19th Workshop on High Performance Computing in Meteorology

Optimization of ACRANEB2 Radiation Kernel on Intel® Xeon® Processors

Vamsi Sripathi, Nitya Hariharan, Ruchira Sasanka

September 23rd, 2021

ACRANEB2

- Radiation component is an expensive part of numerical weather prediction (NWP) models
- ACRANEB2 is a radiation dwarf/kernel, part of ESCAPE* project. Used in production setups of ALADIN community
- Written in F90, OpenMP based parallelism (embarrassingly parallel/no thread synchronization)
- Unlike other components of NWP, it is compute bound (floating-point intensive)
- Motivation for tuning: Faster execution enables NWP community to run radiation physics at every time step of the model

ESCAPE: Energy-efficient Scalable Algorithms for Weather Prediction at Exascale

Anatomy of ACRANEB2 Kernel

PREFIX-SUM

```
do jlev2=jlev1+1,klev
  zdu1 = zdu1k(jlev2)
  zdu2 = zdu2k(jlev2)
  zdu3 = zdu3k(jlev2)
  zt_sca  = pt(jlev2,jlon)
  zp_sca  = paprsf(jlev2,jlon)
  zzirhov = pr(jlev2,jlon)*zt_sca

  jjlev2 = jlev2 + 1

  zc_uck(jjlev2)  = zc_uck(jjlev2-1) + zzc_uck(jlev2)
  zc_uw1k(jjlev2) = zc_uw1k(jjlev2-1)+ zzc_uw1k(jlev2)
  zc_us1k(jjlev2) = zc_us1k(jjlev2-1)+ zzc_us1k(jlev2)*zp_sca
  zc_us_irhov1k(jjlev2) = zc_us_irhov1k(jjlev2-1)+ zzc_us1k(jlev2)*zzirhov
  zc_u1k(jjlev2)  = zc_u1k(jjlev2-1) + zdu1
  zc_pu1k(jjlev2) = zc_pu1k(jjlev2-1)+ zp_sca*zdu1
  zc_tu1k(jjlev2) = zc_tu1k(jjlev2-1)+ zt_sca*zdu1

  zc_uw2k(jjlev2) = zc_uw2k(jjlev2-1)+ zzc_uw2k(jlev2)
  zc_us2k(jjlev2) = zc_us2k(jjlev2-1)+ zzc_us2k(jlev2)*zp_sca
  zc_us_irhov2k(jjlev2) = zc_us_irhov2k(jjlev2-1)+ zzc_us2k(jlev2)*zzirhov
  zc_u2k(jjlev2)  = zc_u2k(jjlev2-1) + zdu2
  zc_pu2k(jjlev2) = zc_pu2k(jjlev2-1)+ zp_sca*zdu2
  zc_tu2k(jjlev2) = zc_tu2k(jjlev2-1)+ zt_sca*zdu2

  zc_uw3k(jjlev2) = zc_uw3k(jjlev2-1)+ zzc_uw3k(jlev2)
  zc_us3k(jjlev2) = zc_us3k(jjlev2-1)+ zzc_us3k(jlev2)*zp_sca
  zc_us_irhov3k(jjlev2) = zc_us_irhov3k(jjlev2-1)+ zzc_us3k(jlev2)*zzirhov
  zc_u3k(jjlev2)  = zc_u3k(jjlev2-1) + zdu3
  zc_pu3k(jjlev2) = zc_pu3k(jjlev2-1)+ zp_sca*zdu3
  zc_tu3k(jjlev2) = zc_tu3k(jjlev2-1)+ zt_sca*zdu3

  zt_uck(jjlev2)  = zt_uck(jjlev2-1) + zzt_uck(jlev2)
  zt_uw1k(jjlev2) = zt_uw1k(jjlev2-1)+ zzt_uw1k(jlev2)
  zt_us1k(jjlev2) = zt_us1k(jjlev2-1)+ zzt_us1k(jlev2)*zp_sca
  zt_us_irhov1k(jjlev2) = zt_us_irhov1k(jjlev2-1)+ zzt_us1k(jlev2)*zzirhov

  zt_uw2k(jjlev2) = zt_uw2k(jjlev2-1)+ zzt_uw2k(jlev2)
  zt_us2k(jjlev2) = zt_us2k(jjlev2-1)+ zzt_us2k(jlev2)*zp_sca
  zt_us_irhov2k(jjlev2) = zt_us_irhov2k(jjlev2-1)+ zzt_us2k(jlev2)*zzirhov

  zt_uw3k(jjlev2) = zt_uw3k(jjlev2-1)+ zzt_uw3k(jlev2)
  zt_us3k(jjlev2) = zt_us3k(jjlev2-1)+ zzt_us3k(jlev2)*zp_sca
  zt_us_irhov3k(jjlev2) = zt_us_irhov3k(jjlev2-1)+ zzt_us3k(jlev2)*zzirhov
enddo
```

Math Ops

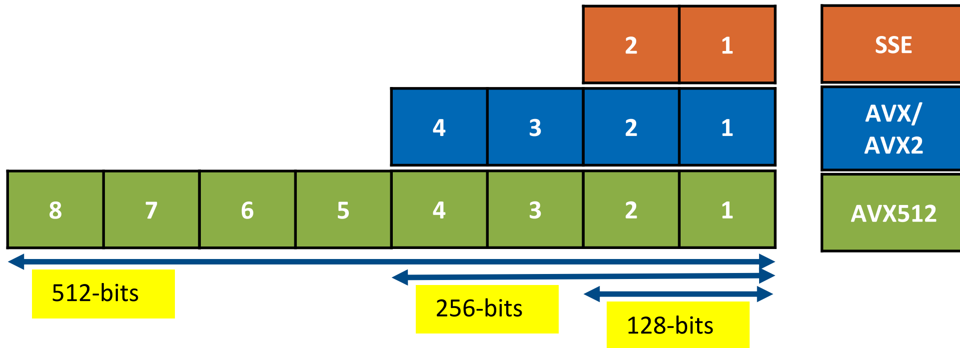
```
do jlev2=1,klev+1
  actoffset = (-1     + jidx)*(1+sign(1,jlev2-(jlev1+2)))/2                  &
          + (jjlev1+jjidx)*(1+sign(1,(jlev1+1)-jlev2))/2

  zp1 = zc_pu1k(jlev2)/zc_u1k(jlev2)
  zt1 = zc_tu1k(jlev2)/zc_u1k(jlev2)
  zp2 = zc_pu2k(jlev2)/zc_u2k(jlev2)
  zt2 = zc_tu2k(jlev2)/zc_u2k(jlev2)
  zp3 = zc_pu3k(jlev2)/zc_u3k(jlev2)
  zt3 = zc_tu3k(jlev2)/zc_u3k(jlev2)

  zdelta1 = zcdelta1(zc_uw1k(jlev2),zc_us1k(jlev2),zc_us_irhov1k(jlev2), &
                     zc_uck(jlev2),zp1,zt1)
  zdelta2 = zcdelta2(zc_uw2k(jlev2),zc_us2k(jlev2),zc_us_irhov2k(jlev2), &
                     zp2,zt2)
  zdel0 = zcdel0(    zc_uw3k(jlev2),zc_us3k(jlev2),zc_us_irhov3k(jlev2), &
                     zp3,zt3,zdelta1,zdelta2)
 ! compute transmissions
  ztau0(jlev2+actoffset) = exp(max(-zdel0,zargli))

  zdelta1 = ztdelta1(zt_uw1k(jlev2),zt_us1k(jlev2),zt_us_irhov1k(jlev2), &
                     zt_uck(jlev2),zp1,zt1)
  zdelta2 = ztdelta2(zt_uw2k(jlev2),zt_us2k(jlev2),zt_us_irhov2k(jlev2), &
                     zp2,zt2)
  zdel1 = ztdel1(    zt_uw3k(jlev2),zt_us3k(jlev2),zt_us_irhov3k(jlev2), &
                     zp3,zt3,zdelta1,zdelta2)
 ! compute transmissions
  ztau1(jlev2+actoffset) = exp(max(-zdel1,zargli))
enddo
```

- Prefix-Sum (3 variants)
  - V1: dst[i] = dst[i-1] + src[i]
  - V2: dst[i] = dst[i-1] + src1[i] * src2[i]
  - V3: dst[i] = dst[i-1] + src1[i] * src2[i] * src3[i]
  - Loop produces 29 output vectors
- Pow() and Log() functions: Results of prefix-sum fed as input

Optimization: AVX512

X86 ISA	Register Width	Register Name	FP64 elems/register	Num. 64b ops per SIMD Add
SSE	128-bit	XMM	2	2
AVX/AVX2	256-bit	YMM	4	4
AVX512	512-bit	ZMM	8	8

Op	AVX512 Vector Intrinsics API	Description
SIMD Load	__m512d _mm512_loadu_pd (void *mem_address);	Load a set of 8 FP64 elements from memory to AVX512 vector register
SIMD Add	__m512d _mm512_add_pd (__m512d src1, __m512d src2);	Add FP64 elements in src1 and src2 and return the results
SIMD Permute with mask (128b lanes)	__m512d _mm512_maskz_permute_pd (__mmask8 k, __m512d src, const int imm8);	Shuffle FP64 elements in src within 128-bit lanes using the 8-bit control in imm8, and store the results using zeromask k (elements are zeroed out when corresponding mask bit is not set)
SIMD Permute with mask (256b lanes)	__m512d mm512_maskz_permutex_pd (__mmask8 k, __m512d src, const int imm8);	Shuffle FP64 elements in src within 256-bit lanes using the 8-bit control in imm8, and store the results using zeromask k (elements are zeroed out when corresponding mask bit is not set)
SIMD Store	void _mm512_storeu_pd (void *mem_address, _m512d src);	Store a set of 8 FP64 elements to memory from AVX512 vector register

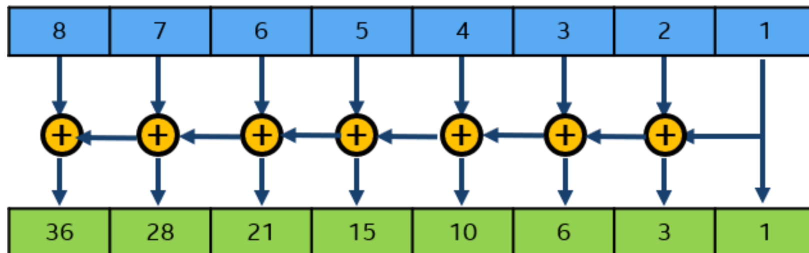


Prefix-Sum

$y_0 = x_0$

$y_1 = x_0 + x_1$

$y_2 = x_0 + x_1 + x_2$



- Prefix-Sum is not SIMD friendly, data dependency leads to non-trivial vectorization
- Developed custom Prefix-Sum kernels using AVX512 Compiler intrinsics

Optimization: Prefix-Sum Kernels

TUNED

BASELINE

```fortran
do jlev2=jlev1+1,klev
  zdu1 = zdu1k(jlev2)
  zdu2 = zdu2k(jlev2)
  zdu3 = zdu3k(jlev2)
  zt_sca  = pt(jlev2,jlon)
  zp_sca  = paprsf(jlev2,jlon)
  zzirhov = pr(jlev2,jlon)*zt_sca

  jjlev2 = jlev2 + 1

  zc_uck(jjlev2)  = zc_uck(jjlev2-1) + zzc_uck(jlev2)
  zc_uw1k(jjlev2) = zc_uw1k(jjlev2-1)+ zzc_uw1k(jlev2)
  zc_us1k(jjlev2) = zc_us1k(jjlev2-1)+ zzc_us1k(jlev2)*zp_sca
  zc_us_irhov1k(jjlev2) = zc_us_irhov1k(jjlev2-1)+ zzc_us1k(jlev2)*zzirhov
  zc_u1k(jjlev2)  = zc_u1k(jjlev2-1) + zdu1
  zc_pu1k(jjlev2) = zc_pu1k(jjlev2-1)+ zp_sca*zdu1
  zc_tu1k(jjlev2) = zc_tu1k(jjlev2-1)+ zt_sca*zdu1

  zc_uw2k(jjlev2) = zc_uw2k(jjlev2-1)+ zzc_uw2k(jlev2)
  zc_us2k(jjlev2) = zc_us2k(jjlev2-1)+ zzc_us2k(jlev2)*zp_sca
  zc_us_irhov2k(jjlev2) = zc_us_irhov2k(jjlev2-1)+ zzc_us2k(jlev2)*zzirhov
  zc_u2k(jjlev2)  = zc_u2k(jjlev2-1) + zdu2
  zc_pu2k(jjlev2) = zc_pu2k(jjlev2-1)+ zp_sca*zdu2
  zc_tu2k(jjlev2) = zc_tu2k(jjlev2-1)+ zt_sca*zdu2

  zc_uw3k(jjlev2) = zc_uw3k(jjlev2-1)+ zzc_uw3k(jlev2)
  zc_us3k(jjlev2) = zc_us3k(jjlev2-1)+ zzc_us3k(jlev2)*zp_sca
  zc_us_irhov3k(jjlev2) = zc_us_irhov3k(jjlev2-1)+ zzc_us3k(jlev2)*zzirhov
  zc_u3k(jjlev2)  = zc_u3k(jjlev2-1) + zdu3
  zc_pu3k(jjlev2) = zc_pu3k(jjlev2-1)+ zp_sca*zdu3
  zc_tu3k(jjlev2) = zc_tu3k(jjlev2-1)+ zt_sca*zdu3

  zt_uck(jjlev2)  = zt_uck(jjlev2-1) + zzt_uck(jlev2)
  zt_uw1k(jjlev2) = zt_uw1k(jjlev2-1)+ zzt_uw1k(jlev2)
  zt_us1k(jjlev2) = zt_us1k(jjlev2-1)+ zzt_us1k(jlev2)*zp_sca
  zt_us_irhov1k(jjlev2) = zt_us_irhov1k(jjlev2-1)+ zzt_us1k(jlev2)*zzirhov

  zt_uw2k(jjlev2) = zt_uw2k(jjlev2-1)+ zzt_uw2k(jlev2)
  zt_us2k(jjlev2) = zt_us2k(jjlev2-1)+ zzt_us2k(jlev2)*zp_sca
  zt_us_irhov2k(jjlev2) = zt_us_irhov2k(jjlev2-1)+ zzt_us2k(jlev2)*zzirhov

  zt_uw3k(jjlev2) = zt_uw3k(jjlev2-1)+ zzt_uw3k(jlev2)
  zt_us3k(jjlev2) = zt_us3k(jjlev2-1)+ zzt_us3k(jlev2)*zp_sca
  zt_us_irhov3k(jjlev2) = zt_us_irhov3k(jjlev2-1)+ zzt_us3k(jlev2)*zzirhov
enddo
```

```fortran
call awe_psum_hybrid_pack11(p_n, zepsu, &
     zdu1k(index_rhs),     zc_u1k(index_lhs),  & !v1
     zzc_uw1k(index_rhs),  zc_uw1k(index_lhs), & !v1
     zzc_uck(index_rhs),   zc_uck(index_lhs),  & !v1
     zzt_uw1k(index_rhs),  zt_uw1k(index_lhs), & !v1
     zzt_uck(index_rhs),   zt_uck(index_lhs),  & !v1
     zdu1k(index_rhs),     paprsf(index_rhs,jlon), zc_pu1k(index_lhs), & !v2
     zzc_us1k(index_rhs),  paprsf(index_rhs,jlon), zc_us1k(index_lhs), & !v2
     zzt_us1k(index_rhs),  paprsf(index_rhs,jlon), zt_us1k(index_lhs), & !v2
     zdu1k(index_rhs),     pt(index_rhs,jlon),     zc_tu1k(index_lhs), & !v2
     zzc_us1k(index_rhs),  pr(index_rhs,jlon), pt(index_rhs,jlon), zc_us_irhov1k(index_lhs), & !v3
     zzt_us1k(index_rhs),  pr(index_rhs,jlon), pt(index_rhs,jlon), zt_us_irhov1k(index_lhs))   !v3

call awe_psum_hybrid_pack9(p_n, zepsu, &
     zdu2k(index_rhs),     zc_u2k(index_lhs),  & !v1
     zzc_uw2k(index_rhs),  zc_uw2k(index_lhs), & !v1
     zzt_uw2k(index_rhs),  zt_uw2k(index_lhs), & !v1
     zdu2k(index_rhs),     paprsf(index_rhs,jlon), zc_pu2k(index_lhs), & !v2
     zzc_us2k(index_rhs),  paprsf(index_rhs,jlon), zc_us2k(index_lhs), & !v2
     zzt_us2k(index_rhs),  paprsf(index_rhs,jlon), zt_us2k(index_lhs), & !v2
     zdu2k(index_rhs),     pt(index_rhs,jlon),     zc_tu2k(index_lhs), & !v2
     zzc_us2k(index_rhs),  pr(index_rhs,jlon), pt(index_rhs,jlon), zc_us_irhov2k(index_lhs), & !v3
     zzt_us2k(index_rhs),  pr(index_rhs,jlon), pt(index_rhs,jlon), zt_us_irhov2k(index_lhs))   !v3

call awe_psum_hybrid_pack9(p_n, zepsu, &
     zdu3k(index_rhs),     zc_u3k(index_lhs),  & !v1
     zzc_uw3k(index_rhs),  zc_uw3k(index_lhs), & !v1
     zzt_uw3k(index_rhs),  zt_uw3k(index_lhs), & !v1
     zdu3k(index_rhs),     paprsf(index_rhs,jlon), zc_pu3k(index_lhs), & !v2
     zzc_us3k(index_rhs),  paprsf(index_rhs,jlon), zc_us3k(index_lhs), & !v2
     zzt_us3k(index_rhs),  paprsf(index_rhs,jlon), zt_us3k(index_lhs), & !v2
     zdu3k(index_rhs),     pt(index_rhs,jlon),     zc_tu3k(index_lhs), & !v2
     zzc_us3k(index_rhs),  pr(index_rhs,jlon),     pt(index_rhs,jlon), zc_us_irhov3k(index_lhs), & !v3
     zzt_us3k(index_rhs),  pr(index_rhs,jlon),     pt(index_rhs,jlon), zt_us_irhov3k(index_lhs))   !v3
```

- Developed two custom prefix-sum kernels using AVX512 Compiler intrinsics –
  - hybrid_pack11( ): Computes prefix-sum for a pack of 11 vectors
  - hybrid_pack9( ): Computes prefix-sum for a pack of 9 vectors
- Since Fortran application, packaged C-based intrinsics kernels into a library
- Split the prefix-sum of 29 vectors into 3 blocks (11 + 9 + 9)

Optimization: Hybrid + Packed Prefix-Sum

```
void awe_psum_hybrid_pack11_ (int *p_n, double *p_init,
                              double *p_v1_src,   double *p_v1_dst,
                              double *p_v2_src,   double *p_v2_dst,
                              double *p_v3_src,   double *p_v3_dst,
                              double *p_v4_src,   double *p_v4_dst,
                              double *p_v5_src,   double *p_v5_dst,
                              double *p_v6_src1,  double *p_v6_src2,  double *p_v6_dst,
                              double *p_v7_src1,  double *p_v7_src2,  double *p_v7_dst,
                              double *p_v8_src1,  double *p_v8_src2,  double *p_v8_dst,
                              double *p_v9_src1,  double *p_v9_src2,  double *p_v9_dst,
                              double *p_v10_src1, double *p_v10_src2, double *p_v10_src3, double *p_v10_dst,
                              double *p_v11_src1, double *p_v11_src2, double *p_v11_src3, double *p_v11_dst)
{
  int n = *p_n;
  __m512d zmm0, zmm1, zmm2, zmm3, zmm4, zmm_tmp, zmm_v6_src1, zmm_v6_src2, zmm_v7_src1, zmm_v8_src1, zmm_v9_src2, zmm_scal;
  __m512d zmm_v1_acc, zmm_v2_acc, zmm_v3_acc, zmm_v4_acc, zmm_v5_acc,
          zmm_v6_acc, zmm_v7_acc, zmm_v8_acc, zmm_v9_acc, zmm_v10_acc, zmm_v11_acc;

  zmm_v1_acc = zmm_v2_acc = zmm_v3_acc = zmm_v4_acc = zmm_v5_acc = zmm_v6_acc =
  zmm_v7_acc = zmm_v8_acc = zmm_v9_acc = zmm_v10_acc = zmm_v11_acc = _mm512_set1_pd(*p_init);
  __m512i zmm_idx     = _mm512_set1_epi64(3);
  __m512i zmm_acc_idx = _mm512_set1_epi64(7);

  for (int j=0; j<(n/N_UNROLL)*N_UNROLL; j+=N_UNROLL) {
    PSUM_V1_ZMM(p_v1_src, p_v1_dst, zmm_v1_acc, zmm0, zmm1, zmm2, zmm3, zmm4, zmm_tmp, zmm_idx, zmm_acc_idx);
    PSUM_V1_ZMM(p_v2_src, p_v2_dst, zmm_v2_acc, zmm0, zmm1, zmm2, zmm3, zmm4, zmm_tmp, zmm_idx, zmm_acc_idx);
    PSUM_V1_ZMM(p_v3_src, p_v3_dst, zmm_v3_acc, zmm0, zmm1, zmm2, zmm3, zmm4, zmm_tmp, zmm_idx, zmm_acc_idx);
    PSUM_V1_ZMM(p_v4_src, p_v4_dst, zmm_v4_acc, zmm0, zmm1, zmm2, zmm3, zmm4, zmm_tmp, zmm_idx, zmm_acc_idx);
    PSUM_V1_ZMM(p_v5_src, p_v5_dst, zmm_v5_acc, zmm0, zmm1, zmm2, zmm3, zmm4, zmm_tmp, zmm_idx, zmm_acc_idx);
  }

  for (int j=0; j<(n/N_UNROLL)*N_UNROLL; j+=N_UNROLL) {
    zmm_v6_src1 = _mm512_loadu_pd(p_v6_src1); zmm_v6_src2 = _mm512_loadu_pd(p_v6_src2);
    PSUM_V2_ZMM(zmm_v6_src1, zmm_v6_src2, p_v6_dst, zmm_v6_acc, zmm0, zmm1, zmm2, zmm3, zmm4, zmm_tmp, zmm_idx, zmm_acc_idx);
    zmm_v7_src1 = _mm512_loadu_pd(p_v7_src1);
    PSUM_V2_ZMM(zmm_v7_src1, zmm_v6_src2, p_v7_dst, zmm_v7_acc, zmm0, zmm1, zmm2, zmm3, zmm4, zmm_tmp, zmm_idx, zmm_acc_idx);
    zmm_v8_src1 = _mm512_loadu_pd(p_v8_src1);
    PSUM_V2_ZMM(zmm_v8_src1, zmm_v6_src2, p_v8_dst, zmm_v8_acc, zmm0, zmm1, zmm2, zmm3, zmm4, zmm_tmp, zmm_idx, zmm_acc_idx);
    zmm_v9_src2 = _mm512_loadu_pd(p_v9_src2);
    PSUM_V2_ZMM(zmm_v6_src1, zmm_v9_src2, p_v9_dst, zmm_v9_acc, zmm0, zmm1, zmm2, zmm3, zmm4, zmm_tmp, zmm_idx, zmm_acc_idx);
    zmm_scal = _mm512_mul_pd(_mm512_loadu_pd(p_v10_src2), zmm_v9_src2);
    PSUM_V3_ZMM(zmm_v7_src1, zmm_scal, p_v10_dst, zmm_v10_acc, zmm0, zmm1, zmm2, zmm3, zmm4, zmm_tmp, zmm_idx, zmm_acc_idx);
    PSUM_V3_ZMM(zmm_v8_src1, zmm_scal, p_v11_dst, zmm_v11_acc, zmm0, zmm1, zmm2, zmm3, zmm4, zmm_tmp, zmm_idx, zmm_acc_idx);

    p_v6_src1  += NUM_ELES_IN_ZMM;
    p_v6_src2  += NUM_ELES_IN_ZMM;
    p_v7_src1  += NUM_ELES_IN_ZMM;
    p_v8_src1  += NUM_ELES_IN_ZMM;
    p_v9_src2  += NUM_ELES_IN_ZMM;
    p_v10_src2 += NUM_ELES_IN_ZMM;
  }
```

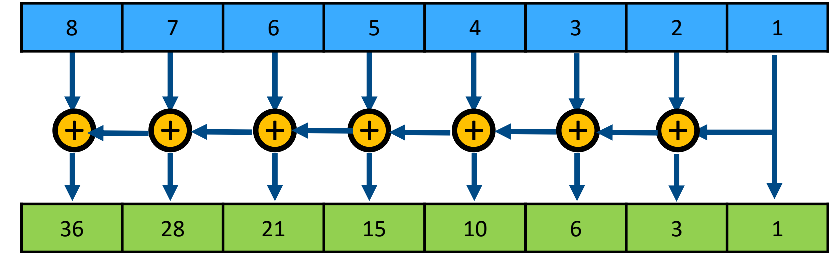
- A pack-11 kernel does three variants of prefix-sum:

V1: 5 vectors of form: dst[i] = dst[i-1] + src[i]
V2: 4 vectors of form: dst[i] = dst[i-1] + src1[i] * src2[i]
V3: 2 vectors of form: dst[i] = dst[i-1] + src1[i] * src2[i] * src3[i]

- Advantages of packed kernels over ad-hoc/1-vector:
  - Avoids redundant loads since some input vectors are shared
  - Avoids redundant multiplies
  - Keeps the 2 AVX512 add units busy and mitigates pipeline stalls by scheduling multiple independent ops
  - Reduces library function call overheads
- Main block using AVX512, tail with AVX (YMM), SSE (XMM, XMM_LO)

Optimization: AVX512 Prefix-Sum

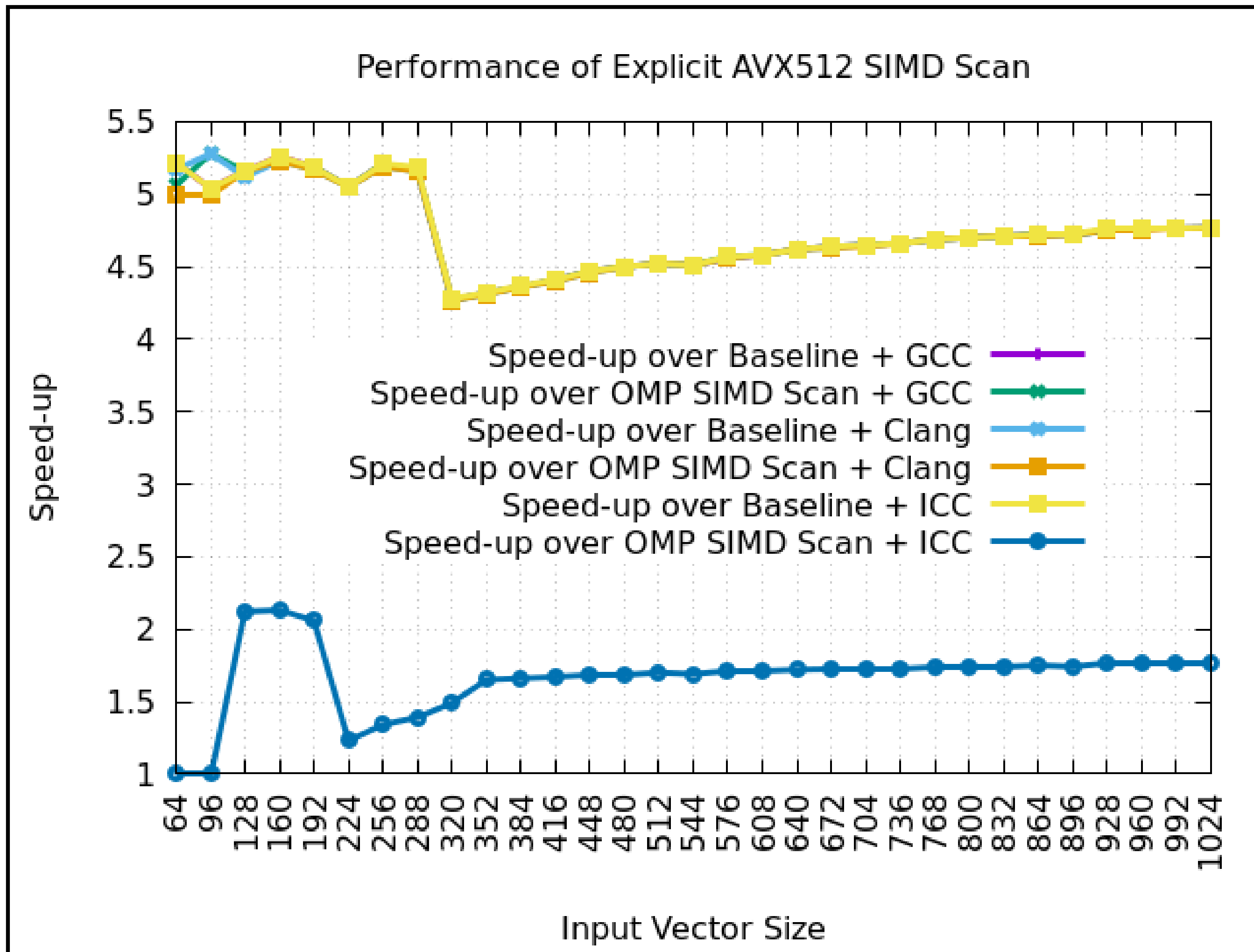
		512-bits				256-bits		128-bits	
Operation	Result Register	511b		383b		255b		127b	0b
_mm512_loadu_pd(p_src)	zmm0	8	7	6	5	4	3	2	1
_mm512_maskz_permute_pd (0xAA, zmm0, 0x00);	zmm1	7	0	5	0	3	0	1	0
_mm512_add_pd (zmm0, zmm1);	zmm2	15	7	11	5	7	3	3	1
_mm512_maskz_permutex_pd (0xCC, zmm0, 0x40);	zmm3	6	5	0	0	2	1	0	0
_mm512_add_pd(zmm2, zmm3);	zmm2	21	12	11	5	9	4	3	1
_mm512_maskz_permute_pd (0xCC, zmm3, 0x44);	zmm3	5	6	0	0	1	2	0	0
_mm512_add_pd(zmm2, zmm3);	zmm2	26	18	11	5	10	6	3	1
_mm512_maskz_permutexvar_pd (0xF0, _mm512_set1_epi64(3), zmm2);	zmm3	10	10	10	10	0	0	0	0
_mm512_add_pd(zmm2, zmm3);	zmm2	36	28	21	15	10	6	3	1




```
#define PSUM_V1_ZMM(p_src,p_dst,zmm_acc,zmm0,zmm1,zmm2,zmm3,zmm4,\
                    zmm_tmp,zmm_idx,zmm_acc_idx) do {\
  zmm0 = _mm512_loadu_pd((p_src));\
  zmm2 = _mm512_maskz_permute_pd(0xAA, zmm0, 0x00);\
  zmm4 = _mm512_add_pd(zmm0, zmm2);\
  zmm1 = _mm512_maskz_permutex_pd(0xCC, zmm0, 0x40);\
  zmm4 = _mm512_add_pd(zmm4, zmm1);\
  zmm1 = _mm512_maskz_permute_pd(0xCC, zmm1, 0x44);\
  zmm4 = _mm512_add_pd(zmm4, zmm1);\
  zmm_tmp = _mm512_maskz_permutexvar_pd(0xF0, zmm_idx, zmm4);\
  zmm4 = _mm512_add_pd(zmm4, zmm_tmp);\
  zmm4 = _mm512_add_pd(zmm4, zmm_acc);\
  zmm_acc = _mm512_permutexvar_pd(zmm_acc_idx, zmm4);\
  _mm512_storeu_pd(p_dst, zmm4);\
  p_src += NUM_ELES_IN_ZMM;\
  p_dst += NUM_ELES_IN_ZMM;\
} while(0)
```

- Simultaneously perform a series of add operations in the lower and upper 256-bit lanes of the 512-bit register after applying the necessary shuffle sequence
- Some of the permutes are independent of add
  - Helps instruction level parallelism and effective utilization of AVX512 execution units
- Operation sequence written to prefer permutes within 128b lane
  - Less costly (1-cycle latency) compared to 256b permutes (3-cycle)

Prefix-Sum: Performance



- Benchmarked explicit AVX512 SIMD Prefix-sum (scan) against GCC, Clang and ICC
- OpenMP 5.0 SIMD construct + Reduction clause + Scan directive
- GCC and Clang unable to vectorize the scan computations as their performance remains unchanged even after using OpenMP SIMD directives.
- Intel Compiler does a great job of auto-vectorization when OpenMP SIMD directives are used and beats both GCC and Clang.
- The average speed-up of explicit SIMD scan over baseline and OpenMP SIMD scan is 4.6x (GCC, Clang) and 1.6x (Intel Compiler) respectively.
- Intel Compiler will integrate the improvements in future releases.
- More details at https://techdecoded.intel.io/resources/optimization-of-scan-operations-using-explicit-vectorization/

Optimization: Cache Blocking

BLOCK-1

```
do jlev1=ktdia-1,klev/2-1
  p_n = klev - jlev1
  index_lhs = jlev1 + 2
  index_rhs = jlev1 + 1
  call awe_psum_hybrid_pack11(p_n, zepsu, &
          zdu1k(index_rhs),     zc_u1k(index_lhs), & !v1
          zzc_uw1k(index_rhs), zc_uw1k(index_lhs), & !v1
          zzc_uck(index_rhs),  zc_uck(index_lhs),  & !v1
          zzt_uw1k(index_rhs), zt_uw1k(index_lhs), & !v1
          zzt_uck(index_rhs),  zt_uck(index_lhs), & !v1
          zdu1k(index_rhs),     paprsf(index_rhs,jlon), zc_pu1k(index_lhs), & !v2
          zzc_us1k(index_rhs), paprsf(index_rhs,jlon), zc_us1k(index_lhs), & !v2
          zzt_us1k(index_rhs), paprsf(index_rhs,jlon), zt_us1k(index_lhs), & !v2
          zdu1k(index_rhs),     pt(index_rhs,jlon),     zc_tu1k(index_lhs), & !v2
          zzc_us1k(index_rhs), pr(index_rhs,jlon), pt(index_rhs,jlon), zc_us_irhov1k(index_lhs),
          zzt_us1k(index_rhs), pr(index_rhs,jlon), pt(index_rhs,jlon), zt_us_irhov1k(index_lhs))
  !DIR$ SIMD VECTORLENGTH(16)
  do jlev2=1,klev+1
    zp1 = zc_pu1k(jlev2)/zc_u1k(jlev2)
    zt1 = zc_tu1k(jlev2)/zc_u1k(jlev2)
    zc_delta1(jlev2,jlev1) = zcdelta1(zc_uw1k(jlev2),zc_us1k(jlev2),zc_us_irhov1k(jlev2), &
                                      zc_uck(jlev2),zp1,zt1)
    zt_delta1(jlev2,jlev1) = ztdelta1(zt_uw1k(jlev2),zt_us1k(jlev2),zt_us_irhov1k(jlev2), &
                                      zt_uck(jlev2),zp1,zt1)
  enddo
enddo ! end cache blocking loop for computations of zc_delta1(:), zt_delta1(:)
```

PREFIX-SUM

SVML

BLOCK-2

```
do jlev1=ktdia-1,klev/2-1
  p_n = klev - jlev1
  index_lhs = jlev1 + 2
  index_rhs = jlev1 + 1
  call awe_psum_hybrid_pack9(p_n, zepsu, &
         zdu2k(index_rhs),     zc_u2k(index_lhs), & !v1
         zzc_uw2k(index_rhs), zc_uw2k(index_lhs), & !v1
         zzt_uw2k(index_rhs), zt_uw2k(index_lhs), & !v1
         zdu2k(index_rhs),     paprsf(index_rhs,jlon), zc_pu2k(index_lhs), & !v2
         zzc_us2k(index_rhs), paprsf(index_rhs,jlon), zc_us2k(index_lhs), & !v2
         zzt_us2k(index_rhs), paprsf(index_rhs,jlon), zt_us2k(index_lhs), & !v2
         zdu2k(index_rhs),     pt(index_rhs,jlon),     zc_tu2k(index_lhs), & !v2
         zzc_us2k(index_rhs), pr(index_rhs,jlon), pt(index_rhs,jlon), zc_us_irhov2k(index_lhs),
         zzt_us2k(index_rhs), pr(index_rhs,jlon), pt(index_rhs,jlon), zt_us_irhov2k(index_lhs))
  !DIR$ SIMD VECTORLENGTH(16)
  do jlev2=1,klev+1
    zp2 = zc_pu2k(jlev2)/zc_u2k(jlev2)
    zt2 = zc_tu2k(jlev2)/zc_u2k(jlev2)
    zc_delta2(jlev2,jlev1) = zcdelta2(zc_uw2k(jlev2),zc_us2k(jlev2),zc_us_irhov2k(jlev2), &
                                      zp2,zt2)
    zt_delta2(jlev2,jlev1) = ztdelta2(zt_uw2k(jlev2),zt_us2k(jlev2),zt_us_irhov2k(jlev2), &
                                      zp2,zt2)
  enddo
enddo ! end cache blocking loop for computations of zc_delta2(:), zt_delta2(:)
```

PREFIX-SUM

SVML

BLOCK-3

```
do jlev1=ktdia-1,klev/2-1
  p_n = klev - jlev1
  index_lhs = jlev1 + 2
  index_rhs = jlev1 + 1
  call awe_psum_hybrid_pack9(p_n, zepsu, &
         zdu3k(index_rhs),     zc_u3k(index_lhs), & !v1
         zzc_uw3k(index_rhs), zc_uw3k(index_lhs), & !v1
         zzt_uw3k(index_rhs), zt_uw3k(index_lhs), & !v1
         zdu3k(index_rhs),     paprsf(index_rhs,jlon), zc_pu3k(index_lhs), & !v2
         zzc_us3k(index_rhs), paprsf(index_rhs,jlon), zc_us3k(index_lhs), & !v2
         zzt_us3k(index_rhs), paprsf(index_rhs,jlon), zt_us3k(index_lhs), & !v2
         zdu3k(index_rhs),     pt(index_rhs,jlon),     zc_tu3k(index_lhs), & !v2
         zzc_us3k(index_rhs), pr(index_rhs,jlon),     pt(index_rhs,jlon), zc_us_irhov3k(index_lhs),
         zzt_us3k(index_rhs), pr(index_rhs,jlon),     pt(index_rhs,jlon), zt_us_irhov3k(index_lhs))
  !DIR$ SIMD VECTORLENGTH(16)
  do jlev2=8,klev+1
    zp3 = zc_pu3k(jlev2)/zc_u3k(jlev2)
    zt3 = zc_tu3k(jlev2)/zc_u3k(jlev2)
    zdel0(jlev2,jlev1) = zcdel0(zc_uw3k(jlev2),zc_us3k(jlev2),zc_us_irhov3k(jlev2), &
                                zp3,zt3,zc_delta1(jlev2,jlev1),zc_delta2(jlev2,jlev1))
    zdel1(jlev2,jlev1) = ztdel1(zt_uw3k(jlev2),zt_us3k(jlev2),zt_us_irhov3k(jlev2), &
                                zp3,zt3,zt_delta1(jlev2,jlev1),zt_delta2(jlev2,jlev1))
  enddo
enddo ! end cache blocking loop for computations of zdel0(:), zdel1(:)
```

PREFIX-SUM

SVML

- Block and fuse the Prefix-sum calculations with Math operations
- Short Vector Math Library (SVML): Intel Fortran Compiler automatically uses the optimized implementations under the hood
- 3 cache loop blocks each operate on 11, 9, 9 vectors (total of 29)
- Aggressive vectorization:
  - Unroll by 16 for SVML loops (w/ SIMD VECTORLENGTH(16)) from default of 8
- Minimize unaligned load/stores:
  - Manual padding of columns for 2D temp arrays to align AVX512 boundary (64B)

Radiation: Performance Results

- Benchmarked two different versions of dwarf, lonlev-0.9 and transt3-v0.1
- Problem size tested was 1200x1200x80
- Tested with Intel Fortran Compiler on Intel® Xeon® Platinum 8380 Processor (code name Ice Lake, 40 cores/socket, supports AVX512)
- 30% speed-up over baseline (with Intel Fortran)

Questions?

vamsi.sripathi@intel.com

Notices & Disclaimers

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

Results have been estimated or simulated.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

Configuration Details

Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 160
On-line CPU(s) list: 0-159
Thread(s) per core: 2
Core(s) per socket: 40
Socket(s): 2
NUMA node(s): 2
Vendor ID: GenuineIntel
CPU family: 6
Model: 106
Model name: Intel(R) Xeon(R) Platinum 8380 CPU @ 2.30GHz
Stepping: 6
CPU MHz: 1900.948
CPU max MHz: 2301.0000
CPU min MHz: 800.0000
BogoMIPS: 4600.00
Virtualization: VT-x
L1d cache: 48K
L1i cache: 32K
L2 cache: 1280K
L3 cache: 61440K
NUMA node0 CPU(s): 0-39,80-119
NUMA node1 CPU(s): 40-79,120-159
Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb cat_l3 invpcid_single ssbd mba ibrs ibpb stibp ibrs_enhanced fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid cqm rdt_a avx512f avx512dq rdseed adx avx512ifma clflushopt clwb intel_pt avx512cd sha_ni avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local split_lock_detect wbnoinvd dtherm ida arat pln pts hwp hwp_act_window hwp_epp hwp_pkg_req avx512vbmi umip pku ospke avx512_vbmi2 gfni vaes vpclmulqdq avx512_vnni avx512_bitalg tme avx512_vpopcntdq la57 rdpid md_clear pconfig flush_l1d arch_capabilities

Intel Fortran Compiler: Intel(R) Parallel Studio XE 2020 Update 4 for Linux*.

intel®