Globally Optimized Superword Level Parallelism Framework (goSLP)

GROUP #28:

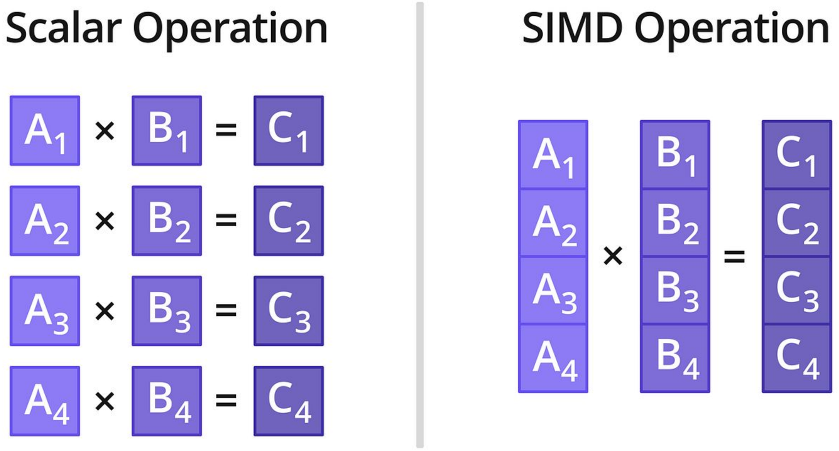
Rishi Rallapalli, Anvita Gollu, Jason Majoros, Luke Nelson

Acronyms and Definitions

- SLP - Superword Level Parallelism
- SIMD - Single Instruction, Multiple Data
- ILP - Integer Linear Programming
- Packing - Taking multiple scalars and packing them into one vector
- Unpacking - Extracting a scalar from a vector
- Shuffling - Reordering elements of the vector to different lanes

Background: What are SIMD and SLP?

SIMD: hardware concept that executes the same operation across multiple elements in parallel
SLP: compiler-level analysis that discovers and exploits SIMD opportunities



https://blog.wasmer.io/webassembly-and-simd-13badb9bf1a8

Problem: Traditional SLP in LLVM

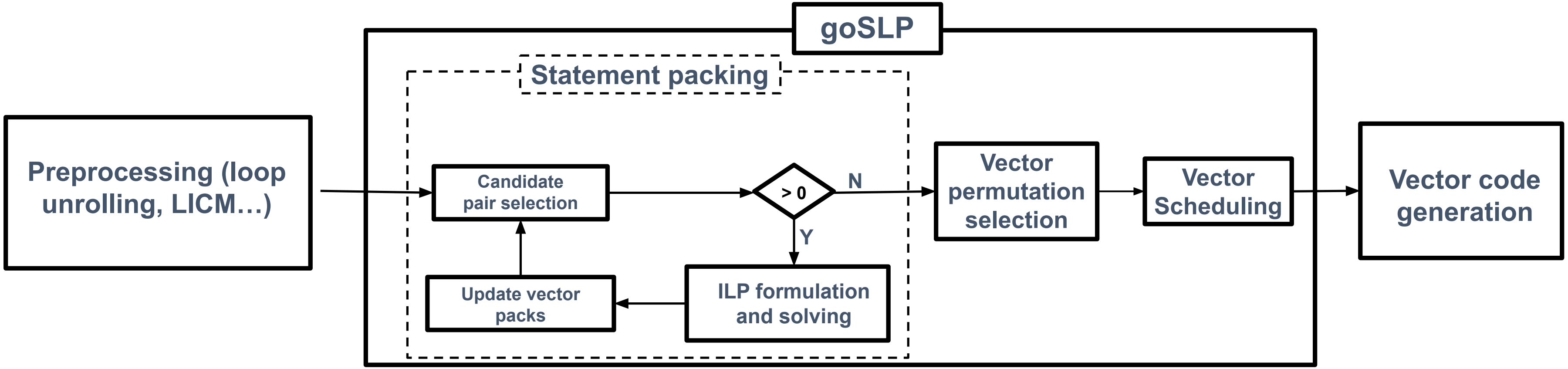
SLP in LLVM works at a **basic block level**
Uses **greedy** algorithms and local heuristics to choose operations to apply SIMD

This may lead to:

- Useless vector packs
- Extra packs/unpacks/shuffles
- Missed global SIMD opportunities that offer better performance

Key Idea of goSLP

goSLP treats SLP vectorization as a global optimization problem across an entire function



Packing - Constraints

For two statements, S_i and S_j to be packed together:

1. They must be **isomorphic** (operate on same data types)
2. They must be **independent**
3. They must be **schedulable** into a pack (protect program semantics)
4. They must be **access adjacent memory locations** (if relevant)
    a. A[i] and A[i+1] can be packed; A[i] and B[j] cannot.

For two packs, P_i and P_j to be packed together:

1. **No circular dependencies** between them
2. **No overlap** between them

Packing - Candidate Pairs and Decision Variables

1. Collect the set of statements f_S that can be paired with statement S
   - f_{S1} : { }, f_{S2} : { }, f_{S3} : $\{S_4\}$, f_{S4} : $\{S_3\}$, f_{S5} : $\{S_6, S_7\}$, f_{S6} : $\{S_5, S_7\}$, f_{S7} : $\{S_5, S_6\}$

2. Create decision variables for each **unique** pairing

$$V\{S_p,S_q\} = \begin{cases} 1 \text{ if the pack is formed} \\ 0 \text{ otherwise} \end{cases}$$

S_1: A_1= load(X)

S_2: A_2= load(X+N)

S_3: B_1= load(Y)

S_4: B_2= load(Y+1)

S_5: C_1= A_1+ B_1

S_6: C_2= A_2+ B_2

S_7: C_3= A_2+ B_1

Minimizing Cost

Goal: Apply vectorization when it reduces total cost

$$\min \; VS + PC_{vec} + PC_{nonvec} + UC$$

Subject to $\quad OC, CC$

VecVecUses = $\{\{S_3, S_4\} \rightarrow \{\{S_5, S_6\} \{S_6, S_7\}\}\}$

NonVecVecUses = $\{\{S_1, S_2\} \rightarrow \{\{S_5, S_7\} \{S_5, S_6\}\}$

$\{S_2, S_2\} \rightarrow \{\{S_6, S_7\}\}$

$\{S_3, S_3\} \rightarrow \{\{S_5, S_7\}\}\}$

S_1: A_1= load(X)

S_2: A_2= load(X+N)

S_3: B_1= load(Y)

S_4: B_2= load(Y+1)

S_5: C_1= A_1+ B_1

S_6: C_2= A_2+ B_2

S_7: C_3= A_2+ B_1

Integer Linear Programming (ILP)

We then feed the objective (minimize total cost), variables, and constraints in an ILP solver:

Returns: $V_p \in 0, 1$ $\quad (1 = \textit{choose pack, } 0 = \textit{don't})$

$$\min \; VS + PC_{vec} + PC_{nonvec} + UC$$

$$\text{Subject to} \quad OC, CC$$

Constraints:

1. No statement can belong to more than one pack
2. No circular dependencies

Iterative Widening

First ILP pass forms only 2-wide vector packs:
e.g., $\{S_3, S_4\}$ and $\{S_5, S_6\}$

Treat each selected pack as a single “vector statement”:
$V_1 = \{S_3, S_4\}$ $V_2 = \{S_5, S_6\}$

Run the ILP again on these vector statements to see if they can be paired:
$\{V_1, V_2\} \rightarrow$ 4-wide vector pack

Repeat until:

- No further profitable packs exist
- Maximum hardware SIMD width is reached

Vectorization Graphs and Lane Permutations

Suppose we have P1 = {a, b} P2 = {c, d} and P3 = {a + c, b + d}

Nodes represents each pack and an edge represents uses

We will build a graph like:

$$\begin{tikzcd} P1 \arrow[r] & P3 \\ P2 \arrow[ur] & \end{tikzcd}$$

Keep in mind P3 could could be packed as {a+c,b+d} or {b+d, a+c}

Constrained Vs Free Nodes

Some packs don't get to choose their order Ex:
Load X[i], X[i+1] or Store X[i], X[i+1]. These are <u>constrained</u> by the hardware

Some can be free like the example previously, P3 could be
{a + c, b + d} or {b + d, a + c}
But if P3 expects it one way over the other then we may need to shuffle P1 or P2

DP Algorithm (Minimize Shuffles)

Consider P1 → P3 → P4 where P1,2 are producers P3 is an add, P4 a store
P2 ↗

DP Algo:

- Start at P4 (Fixed due to a store) say X[i], X[i+1]
- P3 can be X[i], X[i+1] or X[i+1], X[i]←Choose the lower cost from P3->P4(option 1)
- P2,P1 same thing, best cost for P2->P3 and P1->P3
- Then walk down using those choices to fix each permutation

1. Start at P4 (Fixed due to a store)
2. P3 can be X[i], X[i+1] or X[i+1], X[i]

   Choose the lower cost from P3->P4(option 1)

   P2,P1 same thing, best cost for P2->P3 and P1->P3
3. Then walk down using those choices to fix each permutation

procedure ComputeMinAndSelectBest
  Inputs: graph G, candidate permutations FP_V for each node $V \in G$
  W = leaves(G)
  while !W.empty() **do**
    V = W.deque()
    for $P_V \in FP_V$ **do**
      $\mathrm{cost}_{min}(P_V, V) = 0$
      for $S \in \mathrm{succ}(V)$ **do**
        $\mathrm{cost}_{min}(P_V, V) \mathrel{+}= \min_{P_S \in FP_S} \mathrm{cost}_{min}(P_S, S) + \mathrm{perm\_cost}(P_S, P_V)$
        $\mathrm{arg}(P_V, V, S) = \mathrm{argmin}_{P_S \in FP_S} \mathrm{cost}_{min}(P_S, S) + \mathrm{perm\_cost}(P_S, P_V)$
    W.enque(pred(V))
  $W = \phi$
  for $R \in \mathrm{roots}(G)$ **do**
    $\mathrm{selected}(R) = \mathrm{argmin}_{P_R \in FP_R} \mathrm{cost}_{min}(P_R, R)$
    W.enque(succ(R))
  while !W.empty() **do**
    R = W.deque()
    P = pred(R)
    $\mathrm{selected}(R) = \mathrm{arg}(\mathrm{selected}(P), P, R)$
    W.enque(succ(R))

Results

- More optimal permutation selection of vector packed instructions
- Avoids unpacking statement bloat generated by LLVM’s SLP

```
A = sc * ij[1]
B = sc * ij[2]

a1 = A * ai - B * bi
a2 = A * a[3] - B * b[3]
a3 = A * a[2] - B * b[2]
a4 = A * a[1] - B * b[1]
```

(a) Scalar code

```
{A,B}   = {sc,sc} * {ij[1],ij[2]}

V1      = {A,B} * {ai,bi}
a1      = V1[0] - V1[1]

V2      = {A,A} * {a[2],a[3]}
V3      = {B,B} * {b[2],b[3]}
{a3,a2} = V2 - V3

V4      = {A,B} * {a[1],b[1]}
a4      = V4[0] - V4[1]
```

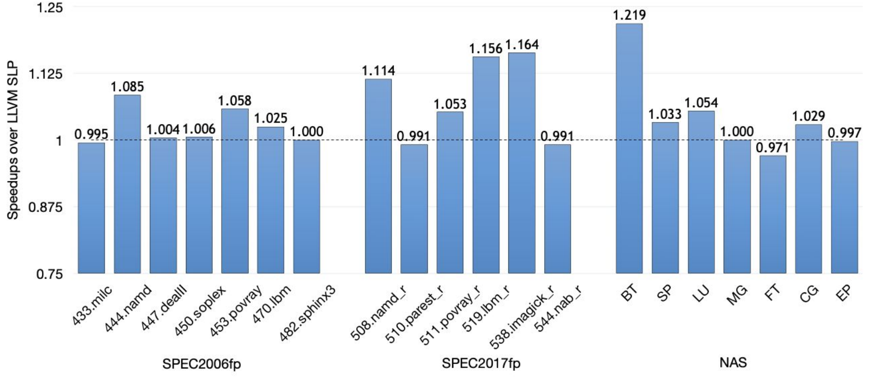
(b) LLVM SLP code

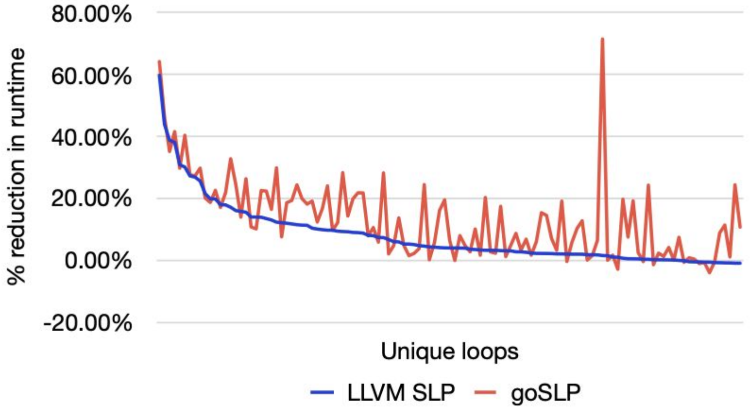
```
{A,B}   = {sc,sc} * {ij[1],ij[2]}

a1      = A * ai - B * bi
a2      = A * a[3] - B * [3]

V1      = {A,A} * {a[1],a[2]}
V2      = {B,B} * {b[1],b[2]}
{a4,a3} = V1 - V2
```

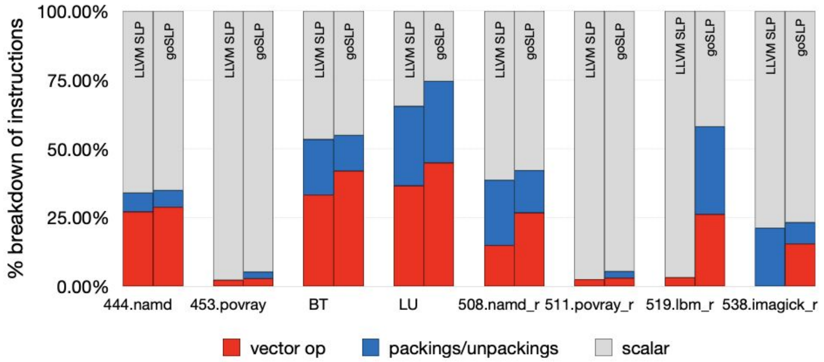
(b) goSLP code

Runtime Performance

Comparing goSLP vs LLVM's SLP auto-vectorizing compiler …

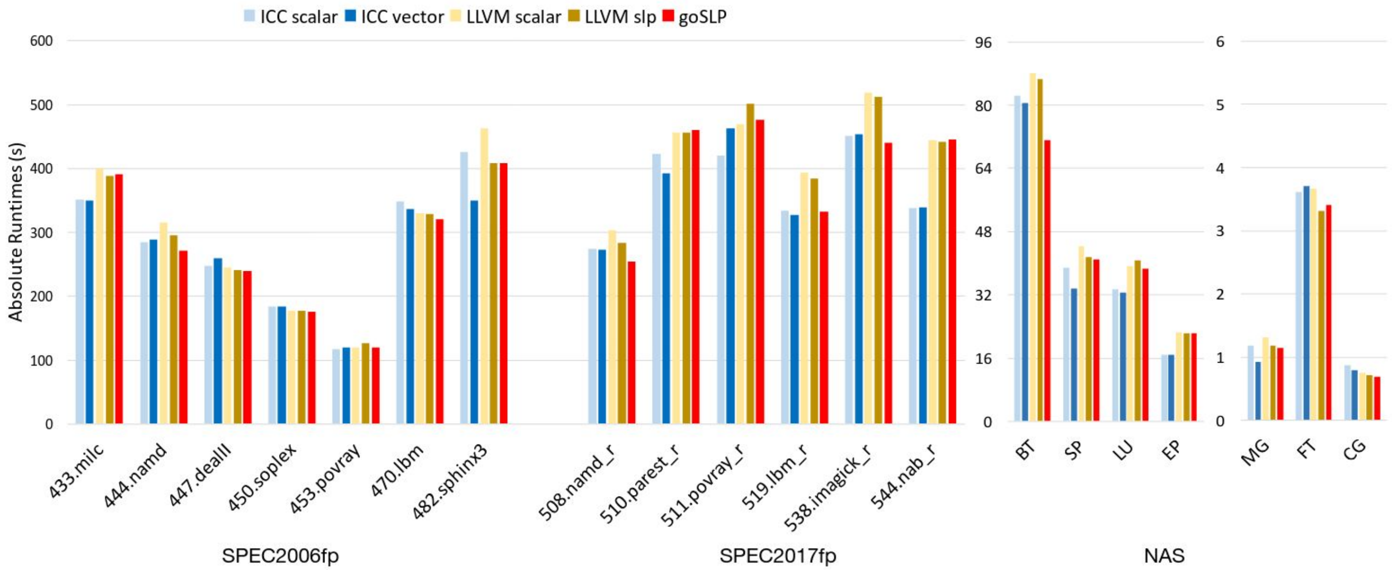






… and vs ICC

- ICC: Intel Commercial Compiler
- ICC is well-known for its loop vectorization approach



Overall

- Greedy algorithms and heuristics are fragile and miss the same vectorization opportunities that are provided by optimal search algorithms
- goSLP leads to objectively faster runtimes with more efficient and intelligent vectorization
- Cost minimization provides optimal framework
    - Theoretically, work remaining is to generate more correct cost function
    - Future research is evaluating this problem
- More flexible, and generalized, than previous local SLP optimizations
    - Allows for globally-reaching vector-packing (i.e. SLP looks beyond basic blocks)

Limitations

- Due to non-convexity of DP algorithm for cost minimization, this optimization scales at a greater than O(n) rate in terms of compilation time
- Comparatively much higher compile time than LLVM SLP, but not infeasible to users looking to eke out the last motes of performance

Benchmark	ILP size	ILP solutions		Compile Time(s)	
		optimal	feasible	goSLP	LLVM SLP
444.namd	61709	65	0	252.84	6.94
453.povray	207553	904	3	444.49	30.6
BT	412974	8	1	125.91	2.23
LU	539138	3	1	129.08	1.54
508.namd_r	174500	108	2	499.74	20.8
511.povray_r	207782	925	4	453.81	34.65
519.lbm_r	109971	13	0	65.44	0.34
538.imagick_r	318137	721	1	172.21	63.06

- The strength of the optimizations is limited by the accuracy of the cost function