Use MKL copy to copy each element from x to y, incrementing 1 step everytime

```
void Copy(const float (&x)[XDIM][YDIM][ZDIM], float (&y)[XDIM][YDIM][ZDIM])
{
#ifdef DO_NOT_USE_MKL

#pragma omp parallel for
    for (int i = 1; i < XDIM - 1; i++)
        for (int j = 1; j < YDIM - 1; j++)
            for (int k = 1; k < ZDIM - 1; k++)
                y[i][j][k] = x[i][j][k];
#else
    cblas_scopy(XDIM * YDIM * ZDIM,
                &x[0][0][0],
                1,
                &y[0][0][0],
                1);

#endif
}
```

```
void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM],
           float (&z)[XDIM][YDIM][ZDIM],
           const float scale)
{
#ifdef DO_NOT_USE_MKL
#pragma omp parallel for
    for (int i = 0; i < XDIM; i++)
        for (int j = 0; j < YDIM; j++)
            for (int k = 0; k < ZDIM; k++)
                z[i][j][k] = x[i][j][k] * scale + y[i][j][k];
#else

    using array_t = float(&)[XDIM][YDIM][ZDIM];
    float *y_temp_raw = new float[XDIM * YDIM * ZDIM];
    array_t y_temp = reinterpret_cast<array_t>(*y_temp_raw);

    cblas_scopy(XDIM * YDIM * ZDIM,
                &y[0][0][0],
                1,
                &y_temp[0][0][0],
                1);

    cblas_saxpy(
        XDIM * YDIM * ZDIM, // Length of vectors
        scale,              // Scale factor
        &x[0][0][0],        // Input vector x, in operation y := x * scale + y
        1,                  // Use step 1 for x
        &y_temp[0][0][0],   // Input/output vector y, in operation y := x * scale +
        1                   // Use step 2 for y
    );
    cblas_scopy(XDIM * YDIM * ZDIM,
                &y_temp[0][0][0],
                1,
                &z[0][0][0],
                1);

#endif
}
```

```
float Norm(const float (&x)[XDIM][YDIM][ZDIM])
{

#ifdef DO_NOT_USE_MKL
    float result = 0.;
#pragma omp parallel for reduction(max \
                                   : result)
    for (int i = 1; i < XDIM - 1; i++)
        for (int j = 1; j < YDIM - 1; j++)
            for (int k = 1; k < ZDIM - 1; k++)
                result = std::max(result, std::abs(x[i][j][k]));

    return result;
#else
    int maxIndex = cblas_isamax(XDIM * YDIM * ZDIM,
                                &x[0][0][0],
                                1);
    // std::cout << "index is " << maxIndex << std::endl;
    return abs(*(&x[0][0][0] + maxIndex));

#endif
}
```

```
float InnerProduct(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM
{
#ifdef DO_NOT_USE_MKL
    double result = 0.;

#pragma omp parallel for reduction(+ \
                                   : result)
    for (int i = 1; i < XDIM - 1; i++)
        for (int j = 1; j < YDIM - 1; j++)
            for (int k = 1; k < ZDIM - 1; k++)
                result += (double)x[i][j][k] * (double)y[i][j][k];

    return (float)result;
#else
    return cblas_dsdot(XDIM * YDIM * ZDIM,
                       &x[0][0][0],
                       1,
                       &y[0][0][0],
                       1);
#endif
}
```

MKL timing：

```
Conjugate Gradients terminated after 23 iterations; residual norm (nu
[Total Laplacian Time : 524.825ms]
[Total Saxpy Time : 378.204ms]
```

```
Conjugate Gradients terminated after 23 iterations; residual norm (nu) = 0.000850816
[Total Laplacian Time : 567.838ms]
[Total Saxpy Time : 449.205ms]
```