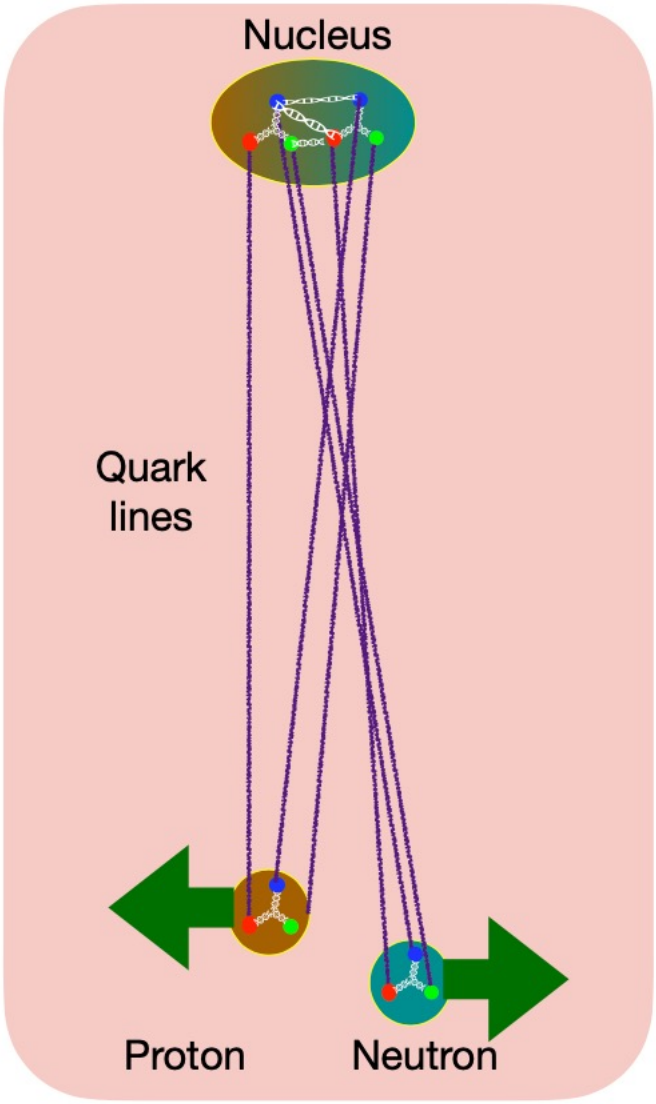
Ryan William Abbott, Saman Amarasinghe, Riyadh Baghdadi, Will Detmold, Andrew Pochinsky, Phiala Shanahan, Richard Sollee, Michael L. Wagman

SciDAC 2022

2022-12-2

Problem Specification

- You have a lattice system that fits the following model:
  - Finite Number of Sites (Sources and Sinks)
  - Allocation of quarks and anti-quarks of specific flavors to each site.
    - Per flavor, # of quarks = # of anti-quarks
- You have already computed the quark propagator via your giant eigenvalue computation.
  - You have already decomposed the interpolating field to a low rank linear combinations of anti-symmetric product of propagators.
    - We could adopt this to other pre-processing schemes.
- You want to measure things:
  - how do you quickly write down and evaluate the correlator?



Prior Work

- Computer Science: Tiramisu (High Performance DSL)
- Physics: Efficient algorithms for specific correlators
- Synthesis of Physics and CS
  - Evaluation of hexaquark and dibaryon correlation functions
- Outstanding Problems in the synthesis

Prior Work: Tiramisu (tiramisu-compiler.org)

- A polyhedral compiler: exploits that many loop nests look like polyhedra of integers
- Designed for optimizing **loop nests manipulating arrays**
- **C++ API** for expressing algorithms and optimizations
- Separates Algorithm from Optimization Commands:
  - Separate set of commands for loop manipulation, explicit hardware features, data layout, and data movement

1. Declare loop computations (unoptimized).

Pseudocode

```
for x in 0 … N
  for y in 0 … N
    out[x, y] = 0;
```

2. Declare optimization commands:
   1. Iteration space/Data layout/Precomputation transformations
   2. Use of hardware parallelism (threads, blocks, atomics, etc..)
   3. Use of different memory technologies (cache, registers, …)
   4. Communication to make this work…

Tiramisu

```
var x(0, N), y(0, N);
computation out(x, y);
out(x, y) = 0;

out.parallelize(x);
out.vectorize(y, 4);
```

Tiramisu Example

Polyhedra

```
input c_B({i, j}, Float64);
input c_C({i, j}, Float64);
computation c_T1_init({i, k}, 0);
computation c_T1({i, j, k}, Float64);
c_T1.set_expression(c_T1(i, j, k) + c_A(i, j) * c_B(j, k));
```

Level 1

```
c_T1.interchange(j, k);
c_T1_init.gpu_tile(i, k, 16, 16, i0, k0, i1, k1);
c_T1.gpu_tile(i, k, 16, 16, i0, k0, i1, k1);
```

Level 2

```
copy_A_to_device.then(copy_B_to_device,
computation::root).then(copy_C_to_device,
computation::root).then(c_T1_init, computation::root).then(c_T1, k1);
```

Level 3

```
c_A.store_in(&b_A_gpu);
c_B.store_in(&b_B_gpu);
c_T1_init.store_in(&b_T1_gpu);
c_T1.store_in(&b_T1_gpu, {i, k});
```

Tiramisu Example (There is also Halide)

Polyhedra

```
input c_B({i, j}, Float64);
input c_C({i, j}, Float64);
computation c_T1_init({i, k}, 0);
computation c_T1({i, j, k}, Float64);
c_T1.set_expression(c_T1(i, j, k) + c_A(i, j) * c_B(j, k));
```

Functions

```
conv(c, x, y, n) = bias(c);
conv(c, x, y, n) += filter(c, r.y, r.z, r.x) *
input(r.x, x + r.y, y + r.z, n);

relu(c, x, y, n) = max(0, conv(c, x, y, n));
```

Schedule

Level 1

```
c_T1.interchange(j, k);
c_T1_init.gpu_tile(i, k, 16, 16, i0, k0, i1, k1);
c_T1.gpu_tile(i, k, 16, 16, i0, k0, i1, k1);
```

Level 2

```
copy_A_to_device.then(copy_B_to_device,
computation::root).then(copy_C_to_device,
computation::root).then(c_T1_init,
computation::root).then(c_T1, k1);
```

```
conv.compute_at(relu, xo)
.store_in(MemoryType::Register)
                .gpu_lanes(c)
                .unroll(x)
                .unroll(y)
                .update()
                .split(r.x, rxo, rxi, 16)
                .split(rxi, rxi, rxii, 2)
                .reorder(c, rxii, x, y, r.y,
                r.z, rxi, rxo)
                .gpu_lanes(c)
                .unroll(x)
```

Level 3

```
c_A.store_in(&b_A_gpu);
c_B.store_in(&b_B_gpu);
c_T1_init.store_in(&b_T1_gpu);
c_T1.store_in(&b_T1_gpu, {i, k});
```

Physics: Efficient algorithms (1) (Roughly Delivered)

- First, the propagators can be post-processed to eliminate redundancy in the tensor product of the propagator's spin color space:

Detmold, W., & Orginos, K. (2013). Nuclear correlation functions in lattice QCD. In Physical Review D (Vol. 87, Issue 11). American Physical Society (APS). https://doi.org/10.1103/physrevd.87.114512

$$N(b) = \sum_{a_1=(x,y,z,f,s,c),\cdots a_n=(x,y,z,f,s,c)} w_{a_1,\cdots,a_n} S(a_1,b_1)\cdots S(a_n,b_n)$$

$$N(b) = \sum_{a'_1=(x,y,z),\cdots,a'_n=(x,y,z)} \sum_{k=1}^{N_w} w_k \sum_{i\in I_k} \sigma(i) S(a'_1,i_1,b_1)\cdots S(a'_n,i_n,b_n)$$

- Critically, $I_{\square}$ will have a product of permutation groups structure.

Physics: Efficient algorithms (2)

- Second, the computation induced via this form sometimes allows a pre-compute that reduces the time complexity, making a few cases tractable.

Detmold, W., & Orginos, K. (2013). Nuclear correlation functions in lattice QCD. In Physical Review D (Vol. 87, Issue 11). American Physical Society (APS). https://doi.org/10.1103/physrevd.87.114512

$$N(b) = \sum_{a'_1=(x,y,z),\cdots,a'_n=(x,y,z)} \sum_{k=1}^{N_w} w_k \sum_{i \in I_k} \sigma(i) S(a'_1, i_1, b_1) \cdots S(a'_n, i_n, b_n)$$

- We want to compute (very roughly) things like:

$$< N, \overline{N} >_A$$

- So this might be useful:

$$N_{x',y',z'}(s', c', f', b_2, \cdots, b_n) = \sum_{a'_1,\cdots,a'_n} \sum_{k=1}^{N_w} w_k \sum_{i \in I_k} \sigma(i) S(a'_1, i_1, x', y', z', s', c', f') \cdots S(a'_n, i_n, b_n)$$

Synthesis

- Appendix C: Pre-compute
  - For a more complex contraction, the one induced via the hexa-quark, precomputes required more care.
  - **The naïve size was 8TBs.**
  - To eliminate this, they precompute on the fly via fusing loop nested together.
    - This sometimes creates redundant work.
  - Trying various versions of this without Tiramisu would be rather intensive.
- More generally: Portable-ish Parallelism.
  - Allowed GPU/Multicore targets.
  - Allowed easily messing around with loop order and data layout.

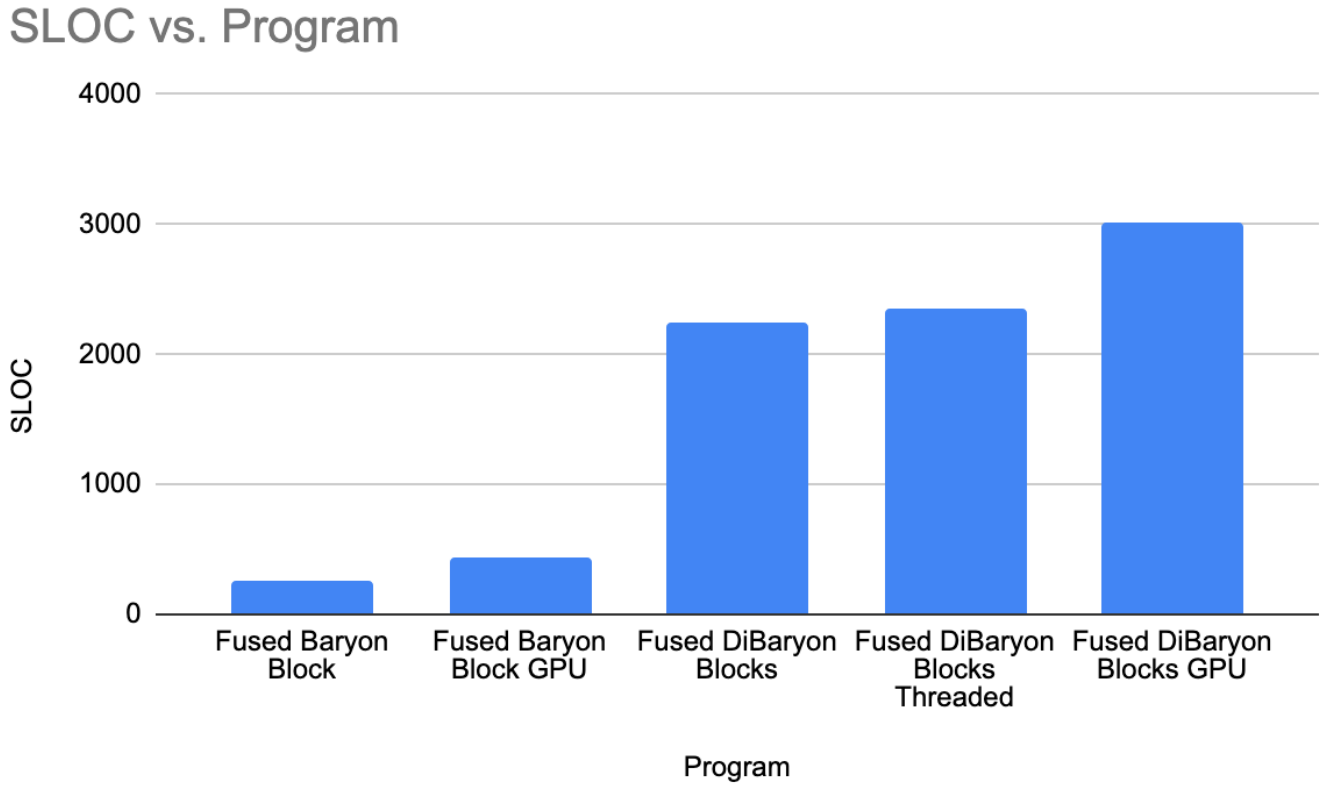
Amarasinghe, S., Baghdadi, R., Davoudi, Z., Detmold, W., Illa, M., Parreno, A., ... & Wagman, M. L. (2021). A variational study of two-nucleon systems with lattice QCD. *arXiv preprint arXiv:2108.10835*.

```
Roughly starting with:
for x in 0 … N
  for y in 0 … N
   for a in 0… N
    precompute[x, y, a] = …; //expensive computation

For z in 0…N
   for x in 0.. N
      for y in 0… N:
         for a in 0..N:
            out[z] = compute(precompute[x,y,a],z,a)
```

```
Go to:
For z in 0…N
   for x in 0.. N
      for y in 0… N:
         for a in 0..N:
            precompute[a] = …
         for a in 0..N:
            out[z] = compute(precompute[a],z,a)
```

New Problem: Scalability of this approach.

- First, someone needs to figure out a good precomputation structure.
  - Doing this by hand will get labor intensive quickly.
  - And then someone needs to code it.
- Second, the amount of code to implement and optimize this in Tiramisu grows alarmingly quickly:
  - Though this is much better than how the equivalent C/Cuda code grows…



See https://github.com/Tiramisu-Compiler/tiramisu/tree/master/benchmarks/tensors
(With much thanks to Mike)

Solution: (Bold is Done, roughly)

- **Automatic generation of a correlator index expression from the specification**
  - **(Efficient) Representation of these programs**
- Automatic Transformation of the index expression to discover efficient algorithms
  - Efficient search of the program space
- **Automatic Transformation of index expressions to Tiramisu**
- Semi-Automatic optimization Tiramisu code.

Automatic generation of a correlator index expression from the specification

- We take a specific of:
  - List of Sites: Sources + Sinks
  - Allocation of quarks + Anti-quarks of specific flavors to sites.

Automatic generation of a correlator index expression from the specification

- We take a specific of:
  - List of Sites: Sources + Sinks
  - Allocation of quarks + Anti-quarks of specific flavors to sites.

- Based on these:
  - Number of sources + sinks determines number of space loops
  - Number of sources/sinks determines wave function arguments

```
For x in srcs //_source 1
  For x’ in snks //snk 1

    phisrc = WaveFunc(y in x)
    phisnk = Wavefunc(y in x’)^{\star}
```

Automatic generation of a correlator index expression from the specification

- We take a specific of:
  - List of Sites: Sources + Sinks
  - Allocation of quarks + Anti-quarks of specific flavors to sites.

- Based on these:
  - Number of sources + sinks determines number of space loops
  - Number of sources/sinks determines wave function arguments
  - We loop over source/sink weights
  - We loop over the permutations.

```
For x in srcs //_source 1
  For x’ in snks //snk 1

    phisrc = WaveFunc(y in x)
    phisnk = Wavefunc(y in x’)^{\star}

    For a_src in range(0, nweights)
     For a_snk in range(0, nweights)
      For perms in S_{num[flv1]}\times …
```

Automatic generation of a correlator index expression from the specification

- We take a specific of:
  - List of Sites: Sources + Sinks
  - Allocation of quarks + Anti-quarks of specific flavors to sites.

- Based on these:
  - Number of sources + sinks determines number of space loops
  - Number of sources/sinks determines wave function arguments
  - We loop over source/sink weights
  - We loop over the permutations.
  - We compute a product for each permutation.
  - For each source/sink quark pairing, we accumulate into our product, using spin/color indices induced by the permutation.

```
For x in srcs //_source 1
  For x’ in snks //snk 1

    phisrc = WaveFunc(y in x)
    phisnk = Wavefunc(y in x’)^{\star}

    For a_src in range(0, nweights)
     For a_snk in range(0, nweights)
      For perms in S_{num[flv1]}\times …
       r = 1.0
       For flv in range(0, nflvs):
        perm = perms[flv]
         For  quark in range(0, num[flv])
           r *=S(flv, x[quark],
                spinColor[a_src, quark],
                x’_{site[perm[quark]]},
                spinColor’[a_src, perm[quark]])
```

Automatic generation of a correlator index expression from the specification

- We take a specific of:
  - List of Sites: Sources + Sinks
  - Allocation of quarks + Anti-quarks of specific flavors to sites.

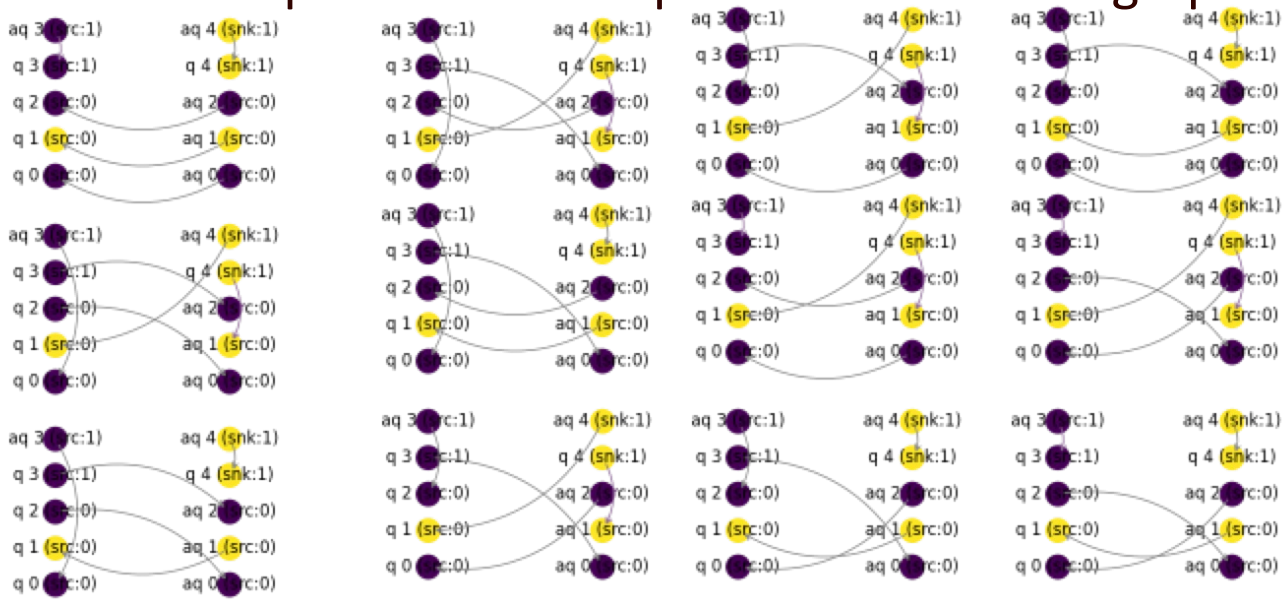
- Based on these:
  - Number of sources + sinks determines number of space loops
  - Number of sources/sinks determines wave function arguments
  - We loop over source/sink weights
  - We loop over the permutations.
  - We compute a product for each permutation.
  - For each source/sink quark pairing, we accumulate into our product, using spin/color indices induced by the permutation.
  - We accumulate the production into our sum.

```
For x in srcs //_source 1
  For x’ in snks //snk 1

    phisrc = WaveFunc(y in x)
    phisnk = Wavefunc(y in x’)^{\star}

    For a_src in range(0, nweights)
     For a_snk in range(0, nweights)
      For perms in S_{num[flv1]}\times …
       r = 1.0
       For flv in range(0, nflvs):
        perm = perms[flv]
         For  quark in range(0, num[flv])
           r *=S(flv, x[quark],
               spinColor[a_src, quark],
               x’_{site[perm[quark]]},
               spinColor’[a_src, perm[quark]])
      out+= r* phisrc*phisnk
```

Automatic generation of a correlator index expression from the specification

- We take a specific of:
  - List of Sites: Sources + Sinks
  - Allocation of quarks + Anti-quarks of specific flavors to sites.

- We could represent the outputs via contractor graphs:



- But this might be pre-mature……

```
For x in srcs //_source 1
  For x’ in snks //snk 1

    phisrc = WaveFunc(y in x)
    phisnk = Wavefunc(y in x’)^{\star}

    For a_src in range(0, nweights)
     For a_snk in range(0, nweights)
      For perms in S_{num[flv1]}\times …
       r = 1.0
       For flv in range(0, nflvs):
        perm = perms[flv]
         For  quark in range(0, num[flv])
           r *=S(flv, x[quark],
                 spinColor[a_src, quark],
                 x’_{site[perm[quark]]},
                 spinColor’[a_src, perm[quark]])
      out+= r* phisrc*phisnk
```

Solution: (Bold is Done, roughly)

- ~~**Automatic generation of a correlator index expression from the specification**~~
  - **(Efficient) Representation of these programs**
- Automatic Transformation of the index expression to discover efficient algorithms
  - Efficient search of the program space
- **Automatic Transformation of index expressions to Tiramisu**
- Semi-Automatic optimization of Tiramisu code.

(Efficient) Representation of these programs

- We augment a modified Einstein Index Notation language: $a_{ijk} = \sum_l b_{ijkl} c_l$
- Some very standard augmentations:
  - Allow for indirect access: $a_i = b_{c_i}$
  - Allow for complex number support.
  - Distinguished tensors for the propagator and wave functions
- Somewhat less standard:
  - We allow for a pre-computation to be represented:
  - Compare: $D_{ij} = A_{ik} B_{kl} C_{lj}$
  - And: Let $P_{rl} = A_{rk} B_{kl}$ IN $D_{ij} = P_{il} C_{lj}$
- **Even less standard but critical for space efficiency:**
  - Sums over products of permutation groups and subgroups:
  - Sign of permutations + Permutations as arrays.

$$\sum_{\sigma \in S^k \times S^k} , \sum_{\sigma \in S^k, \sigma(1)=1}$$

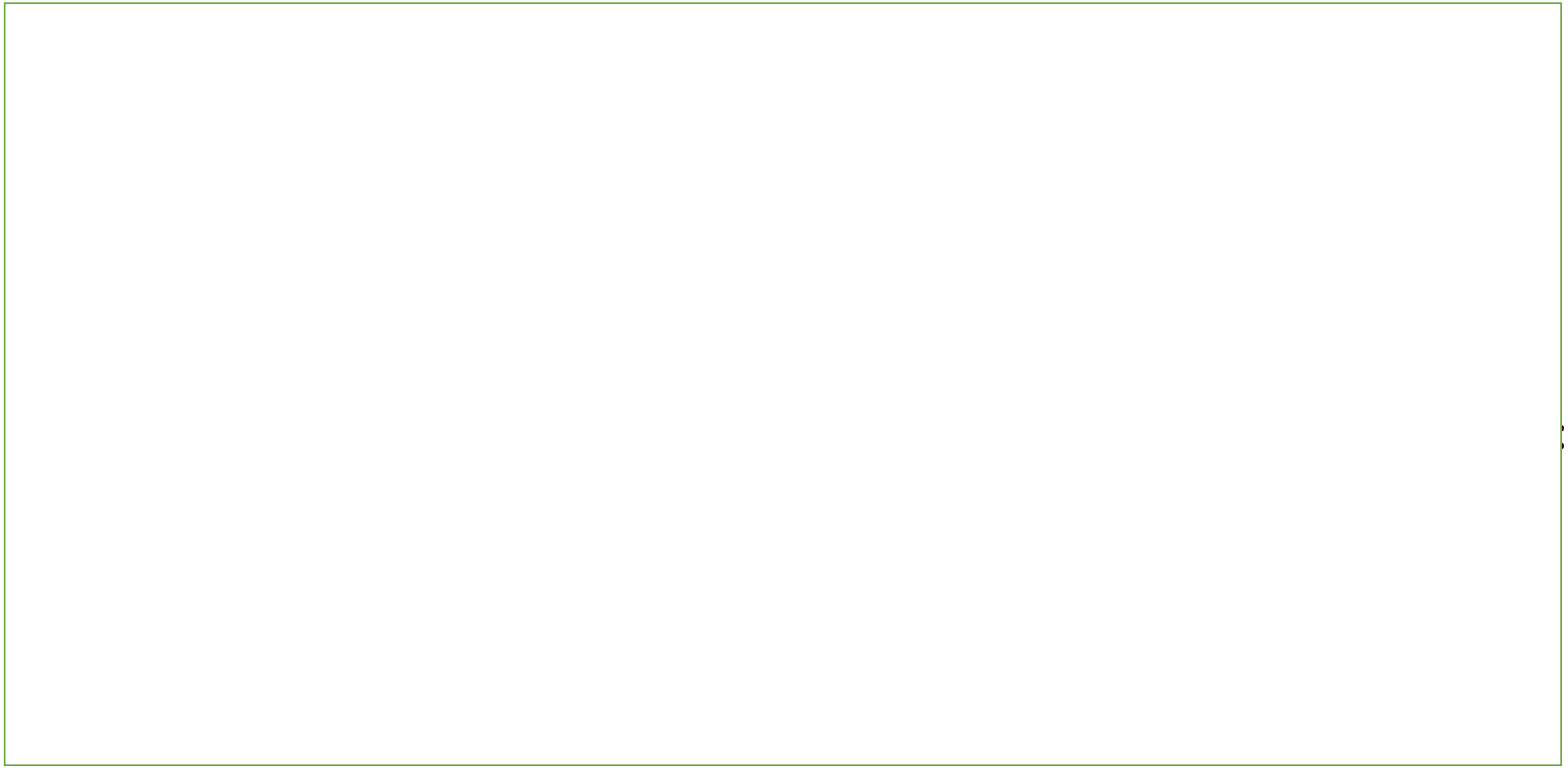
Solution: (Bold is Done, roughly)

- ~~**Automatic generation of a correlator index expression from the specification**~~
  - ~~**(Efficient) Representation of these programs**~~
- Automatic Transformation of the index expression to discover efficient algorithms
  - Efficient search of the program space
- **Automatic Transformation of index expressions to Tiramisu**
- Semi-Automatic optimization of Tiramisu code.

Automatic Transformation of the index expression to discover efficient algorithms

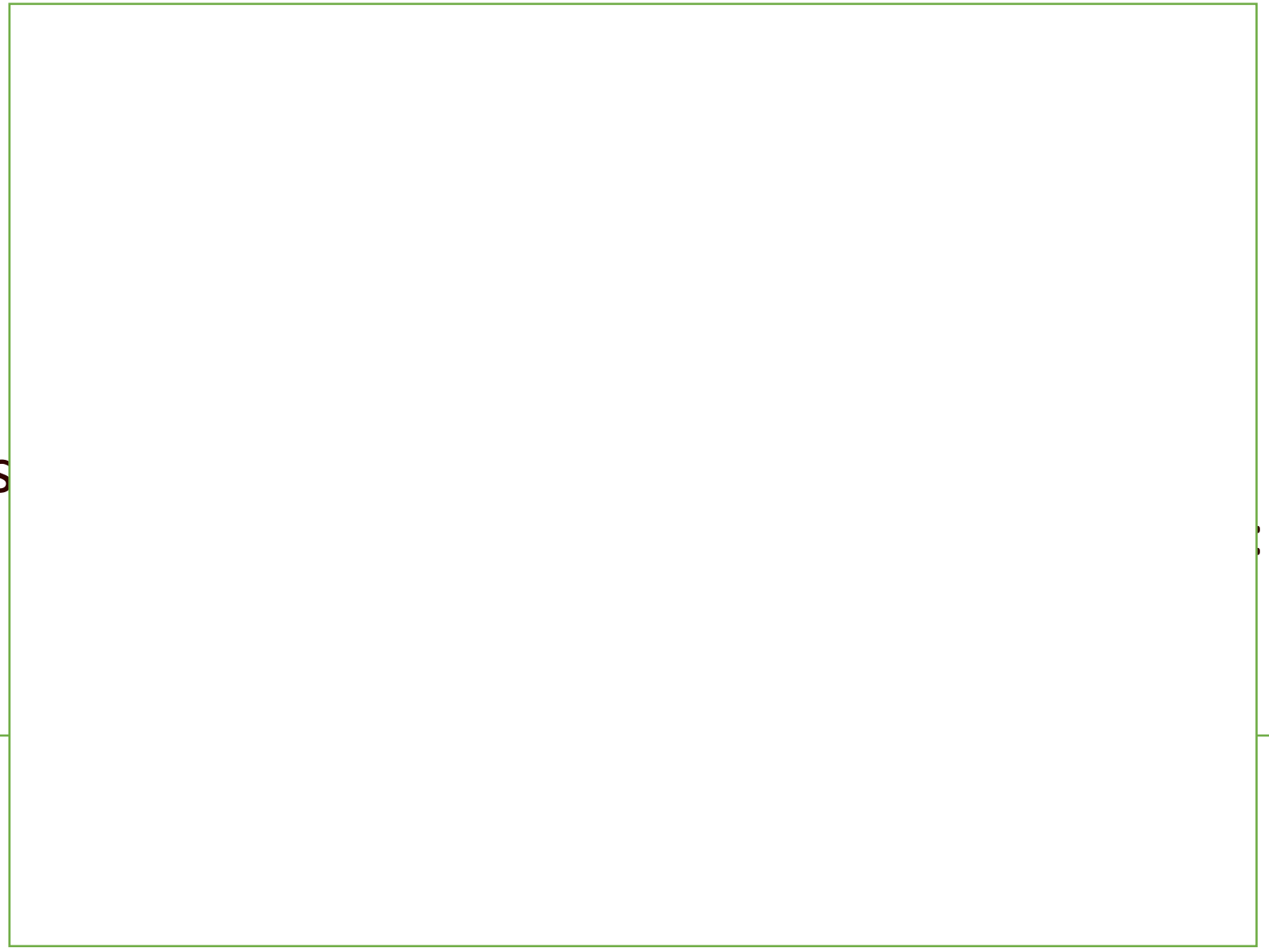
- We will use rewrite rules to discover equivalent versions of the correlation contraction.
- Example rewrite rules:
  - Start: $D_{ij} = A_{ik} B_{kl} C_{lj}$
  - To: Let $P_{rl} = A_{rk} B_{kl}$ in $D_{ij} = P_{il} C_{lj}$
  - Start: x+y
  - To: y+x
- We will need a more specialized collection of rules.
- We will need to search it in a scalable way

Rewrite rules, roughly



Rewrite rules, roughly

- Standard Loop rewrite rules:
  - Loop invariant code motion
  - Pre computations
  - Loop reordering
  - Unrolling
  - Constant Prop



Rewrite rules, roughly

- Standard Loop rewrite rules:
  - Loop invariant code motion
  - Pre computations
  - Loop reordering
  - Unrolling
  - Constant Prop
- Physics Rules:
  - Flavor Symmetry
  - Gamma-5 Hermiticity:

$$S(x, sc, y, sc') = \gamma^5 \overline{S(y, sc', x, sc)} \gamma^5$$

Rewrite rules, roughly

- Standard Loop rewrite rules:
  - Loop invariant code motion
  - Pre computations
  - Loop reordering
  - Unrolling
  - Constant Prop
- Physics Rules:
  - Flavor Symmetry
  - Gamma-5 Hermiticity:

$$S(x, sc, y, sc') = \gamma^5 \overline{S(y, sc', x, sc)} \gamma^5$$

- In-between:
  - Basic Math

$$x * 1 = x$$

  - Complex numbers

$$a\bar{a} = |a|^2$$

  - Permutation Groups:

$$\sum_{\sigma \in S^k} = \sum_{i=0}^{k} \sum_{\sigma \in S^k, \sigma(0)=i}$$

Rewrite rules, roughly

- Standard Loop rewrite rules:
  - Loop invariant code motion
  - Pre computations
  - Loop reordering
  - Unrolling
  - Constant Prop
- Physics Rules:
  - Flavor Symmetry
  - Gamma-5 Hermiticity:

$$S(x, sc, y, sc') = \gamma^5 \overline{S(y, sc', x, sc)} \gamma^5$$

- In-between:
  - Basic Math

$$x * 1 = x$$

  - Complex numbers

$$a\bar{a} = |a|^2$$

  - Permutation Groups:

$$\sum_{\sigma \in S^k} = \sum_{i=0}^{k} \sum_{\sigma \in S^k, \sigma(0)=i}$$

We believe these three categories combined should allow us to identified similar pre-computation strategies to those identified in previous papers but for new operators.

Solution: (Bold is Done, roughly)

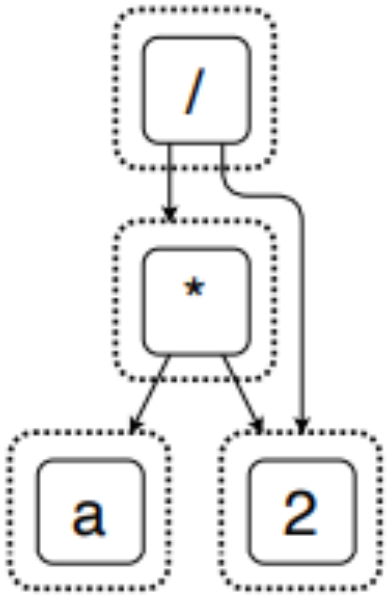
- ~~**Automatic generation of a correlator index expression from the specification**~~
  - ~~**(Efficient) Representation of these programs**~~
- ~~Automatic Transformation of the index expression to discover efficient algorithms~~
  - Efficient search of the program space:
    - (E-graphs, Rule Application strategies)
- **Automatic Transformation of index expressions to Tiramisu**
- Semi-Automatic optimization of Tiramisu code.

Efficient search I: E-graphs

- Rewrite Rules produce lots of duplicate/overlapping results
- E-Graphs are a data structure for storing these results
- We borrow from:

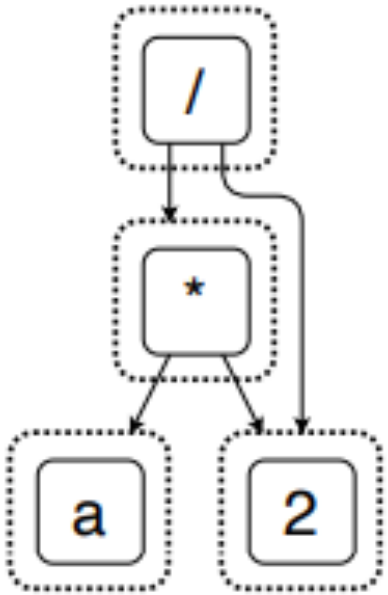
Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and extensible equality saturation. Proc. ACM Program. Lang. 5, POPL, Article 23 (January 2021), 29 pages. https://doi-org.libproxy.mit.edu/10.1145/3434304

Efficient search I: E-graphs

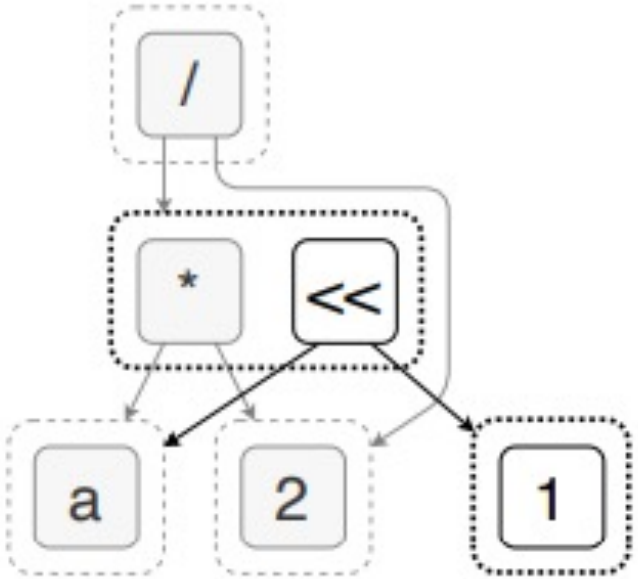


(a) Initial e-graph contains $(a \times 2)/2$.

Efficient search I: E-graphs

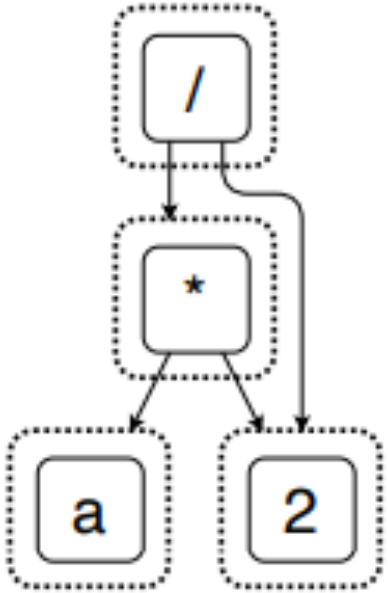


(a) Initial e-graph contains $(a \times 2)/2$.

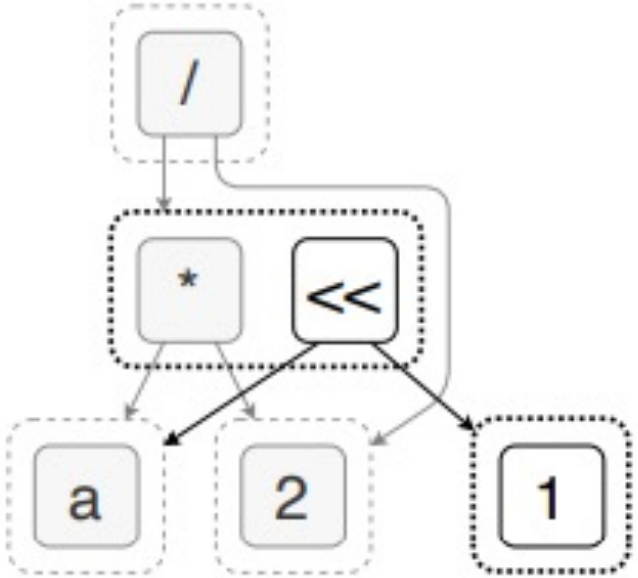


(b) After applying rewrite $x \times 2 \rightarrow x \ll 1$.

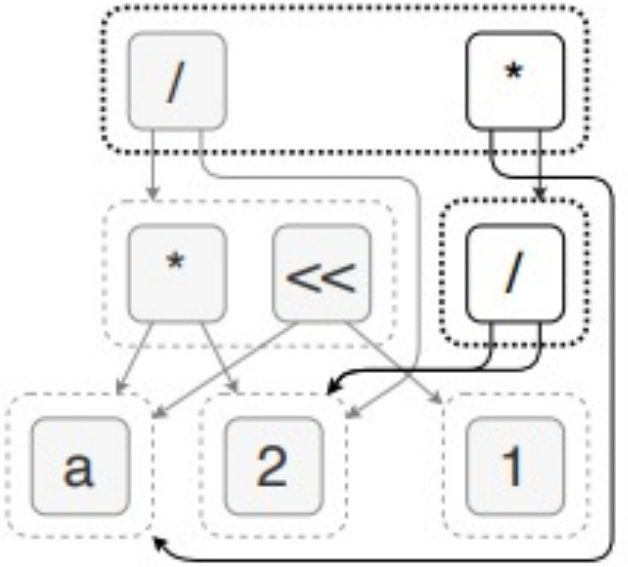
Efficient search I: E-graphs



(a) Initial e-graph contains $(a \times 2)/2$.



(b) After applying rewrite $x \times 2 \rightarrow x \ll 1$.



(c) After applying rewrite $(x \times y)/z \rightarrow x \times (y/z)$.

Solution: (Bold is Done, roughly)

- ~~**Automatic generation of a correlator index expression from the specification**~~
  - ~~**(Efficient) Representation of these programs**~~
- ~~Automatic Transformation of the index expression to discover efficient algorithms~~
  - Efficient search of the program space:
    - (~~E graphs~~, Rule Application strategies)
- **Automatic Transformation of index expressions to Tiramisu**
- Semi-Automatic optimization Tiramisu code.

Efficient search II: Limiting Rules

- Number of variants:
  - O(!(Nloops) * (NExprs) * 2^(Num_sub_exprs_per_expr) * Nloops …)
  - It is a more complex combinatorial problem…
- But Other similar approaches have run into this:

Humphrey, N., Detmold, W., Young, R. D., & Zanotti, J. M. (2022). Novel Algorithms for Computing Correlation Functions of Nuclei. *arXiv preprint arXiv:2201.04269*.

```
For x …
   For y…
      For z….
          For a …
            For b …
              For c…
                    lhs += expr2[a,b,c] * expr3[x,yz] *expr4[x,a,c]
```

```
For x …
   For y…
      For z….
          For c …
            For a …
              For b…
                    lhs += expr2[a,b,c] * expr3[x,yz] *expr4[x,a,c]
```

Efficient search II: Limiting Rules

- We want to restrain this search:
  - Key technical difficulty
- We believe:
  - Many loop re-orderings are pointless – only a few needed
  - Precomputes that produce too much intermediate memory are pointless.
  - Permutation Splitting guided by potential use of symmetries and pre-computations
- Build on the intuition of how you might manually find these
  - Even if it is not perfect.

Solution: (Bold is Done, roughly)

- ~~**Automatic generation of a correlator index expression from the specification**~~
  - ~~**(Efficient) Representation of these programs**~~
- ~~Automatic Transformation of the index expression to discover efficient algorithms~~
  - ~~Efficient search of the program space:~~
    - ~~(E-graphs, Rule Application strategies)~~
- **Automatic Transformation of index expressions to Tiramisu**
- Semi-Automatic optimization of Tiramisu code.

Automatic Transformation of index expressions to Tiramisu

- This is straightforward:
  - We wrote python bindings: https://github.com/wraith1995/tiramisu/tree/new-halide%2Bllvm/python_bindings
- The programs are reasonable looping structures.
  - Let Statements are a bit tricky: they amount to placing computations at the right loop levels
  - Permutations require tricks: representing permutations of k things as integers (O(k log k) bits)

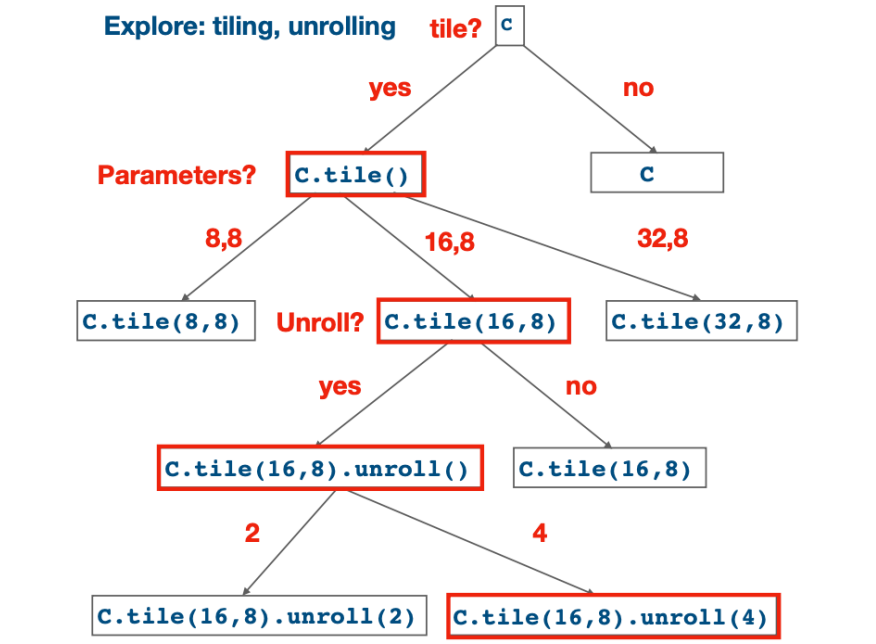
Solution: (Bold is Done, roughly)

- ~~**Automatic generation of a correlator index expression from the specification**~~
  - ~~**(Efficient) Representation of these programs**~~
- ~~Automatic Transformation of the index expression to discover efficient algorithms~~
  - ~~Efficient search of the program space:~~
    - ~~(E-graphs, Rule Application strategies)~~
- ~~**Automatic Transformation of index expressions to Tiramisu**~~
- Semi-Automatic optimization of low level Tiramisu code.

Optimization of Tiramisu code.

- A Few Possible Approaches:
  - Use Tiramisu Auto-scheduling:
    - Scaling is unclear.
  - Take sane guesses:
    - (e.g. storage order can just be chosen from loop order sometimes)
  - Expose saner scheduling templates to the user
    - Allow scheduling
    - but in a simpler interface.
  - Some Hybrid

Baghdadi, R., Merouani, M., Leghettas, M.-H., Abdous, K., Arbaoui, T., Benatchba, K., & Amarasinghe, S. (2021). A Deep Learning Based Cost Model for Automatic Code Optimization. Στο A. Smola, A. Dimakis, & I. Stoica *Proceedings of Machine Learning and Systems*



Questions?

- Automatic generation of a correlator index expression from the specification
  - (Efficient) Representation of these programs
- Automatic Transformation of the index expression to discover efficient algorithms
  - Efficient search of the program space
- Automatic Transformation of index expressions to Tiramisu
- Semi-Automatic optimization of low level Tiramisu code.