```
! Treat A and B each as n m-by-m square matrices (with the n dimension
! varying fastest) and perform matrix multiplications on all n matrix pairs
mat_x_mat = 0.0_jprb ! Array-wise assignment
mblock = m/3
m2block = 2*mblock
if (i_actual_matrix_pattern == IMatrixPatternShortwave) then
  ! Matrix has a sparsity pattern
  !     (C D E)
  !     (F G H)
  !     (0 0 I)
  ! Do the top-left (C, D, F, G)
  do j2 = 1,m2block ! 1,6
    do j1 = 1,m2block ! 1,6
      do j3 = 1,m2block ! 1,6
        mat_x_mat(1:ng3D,j1,j2) = mat_x_mat(1:ng3D,j1,j2) &
             &                  + A(1:ng3D,j1,j3)*B(1:ng3D,j3,j2)
      end do
    end do
  end do
  do j2 = m2block+1,m ! 7,9
    ! Do the top-right (E & H)
    do j1 = 1,m2block ! 1,6
      do j3 = 1,m
        mat_x_mat(1:ng3D,j1,j2) = mat_x_mat(1:ng3D,j1,j2) &
             &                  + A(1:ng3D,j1,j3)*B(1:ng3D,j3,j2)
      end do
    end do
    ! Do the bottom-right (I)
    do j1 = m2block+1,m ! 7,9
      do j3 = m2block+1,m ! 7,9
        mat_x_mat(1:ng3D,j1,j2) = mat_x_mat(1:ng3D,j1,j2) &
             &                  + A(1:ng3D,j1,j3)*B(1:ng3D,j3,j2)
      end do
    end do
  end do
else
...
```

```
pure subroutine mat_x_mat_sw_repeats(ng_sw_in, nlev_b, A, B, C)
  integer,    intent(in)                                  :: ng_sw_in, nlev_b
  real(jprb), intent(in), dimension(ng_sw*nlev_b,9,9) :: A, B
  real(jprb), intent(out),dimension(ng_sw*nlev_b,9,9) :: C
  integer    :: j1, j2, j22
  !dir$ assume_aligned A:64,B:64,C:64
  ! Input matrices have pattern:
  !   (C          D          E)
  !   (F=-D       G=-C       H)
  !   (0          0          I), where each element is a 3-by-3 matrix
  ! As a result, output matrices have pattern:
  !   (C          D          E)
  !   (F=D        G=C        H)
  !   (0          0          I)
  do j2 = 1,3
    j22 = j2 + 6
    do j1 = 1,6
      ! Do the top-left (C, F)
      ! Unroll innermost matmul loop: more work for each iteration of SIMD loop
      C(:,j1,j2) = A(:,j1,1)*B(:,1,j2) + A(:,j1,2)*B(:,2,j2) + A(:,j1,3)*B(:,3,j2) &
      &          + A(:,j1,4)*B(:,4,j2) + A(:,j1,6)*B(:,6,j2)
      ! Do the top-right (E & H)
      C(:,j1,j22) = A(:,j1,1)*B(:,1,j22) + A(:,j1,2)*B(:,2,j22) + A(:,j1,3)*B(:,3,j22) &
      &      + A(:,j1,4)*B(:,4,j22) + A(:,j1,5)*B(:,5,j22) + A(:,j1,6)*B(:,6,j22) &
      &      + A(:,j1,7)*B(:,7,j22) + A(:,j1,8)*B(:,8,j22) + A(:,j1,9)*B(:,9,j22)
    end do
    do j1 = 7,9  ! Do the bottom-right (I)
      C(:,j1,j22) = A(:,j1,7)*B(:,7,j22) + A(:,j1,8)*B(:,8,j22) + A(:,j1,9)*B(:,9,j22)
    end do
  end do
  C(:,1:3,4:6) = C(:,4:6,1:3) ! D = F
  C(:,4:6,4:6) = C(:,1:3,1:3) ! G = C
  C(:,7:9,1:6) = 0.0_jprb     ! Lower left corner
```

Figure 1: Reference (top) and optimized (bottom) versions of the matrix-matrix multiplication kernel used in the shortwave matrix exponential computations. The latter unrolls loops and reduces work by exploiting that some matrix elements are repeated. For this performance-critical code, further speedup was gained by data alignment. The Intel compiler reported aligned data access only after declaring `ng_sw` at compile-time.