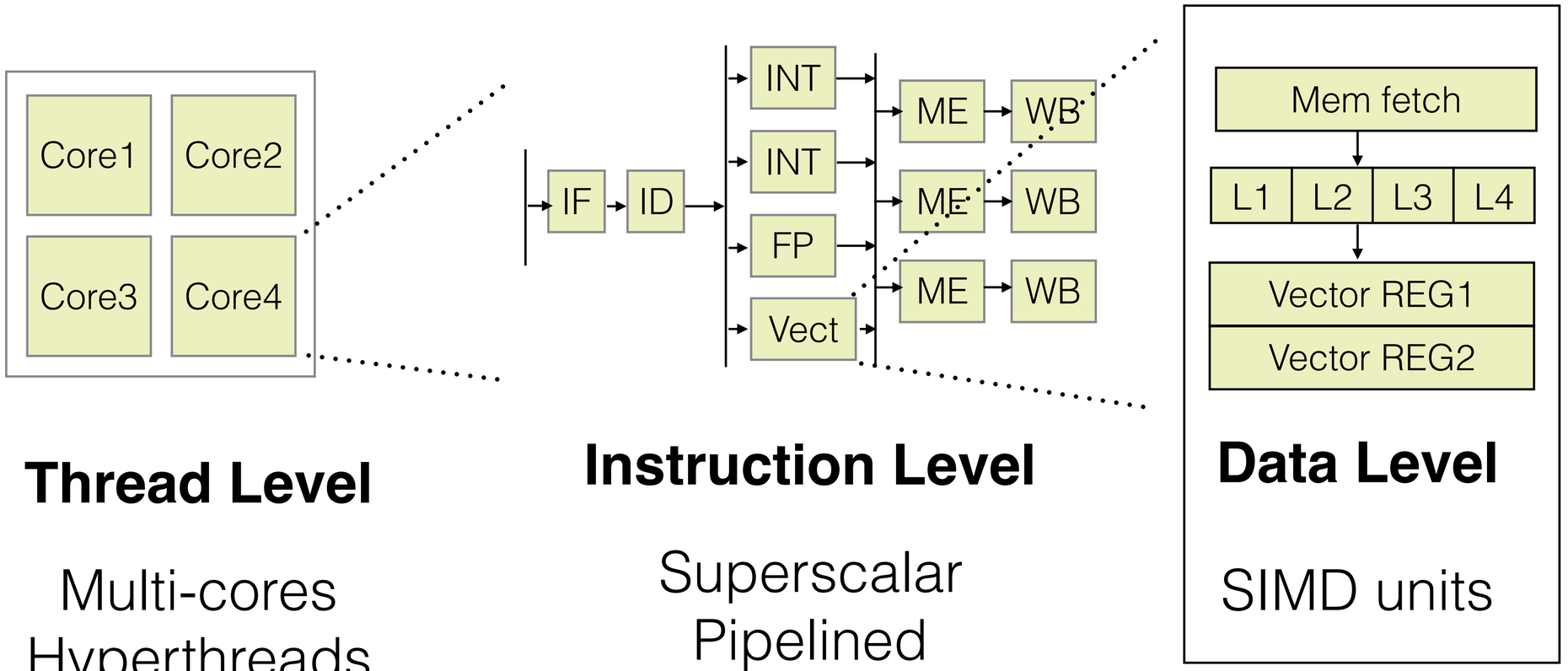
Revec: Program Rejuvenation through Revectorization

Charith Mendis *
Ajay Jain *
Paras Jain
Saman Amarasinghe

* equal contribution

Parallelism in Processors



Thread Level

Multi-cores
Hyperthreads

Instruction Level

Superscalar
Pipelined

Data Level

SIMD units

Exploiting SIMD parallelism

Use compiler auto-vectorization.

```
for (int i = 0; i < N; i++) {
    av[i] = sqrt(bv[i]);
}
```

Portable
Complete?

```
sqrtpd  80(%rdx,%rax), %xmm0
sqrtpd  96(%rdx,%rax), %xmm1
vmovdqu %xmm0, 40(%rdi,%rax)
vmovdqu %xmm1, 56(%rdi,%rax)
```

SSE2 (128 bit)

Portable?
Complete

Hand-vectorization using compiler intrinsics

```
for (int i = 0; i < N/4; i+=4) {
    av[i] = _mm_sqrt_pd(bv[i]);
    av[i+2] = _mm_sqrt_pd(bv[i+2]);
}
```

Exploiting SIMD parallelism

Use compiler auto-vectorization.

```
for (int i = 0; i < N; i++) {
    av[i] = sqrt(bv[i]);
}
```

Portable
Complete?

```
vsqrtpd 80(%rdx,%rax), %zmm0
vsqrtpd 144(%rdx,%rax),%zmm1
vmovupd %zmm0, 40(%rdi,%rax)
vmovupd %zmm1, 104(%rdi,%rax)
```

AVX-512 (512 bit)

Hand-vectorization using compiler intrinsics

Portable?
Complete

```
for (int i = 0; i < N/4; i+=4) {
    av[i] = _mm_sqrt_pd(bv[i]);
    av[i+2] = _mm_sqrt_pd(bv[i+2]);
}
```

fixed to 128 bits

Intel Vector-ISA Generations

32-bit scalar only

64-bit vector (MMX)

1997

128-bit vector (SSE2)

2000

256-bit vector (AVX2)

2011

512-bit vector (AVX512)

2016

Increase in bit-width

Diversity in Instruction Set

Exploiting SIMD parallelism

Use compiler auto-vectorization.

```
for (int i = 0; i < N; i++) {
    av[i] = sqrt(bv[i]);
}
```

Portable
Complete?

```
vsqrtpd 80(%rdx,%rax), %zmm0
vsqrtpd 144(%rdx,%rax),%zmm1
vmovupd %zmm0, 40(%rdi,%rax)
vmovupd %zmm1, 104(%rdi,%rax)
```

Hand-vectorization using compiler intrinsics

```
for (int i = 0; i < N/4; i+=4) {
    av[i] = _mm_sqrt_pd(bv[i]);
    av[i+2] = _mm_sqrt_pd(bv[i+2]);
}
```

Portable?
Complete

AVX-512 (512 bit)

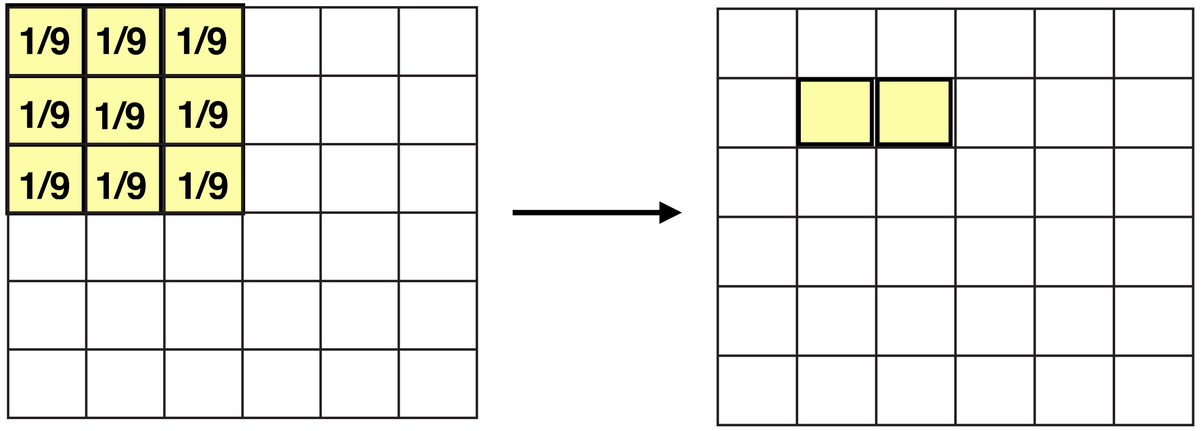
Portable
Complete

Revec

Naive implementation auto-vectorized by compiler

Unoptimized MeanFilter3x3

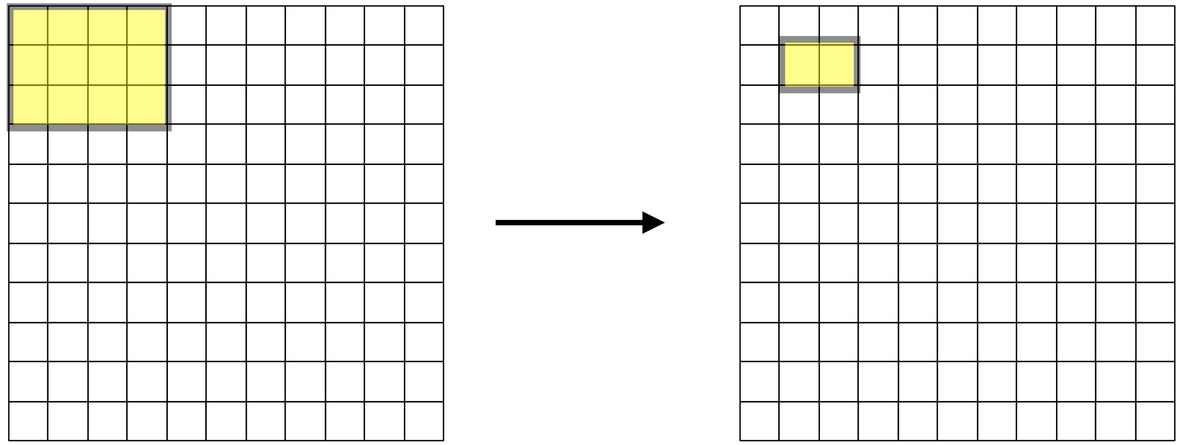
```
for (i = 1; i < H - 1; ++i)
   for (j = 1; j < W - 1; ++j)
          dst[i][j] =
            1/9 * (  in[i-1][j-1]  + in[i-1][j] + in[i-1][j+1]
                  +  in[i][j-1]    + in[i][j]   + in[i][j+1]
                  +  in[i+1][j-1]  + in[i+1][j] + in[i+1][j+1])
```



Naive implementation auto-vectorized by compiler

Unoptimized MeanFilter3x3

```
for (i = 1; i < H - 1; ++i)
   for (j = 1; j < W - 1; ++j)
          dst[i][j] =
           1/9 * (  in[i-1][j-1]  + in[i-1][j] + in[i-1][j+1]
                 +  in[i][j-1]    + in[i][j]   + in[i][j+1]
                 +  in[i+1][j-1]  + in[i+1][j] + in[i+1][j+1])
```



Input Output

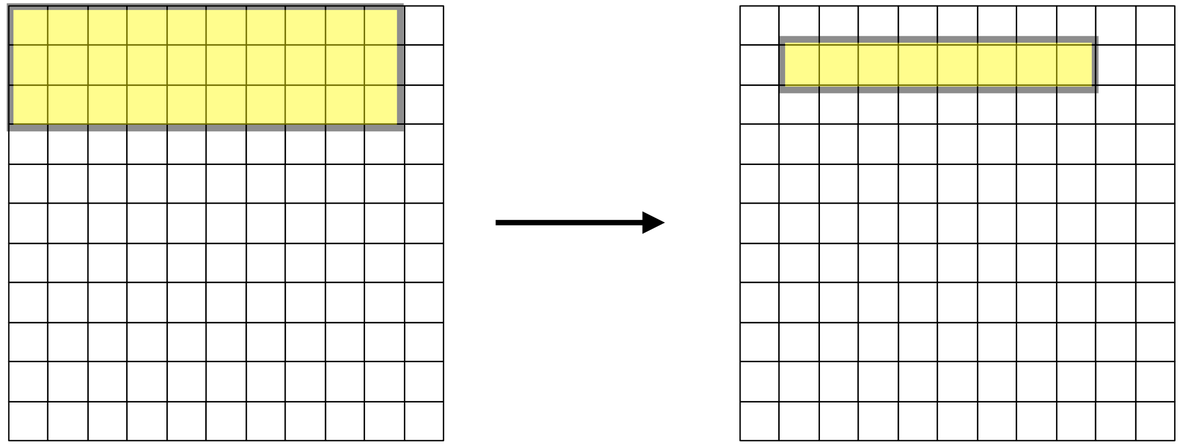
```
for (i = 1; i < H - 1; ++i)
   for (j = 1; j < W - 1; j+=8)
          dst[i][j:j+7] =
           1/9 * (  in[i-1][j-1:j+6]  + in[i-1][j:j+7] + in[i-1][j+1:j+8]
                 +  in[i]  [j-1:j+6]  + in[i]  [j:j+7] + in[i]  [j+1:j+8]
                 +  in[i+1][j-1:j+6]  + in[i+1][j:j+7] + in[i+1][j+1:j+8])
```

Auto-vectorized SSE2 - 128-bit

Naive implementation auto-vectorized by compiler

Unoptimized MeanFilter3x3

```
for (i = 1; i < H - 1; ++i)
   for (j = 1; j < W - 1; ++j)
          dst[i][j] =
           1/9 * (  in[i-1][j-1]  + in[i-1][j] + in[i-1][j+1]
                 +  in[i][j-1]    + in[i][j]   + in[i][j+1]
                 +  in[i+1][j-1]  + in[i+1][j] + in[i+1][j+1])
```



Input

Output

```
for (i = 1; i < H - 1; ++i)
   for (j = 1; j < W - 1; j+=8)
          dst[i][j:j+31] =
           1/9 * (  in[i-1][j-1:j+30]  + in[i-1][j:j+31] + in[i-1][j+1:j+32]
                 +  in[i]  [j-1:j+30]  + in[i]  [j:j+31] + in[i]  [j+1:j+32]
                 +  in[i+1][j-1:j+30]  + in[i+1][j:j+31] + in[i+1][j+1:j+32])
```

Auto-vectorized AVX-512 - 512-bit

Optimized Implementation hand-vectorized by programmer

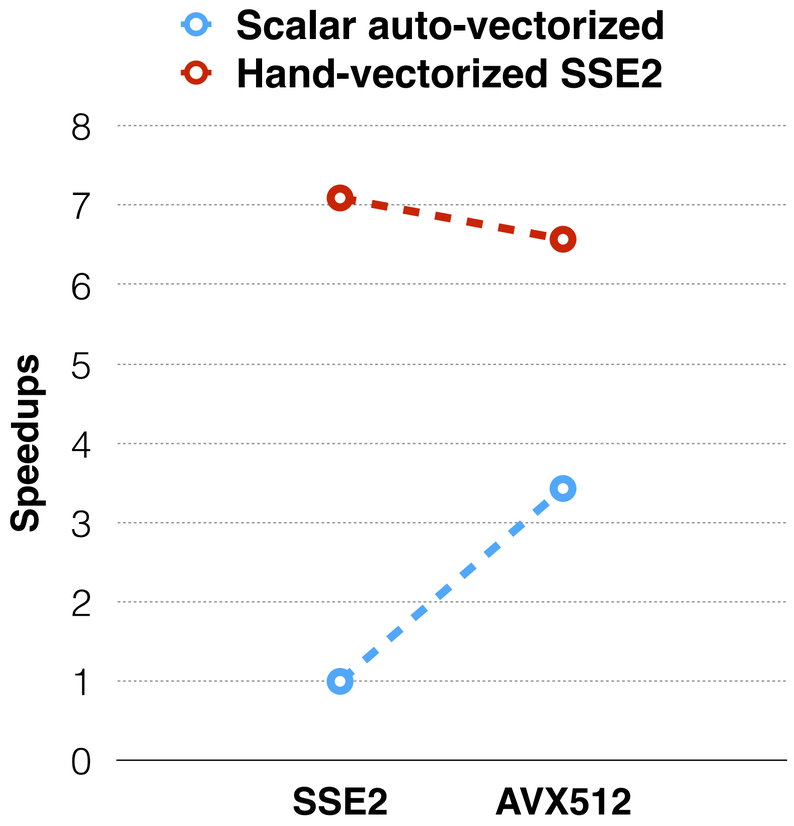
Optimized MeanFilter3x3

```
#define A 8
#define F (1 << 16)/9
__m128i div9 = _mm_set_epi16(F,F,F,F,F,F,F,F);
uint16_t colsum[3 * W];
__m128i * buf1 = &colsum[0 * W];
__m128i * buf2 = &colsum[1 * W];
__m128i * buf3 = &colsum[2 * W];

//code to compute column sums for first two rows in buf1, buf2

for (i = 2; i < H; ++i){
   for (j = 1; j < W - 1; j += A){
     a0 = _mm_loadu_si128(in[i][j-1]);
     a1 = _mm_loadu_si128(in[i][j]);
     a2 = _mm_loadu_si128(in[i][j+1]);
     buf3[j/A] = _mm_add_epi16(a0, _mm_add_epi16(a1,a2)));
       dst[i - 1][j] = _mm_mulhi_epu16(div9,
            _mm_add_epi16(buf1[j/A]
         _mm_add_epi16(buf2[j/A],buf3[j/A])));}
     //swap buffer colsums for next iteration
      __m128i * temp = buf1;
      buf1 = buf2;
      buf2 = buf3;
      buf3 = temp;
}
```

- Rotating buffers
- Hand-vectorized using compiler intrinsics targeting a 128-bit SSE2 machine

`_mm_loadu_si128` `_mm_add_epi16`

Hand-vectorization inhibits performance portability

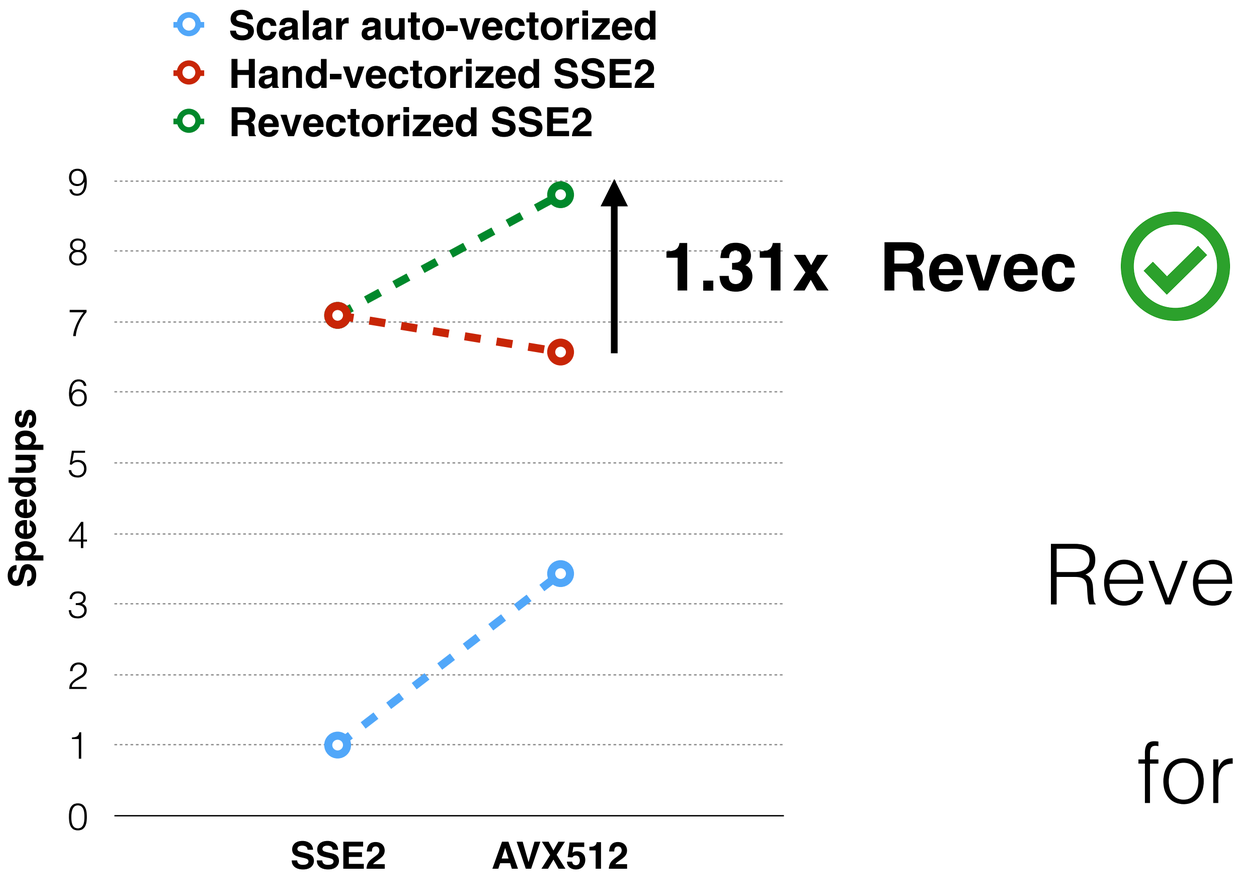


- Auto-vectorizers do not handle already vectorized code
- Hand-vectorized code does not utilize features of newer vector ISA

- Retarget hand-vectorized codes to use new vector instruction sets when available

Revectorization

- We built a compiler pass **Revec** to perform revectorization



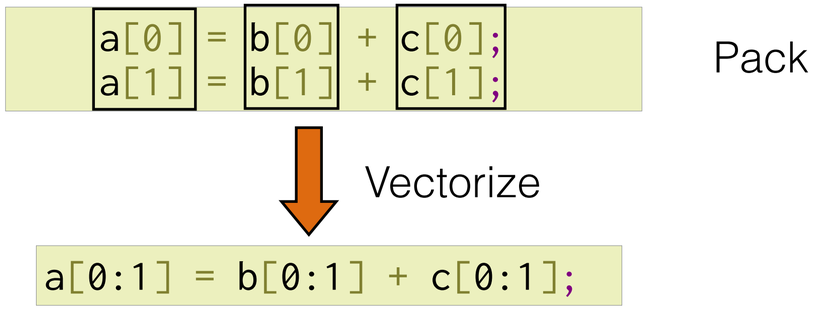
Revec reinstates performance portability
for hand-vectorized code

Revec

- Revectorizes code **transparently** — implemented as a regular compiler transformation in LLVM
- Based on SLP vectorization
- Enables performance portability for hand-vectorized code

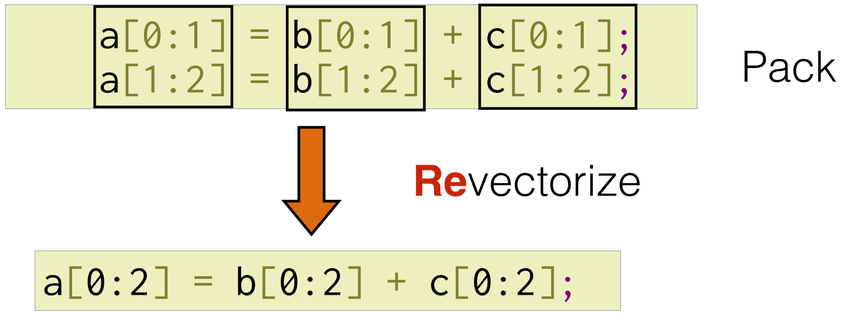
SLP Vectorization

Isomorphic and **independent** statements
can be vectorized.



Revec: Revectorization

Isomorphic and **independent vectorized** statements
can be **re**vectorized.



However, need to adapt to handle

Vector Shuffles

Opaque Intrinsics

Hand-vectorized example

```
__m128i zeros = _mm_set_epi16(0,0,0,0,0,0,0,0);
__m128i cons = _mm_set_epi32(127,127,127,127);

for(int i = 0; i < H * W; i+= 8){
  __m128i inval = _mm_loadu_si128(in[i]);
  __m128i lo = _mm_unpacklo_epi16(inval,zeros);
  __m128i hi = _mm_unpackhi_epi16(inval,zeros);
  __m128i lo_plus = _mm_add_epi32(lo,cons);
  __m128i hi_plus = _mm_add_epi32(hi,cons);
  __m128i final = _mm_packus_epi32(lo_plus, hi_plus);
  _mm_storeu_si128(out[i],final);
}
```

Hand-vectorized code (128-bit)


```
%1 = load <8 x i16>, <8 x i16>* %in
%2 = shufflevector <8 x i16> %1, const_vec_1, mask_1
%3 = shufflevector <8 x i16> %1, const_vec_2, mask_2
%4 = bitcast <8 x i16> %2 to <4 x i32>
%5 = add <4 x i32> %4, <127, 127, 127, 127>
%6 = bitcast <8 x i16> %3 to <4 x i32>
%7 = add <4 x i32> %6, <127, 127, 127, 127>
%8 = call <8 x i16> @llvm.x86.sse41.packusdw(%5, %7)
store <8 x i16> %8, <8 x i16>* %out
```

LLVM IR

Revec revectorizes LLVM IR

Vector IR

Vector Shuffles

Opaque Intrinsics

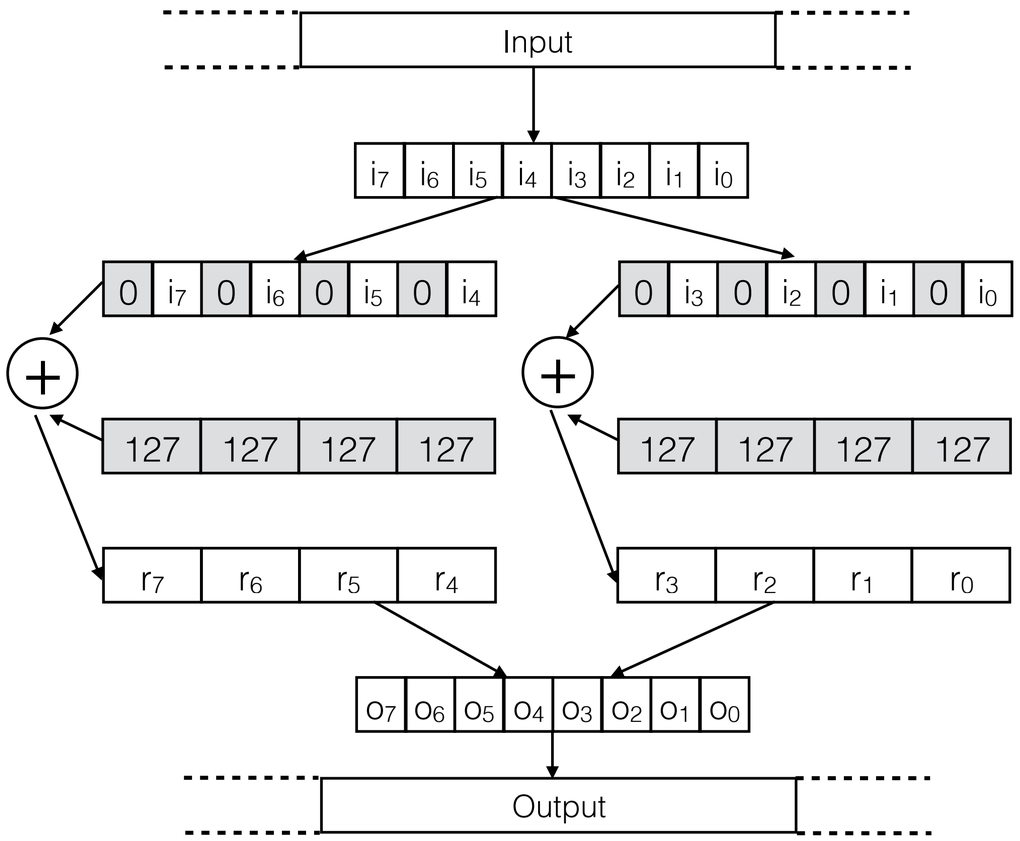
Hand-vectorized example

```
__m128i zeros = _mm_set_epi16(0,0,0,0,0,0,0,0);
__m128i cons = _mm_set_epi32(127,127,127,127);

for(int i = 0; i < H * W; i+= 8){
   __m128i inval = _mm_loadu_si128(in[i]);
   __m128i lo = _mm_unpacklo_epi16(inval,zeros);
   __m128i hi = _mm_unpackhi_epi16(inval,zeros);
   __m128i lo_plus = _mm_add_epi32(lo,cons);
   __m128i hi_plus = _mm_add_epi32(hi,cons);
   __m128i final = _mm_packus_epi32(lo_plus, hi_plus);
   _mm_storeu_si128(out[i],final);
}
```

Hand-vectorized code (128-bit)

```
%1 = load <8 x i16>, <8 x i16>* %in
%2 = shufflevector <8 x i16> %1, const_vec_1, mask_1
%3 = shufflevector <8 x i16> %1, const_vec_2, mask_2
%4 = bitcast <8 x i16> %2 to <4 x i32>
%5 = add <4 x i32> %4, <127, 127, 127, 127>
%6 = bitcast <8 x i16> %3 to <4 x i32>
%7 = add <4 x i32> %6, <127, 127, 127, 127>
%8 = call <8 x i16> @llvm.x86.sse41.packusdw(%5, %7)
store <8 x i16> %8, <8 x i16>* %out
```

LLVM IR

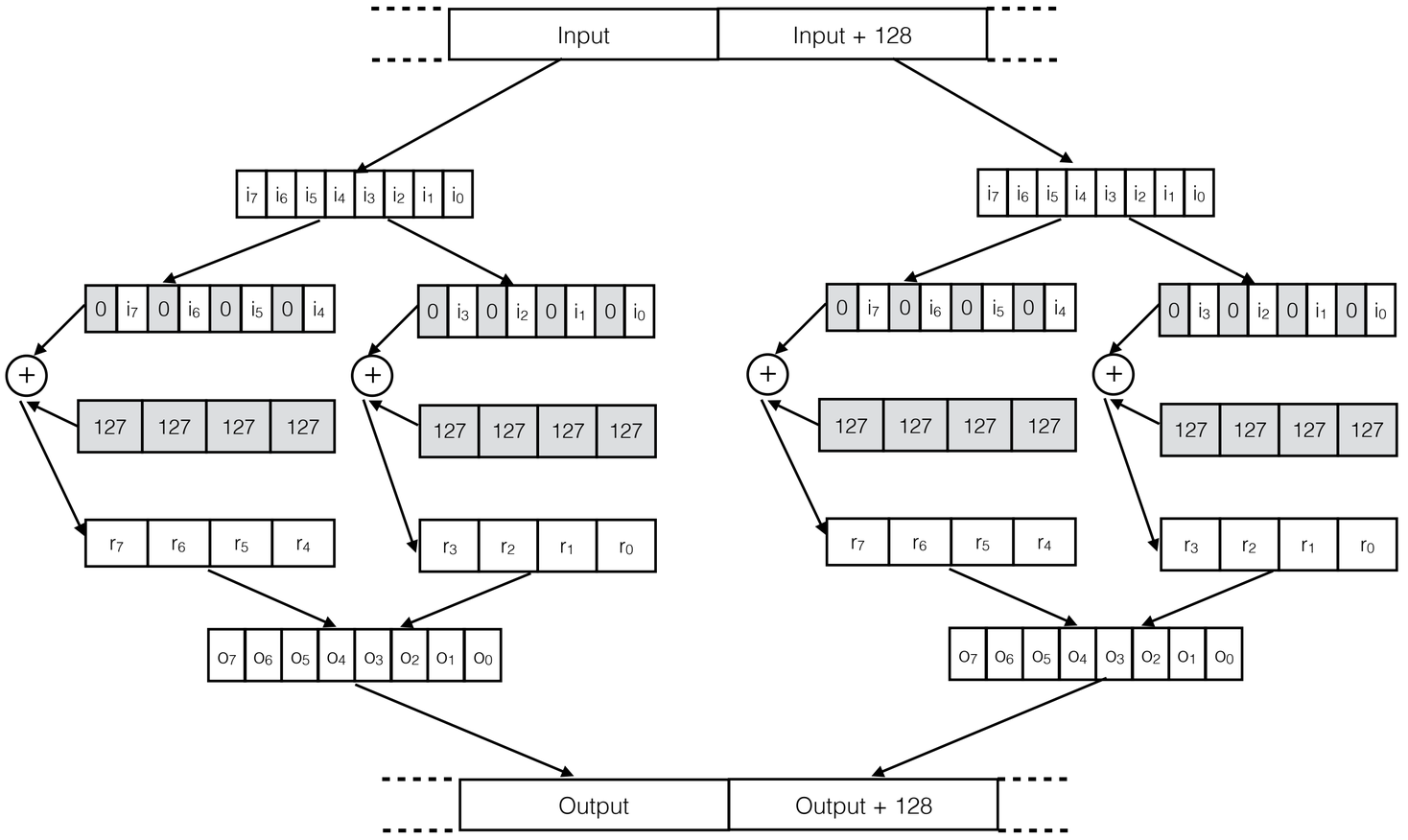


Loop Unrolling

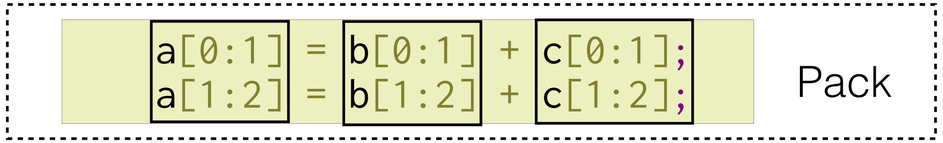
```
%1 = load <8 x i16>, <8 x i16>* %in
%2 = shufflevector <8 x i16> %1, const_vec_1, mask_1
%3 = shufflevector <8 x i16> %1, const_vec_2, mask_2
%4 = bitcast <8 x i16> %2 to <4 x i32>
%5 = add <4 x i32> %4, <127, 127, 127, 127>
%6 = bitcast <8 x i16> %3 to <4 x i32>
%7 = add <4 x i32> %6, <127, 127, 127, 127>
%8 = call <8 x i16> @llvm.x86.sse41.packusdw(%5, %7)
store <8 x i16> %8, <8 x i16>* %out
```

```
%9 = load <8 x i16>, <8 x i16>* %in_1
%10 = shufflevector <8 x i16> %9, const_vec_3, mask_3
%11 = shufflevector <8 x i16> %9, const_vec_4, mask_4
%12 = bitcast <8 x i16> %10 to <4 x i32>
%13 = add <4 x i32> %12, <127, 127, 127, 127>
%14 = bitcast <8 x i16> %11 to <4 x i32>
%15 = add <4 x i32> %14, <127, 127, 127, 127>
%16 = call <8 x i16> @llvm.x86.sse41.packusdw(%13,%15)
store <8 x i16> %16, <8 x i16>* %out_1
```

Targeting a 256-vector machine
Unroll once



Revec Transformation



1. Start at root packs
 (adjacent stores, reduction roots)

2. Revectorize the pack

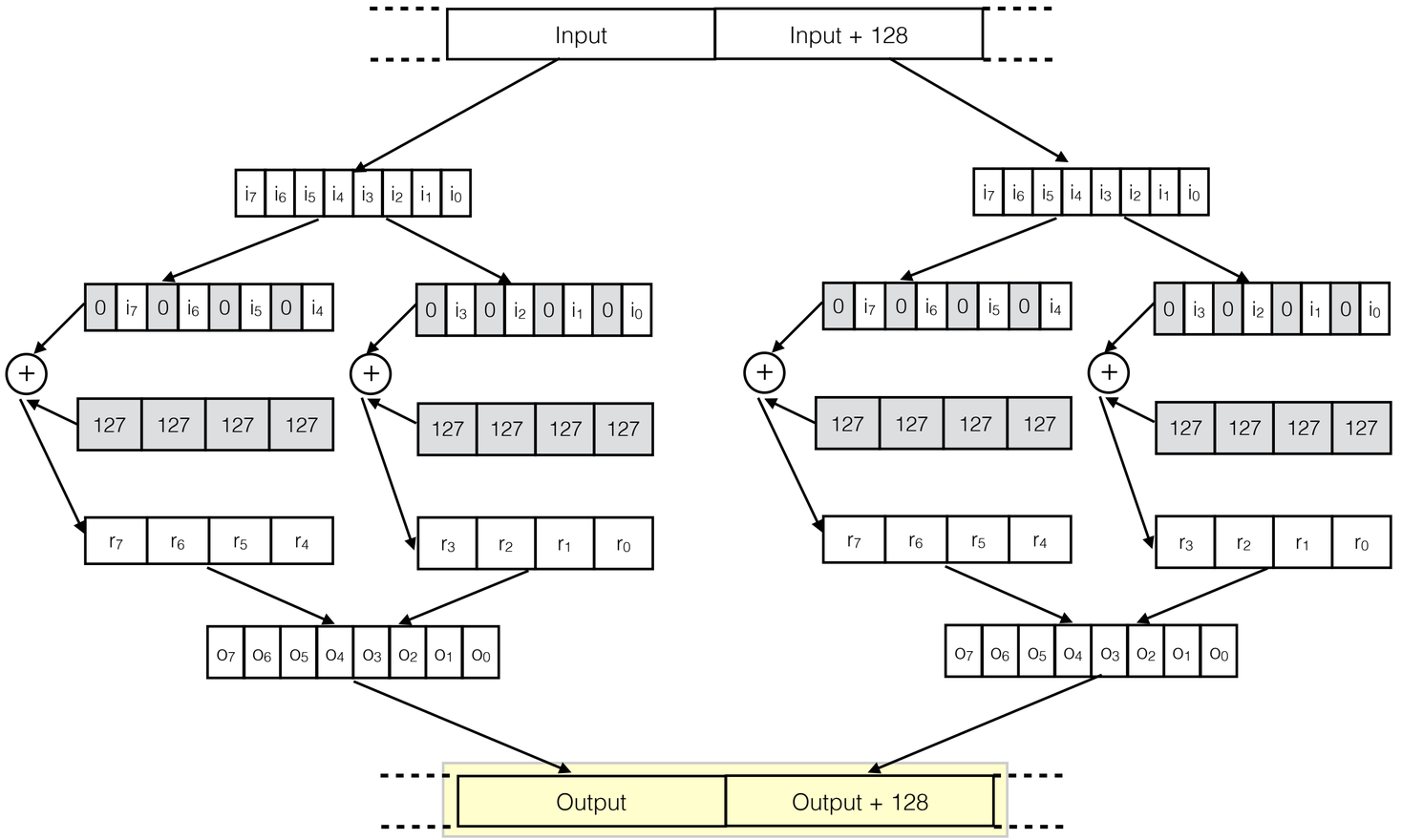
3. Recursively traverse the use-def chain

4. Stop when packs are not revectorizable

Loop-aware SLP in GCC [Rosen, et al. 2007]

Revec Transformation

```
%1 = load <8 x i16>, <8 x i16>* %in
%2 = shufflevector <8 x i16> %1, const_vec_1, mask_1
%3 = shufflevector <8 x i16> %1, const_vec_2, mask_2
%4 = bitcast <8 x i16> %2 to <4 x i32>
%5 = add <4 x i32> %4, <127, 127, 127, 127>
%6 = bitcast <8 x i16> %3 to <4 x i32>
%7 = add <4 x i32> %6, <127, 127, 127, 127>
%8 = call <8 x i16> @llvm.x86.sse41.packusdw(%5, %7)
store <8 x i16> %8, <8 x i16>* %out


%9 = load <8 x i16>, <8 x i16>* %in_1
%10 = shufflevector <8 x i16> %9, const_vec_3, mask_3
%11 = shufflevector <8 x i16> %9, const_vec_4, mask_4
%12 = bitcast <8 x i16> %10 to <4 x i32>
%13 = add <4 x i32> %12, <127, 127, 127, 127>
%14 = bitcast <8 x i16> %11 to <4 x i32>
%15 = add <4 x i32> %14, <127, 127, 127, 127>
%16 = call <8 x i16> @llvm.x86.sse41.packusdw(%13,%15)
store <8 x i16> %16, <8 x i16>* %out_1
```



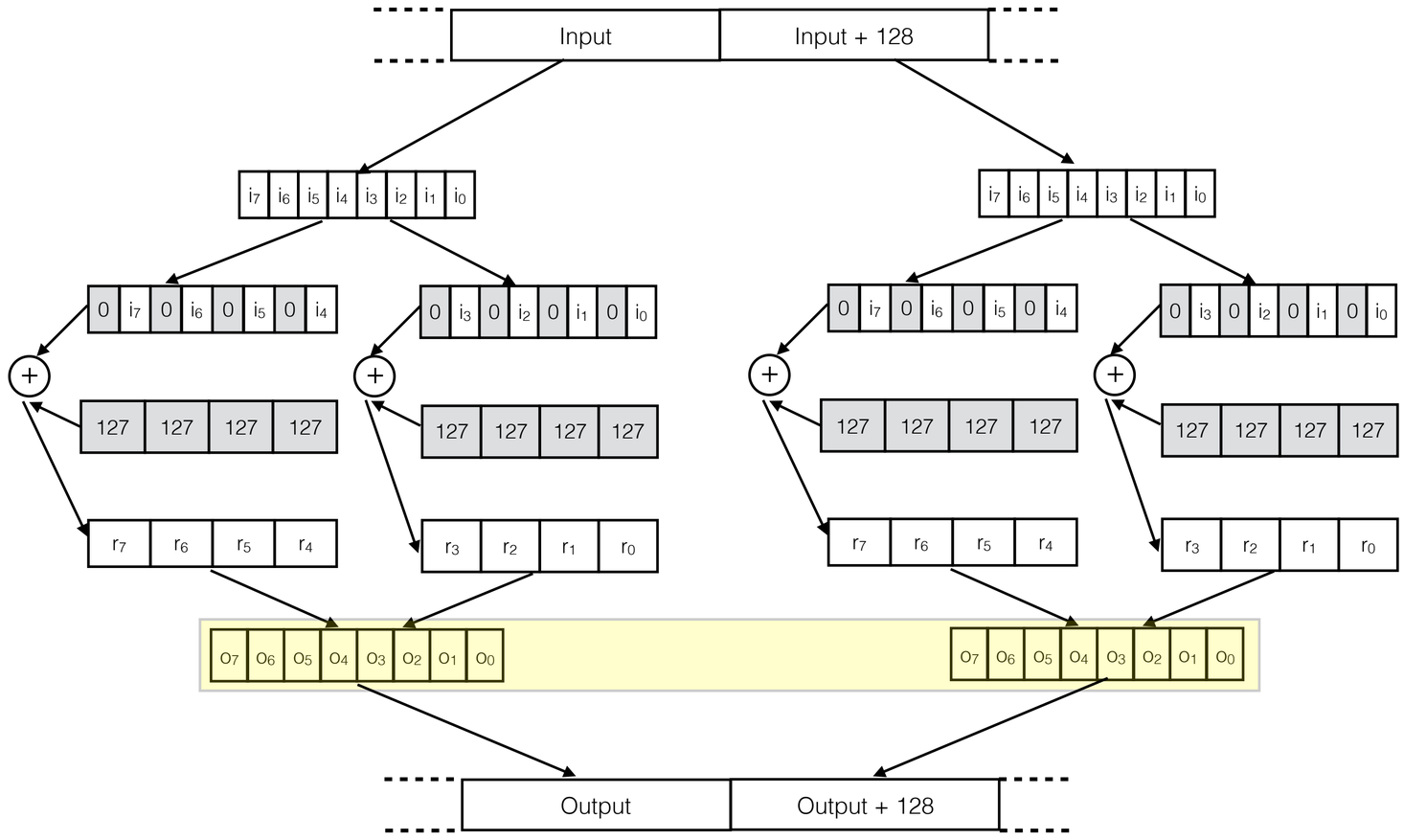
Revec Transformation

```
%1 = load <8 x i16>, <8 x i16>* %in
%2 = shufflevector <8 x i16> %1, const_vec_1, mask_1
%3 = shufflevector <8 x i16> %1, const_vec_2, mask_2
%4 = bitcast <8 x i16> %2 to <4 x i32>
%5 = add <4 x i32> %4, <127, 127, 127, 127>
%6 = bitcast <8 x i16> %3 to <4 x i32>
%7 = add <4 x i32> %6, <127, 127, 127, 127>
%8 = call <8 x i16> @llvm.x86.sse41.packusdw(%5, %7)
store <8 x i16> %8, <8 x i16>* %out


%9 = load <8 x i16>, <8 x i16>* %in_1
%10 = shufflevector <8 x i16> %9, const_vec_3, mask_3
%11 = shufflevector <8 x i16> %9, const_vec_4, mask_4
%12 = bitcast <8 x i16> %10 to <4 x i32>
%13 = add <4 x i32> %12, <127, 127, 127, 127>
%14 = bitcast <8 x i16> %11 to <4 x i32>
%15 = add <4 x i32> %14, <127, 127, 127, 127>
%16 = call <8 x i16> @llvm.x86.sse41.packusdw(%13,%15)
store <8 x i16> %16, <8 x i16>* %out_1
```

Opaque Intrinsics

```
store <16 x i16> ??, <16 x i16>* %out
```

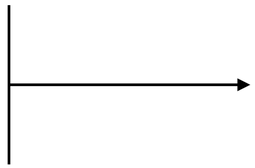
Revectorized code



Finding Intrinsic Equivalences

Does there exist a matching 256-bit intrinsic?

```
%8 = call <8 x i16> @llvm.x86.sse41.packusdw(%5, %7)
```



```
%16 = call <8 x i16> @llvm.x86.sse41.packusdw(%13,%15)
```

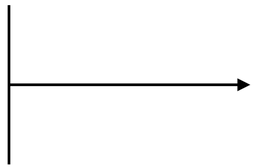
Opaque Intrinsics

Finding Intrinsic Equivalences

Does there exist a matching 256-bit intrinsic?

```
%8 = call <8 x i16> @llvm.x86.sse41.packusdw(%5, %7)
```

```
%16 = call <8 x i16> @llvm.x86.sse41.packusdw(%13,%15)
```



Populate a dictionary of equivalences through enumerative search (offline)

- K:1 equivalences
- Random and corner case testing

Compile-time - dictionary look-up

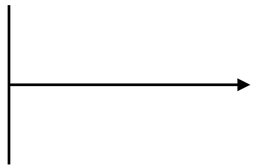
Opaque Intrinsics

Finding Intrinsic Equivalences

Does there exist a matching 256-bit intrinsic?

```
%8 = call <8 x i16> @llvm.x86.sse41.packusdw(%5, %7)
```

```
%16 = call <8 x i16> @llvm.x86.sse41.packusdw(%13,%15)
```

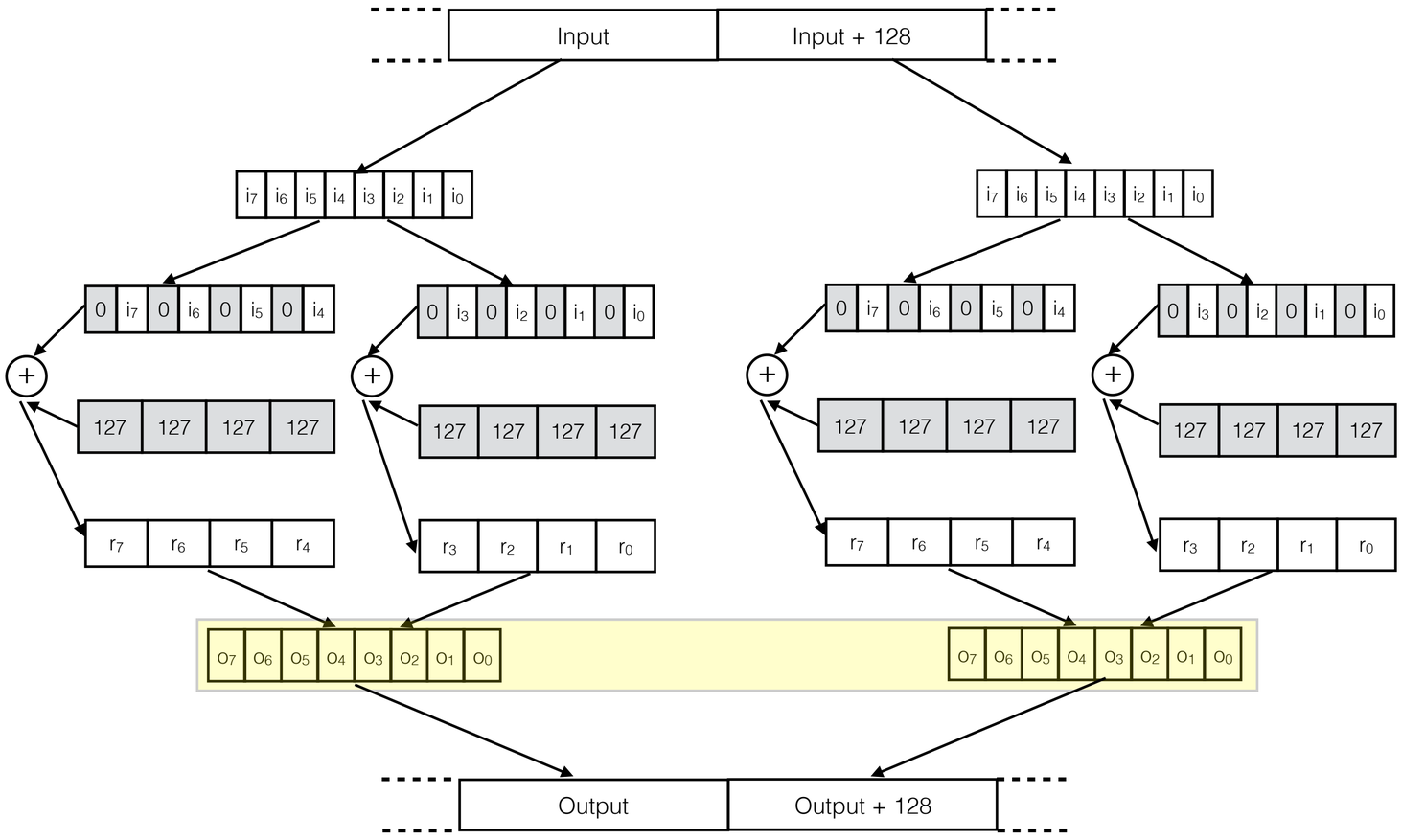


```
call <16 x i16> @llvm.x86.avx2.packusdw(??,??)
```

Opaque Intrinsics

Revec Transformation

```
%1 = load <8 x i16>, <8 x i16>* %in
%2 = shufflevector <8 x i16> %1, const_vec_1, mask_1
%3 = shufflevector <8 x i16> %1, const_vec_2, mask_2
%4 = bitcast <8 x i16> %2 to <4 x i32>
%5 = add <4 x i32> %4, <127, 127, 127, 127>
%6 = bitcast <8 x i16> %3 to <4 x i32>
%7 = add <4 x i32> %6, <127, 127, 127, 127>
%8 = call <8 x i16> @llvm.x86.sse41.packusdw(%5, %7)
store <8 x i16> %8, <8 x i16>* %out


%9 = load <8 x i16>, <8 x i16>* %in_1
%10 = shufflevector <8 x i16> %9, const_vec_3, mask_3
%11 = shufflevector <8 x i16> %9, const_vec_4, mask_4
%12 = bitcast <8 x i16> %10 to <4 x i32>
%13 = add <4 x i32> %12, <127, 127, 127, 127>
%14 = bitcast <8 x i16> %11 to <4 x i32>
%15 = add <4 x i32> %14, <127, 127, 127, 127>
%16 = call <8 x i16> @llvm.x86.sse41.packusdw(%13,%15)
store <8 x i16> %16, <8 x i16>* %out_1
```

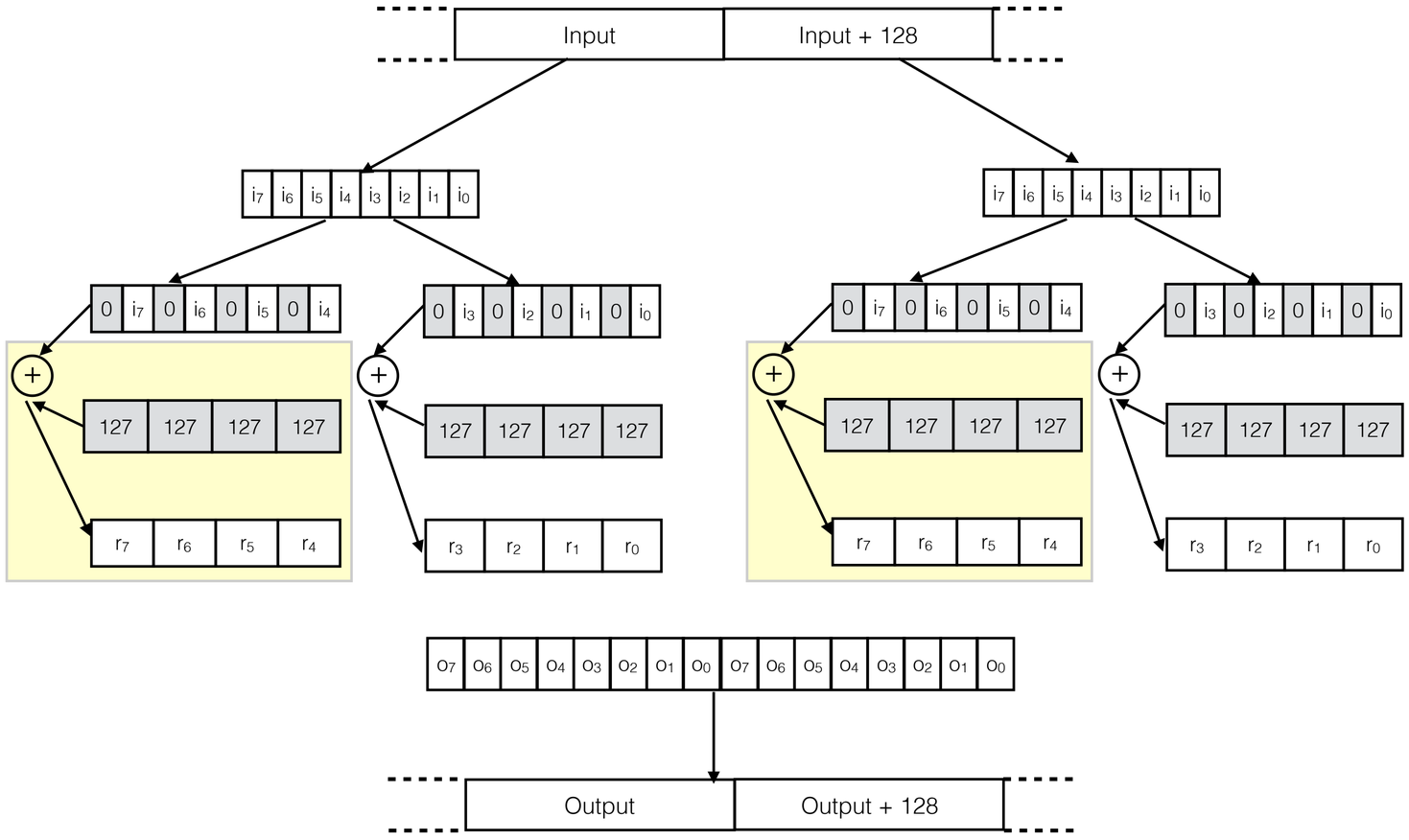



```
store <16 x i16> ??, <16 x i16>* %out
```

Revectorized code

Revec Transformation

```
%1 = load <8 x i16>, <8 x i16>* %in
%2 = shufflevector <8 x i16> %1, const_vec_1, mask_1
%3 = shufflevector <8 x i16> %1, const_vec_2, mask_2
%4 = bitcast <8 x i16> %2 to <4 x i32>
%5 = add <4 x i32> %4, <127, 127, 127, 127>
%6 = bitcast <8 x i16> %3 to <4 x i32>
%7 = add <4 x i32> %6, <127, 127, 127, 127>
%8 = call <8 x i16> @llvm.x86.sse41.packusdw(%5, %7)
store <8 x i16> %8, <8 x i16>* %out


%9 = load <8 x i16>, <8 x i16>* %in_1
%10 = shufflevector <8 x i16> %9, const_vec_3, mask_3
%11 = shufflevector <8 x i16> %9, const_vec_4, mask_4
%12 = bitcast <8 x i16> %10 to <4 x i32>
%13 = add <4 x i32> %12, <127, 127, 127, 127>
%14 = bitcast <8 x i16> %11 to <4 x i32>
%15 = add <4 x i32> %14, <127, 127, 127, 127>
%16 = call <8 x i16> @llvm.x86.sse41.packusdw(%13,%15)
store <8 x i16> %16, <8 x i16>* %out_1
```

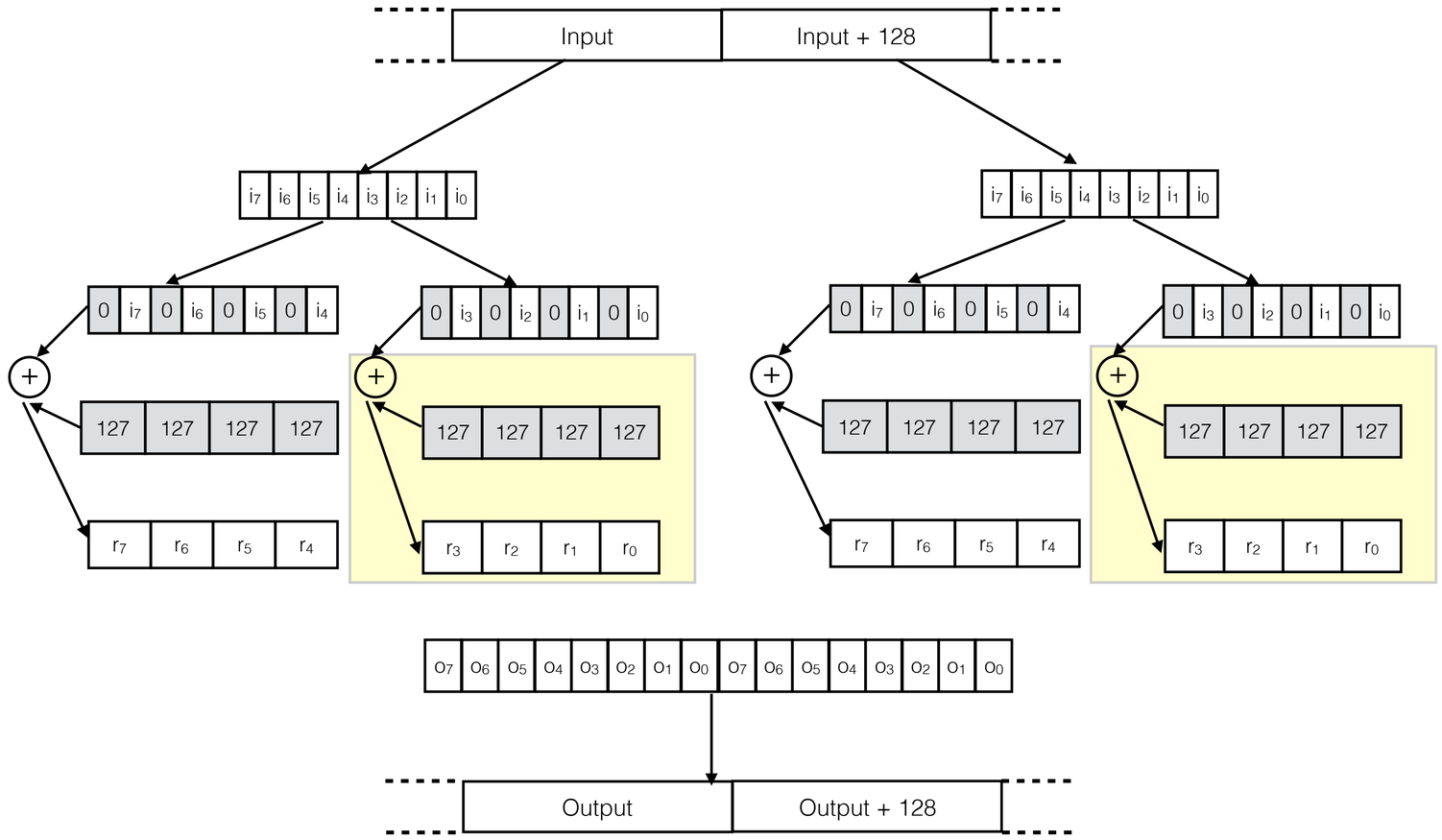



```
%8a = call <16 x i16> @llvm.x86.avx2.packusdw(??,??)
store <16 x i16> %8a, <16 x i16>* %out
```

Revectorized code

Revec Transformation

```
%1 = load <8 x i16>, <8 x i16>* %in
%2 = shufflevector <8 x i16> %1, const_vec_1, mask_1
%3 = shufflevector <8 x i16> %1, const_vec_2, mask_2
%4 = bitcast <8 x i16> %2 to <4 x i32>
%5 = add <4 x i32> %4, <127, 127, 127, 127>
%6 = bitcast <8 x i16> %3 to <4 x i32>
%7 = add <4 x i32> %6, <127, 127, 127, 127>
%8 = call <8 x i16> @llvm.x86.sse41.packusdw(%5, %7)
store <8 x i16> %8, <8 x i16>* %out


%9 = load <8 x i16>, <8 x i16>* %in_1
%10 = shufflevector <8 x i16> %9, const_vec_3, mask_3
%11 = shufflevector <8 x i16> %9, const_vec_4, mask_4
%12 = bitcast <8 x i16> %10 to <4 x i32>
%13 = add <4 x i32> %12, <127, 127, 127, 127>
%14 = bitcast <8 x i16> %11 to <4 x i32>
%15 = add <4 x i32> %14, <127, 127, 127, 127>
%16 = call <8 x i16> @llvm.x86.sse41.packusdw(%13,%15)
store <8 x i16> %16, <8 x i16>* %out_1
```

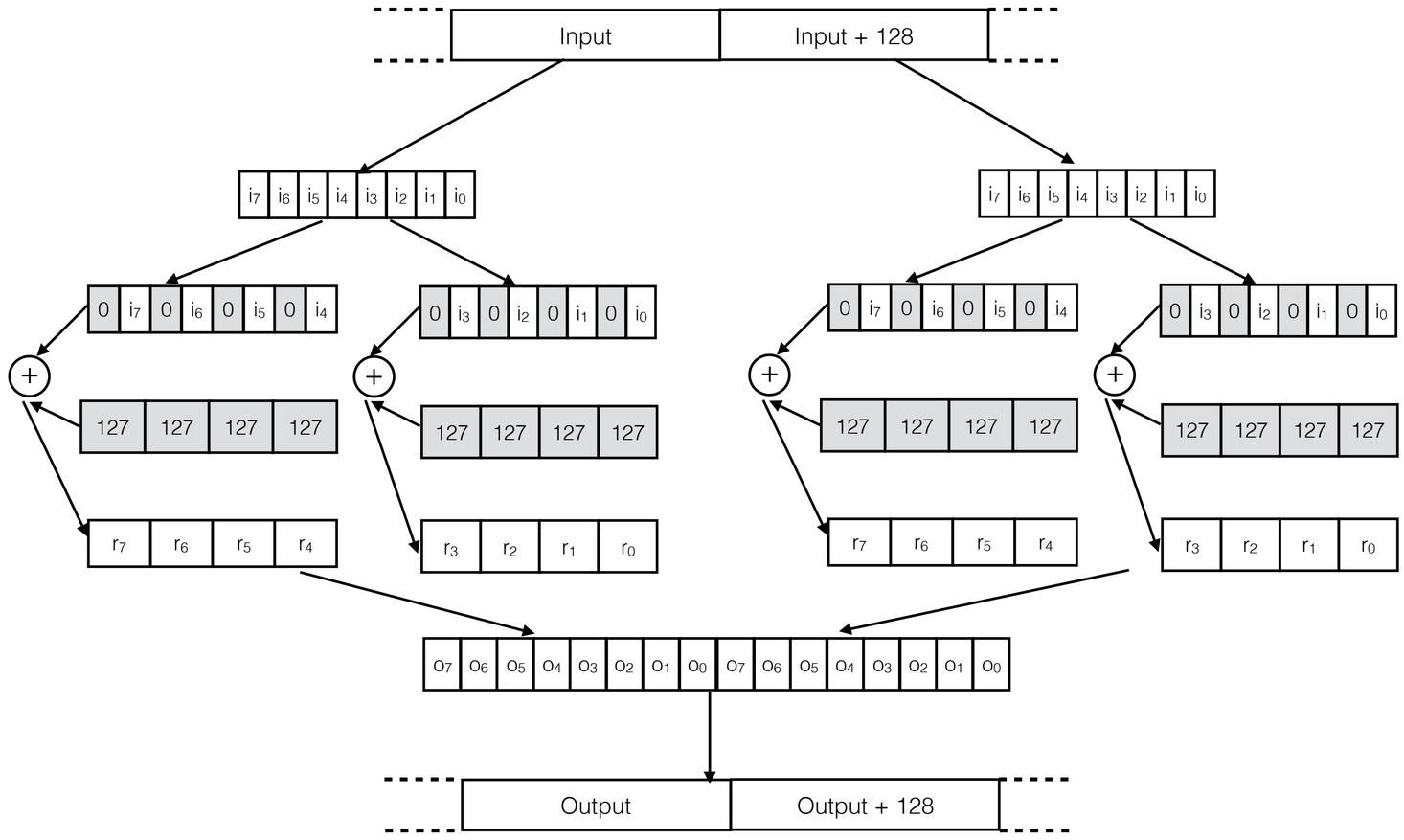



```
%8a = call <16 x i16> @llvm.x86.avx2.packusdw(??,??)
store <16 x i16> %8a, <16 x i16>* %out
```

Revectorized code

Revec Transformation

```
%1 = load <8 x i16>, <8 x i16>* %in
%2 = shufflevector <8 x i16> %1, const_vec_1, mask_1
%3 = shufflevector <8 x i16> %1, const_vec_2, mask_2
%4 = bitcast <8 x i16> %2 to <4 x i32>
%5 = add <4 x i32> %4, <127, 127, 127, 127>
%6 = bitcast <8 x i16> %3 to <4 x i32>
%7 = add <4 x i32> %6, <127, 127, 127, 127>
%8 = call <8 x i16> @llvm.x86.sse41.packusdw(%5, %7)
store <8 x i16> %8, <8 x i16>* %out


%9 = load <8 x i16>, <8 x i16>* %in_1
%10 = shufflevector <8 x i16> %9, const_vec_3, mask_3
%11 = shufflevector <8 x i16> %9, const_vec_4, mask_4
%12 = bitcast <8 x i16> %10 to <4 x i32>
%13 = add <4 x i32> %12, <127, 127, 127, 127>
%14 = bitcast <8 x i16> %11 to <4 x i32>
%15 = add <4 x i32> %14, <127, 127, 127, 127>
%16 = call <8 x i16> @llvm.x86.sse41.packusdw(%13,%15)
store <8 x i16> %16, <8 x i16>* %out_1
```




```
%8a = call <16 x i16> @llvm.x86.avx2.packusdw(??,??)
store <16 x i16> %8a, <16 x i16>* %out
```

Revectorized code

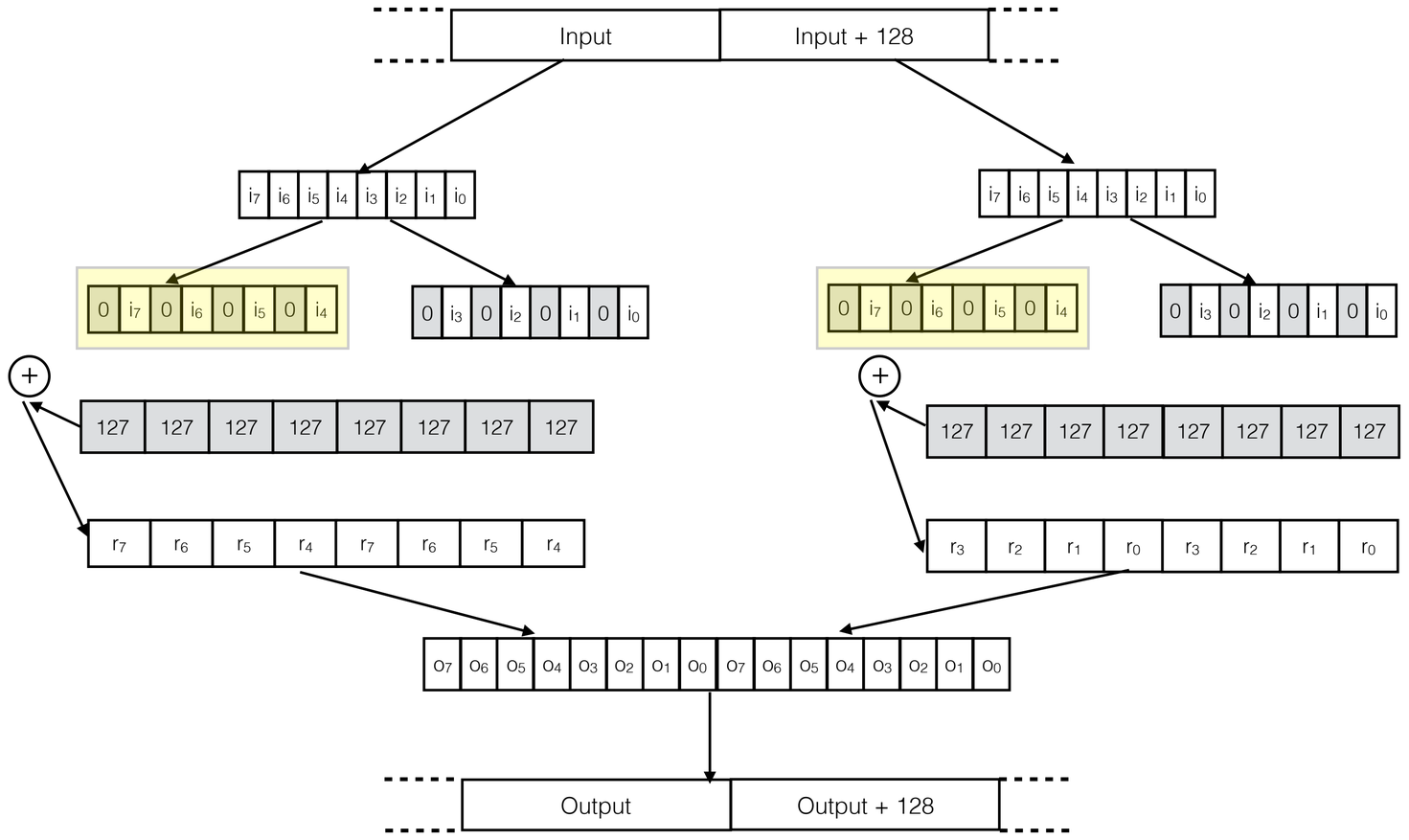
Revec Transformation

```
%1 = load <8 x i16>, <8 x i16>* %in
%2 = shufflevector <8 x i16> %1, const_vec_1, mask_1
%3 = shufflevector <8 x i16> %1, const_vec_2, mask_2
%4 = bitcast <8 x i16> %2 to <4 x i32>
%5 = add <4 x i32> %4, <127, 127, 127, 127>
%6 = bitcast <8 x i16> %3 to <4 x i32>
%7 = add <4 x i32> %6, <127, 127, 127, 127>
%8 = call <8 x i16> @llvm.x86.sse41.packusdw(%5, %7)
store <8 x i16> %8, <8 x i16>* %out


%9 = load <8 x i16>, <8 x i16>* %in_1
%10 = shufflevector <8 x i16> %9, const_vec_3, mask_3
%11 = shufflevector <8 x i16> %9, const_vec_4, mask_4
%12 = bitcast <8 x i16> %10 to <4 x i32>
%13 = add <4 x i32> %12, <127, 127, 127, 127>
%14 = bitcast <8 x i16> %11 to <4 x i32>
%15 = add <4 x i32> %14, <127, 127, 127, 127>
%16 = call <8 x i16> @llvm.x86.sse41.packusdw(%13,%15)
store <8 x i16> %16, <8 x i16>* %out_1
```

Vector Shuffles

```
%4a = bitcast <16 x i16> ?? to <8 x i32>
%5a = add <8 x i32> %4a, <127,127,127,127,127,127,127,127>
%6a = bitcast <16 x i16> ?? to <8 x i32>
%7a = add <8 x i32> %6a, <127,127,127,127,127,127,127,127>
%8a = call <16 x i16> @llvm.x86.avx2.packusdw(%5a,%7a)
store <16 x i16> %8a, <16 x i16>* %out
```

Revectorized code



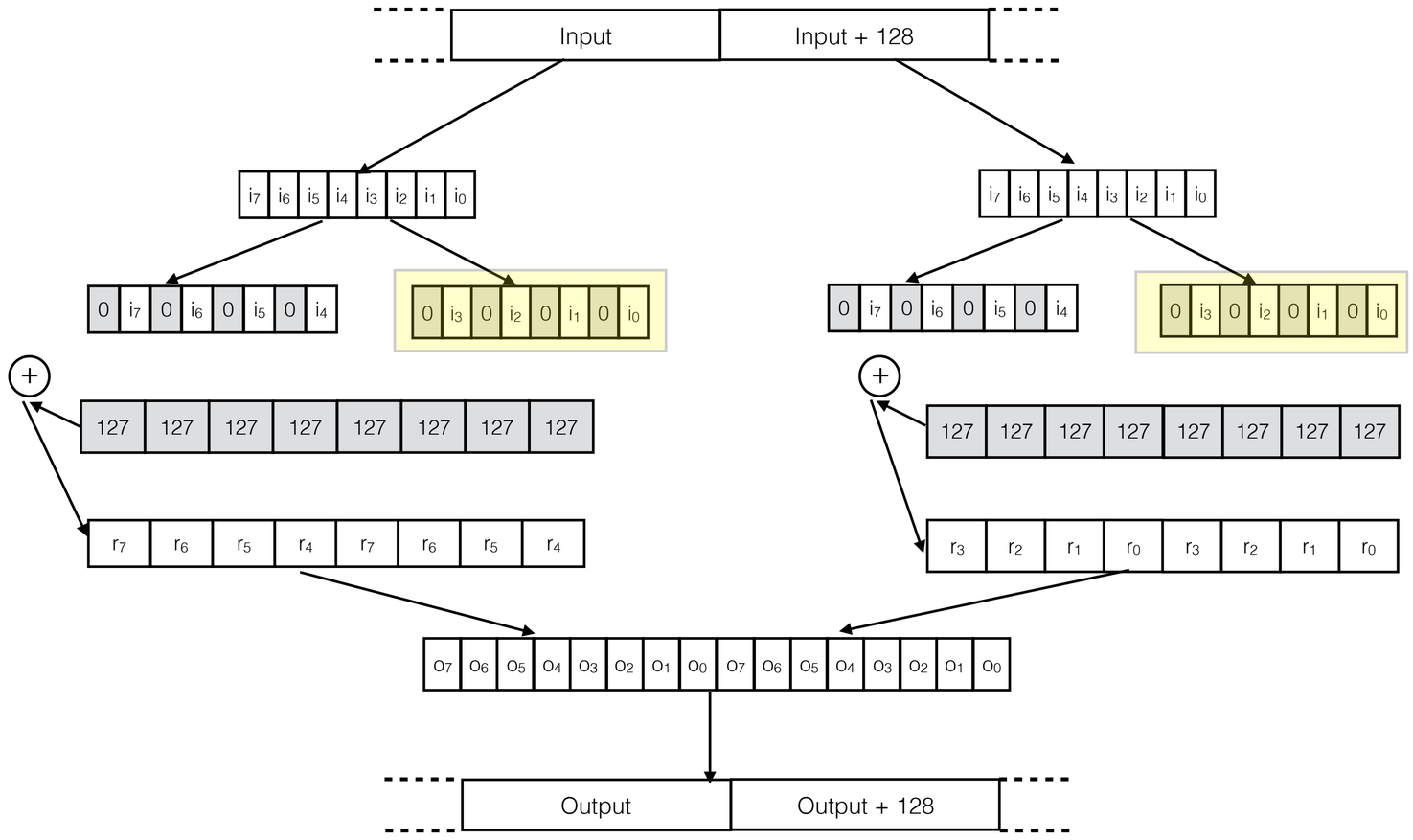
Revec Transformation

```
%1 = load <8 x i16>, <8 x i16>* %in
%2 = shufflevector <8 x i16> %1, const_vec_1, mask_1
%3 = shufflevector <8 x i16> %1, const_vec_2, mask_2
%4 = bitcast <8 x i16> %2 to <4 x i32>
%5 = add <4 x i32> %4, <127, 127, 127, 127>
%6 = bitcast <8 x i16> %3 to <4 x i32>
%7 = add <4 x i32> %6, <127, 127, 127, 127>
%8 = call <8 x i16> @llvm.x86.sse41.packusdw(%5, %7)
store <8 x i16> %8, <8 x i16>* %out


%9 = load <8 x i16>, <8 x i16>* %in_1
%10 = shufflevector <8 x i16> %9, const_vec_3, mask_3
%11 = shufflevector <8 x i16> %9, const_vec_4, mask_4
%12 = bitcast <8 x i16> %10 to <4 x i32>
%13 = add <4 x i32> %12, <127, 127, 127, 127>
%14 = bitcast <8 x i16> %11 to <4 x i32>
%15 = add <4 x i32> %14, <127, 127, 127, 127>
%16 = call <8 x i16> @llvm.x86.sse41.packusdw(%13,%15)
store <8 x i16> %16, <8 x i16>* %out_1
```

Vector Shuffles

```
%4a = bitcast <16 x i16> ?? to <8 x i32>
%5a = add <8 x i32> %4a, <127,127,127,127,127,127,127,127>
%6a = bitcast <16 x i16> ?? to <8 x i32>
%7a = add <8 x i32> %6a, <127,127,127,127,127,127,127,127>
%8a = call <16 x i16> @llvm.x86.avx2.packusdw(%5a,%7a)
store <16 x i16> %8a, <16 x i16>* %out
```

Revectorized code



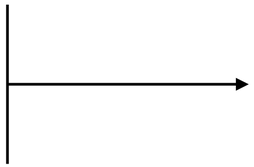
Shuffle Merge Rules

```
shufflevector operand_1, operand_2, mask
```

mask selects values from operand_1 or operand_2 to be in the final output

```
shufflevector operand_1, operand_2, mask_1
```

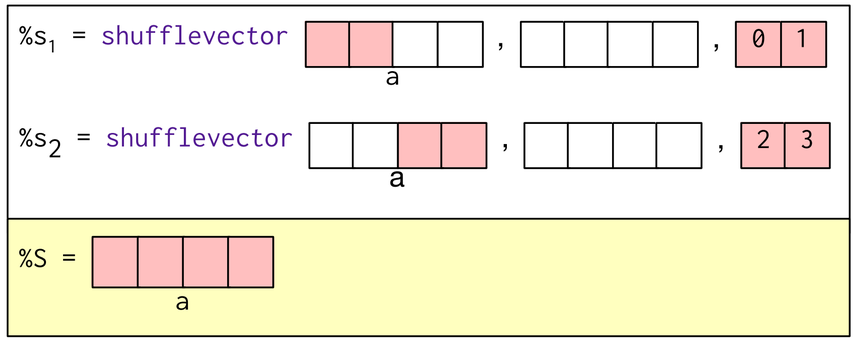
```
shufflevector operand_3, operand_4, mask_2
```



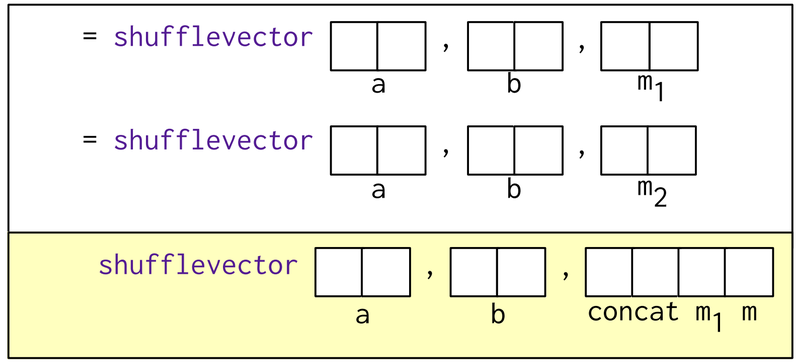
Depending on the mask and operand type rules for merging vector shuffles change

Four different vector shuffle patterns to handle merging

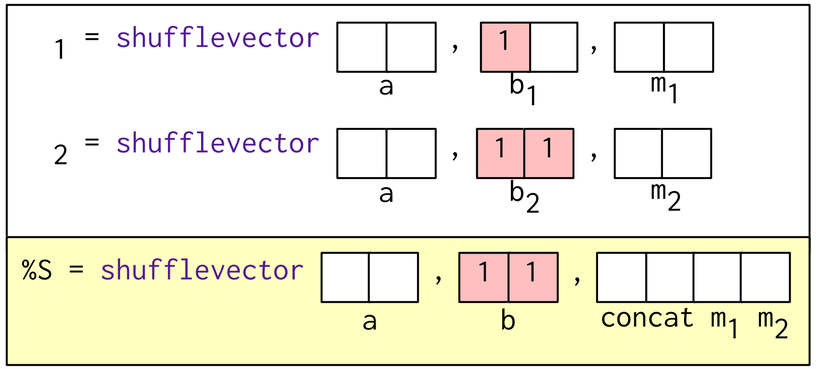
Shuffle Patterns



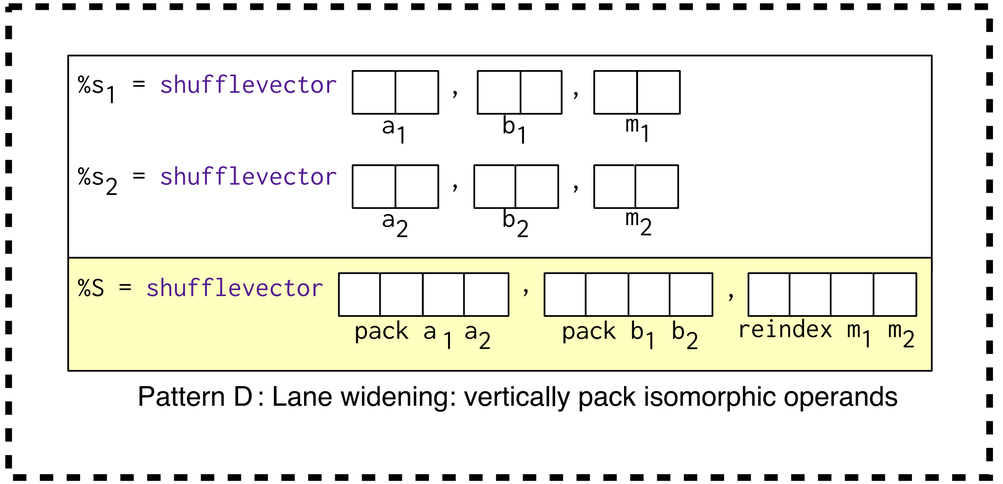
Pattern A: Sequential subvector extracts



Pattern B: Permutations of identical operands

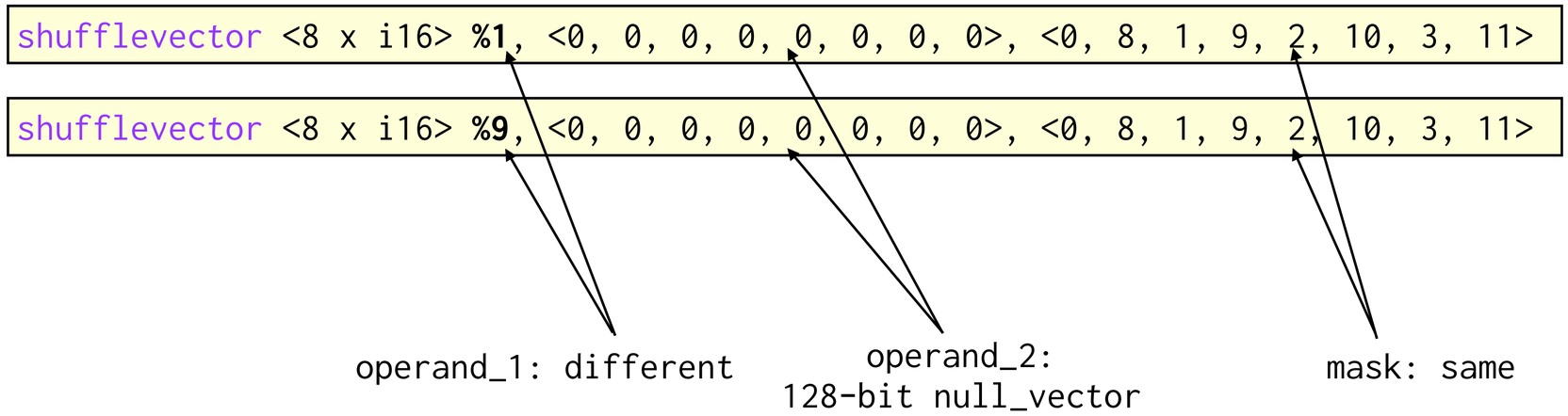


Pattern C: Mergeable constant vector operand

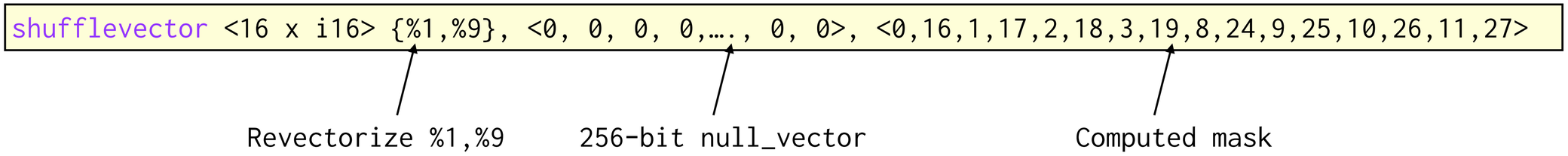


Pattern D: Lane widening: vertically pack isomorphic operands

Shuffle Merge Rules

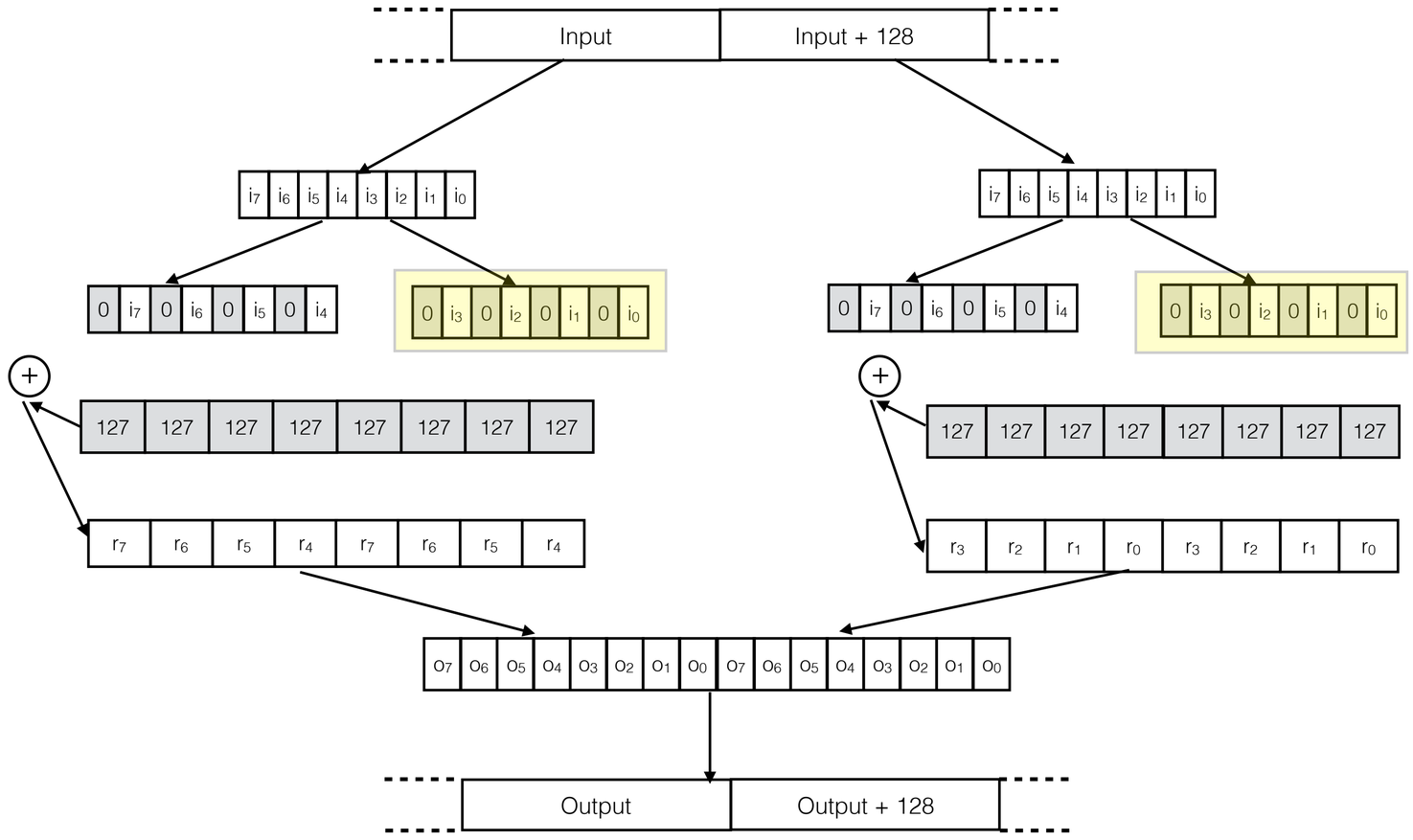


Generic Lane Widening



Revec Transformation

```
%1 = load <8 x i16>, <8 x i16>* %in
%2 = shufflevector <8 x i16> %1, const_vec_1, mask_1
%3 = shufflevector <8 x i16> %1, const_vec_2, mask_2
%4 = bitcast <8 x i16> %2 to <4 x i32>
%5 = add <4 x i32> %4, <127, 127, 127, 127>
%6 = bitcast <8 x i16> %3 to <4 x i32>
%7 = add <4 x i32> %6, <127, 127, 127, 127>
%8 = call <8 x i16> @llvm.x86.sse41.packusdw(%5, %7)
store <8 x i16> %8, <8 x i16>* %out


%9 = load <8 x i16>, <8 x i16>* %in_1
%10 = shufflevector <8 x i16> %9, const_vec_3, mask_3
%11 = shufflevector <8 x i16> %9, const_vec_4, mask_4
%12 = bitcast <8 x i16> %10 to <4 x i32>
%13 = add <4 x i32> %12, <127, 127, 127, 127>
%14 = bitcast <8 x i16> %11 to <4 x i32>
%15 = add <4 x i32> %14, <127, 127, 127, 127>
%16 = call <8 x i16> @llvm.x86.sse41.packusdw(%13,%15)
store <8 x i16> %16, <8 x i16>* %out_1
```

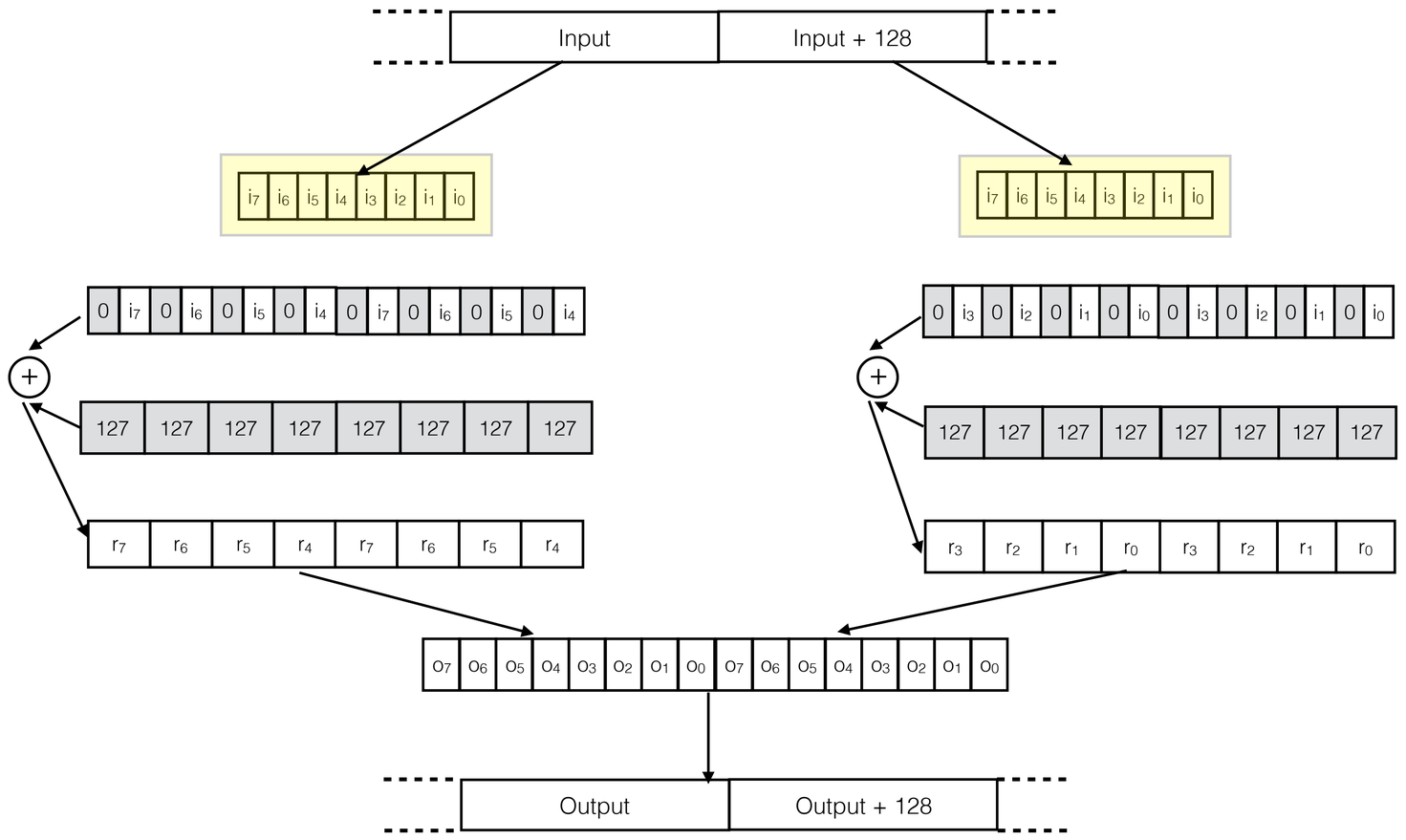



```
%4a = bitcast <16 x i16> ?? to <8 x i32>
%5a = add <8 x i32> %4a, <127,127,127,127,127,127,127,127>
%6a = bitcast <16 x i16> ?? to <8 x i32>
%7a = add <8 x i32> %6a, <127,127,127,127,127,127,127,127>
%8a = call <16 x i16> @llvm.x86.avx2.packusdw(%5a,%7a)
store <16 x i16> %8a, <16 x i16>* %out
```

Revectorized code

Revec Transformation

```
%1 = load <8 x i16>, <8 x i16>* %in
%2 = shufflevector <8 x i16> %1, const_vec_1, mask_1
%3 = shufflevector <8 x i16> %1, const_vec_2, mask_2
%4 = bitcast <8 x i16> %2 to <4 x i32>
%5 = add <4 x i32> %4, <127, 127, 127, 127>
%6 = bitcast <8 x i16> %3 to <4 x i32>
%7 = add <4 x i32> %6, <127, 127, 127, 127>
%8 = call <8 x i16> @llvm.x86.sse41.packusdw(%5, %7)
store <8 x i16> %8, <8 x i16>* %out

%9 = load <8 x i16>, <8 x i16>* %in_1
%10 = shufflevector <8 x i16> %9, const_vec_3, mask_3
%11 = shufflevector <8 x i16> %9, const_vec_4, mask_4
%12 = bitcast <8 x i16> %10 to <4 x i32>
%13 = add <4 x i32> %12, <127, 127, 127, 127>
%14 = bitcast <8 x i16> %11 to <4 x i32>
%15 = add <4 x i32> %14, <127, 127, 127, 127>
%16 = call <8 x i16> @llvm.x86.sse41.packusdw(%13,%15)
store <8 x i16> %16, <8 x i16>* %out_1
```




```
%2a = shufflevector <16 x i16> ??, const_vec_n1, mask_n1
%3a = shufflevector <16 x i16> ??, const_vec_n2, mask_n2
%4a = bitcast <16 x i16> %2a to <8 x i32>
%5a = add <8 x i32> %4a, <127,127,127,127,127,127,127,127>
%6a = bitcast <16 x i16> %3a to <8 x i32>
%7a = add <8 x i32> %6a, <127,127,127,127,127,127,127,127>
%8a = call <16 x i16> @llvm.x86.avx2.packusdw(%5a,%7a)
store <16 x i16> %8a, <16 x i16>* %out
```

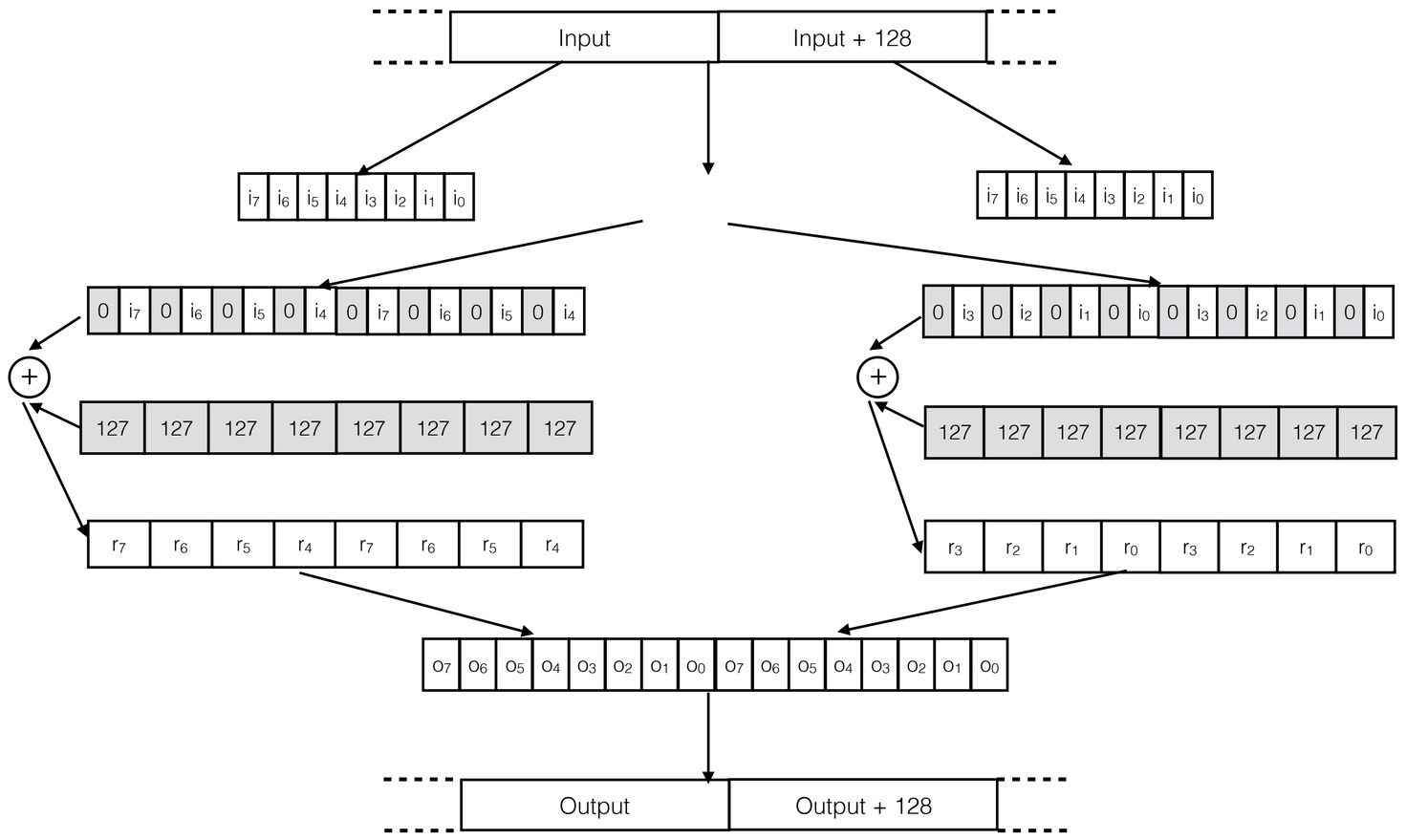
Revectorized code

Revec Transformation

```
%1 = load <8 x i16>, <8 x i16>* %in
%2 = shufflevector <8 x i16> %1, const_vec_1, mask_1
%3 = shufflevector <8 x i16> %1, const_vec_2, mask_2
%4 = bitcast <8 x i16> %2 to <4 x i32>
%5 = add <4 x i32> %4, <127, 127, 127, 127>
%6 = bitcast <8 x i16> %3 to <4 x i32>
%7 = add <4 x i32> %6, <127, 127, 127, 127>
%8 = call <8 x i16> @llvm.x86.sse41.packusdw(%5, %7)
store <8 x i16> %8, <8 x i16>* %out


%9 = load <8 x i16>, <8 x i16>* %in_1
%10 = shufflevector <8 x i16> %9, const_vec_3, mask_3
%11 = shufflevector <8 x i16> %9, const_vec_4, mask_4
%12 = bitcast <8 x i16> %10 to <4 x i32>
%13 = add <4 x i32> %12, <127, 127, 127, 127>
%14 = bitcast <8 x i16> %11 to <4 x i32>
%15 = add <4 x i32> %14, <127, 127, 127, 127>
%16 = call <8 x i16> @llvm.x86.sse41.packusdw(%13,%15)
store <8 x i16> %16, <8 x i16>* %out_1
```

```
%1a = load <16 x i16>, <16 x i16>* %in
%2a = shufflevector <16 x i16> %1a, const_vec_n1, mask_n1
%3a = shufflevector <16 x i16> %1a, const_vec_n2, mask_n2
%4a = bitcast <16 x i16> %2a to <8 x i32>
%5a = add <8 x i32> %4a, <127,127,127,127,127,127,127,127>
%6a = bitcast <16 x i16> %3a to <8 x i32>
%7a = add <8 x i32> %6a, <127,127,127,127,127,127,127,127>
%8a = call <16 x i16> @llvm.x86.avx2.packusdw(%5a,%7a)
store <16 x i16> %8a, <16 x i16>* %out
```

Revectorized code



Revectorization complete

Evaluation

SIMD-scan
(databases)

Video Compression

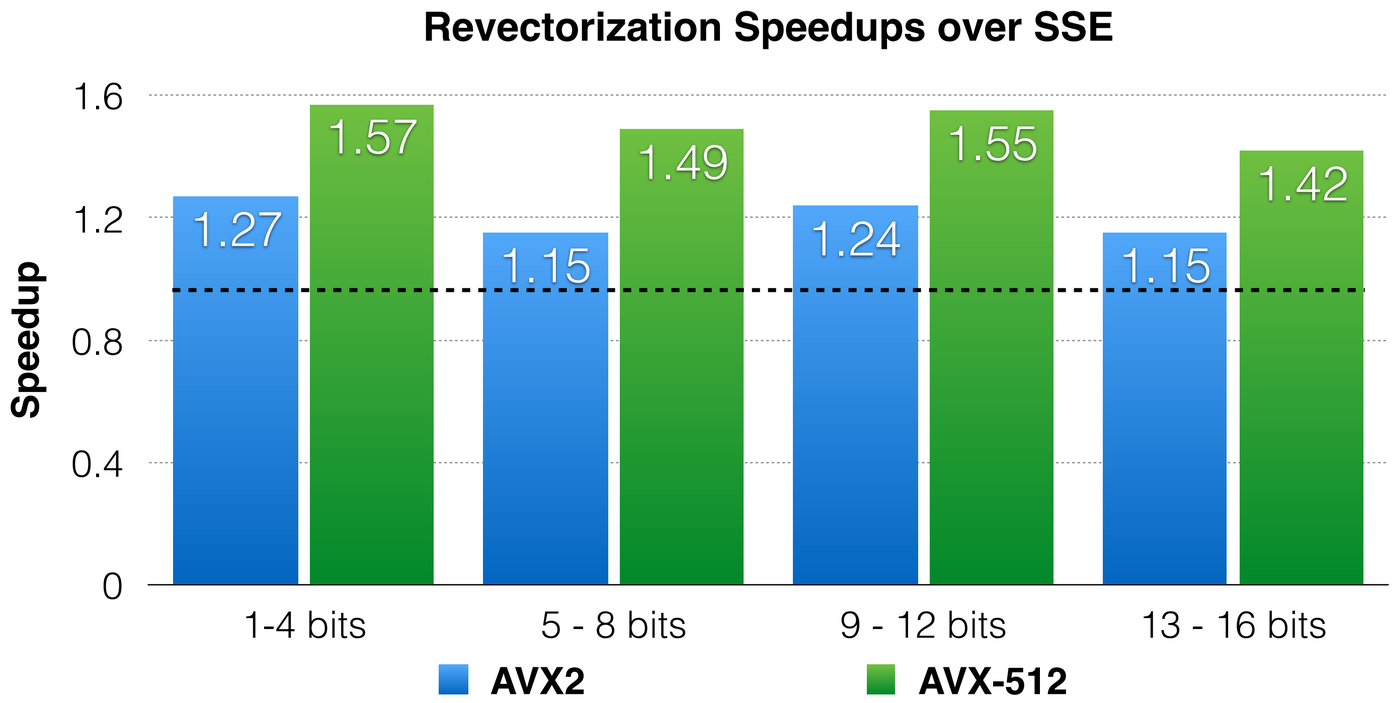
x265

Image Processing

High Performance Kernels

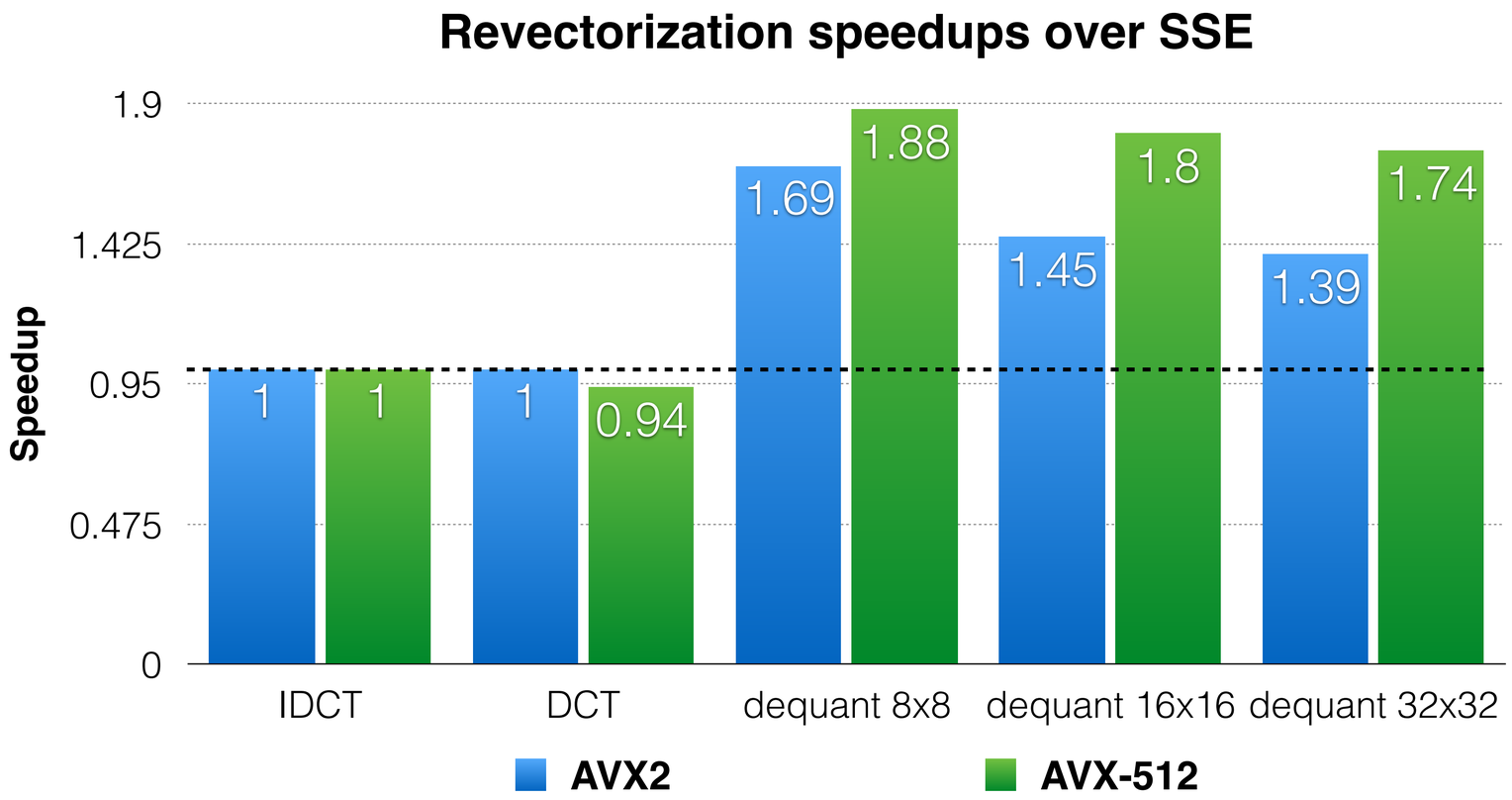
SIMD-Scan

- Variable width integer packing and unpacking using SIMD
- FastPFor's SSE implementation [Lemire, et al.]



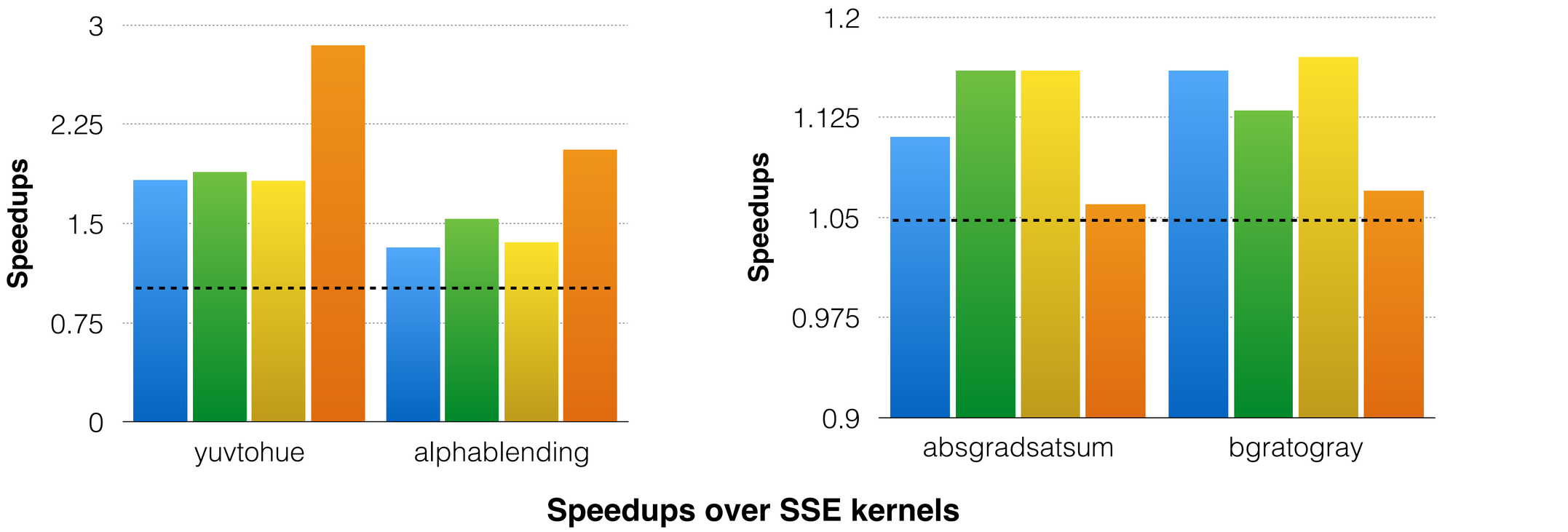
x265

- Implements H.265 Video Codec
- Best Software based video encoding for H.265 (http://x265.org)



Simd image processing

- **216** intrinsic-rich image processing and stencil operations
- Compare against hand-vectorized AVX2, AVX512 implementations
- Extensively hand-tuned



Conclusion

- Revectorization allows retargeting hand-vectorized code
- The compiler transformation **transparently** rejuvenates performance
- Helps **preserve performance portability** of hand-vectorized implementations

Try **Revec** and

VectorBench (benchmark suite)
>200 hand-vectorized kernels extracted
from popular repos

https://www.nextgenvec.org

This Work Supported By: