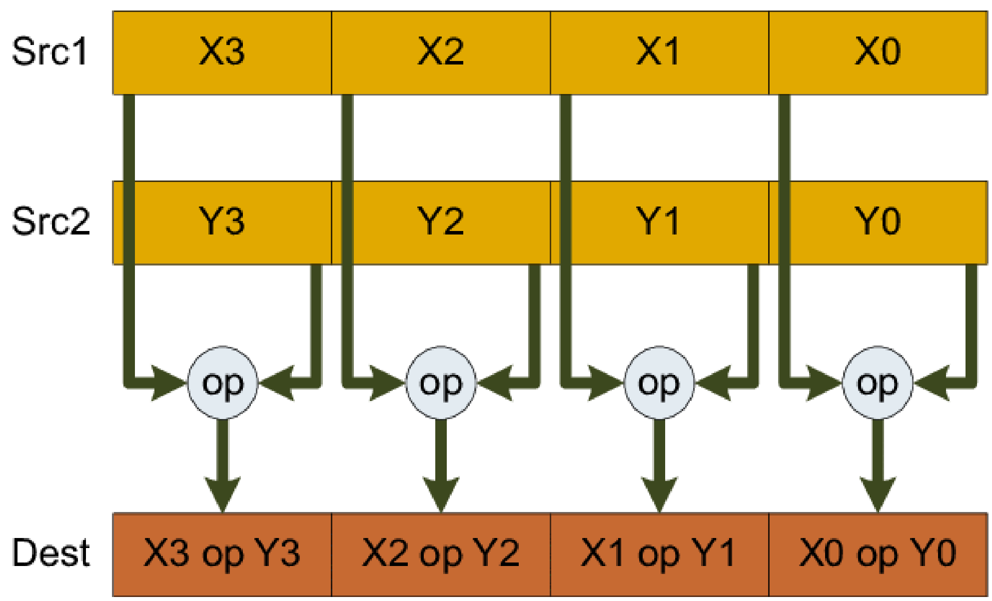
Superword Level Parallelism

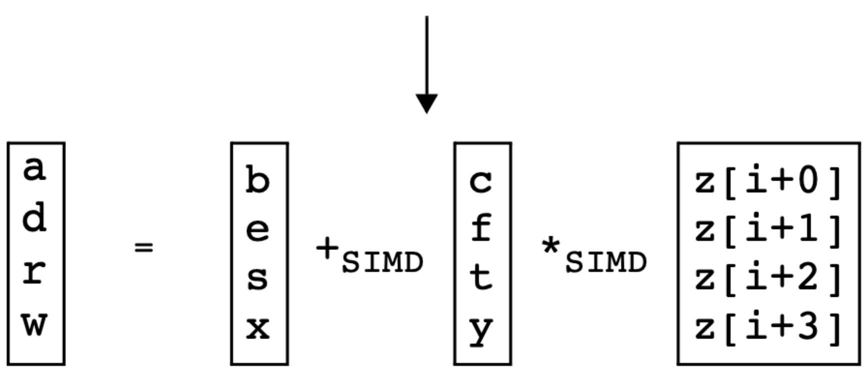
Ashwin Adulla (aadulla@andrew.cmu.edu), Li Shi (lishi@andrew.cmu.edu)

Introduction

Nowadays, many computer architectures have added support for multimedia extensions in the form of SIMD instructions. SIMD instructions allow the same operation to be performed on multiple pieces of data simultaneously, thus exploiting the inherent parallelism present in the application. Compilers employ various auto-vectorization strategies to recognize this parallelism and utilize the SIMD instructions. However, many of these strategies are very limited in their scope and sometimes require explicit indication by the user of what to parallelize through the use of library functions or annotated pragmas. As a result, the user often holds the burden of writing and formatting their code in such a way that the compiler can easily determine the inherent parallelism.




```
a = b + c * z[i+0]
d = e + f * z[i+1]
r = s + t * z[i+2]
w = x + y * z[i+3]
```



Algorithm

SLP extraction algorithm	A motivating example

<pre>SLP_extract: BasicBlock B → BasicBlock
    PackSet P ← EmptySet
    P ← find_adj_refs(B, P)
    P ← extend_packlist(B, P)
    P ← combine_packs(P)
    return schedule(B, [], P)</pre>
	<pre>b0 = Load A[i + 0]
	b1 = Load A[i + 1]
	b2 = Load A[i + 2]
	d0 = c0 + b0
	d1 = c1 + b1
	d2 = c2 + b2
	Store B[i + 0], d0
	Store B[i + 1], d1
	Store B[i + 2], d2</pre>

Step 1. Find adjacent memory references	

find_adj_refs: BasicBlock $B \times$ PackSet $P \rightarrow$ PackSet
  foreach Stmt $s \in B$ **do**
    foreach Stmt $s' \in B$ **where** $s \neq s'$ **do**
      if has_mem_ref(s) $\wedge$ has_mem_ref(s') **then**
        if adjacent(s, s') **then**
          Int $align \leftarrow$ get_alignment(s)
          if stmts_can_pack($B, P, s, s', align$) **then**
            $P \leftarrow P \cup \{\langle s, s' \rangle\}$
  return P
	<pre>Pack 0
	  b0 = Load A[i + 0]
	  b1 = Load A[i + 1]
	Pack 1
	  b1 = Load A[i + 1]
	  b2 = Load A[i + 2]
	Pack 2
	  Store B[i + 0], d0
	  Store B[i + 1], d1
	Pack 3
	  Store B[i + 1], d1
	  Store B[i + 2], d2</pre>

Step 2. Extend the pack lists	

extend_packlist: BasicBlock $B \times$ PackSet $P \rightarrow$ PackSet
  repeat
    PackSet $P_{prev} \leftarrow P$
    foreach Pack $p \in P$ **do**
      $P \leftarrow$ follow_use_defs(B, P, p)
      $P \leftarrow$ follow_def_uses(B, P, p)
  until $P \equiv P_{prev}$
  return P
	<pre>Pack 0
	  b0 = Load A[i + 0]
	  b1 = Load A[i + 1]
	Pack 1
	  b1 = Load A[i + 1]
	  b2 = Load A[i + 2]
	Pack 2
	  Store B[i + 0], d0
	  Store B[i + 1], d1
	Pack 3
	  Store B[i + 1], d1
	  Store B[i + 2], d2
	Pack 4
	  d0 = c0 + b0
	  d1 = c1 + b1
	Pack 5
	  d1 = c1 + b1
	  d2 = c2 + b2</pre>

Step 3. Combine packs

combine_packs: PackSet $P \rightarrow$ PackSet
repeat
PackSet $P_{prev} \leftarrow P$
foreach Pack $p = \langle s_1, \ldots, s_n \rangle \in P$ **do**
foreach Pack $p' = \langle s'_1, \ldots, s'_m \rangle \in P$ **do**
if $s_n \equiv s'_1$ **then**
$P \leftarrow P - \{p, p'\} \cup \{\langle s_1, \ldots, s_n, s'_2, \ldots, s'_m \rangle\}$
until $P \equiv P_{prev}$
return P

```
Pack 0
  b0 = Load A[i + 0]
  b1 = Load A[i + 1]
  b2 = Load A[i + 2]
Pack 2
  Store B[i + 0], d0
  Store B[i + 1], d1
  Store B[i + 2], d2
Pack 4
  d0 = c0 + b0
  d1 = c1 + b1
  d2 = c2 + b2
```

Step 4. Check dependency and schedule packs

schedule: BasicBlock B $\times$ BasicBlock B' $\times$ PackSet P
$\rightarrow$ BasicBlock
for $i \leftarrow 0$ **to** $|B|$ **do**
if $\exists p = \langle \ldots, s_i, \ldots \rangle \in P$ **then**
if $\forall s \in p$. deps_scheduled(s, B') **then**
foreach Stmt $s \in p$ **do**
$B \leftarrow B - s$
$B' \leftarrow B' \cdot s$
return schedule(B, B', P)
else if deps_scheduled(s_i, B') **then**
return schedule$(B - s_i, B' \cdot s_i, P)$
if $|B| \neq 0$ **then**
$P \leftarrow P - \{p\}$ **where** $p =$ first(B, P)
return schedule(B, B', P)
return B'

```
Pack 0
  b0 = Load A[i + 0]
  b1 = Load A[i + 1]
  b2 = Load A[i + 2]
Pack 4
  d0 = c0 + b0
  d1 = c1 + b1
  d2 = c2 + b2
Pack 2
  Store B[i + 0], d0
  Store B[i + 1], d1
  Store B[i + 2], d2
```

Step 5. Emit LLVM IR code

```
code_emit: BasicBlock B, PackSet P
    P ← find_pre_pack(B, P)
    P ← find_post_pack(B, P)
    code_gen(B, P)
```

```
Find pre pack:
 <0, 1, 2> (array indices)
 <c0, c1, c2>

Find post pack:
 None

Code generation:
 ...
 load <4 x float>, <4 x float>* %0
 ...
 %9 = fadd <4 x float> %6, %8
 ...
 store <4 x float> %9, <4 x float>* %10
 ...
```

```c
int test1(int a, int b, int c, int d, long i) {
  int e = a * A[i];
  int f = b * A[i + 1];
  int g = c * A[i + 2];
  int h = d * A[i + 3];
  return e + f + g + h;
}
```

→ <e, f, g, h> = <a, b, c, d> * A[i:i+3]
return e + f + g + h

<a, b, c, d> →

```
%2 = insertelement <4 x i32> undef, i32 %arg, i64 0
%3 = insertelement <4 x i32> %2, i32 %arg1, i64 1
%4 = insertelement <4 x i32> %3, i32 %arg2, i64 2
%5 = insertelement <4 x i32> %4, i32 %arg3, i64 3
```

A[i:i+3] →

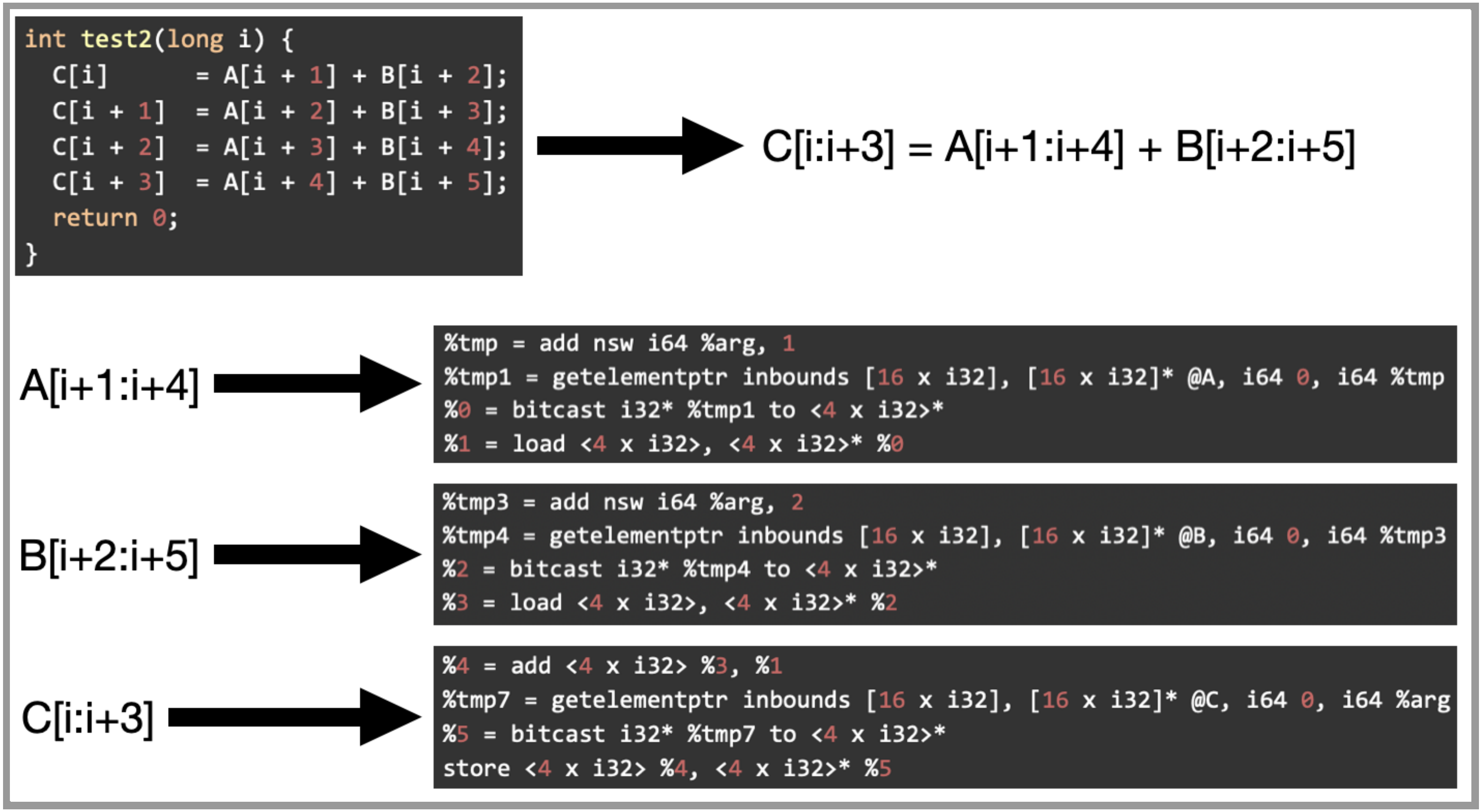
```
%tmp = getelementptr inbounds [16 x i32], [16 x i32]* @A, i64 0, i64 %arg4
%0 = bitcast i32* %tmp to <4 x i32>*
%1 = load <4 x i32>, <4 x i32>* %0
```

<e, f, g, h> →

```
%6 = mul <4 x i32> %1, %5
```

return e + f + g + h →

```
%7 = extractelement <4 x i32> %6, i64 0
%8 = extractelement <4 x i32> %6, i64 1
%9 = extractelement <4 x i32> %6, i64 2
%10 = extractelement <4 x i32> %6, i64 3
%tmp19 = add nsw i32 %8, %7
%tmp20 = add nsw i32 %tmp19, %9
%tmp21 = add nsw i32 %tmp20, %10
ret i32 %tmp21
```



int test2(long i) {
C[i] = A[i + 1] + B[i + 2];
C[i + 1] = A[i + 2] + B[i + 3];
C[i + 2] = A[i + 3] + B[i + 4];
C[i + 3] = A[i + 4] + B[i + 5];
return 0;
}
C[i:i+3] = A[i+1:i+4] + B[i+2:i+5]
A[i+1:i+4]
%tmp = add nsw i64 %arg, 1
%tmp1 = getelementptr inbounds [16 x i32], [16 x i32]* @A, i64 0, i64 %tmp
%0 = bitcast i32* %tmp1 to <4 x i32>*
%1 = load <4 x i32>, <4 x i32>* %0
B[i+2:i+5]
%tmp3 = add nsw i64 %arg, 2
%tmp4 = getelementptr inbounds [16 x i32], [16 x i32]* @B, i64 0, i64 %tmp3
%2 = bitcast i32* %tmp4 to <4 x i32>*
%3 = load <4 x i32>, <4 x i32>* %2
C[i:i+3]
%4 = add <4 x i32> %3, %1
%tmp7 = getelementptr inbounds [16 x i32], [16 x i32]* @C, i64 0, i64 %arg
%5 = bitcast i32* %tmp7 to <4 x i32>*
store <4 x i32> %4, <4 x i32>* %5

```
test1:                                  // @test1
        adrp    x8, A
        add     x8, x8, :lo12:A
        mov     w9, #4
        mov     w10, w9
        mul     x10, x10, x4
        ldr     q1, [x8, x10]          // q1 = A[i:i+3]
        mov     v0.s[0], w0            // v0.s[0] = a
        mov     v0.s[1], w1            // v0.s[1] = b
        mov     v0.s[2], w2            // v0.s[2] = c
        mov     v0.s[3], w3            // v0.s[3] = d
        mul     v0.4s, v1.4s, v0.4s    // v0 = v1 * v0 = A[i:i+3] * <a, b, c, d>
        mov     s2, v0.s[0]            // s2 = v1[0] = e
        mov     s3, v0.s[1]            // s3 = v1[1] = f
        mov     s4, v0.s[2]            // s4 = v1[2] = g
        mov     s5, v0.s[3]            // s5 = v1[3] = h
        fmov    w9, s3
        fmov    w11, s2
        add     w9, w9, w11
        fmov    w11, s4
        add     w9, w9, w11
        fmov    w11, s5
        add     w0, w9, w11
        ret
```

```
test2:
        adrp   x8, A
        add    x8, x8, :lo12:A
        adrp   x9, B
        add    x9, x9, :lo12:B
        adrp   x10, C
        add    x10, x10, :lo12:C
        mov    w11, wzr
        add    x12, x0, #1
        mov    w13, #4
        mov    w14, w13
        mul    x12, x14, x12
        add    x15, x0, #2
        mul    x15, x14, x15
        mul    x14, x14, x0
        ldr    q0, [x8, x12]              // q0 = A[i+1:i+4]
        ldr    q1, [x9, x15]              // q1 = B[i+2:i+5]
        add    v0.4s, v1.4s, v0.4s        // v0 = v1 + v0 = A[i+1:i+4] + B[i+2:i+5]
        str    q0, [x10, x14]             // C[i:i+3] = q0
        mov    w0, w11
        ret
```

Experiment Setup

System & Compilation	Simulation & Evaluation
OS: Ubuntu 20.04.3 LLVM: version 10.0.0 GCC toolchain: version 9.4.0	Gem5 (commit 141cc37c2d) CPU type: `O3_ARM_v7a_3` L1 instruction cache: 32 KB L1 data cache: 64 KB L2 cache: 2 MB

Evaluation Results – Performance

	O1	O1+unroll	O1+unroll+SLP	O2
memcpy	14833	14031	13768	7085
axpy	5011	4056	2628	1995
dotprod	5675	4443	3582	2969
mmm	40158	37575	37212	33945
arithmetic	20073	19752	17934	10102

Table 1. Execution time of 5 benchmark programs in microseconds.

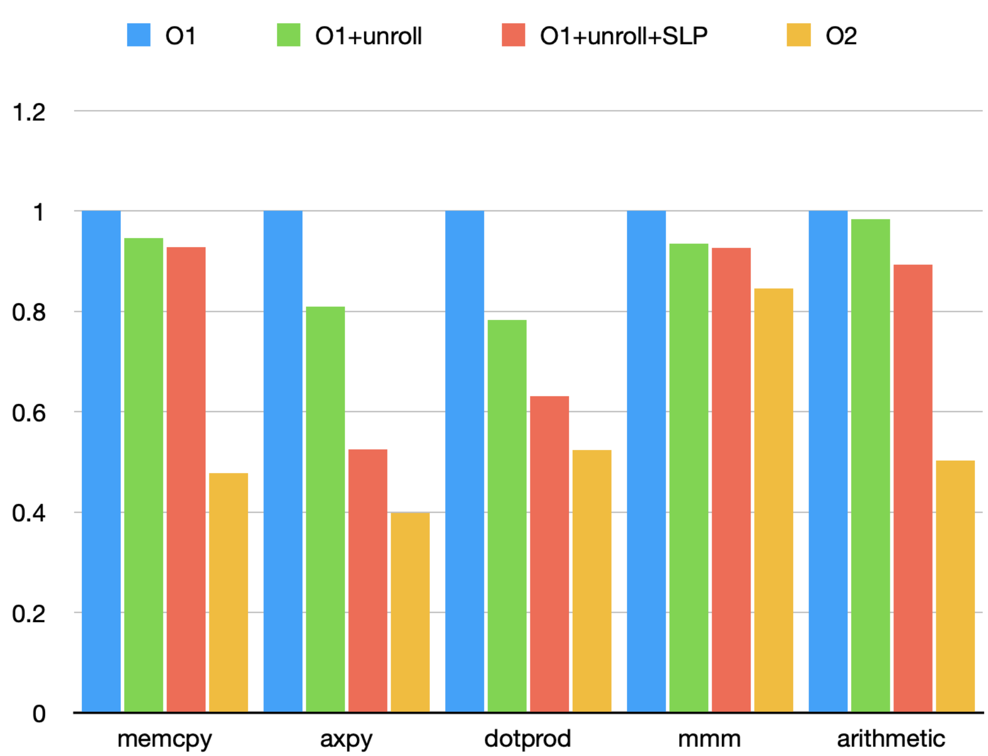


Figure 1. Execution time in different compilation setups, O1 is 1*x*.

Evaluation Results – Code Size

	O1	O1+unroll	O1+unroll+SLP	O2
memcpy	20	64	59	71
axpy	25	43	43	90
dotprod	42	70	59	90
mmm	82	110	108	123
arithmetic	50	131	146	50

Table 2. Instruction counts of computation kernels in 5 benchmark programs.

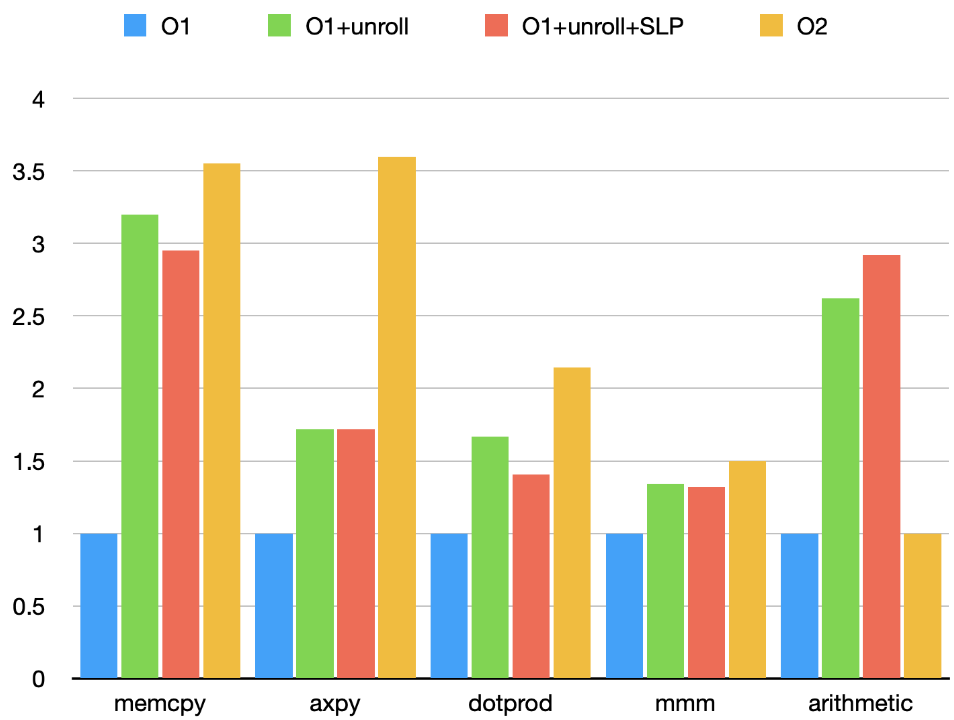


Figure 2. Instruction counts of computation kernels in different compilation setups, O1 is 1*x*.

Conclusion

In this project, we implement the SLP algorithms in LLVM and measure the effectiveness. Our algorithms can automatically exploit SLP within a basic block and pack operations into SIMD instructions. Our results show that basic SLP algorithms improve the runtime performance by 1.31*x* compared to O1 and 1.17*x* compared to O1 with loop unrolling with a smaller program size compared with direct loop unrolling.

References

[1] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. PLDI, 2000.

[2] J. Liu, Y. Zhang, O. Jang, W. Ding, and M. Kandemir. A compiler framework for extracting superword level parallelism. PLDI, 2012.

[3] Auto-vectorization in LLVM. https://llvm.org/docs/Vectorizers.html.