

[Jump to navigation](#)

Developer Zone

[Join today](#) [Log in](#)

Development

OS

[Android\\*](#) [Chrome\\*](#) [HTML5](#) [Windows\\*](#)

Device

[2-in-1 & Ultrabook](#),[ç](#) [Business Client](#) [Embedded Systems](#) [IoT](#) [Server, Workstation, HPC](#)

Technology

[Big Data](#) [Intel® Modem Code](#) [Dual Screen](#) [Game Dev](#) [Intel® RealSense](#),[ç](#) [ISA Extensions](#) [Networking](#) [Open](#)[Source](#) [Storage](#)

Tools

Developer Type

[Embedded Systems](#) [Game Dev](#) [Media](#) [Technical, Enterprise, HPC](#) [Web](#)

OS

[All Tools](#) [Android\\*](#) [HTML5](#) [Linux\\*](#) [OS X\\*](#) [Windows\\*](#)

Resources

[Contact Support](#) [Documentation](#) [Free Software & Code Samples](#) [Intel Registration Center](#) [Product Forums](#) [SDKs](#)

Resources

Partner with Intel

[Academic Program](#) [Partner Spotlight](#) [Black Belt Developer](#) [Innovator Program](#) [Success Stories](#)

Learn

[Blog](#) [Events](#) [Videos](#)

Support

[Contact Support](#) [Developer Evangelists](#) [FAQs](#) [Forums](#)**Search form**Search **Case Study: Optimized Code for Neural Cell Simulations**Submitted by [Daniel Ve](#) on January 4, 2016[Share](#) [Tweet](#) [Share](#)**About**

Intel held the [Intel® Modem Code Developer Challenge](#) that had about 2,000 students from 130 universities in 19 countries registered to participate in the Challenge. They were provided access to Intel® Xeon® processors and Intel® Xeon Phi, ç coprocessors to optimize code used in a CERN openlab brain simulation research project. The goal of the research project is to find treatments and cures for neurological disorders, such as schizophrenia, epilepsy, and autism. The contestants task was to look at the code for cell clustering and 3D movement and then modify the algorithms for parallel performance by optimizing the code to reduce the runtime, all while maintaining correctness.

In this article Daniel Ve Falguera (one of the Challenge winners) shares the original code as well as the *optimized code* (Note: **Changed code line(s)**) and describes many of the optimizations he implemented. In some cases, the optimizations did not work, but he gives insight into how other changes to the code would work.

[Herunterladen](#) [Download](#)**INCLUDES****Original Code**

```
#include <cstring>
#include <cstdlib>
#include <ctime>
#include <cmath>
#include <getopt.h>
#include "util.hpp"
```

**Optimized Code**

```
#include <cstring>
#include <cstdlib>
#include <ctime>
#include <cmath>
#include <omp.h>
#include <getopt.h>
#include "util.hpp"
#include <malloc.h> //Useless
#include <mkl.h> //Useless
#include <cilk/cilk.h> //Useless
```

**Changed code line: 5****Optimization Notes**

The addition of `#include <omp.h>` is required to run OpenMP\*. The OpenMP functions and clauses are used in the optimized code.

In the completed code, `malloc.h`, `mkl.h`, and `cilk.h` were included during the development process to optimize memory blocks (among other things), but they didn't show any improvements. They are included to show that there was an attempt to use them. More information on the Intel® Math Kernel Library (Intel® MKL) and Intel® Cilk, ç Plus is given in the Other Optimizations section.

**RandomFloatPos**

The function `RandomFloatPos()` is used to generate a random number three times to generate a random 3D position inside the `cellMovementAndDuplication` function.

**NOTE:** Only the computational related functions are referenced in this paper, so functions such as `printf` and others aren't discussed.

#### Original Code

```
static float RandomFloatPos() {
    // returns a random number between a given minimum and maximum
    float random = ((float) rand()) / (float) RAND_MAX;
    float a = 0;
    float r = random;
    return a + r;
}
```

**Changed code line(s):** 3, 4, 5, 6

#### Optimized Code

```
static void RandomFloatPos(float input[3], unsigned sedin) {
    // returns a random number between a given minimum and maximum
    __assume_aligned((float*)input, 64);
    unsigned int seed=sedin,i;
    for(i=0; i<3; ++i){
        input[i]=(((float)rand_r(&seed))/(float)(RAND_MAX))-0.5;
    }
}
```

**Changed code line:** 1

#### Optimization Notes

The original function returns a random number doing some needless steps (for instance, adding zero to the random number). The original code can't be parallelized directly due to the function `rand()`, which can only be executed one thread at a time. The parallelization problem was fixed using the `rand_r()` function. The `rand_r()` function allows each thread to execute at the same time.

The function `RandomFloatPos()` is called to generate a random 3D position minus a constant offset of 0.5, so it can be simplified as a for loop with three iterations. The optimized code does exactly this. The offset is used to make the position values range from -0.5 to 0.5, placing the (0,0,0) point in the middle of the space of cell movement.

The optimized function has two parameters: `input[3]` and `sedin`. `input[3]` returns the values of the 3D random generated position and is memory aligned(64). The `sedin` is passed as a parameter and contains the value of the seed generated for each call to this function.

The for loop generates the 3D coordinate and stores it into `input`.

#### getNorm

##### Original Code

```
static float getNorm(float* currArray) {
    // computes L2 norm of input array
    int c;
    float arraySum=0;
    for (c=0; c<3; c++) {
        arraySum += currArray[c]*currArray[c];
    }
    float res = sqrt(arraySum);
    return res;
}
```

**Changed code line(s):** 3, 6, 8, 9

##### Optimized Code

```
static float getNorm(float* currArray) {
    // computes L2 norm of input array
    float arraySum=0;
    for (int c=0; c<3; ++c) {
        arraySum += pow(currArray[c],2);
    }
    return sqrt(arraySum);
}
```

**Changed code line(s):** 5, 7

#### Optimization Notes

This computes the norm from a given array of numbers by the input float `currArray`. There are two ways that the code was optimized for this function:

The `res` variable was removed. The `pow()` function was added. Extensive research done afterwards has shown that the `pow()` is not a significant improvement. Further, given the Intel® Xeon Phi's coprocessor's ability to execute multiple floating point operations per cycle, the original `currArray[c]*currArray[c]` is provably faster in those cases. The Intel compiler vectorizes the `pow()` function, but adds more code to do so, resulting in slower runtime speed.

#### getL2Distance

This function is used to determine the linear distance between two points in 3D space.

##### Original Code

```
static float getL2Distance(float pos1x, float pos1y, float pos1z, float
pos2x, float pos2y, float pos2z) {
    // returns distance (L2 norm) between two positions in 3D
    float distArray[3];
    distArray[0] = pos2x-pos1x;
    distArray[1] = pos2y-pos1y;
    distArray[2] = pos2z-pos1z;
    float l2Norm = getNorm(distArray);
    return l2Norm;
}
```

**Changed code line(s):** 1, 2, 5, 6, 7, 8, 9

##### Optimized Code

```
static float getL2Distance(float* pos1, float* pos2) {
    // returns distance (L2 norm) between two positions in 3D
    float distArray[3] __attribute__((aligned(64)));
    distArray[0] = pos2[0]-pos1[0];
    distArray[1] = pos2[1]-pos1[1];
    distArray[2] = pos2[2]-pos1[2];
    return getNorm(distArray);
}
```

**Changed code line(s):** 1-8

#### Optimization Notes

The original function has six inputs that represent two 3D points to calculate the distance between them. The optimized function has only two inputs; each input is an array of three elements representing the 3D point. The variable used inside is aligned.

The optimized function seems to be SIMD executable, but when using the vector notation (for example,  $P[0:2]=a[0:2]*b[0:2]$ ) the execution time was slower. It may be that defining and using an elemental function will further optimize the function.

A white paper on elemental functions is available at:

<http://software.intel.com/sites/default/files/article/181418/whitepaperonelementalfuncions.pdf>

#### produceSubstances

This function increases the concentration of substances for each cell position to a maximum limit of 1 unit per cell position.

#### Original Code

```
static void produceSubstances(float*** Conc, float** posAll, int* typesAll, int L, int n){
    produceSubstances_sw.reset();
    // increases the concentration of substances at the location of the cells
    float sideLength = 1/(float)L; // length of a side of a diffusion voxel
    int c, i1, i2, i3;
    for (c=0; c< n; c++) {
        i1 = std::min((int)floor(posAll[c][0]/sideLength),(L-1));
        i2 = std::min((int)floor(posAll[c][1]/sideLength),(L-1));
        i3 = std::min((int)floor(posAll[c][2]/sideLength),(L-1));
        if (typesAll[c]==1) {
            Conc[0][i1][i2][i3]+=0.1;
            if (Conc[0][i1][i2][i3]>1) {
                Conc[0][i1][i2][i3]=1;
            }
        } else {
            Conc[1][i1][i2][i3]+=0.1;
            if (Conc[1][i1][i2][i3]>1) {
                Conc[1][i1][i2][i3]=1;
            }
        }
    }
    produceSubstances_sw.mark();
}
```

**Changed code line(s):** 1, 5, 6, 8, 9, 10, 11, 13, 14, 17, 18, 19

#### Optimized Code

```
static void produceSubstances(int L, float Conc[2][L][L][L], float posAll[][3], int* typesAll, int n) {
    produceSubstances_sw.reset();
    // increases the concentration of substances at the location of the cells

    const int auxL=L;
    --L;
    int c,i[3] __attribute__((aligned(32))); //i array aligned
    omp_set_num_threads(240);

    #pragma omp parallel for schedule(static) private(i,c)
    for (c=0; c< n; ++c) {
        __assume_aligned((int*)i, 32);
        __assume_aligned((float*)posAll, 64);
        i[0] = std::min((int)floor(posAll[c][0]*auxL),L);
        i[1] = std::min((int)floor(posAll[c][1]*auxL),L);
        i[2] = std::min((int)floor(posAll[c][2]*auxL),L);

        if (typesAll[c]==1) {
            (Conc[0][i[0]][i[1]][i[2]]>0.9)?
            Conc[0][i[0]][i[1]][i[2]]=1 :
            Conc[0][i[0]][i[1]][i[2]]+=0.1;
        } else {
            (Conc[1][i[0]][i[1]][i[2]]>0.9)?
            Conc[1][i[0]][i[1]][i[2]]=1 :
            Conc[1][i[0]][i[1]][i[2]]+=0.1;
        }
    }
    produceSubstances_sw.mark();
}
```

**Changed code line(s):** 1,6-11, 13-17, 20-22, 24-26

#### Optimization Notes

The optimized function inputs order changed to define in the header of the function the size of the arrays. This way we can avoid the use of pointers and work directly with the arrays so the compiler knows beforehand the size of the elements we pass into the function.

The original code used pointers to initialize the arrays, but since those arrays are static (the length is static and doesn't change) it is easier and faster to declare it directly as arrays with a defined size without using pointers.

The optimized code includes the use of one OpenMP parallel for function with static scheduling to distribute the load to the other cores equally. This is because this function can be executed in parallel without affecting the result, that is, every cycle of the for loop can be executed individually without affecting the next iterations.

During the execution of the main code (described in a later section), this function is called multiple times, each time increasing the value of n. So while it is possible to make the function parallel, this would only be useful while the values of n are low (less than 10000). Otherwise the added overhead may slow the code.

The operation  $posAll[c][0]/sideLength$  is the same as  $(posAll[c][0]/1/L)$  or  $(posAll[c][0]*L)$ . Since  $posAll[c][0]/sideLength$  generates the same result with fewer calculations, it is a benefit.

The use of the L variable was changed in optimization. The original code uses the operation L-1 three times, using three extra operations. In the optimized version we simply decrement previously the L variable. The auxL constant (used to optimize as a read-only variable is faster than a read/write variable) stores the original value of L and will not change during the execution of this function.

Similarly the method for using the Conc variable was changed to optimize the code. In the original code the if clause will increment the Conc variable by 0.1, then check if the value is greater than 1 and if true, limit its value to 1, making the previous addition useless. The solution was to check if the value of Conc is greater than 0.9, and if true set the value of Conc to 1; if false then increment Conc by 0.1.

### runDiffusionStep

This function has two parts. The first part of the function copies the Conc variable to tempConc. The second part iterates through the Conc array checking upper and 3D boundaries.

#### Original Code

```
static void runDiffusionStep(float**** Conc, int L, float D) {
    runDiffusionStep_sw.reset();
    // computes the changes in substance concentrations due to diffusion

    int i1,i2,i3, subInd;
    float tempConc[2][L][L][L];
    for (i1 = 0; i1 < L; i1++) {
        for (i2 = 0; i2 < L; i2++) {
            for (i3 = 0; i3 < L; i3++) {
                tempConc[0][i1][i2][i3] = Conc[0][i1][i2][i3];
                tempConc[1][i1][i2][i3] = Conc[1][i1][i2][i3];
            }
        }
    }

    int xUp, xDown, yUp, yDown, zUp, zDown;

    for (i1 = 0; i1 < L; i1++) {
        for (i2 = 0; i2 < L; i2++) {
            for (i3 = 0; i3 < L; i3++) {
                xUp = (i1+1);
                xDown = (i1-1);
                yUp = (i2+1);
                yDown = (i2-1);
                zUp = (i3+1);
                zDown = (i3-1);
                for (subInd = 0; subInd < 2; subInd++) {
                    if (xUp<L) {
                        Conc[subInd][i1][i2][i3] += (tempConc[subInd][xUp][i2][i3]-tempConc[subInd][i1][i2][i3])*D/6;
                    }
                    if (xDown>=0) {
                        Conc[subInd][i1][i2][i3] += (tempConc[subInd][xDown][i2][i3]-tempConc[subInd][i1][i2][i3])*D/6;
                    }
                    if (yUp<L) {
                        Conc[subInd][i1][i2][i3] += (tempConc[subInd][i1][yUp][i3]-tempConc[subInd][i1][i2][i3])*D/6;
                    }
                    if (yDown>=0) {
                        Conc[subInd][i1][i2][i3] += (tempConc[subInd][i1][yDown][i3]-tempConc[subInd][i1][i2][i3])*D/6;
                    }
                    if (zUp<L) {
                        Conc[subInd][i1][i2][i3] += (tempConc[subInd][i1][i2][zUp]-tempConc[subInd][i1][i2][i3])*D/6;
                    }
                    if (zDown>=0) {
                        Conc[subInd][i1][i2][i3] += (tempConc[subInd][i1][i2][zDown]-tempConc[subInd][i1][i2][i3])*D/6;
                    }
                }
            }
        }
    }
    runDiffusionStep_sw.mark();
}
```

**Changed code line(s):** 1, 5, 9-12, 16, 21-45

#### Optimized Code

```
static void runDiffusionStep(int L, float Conc[2][L][L][L], float D) {
    runDiffusionStep_sw.reset();
    // computes the changes in substance concentrations due to diffusion
    int i1,i2,i3,auxx;
    const float auxD=D/6;
    const int auxL=L-1;
    float tempConc[2][L][L][L] __attribute__((aligned(64)));
    omp_set_num_threads(240);
    #pragma omp parallel
    {
        #pragma omp for schedule(static) private(i1,i2) collapse(2)
        for (i1 = 0; i1 < L; ++i1) {
            for (i2 = 0; i2 < L; ++i2) {
                memcpy(tempConc[0][i1][i2],
                    Conc[0][i1][i2],sizeof(float)*L);
                memcpy(tempConc[1][i1][i2],
                    Conc[1][i1][i2],sizeof(float)*L);
            }
        }

        #pragma omp for schedule(static) private(i1,i2,i3) collapse(2)
        for (i1 = 0; i1 < L; ++i1) {
            for (i2 = 0; i2 < L; ++i2) {
                Conc[0][i1][i2][0] += (tempConc[0][i1][i2][1]-
                    tempConc[0][i1][i2][0])*auxD;
                Conc[1][i1][i2][0] += (tempConc[1][i1][i2][1]-
                    tempConc[1][i1][i2][0])*auxD;
                for (i3 = 1; i3 < auxL; ++i3) {
                    const float aux=tempConc[0][i1][i2][i3];
                    const float aux1=tempConc[1][i1][i2][i3];
                    __assume_aligned((float*)tempConc[0], 64);
                    __assume_aligned((float*)tempConc[1], 64);
                    __assume_aligned((float*)Conc[0], 64);
                    __assume_aligned((float*)Conc[1], 64);
                    if (i1<auxL) {
                        Conc[0][i1][i2][i3] +=
                            (tempConc[0][(i1+1)][i2][i3]-aux)*auxD;
                        Conc[1][i1][i2][i3] +=
                            (tempConc[1][(i1+1)][i2][i3]-aux1)*auxD;
                    }
                    if (i1>0) {
                        Conc[0][i1][i2][i3] +=
                            (tempConc[0][(i1-1)][i2][i3]-aux)*auxD;
                    }
                }
            }
        }
    }
}
```



```

    for (i2 = 0; i2 < L; ++i2) {
        for (i3 = 0; i3 < L; ++i3) {
            __assume_aligned((float*)Conc[0][i1][i2], 64);
            __assume_aligned((float*)Conc[1][i1][i2], 64);
            Conc[0][i1][i2][i3] = Conc[0][i1][i2][i3]*muu;
            Conc[1][i1][i2][i3] = Conc[1][i1][i2][i3]*muu;
        }
    }
}
runDecayStep_sw.mark();
}

```

**Changed code line(s):** 1, 4, 6-8, 12-15

#### Optimization Notes

There are a few simple changes to optimize this function.

The input arrays of the function are defined on the header. The variables are optimized as in previous functions (using constants when possible and aligning the arrays). Loops are collapsed under the OpenMP pragma header. The use of #pragma simd loop. This loop can be vectorized with SIMD instructions, which improve the execution time on the optimized code. The use of SIMD instructions gives the possibility of executing multiple operations at the same time.

Additional optimizations that could be made are to use the Intel MKL functions to their fullest possibilities. The Intel MKL functions are optimized for large arrays of data, so if the Conc array is flattened to one dimension, the Intel MKL will perform faster.

#### cellMovementAndDuplication

This function is used to generate the random movement of each cell and each cell's duplicate. However, not every cell duplicates.

#### Original Code

```

static int cellMovementAndDuplication(float** posAll, float* pathTraveled,
int* typesAll, int* numberDivisions, float pathThreshold, int
divThreshold, int n) {
    cellMovementAndDuplication_sw.reset();
    int c;
    currentNumberCells = n;
    float currentNorm;
    float currentCellMovement[3];
    float duplicatedCellOffset[3];
    for (c=0; c<n; c++) {
        // random cell movement
        currentCellMovement[0]=RandomFloatPos()-0.5;
        currentCellMovement[1]=RandomFloatPos()-0.5;
        currentCellMovement[2]=RandomFloatPos()-0.5;
        currentNorm = getNorm(currentCellMovement);
        posAll[c][0]+=0.1*currentCellMovement[0]/currentNorm;
        posAll[c][1]+=0.1*currentCellMovement[1]/currentNorm;
        posAll[c][2]+=0.1*currentCellMovement[2]/currentNorm;
        pathTraveled[c]+=0.1;
        // cell duplication if conditions fulfilled
        if (numberDivisions[c]<divThreshold) {
            if (pathTraveled[c]>pathThreshold) {
                pathTraveled[c]=pathThreshold;
                numberDivisions[c]++; // update number of divisions this cell has undergone
                currentNumberCells++; // update number of cells in
                the simulation
                numberDivisions[currentNumberCells-1]=numberDivisions[c]; // update number of divisions the duplicated cell has undergone
                typesAll[currentNumberCells-1]=typesAll[c];
                // assign type of duplicated cell (opposite to current cell)

                // assign location of duplicated cell
                duplicatedCellOffset[0]=RandomFloatPos()-0.5;
                duplicatedCellOffset[1]=RandomFloatPos()-0.5;
                duplicatedCellOffset[2]=RandomFloatPos()-0.5;
                currentNorm = getNorm(duplicatedCellOffset);
                posAll[currentNumberCells-1][0]=posAll[c][0]+0.05*duplicatedCellOffset[0]/currentNorm;
                posAll[currentNumberCells-1][1]=posAll[c][1]+0.05*duplicatedCellOffset[1]/currentNorm;
                posAll[currentNumberCells-1][2]=posAll[c][2]+0.05*duplicatedCellOffset[2]/currentNorm;
            }
        }
    }
    cellMovementAndDuplication_sw.mark();
    return currentNumberCells;
}

```

**Changed code line(s):** 1, 12-14, 16-18, 21, 22, 24-34, 36-38

#### Optimized Code

```

static int cellMovementAndDuplication(float posAll[][3], float* pathTraveled,
int* typesAll, int* numberDivisions, float pathThreshold, int
divThreshold, int n) {
    cellMovementAndDuplication_sw.reset();
    int c, currentNumberCells = n;
    float currentNorm;
    unsigned int seed=rand();
    float currentCellMovement[3] __attribute__((aligned(64)));
    float duplicatedCellOffset[3] __attribute__((aligned(64)));
    omp_set_num_threads(240);

#pragma omp parallel for simd schedule (static) shared(posAll) private(c,currentNorm,currentCellMovement)

    for (c=0; c<n; ++c) {
        // random cell movement
        RandomFloatPos(currentCellMovement, seed+c);
        currentNorm = getNorm(currentCellMovement)*10;
        __assume_aligned((float*)posAll, 64);
        __assume_aligned((float*)currentCellMovement, 64);
        posAll[c][0:3]+=currentCellMovement[0:3]/currentNorm;
        __assume_aligned((float*)pathTraveled, 64);
        pathTraveled[c]+=0.1;
    }
    seed=rand();
    for (c=0; c<n; ++c) {
        if ((numberDivisions[c]<divThreshold) && (pathTraveled[c]>pathThreshold)) {
            pathTraveled[c]=pathThreshold;
            ++numberDivisions[c]; // update number of divisions this cell has undergone
            numberDivisions[currentNumberCells]=numberDivisions[c]; // update number of divisions the duplicated cell has undergone
            typesAll[currentNumberCells]=typesAll[c]; // assign type of
            duplicated cell (opposite to current cell)
        }
    }
}

```

```

        // assign location of duplicated cell
        RandomFloatPos(duplicatedCellOffset,seed+c); //The seed+c value will be different for each thread and iteration, this way the random
        currentNorm = getNorm(duplicatedCellOffset)*20;
        __assume_aligned((float*)posAll, 64);
        __assume_aligned((float*)duplicatedCellOffset, 64);
        posAll[currentNumberCells][0:3]=posAll[c][0:3]+duplicatedCellOffset[0:3]/currentNorm;
        ++currentNumberCells; // update number of cells in the simulation
    }
}
cellMovementAndDuplication_sw.mark();
return currentNumberCells;
}

```

**Changed code line(s)**: 1, 8-12, 16-21, 26, 28, 29, 34, 36-39

### Optimization Notes

The optimized code for cellMovementAndDuplication has two portions that were optimized in the same way as described for previous functions:

RandomFloatPos() functions. The optimized function can be multithreaded and returns the whole position array (the 3D position array). The code lines incorporating currentNorm can be vectorized with SIMD and also mathematically simplified.

Additionally, the optimized code updates the header and aligns the arrays.

The new function RandomFloatPos() requires a random generated seed before calling seed=rand(), and each thread needs a different value of this seed. To achieve this, every thread increases the random generated seed value with their corresponding for loop iteration. This way the number is both random and different for each thread.

To multithread this function I divided it into two big loops. The first (random cell movement) can be parallelized without problems. The second loop (starting with seed=rand();) must be executed in a single thread, because every iteration depends on the previous iteration's values.

### runDiffusionClusterStep

This function is used to determine cell movement based on the substance gradients.

### Original Code

```

static void runDiffusionClusterStep(float**** Conc, float** movVec, float** posAll, int* typesAll, int n, int L, float speed) {
    runDiffusionClusterStep_sw.reset();
    // computes movements of all cells based on gradients of the two substances
    float sideLength = 1/(float)L; // length of a side of a diffusion voxel

    float gradSub1[3];
    float gradSub2[3];
    float normGrad1, normGrad2;
    int c, i1, i2, i3, xUp, xDown, yUp, yDown, zUp, zDown;

    for (c = 0; c < n; c++) {
        i1 = std::min((int)floor(posAll[c][0]/sideLength), (L-1));
        i2 = std::min((int)floor(posAll[c][1]/sideLength), (L-1));
        i3 = std::min((int)floor(posAll[c][2]/sideLength), (L-1));

        xUp = std::min((i1+1), L-1);
        xDown = std::max((i1-1), 0);
        yUp = std::min((i2+1), L-1);
        yDown = std::max((i2-1), 0);
        zUp = std::min((i3+1), L-1);
        zDown = std::max((i3-1), 0);

        gradSub1[0] = (Conc[0][xUp][i2][i3]-Conc[0][xDown][i2][i3])/(sideLength*(xUp-xDown));
        gradSub1[1] = (Conc[0][i1][yUp][i3]-Conc[0][i1][yDown][i3])/(sideLength*(yUp-yDown));
        gradSub1[2] = (Conc[0][i1][i2][zUp]-Conc[0][i1][i2][zDown])/(sideLength*(zUp-zDown));
        gradSub2[0] = (Conc[1][xUp][i2][i3]-Conc[1][xDown][i2][i3])/(sideLength*(xUp-xDown));
        gradSub2[1] = (Conc[1][i1][yUp][i3]-Conc[1][i1][yDown][i3])/(sideLength*(yUp-yDown));
        gradSub2[2] = (Conc[1][i1][i2][zUp]-Conc[1][i1][i2][zDown])/(sideLength*(zUp-zDown));
        normGrad1 = getNorm(gradSub1);
        normGrad2 = getNorm(gradSub2);
        if ((normGrad1>0)&&(normGrad2>0)) {
            movVec[c][0]=typesAll[c]*(gradSub1[0]/normGrad1-gradSub2[0]/normGrad2)*speed;
            movVec[c][1]=typesAll[c]*(gradSub1[1]/normGrad1-gradSub2[1]/normGrad2)*speed;
            movVec[c][2]=typesAll[c]*(gradSub1[2]/normGrad1-gradSub2[2]/normGrad2)*speed;
        } else {
            movVec[c][0]=0;
            movVec[c][1]=0;
            movVec[c][2]=0;
        }
    }
    runDiffusionClusterStep_sw.mark();
}

```

**Changed code line(s)**: 1-2, 5-10, 13-39

### Optimized Code

```

static void runDiffusionClusterStep(int L, float Conc[2][L][L][L], float movVec[][3], float posAll[][3], int* typesAll, int n, float speed) {
    runDiffusionClusterStep_sw.reset();
    // computes movements of all cells based on gradients of the two substances
    const float auxL=L;
    --L;

    float gradSub[6] __attribute__((aligned(64)));
    float aux[3] __attribute__((aligned(64)));
    int i[3] __attribute__((aligned(32)));
    float normGrad[2] __attribute__((aligned(64)));
    int c, xUp, xDown, yUp, yDown, zUp, zDown;
    omp_set_num_threads(240);

#pragma omp parallel for schedule(static) private(i,xUp,xDown,yUp,yDown,zUp,zDown,gradSub,normGrad,aux) if (n>240)
    for (c = 0; c < n; ++c) {
        __assume_aligned((int*)i, 32);
        __assume_aligned((float*)posAll, 64);
        __assume_aligned((float*)gradSub, 64);
        __assume_aligned((float*)normGrad, 64);
        __assume_aligned((float*)movVec, 64);
        __assume_aligned((float*)typesAll, 64);
        i[0:3] = std::min((int)floor(posAll[c][0:3]*auxL), L)-1;
        xDown = std::max(i[0], 0);
        yDown = std::max(i[1], 0);
        zDown = std::max(i[2], 0);
        xUp = std::min((i[0]+2), L);
        yUp = std::min((i[1]+2), L);
        zUp = std::min((i[2]+2), L);
    }
}

```

```

aux[0]=auxL/((xUp-xDown));
aux[1]=auxL/((yUp-yDown));
aux[2]=auxL/((zUp-zDown));
gradSub[0] = (Conc[0][xUp][i[1]][i[2]]-Conc[0][xDown][i[1]][i[2]])*aux[0];
gradSub[1] = (Conc[0][i[0]][yUp][i[2]]-Conc[0][i[0]][yDown][i[2]])*aux[1];
gradSub[2] = (Conc[0][i[0]][i[1]][zUp]-Conc[0][i[0]][i[1]][zDown])*aux[2];
normGrad[0] = getNorm(gradSub);
if (normGrad[0]>0){
    gradSub[3] = (Conc[1][i[0]][yUp][i[2]]-Conc[1][i[0]][yDown][i[2]])*aux[1];
    gradSub[4] = (Conc[1][xUp][i[1]][i[2]]-Conc[1][xDown][i[1]][i[2]])*aux[0];
    gradSub[5] = (Conc[1][i[0]][i[1]][zUp]-Conc[1][i[0]][i[1]][zDown])*aux[2];
    normGrad[1] = getNorm(gradSub+3);
    if ( normGrad[1]>0) {
        movVec[c][0:3]=typesAll[c]*(gradSub[0:3]/normGrad[0]-gradSub[3:3]/normGrad[1])*speed;
    } else movVec[c][0:3]=0;
}
}
runDiffusionClusterStep_sw.mark();
}

```

**Changed code line(s)**: 1, 5-10, 12-14, 12-43

### Optimization Notes

This function is optimized using some small simplifications, vectorization and multithreading, much in the same way other functions were optimized.

In the header the dimensions of the input arrays are specified. It is array aligned. Replacing variables with constants whenever possible. OpenMP parallelization. Simplification of operations. Simplification of vectorization.

The if condition can filter some of the operations out. An example is the calculation of normGrad[1] is unnecessary if normGrad[0] is not > 0. Using the if condition results in fewer comparisons and fewer useless operations are calculated.

This function can be further improved using more SIMD optimizations. However, to use the SIMD operations, the max and min functions must be implemented differently.

### getEnergy

This function computes the energy measure of a subvolume of cells by assuming uniform distribution within the entire volume and determining a volume of a target number of cells.

### Original Code

```

Original Code
static float getEnergy(float** posAll, int* typesAll, int n, float spatialRange, int targetN) {
    getEnergy_sw.reset();
    // Computes an energy measure of clusteredness within a subvolume. The
    // size of the subvolume is computed by assuming roughly uniform
    // distribution within the whole volume, and selecting a volume
    // comprising approximately targetN cells.
    int i1, i2;
    float currDist;
    float** posSubvol=0; // array of all 3 dimensional cell positions
    posSubvol = new float*[n];
    int typesSubvol[n];
    float subVolMax = pow(float(targetN)/float(n),1.0/3.0)/2;
    if(quiet < 1)
        printf("subVolMax: %f\n", subVolMax);
    int nrCellsSubVol = 0;
    float intraClusterEnergy = 0.0;
    float extraClusterEnergy = 0.0;
    float nrSmallDist=0.0;

    for (i1 = 0; i1 < n; i1++) {
        posSubvol[i1] = new float[3];
        if ((fabs(posAll[i1][0]-0.5)<subVolMax) && (fabs(posAll[i1][1]-0.5)<subVolMax) && (fabs(posAll[i1][2]-0.5)<subVolMax)) {
            posSubvol[nrCellsSubVol][0] = posAll[i1][0];
            posSubvol[nrCellsSubVol][1] = posAll[i1][1];
            posSubvol[nrCellsSubVol][2] = posAll[i1][2];
            typesSubvol[nrCellsSubVol] = typesAll[i1];
            nrCellsSubVol++;
        }
    }

    for (i1 = 0; i1 < nrCellsSubVol; i1++) {
        for (i2 = i1+1; i2 < nrCellsSubVol; i2++) {
            currDist = getL2Distance(posSubvol[i1][0],posSubvol[i1][1],posSubvol[i1][2],posSubvol[i2][0],
            posSubvol[i2][1],posSubvol[i2][2]);
            if (currDist<spatialRange) {
                nrSmallDist = nrSmallDist+1;//currDist/spatialRange;
                if (typesSubvol[i1]*typesSubvol[i2]>0) {
                    intraClusterEnergy = intraClusterEnergy+fmin(100.0,spatialRange/currDist);
                } else {
                    extraClusterEnergy = extraClusterEnergy+fmin(100.0,spatialRange/currDist);
                }
            }
        }
    }
    float totalEnergy = (extraClusterEnergy-intraClusterEnergy)/(1.0+100.0*nrSmallDist);
    getEnergy_sw.mark();
    return totalEnergy;
}

```

**Changed code line(s)**: 1, 22, 34, 35, 37-41, 46, 48

### Optimized Code

```

static float getEnergy(float posAll[][3], int* typesAll, int n, float spatialRange, int targetN) {
    getEnergy_sw.reset();
    // Computes an energy measure of clusteredness within a subvolume. The
    // size of the subvolume is computed by assuming roughly uniform
    // distribution within the whole volume, and selecting a volume
    // comprising approximately targetN cells.
    int i1, i2;
    float currDist;
    float posSubvol[n][3] __attribute__((aligned(64)));
    int typesSubvol[n] __attribute__((aligned(64)));
    const float subVolMax = pow(float(targetN)/float(n),1.0/3.0)/2;
    if(quiet < 1)printf("subVolMax: %f\n", subVolMax);

    int nrCellsSubVol = 0;
    float intraClusterEnergy = 0.0;
    float extraClusterEnergy = 0.0;
    float nrSmallDist=0.0;

```



```

for (i1 = 0; i1 < n; ++i1) {
    __assume_aligned((float*)posAll, 64);
    if ((fabs(posAll[i1][0]-0.5)<subVolMax) && (fabs(posAll[i1][1]-0.5)<subVolMax) && (fabs(posAll[i1][2]-
        0.5)<subVolMax)) {
        __assume_aligned((float*)posSubvol[nrCellsSubVol], 64);
        __assume_aligned((float*)typesAll, 64);
        __assume_aligned((int*)typesSubvol, 64);
        posSubvol[nrCellsSubVol][0] = posAll[i1][0];
        posSubvol[nrCellsSubVol][1] = posAll[i1][1];
        posSubvol[nrCellsSubVol][2] = posAll[i1][2];
        typesSubvol[nrCellsSubVol] = typesAll[i1];
        ++nrCellsSubVol;
    }
}
omp_set_num_threads(240);
#pragma omp parallel for schedule(static) reduction(+:nrSmallDist,intraClusterEnergy,extraClusterEnergy) private(i1,i2,currDist)
for (i1 = 0; i1 < nrCellsSubVol; ++i1) {
    for (i2 = i1+1; i2 < nrCellsSubVol; ++i2) {
        currDist = getL2Distance(posSubvol[i1],posSubvol[i2]);
        if (currDist<spatialRange) {
            ++nrSmallDist; //currDist/spatialRange;
            (typesSubvol[i1]*typesSubvol[i2]>0)? intraClusterEnergy += fmin(100.0,spatialRange/currDist) :
            extraClusterEnergy += fmin(100.0,spatialRange/currDist);
        }
    }
}
getEnergy_sw.mark();
return (extraClusterEnergy-intraClusterEnergy)/(1.0+100.0*nrSmallDist);
}

```

**Changed code line(s)**: 1, 9-11, 19, 22-24, 32, 33, 36-40

### Optimization Notes

Some of the things in this code that were easy targets for optimization are the `new` constructors, some variables that are constants, and the fact that the second loop depends on the first. The first loop isn't parallelizable since it depends on the value of the `nrCellsSubVol` and the `if` with three conditions inside this loop makes it even more difficult to parallelize. The second loop, however, can be parallelized without problems.

The specific optimizations done to this function are much the same as in the other functions:

The dimensions of the input arrays are specified in the header. Use of constants when possible. Array aligned. OpenMP parallelization on the second `for` loop using a reduction.

The next step optimization would be to parallelize the first loop if condition. It is difficult but possible and would result in provable gains.

### getCriterion

This function is used to determine if the cells in the subvolume are in clusters.

**NOTE:** Both the original code and the optimized code are truncated versions, showing only the code portions relevant to this paper.

### Original Code

```

static bool getCriterion(float** posAll, int* typesAll, int n, float spatialRange, int targetN) {
    getCriterion_sw.reset();
    // Returns 0 if the cell locations within a subvolume of the total
    // system, comprising approximately targetN cells, are arranged as clusters, and 1 otherwise.
    int i1, i2;
    int nrClose=0; // number of cells that are close (i.e. within a distance of spatialRange)
    float currDist;
    int sameTypeClose=0; // number of cells of the same type, and that are close (i.e. within a distance of spatialRange)
    int diffTypeClose=0; //number of cells of opposite types, and that are close (i.e. within a distance of spatialRange)
    float** posSubvol=0; // array of all 3 dimensional cell positions in the subcube
    posSubvol = new float*[n];
    int typesSubvol[n];
    float subVolMax = pow(float(targetN)/float(n),1.0/3.0)/2;
    int nrCellsSubVol = 0;

    // the locations of all cells within the subvolume are copied to array PosSubvol
    for (i1 = 0; i1 < n; i1++) {
        posSubvol[i1] = new float[3];
        if ((fabs(posAll[i1][0]-0.5)<subVolMax) && (fabs(posAll[i1][1]-0.5)<subVolMax) && (fabs(posAll[i1][2]-
            0.5)<subVolMax)) {
            posSubvol[nrCellsSubVol][0] = posAll[i1][0];
            posSubvol[nrCellsSubVol][1] = posAll[i1][1];
            posSubvol[nrCellsSubVol][2] = posAll[i1][2];
            typesSubvol[nrCellsSubVol] = typesAll[i1];
            nrCellsSubVol++;
        }
    }
}
[section of truncated code]

for (i1 = 0; i1 < nrCellsSubVol; i1++) {
    for (i2 = i1+1; i2 < nrCellsSubVol; i2++) {
        currDist = getL2Distance(posSubvol[i1][0],posSubvol[i1][1],posSubvol[i1][2],posSubvol[i2][0], posSubvol[i2][1],posSubvol[i2][2]);
        if (currDist<spatialRange) {
            nrClose++;
            if (typesSubvol[i1]*typesSubvol[i2]<0) {
                diffTypeClose++;
            } else {
                sameTypeClose++;
            }
        }
    }
}
[section of truncated code]
}

```

**Changed code line(s)**: 9, 10, 32-39

### Optimized Code

```

static bool getCriterion(float posAll[][3], int* typesAll, int n, float spatialRange, int targetN) {
    getCriterion_sw.reset();
    // Returns 0 if the cell locations within a subvolume of the total
    // system, comprising approximately targetN cells, are arranged as clusters, and 1 otherwise.
    int i1, i2;
    int nrClose=0; // number of cells that are close (i.e. within a distance of spatialRange)
    int sameTypeClose=0; // number of cells of the same type, and that are close (i.e. within a distance of spatialRange)

```

```

int diffTypeClose=0; // number of cells of opposite types, and that are close (i.e. within a distance of spatialRange)
float posSubVol[n][3] __attribute__((aligned(64)));
int typesSubVol[n] __attribute__((aligned(64)));
const float subVolMax = pow(float(targetN)/float(n),1.0/3.0)/2;
int nrCellsSubVol = 0;

// the locations of all cells within the subvolume are copied to array posSubVol

for (i1 = 0; i1 < n; ++i1) {
    __assume_aligned((float*)posAll, 64);
    if ((fabs(posAll[i1][0]-0.5)<subVolMax) && (fabs(posAll[i1][1]-0.5)<subVolMax) && (fabs(posAll[i1][2]
        -0.5)<subVolMax)) {
        __assume_aligned((float*)posSubVol[nrCellsSubVol], 64);
        __assume_aligned((float*)typesAll, 64);
        __assume_aligned((int*)typesSubvol, 64);
        posSubVol[nrCellsSubVol][0] = posAll[i1][0];
        posSubVol[nrCellsSubVol][1] = posAll[i1][1];
        posSubVol[nrCellsSubVol][2] = posAll[i1][2];
        typesSubVol[nrCellsSubVol] = typesAll[i1];
        ++nrCellsSubVol;
    }
}

[section of truncated code]

omp_set_num_threads(240);
#pragma omp parallel for schedule(static)
reduction(+:nrClose,diffTypeClose,sameTypeClose) private(i1,i2)
for (i1 = 0; i1 < nrCellsSubVol; ++i1) {
    for (i2 = i1+1; i2 < nrCellsSubVol; ++i2) {
        if (getL2Distance(posSubVol[i1],posSubVol[i2])<spatialRange) {
            ++nrClose;
            (typesSubVol[i1]*typesSubVol[i2]<0) ? ++diffTypeClose: ++sameTypeClose;
        }
    }
}

[section of truncated code]
}

```

**Changed code line(s):** 1, 9-11, 17, 20-22, 33-35, 38-40

### Optimization Notes

The optimizable parts of this function include the variable declaration and initialization. The first loop is less optimizable, but optimizing the second loop can make up the difference.

The specific optimizations done to this function are much the same as in the other functions:

The dimensions of the input arrays are specified in the header. Use of constants when possible. OpenMP parallelization on the second `for` loop using a reduction.

For further optimization, it may be possible to parallelize the first loop if condition. Additionally, flattening the `posAll` and `posSubVol` arrays can lead to a faster execution.

### Main `Variable Declaration`

The main code is large so for this paper it is divided into meaningful sections. This section deals with the code and optimized code for the variable declarations.

#### Original Code

```

int i,c,d;
int i1, i2, i3, i4;
float energy; // value that quantifies the quality of the cell clustering output. The smaller this value, the better the clustering
float** posAll=0; // array of all 3 dimensional cell positions
posAll = new float*[finalNumberCells];
float** currMov=0; // array of all 3 dimensional cell movements at the last time point
currMov = new float*[finalNumberCells]; // array of all cell movements in the last time step
float zeroFloat = 0.0;
float pathTraveled[finalNumberCells]; // array keeping track of length of path traveled until cell divides
int numberDivisions[finalNumberCells]; //array keeping track of number of division a cell has undergone
int typesAll[finalNumberCells]; // array specifying cell type (+1 or -1)

```

**Changed code line(s):** 2, 4-7,

#### Optimized Code

```

int i,c,d,i1;
float energy; // value that quantifies the quality of the cell clustering output. The smaller this value, the better the clustering
float posAll[finalNumberCells][3] __attribute__((aligned(64)));
float currMov[finalNumberCells][3] __attribute__((aligned(64))); //array of
// all cell movements in the last time step
float pathTraveled[finalNumberCells] __attribute__((aligned(64))); // array
// keeping track of length of path traveled until cell divides
int numberDivisions[finalNumberCells] __attribute__((aligned(64))); //array
// keeping track of number of division a cell has undergone
int typesAll[finalNumberCells] __attribute__((aligned(64))); // array
// specifying cell type (+1 or -1)
float Conc[2][L][L][L] __attribute__((aligned(64)));

```

**Changed code line(s):** 3-12

### Optimization Notes

There were many ways to optimize the code in this small section.

Avoid the use of `new` constructors. Delete useless variables (for example, `zeroFloat`). Avoid using pointers when we can declare an array of a known size and static. Memory alignment; memory access is faster though it can use more memory. Define the variable value at initialization time; this is faster than defining the value later.

### Main `Variable Initialization`

The main code is large so for this paper it is divided into meaningful sections. This section deals with the code and optimized code for the variable initialization. The size and values of the variables are declared in this code, including the largest arrays `currMov`, `posAll`, `pathTraveled`, and `Conc`.

#### Original Code

```
// Initialization of the various arrays
for (i1 = 0; i1 < finalNumberCells; i1++) {
    currMov[i1] = new float[3];
    posAll[i1] = new float[3];
    pathTraveled[i1] = zeroFloat;
    pathTraveled[i1] = 0;
    for (i2 = 0; i2 < 3; i2++) {
        currMov[i1][i2] = zeroFloat;
        posAll[i1][i2] = 0.5;
    }
}
// create 3D concentration matrix
float*** Conc;
Conc = new float***[L];
for (i1 = 0; i1 < 2; i1++) {
    Conc[i1] = new float**[L];
    for (i2 = 0; i2 < L; i2++) {
        Conc[i1][i2] = new float*[L];
        for (i3 = 0; i3 < L; i3++) {
            Conc[i1][i2][i3] = new float[L];
            for (i4 = 0; i4 < L; i4++) {
                Conc[i1][i2][i3][i4] = zeroFloat;
            }
        }
    }
}
}
```

**Changed code line(s):** 3-5, 7-10, 13, 14, 16, 18, 20, 22

#### Optimized Code

```
// Initialization of the various arrays
omp_set_num_threads(240);
#pragma omp parallel
{
    #pragma omp for simd schedule (static) private (i1) nowait
    for (i1 = 0; i1 < finalNumberCells; ++i1) {
        __assume_aligned((float*)posAll, 64);
        __assume_aligned((float*)currMov[i1], 64);
        __assume_aligned((float*)pathTraveled, 64);
        currMov[i1][0:3]=0;
        posAll[i1][0]=0.5;
        posAll[i1][1]=0.5;
        posAll[i1][2]=0.5;
        pathTraveled[i1] = 0;
    }
    #pragma omp for schedule (static) private (i1,c,d) collapse(3)
    for (i1 = 0; i1 < 2; ++i1) {
        for (c = 0; c < L; ++c) {
            for (d = 0; d < L; ++d) {
                Conc[i1][c][d][0:L]=0;
            }
        }
    }
}
}
```

**Changed code line(s):** 2-5, 7-13, 16, 20

#### Optimization Notes

There are three ways the code in this portion was optimized. The most basic is the reduction of the for loops from four to three. The other optimizations are memory alignment and the OpenMP thread parallelization with SIMD.

#### Main "Phase 1"

The main code is large, so for this paper it is divided into meaningful sections. This section deals with the first steps of the simulation.

#### Original Code

```
// Phase 1: Cells move randomly and divide until final number of cells is reached
while (n<finalNumberCells) {
    produceSubstances(Conc, posAll, typesAll, L, n); // Cells produce
    // substances. Depending on the cell type, one of the two substances is produced.
    runDiffusionStep(Conc, L, D); // Simulation of substance diffusion
    runDecayStep(Conc, L, mu);
    n = cellMovementAndDuplication(posAll, pathTraveled, typesAll, numberDivisions, pathThreshold, divThreshold, n);

    for (c=0; c<n; c++) {
        // boundary conditions
        for (d=0; d<3; d++) {
            if (posAll[c][d]<0) {posAll[c][d]=0;}
            if (posAll[c][d]>1) {posAll[c][d]=1;}
        }
    }
}
}
```

**Changed code line(s):** 4, 6, 7, 14

#### Optimized Code

```
// Phase 1: Cells move randomly and divide until final number of cells is reached
while (n<finalNumberCells) {
    produceSubstances(L,Conc, posAll, typesAll, n); // Cells produce substances. Depending on the cell type, one of the two substances is produced.
    runDiffusionStep(L,Conc, D); // Simulation of substance diffusion
    runDecayStep(L,Conc, mu);
    n = cellMovementAndDuplication(posAll, pathTraveled, typesAll, numberDivisions, pathThreshold, divThreshold, n);
    omp_set_num_threads(240);
    #pragma omp parallel for simd schedule (static) private (c,d) if (n>500)
    for (c=0; c<n; ++c) {
        // boundary conditions
        for (d=0; d<3; d++) {
            if (posAll[c][d]<0) {
                posAll[c][d]=0;
            }else if (posAll[c][d]>1) posAll[c][d]=1;
        }
    }
}
}
```

**Changed code line(s):** 4-6, 8, 9, 15

#### Optimization Notes

There are two optimizations used in this portion of the code: OpenMP thread parallelization with SIMD and if clause reduction.

The OpenMP thread parallelization with SIMD is a good change because the schedule is static. Further, we know the size of the loop and the cost of each iteration is the same so the load cost will be equal for all threads. The if n>500 condition helps to reduce the added overhead for the first 500 n values.

The original code always does two checks. The optimized code reduces these if checks by doing one, and when necessary doing the second if check. This reduces the total number of comparisons done.

### Main " Phase 2 Ending

#### Original code

```
for (c=0; c<n; c++) {
    posAll[c][0] = posAll[c][0]+currMov[c][0];
    posAll[c][1] = posAll[c][1]+currMov[c][1];
    posAll[c][2] = posAll[c][2]+currMov[c][2];

    // boundary conditions: cells can not move out of the cube [0,1]^3
    for (d=0; d<3; d++) {
        if (posAll[c][d]<0) {posAll[c][d]=0;}
        if (posAll[c][d]>1) {posAll[c][d]=1;}
    }
}
```

Changed code line(s): 2-4, 9

#### Optimized code

```
#pragma omp parallel for simd schedule (static) private (c,d) if (n>500)
for (c=0; c<n; ++c) {
    __assume_aligned((float*)posAll, 64);
    __assume_aligned((float*)currMov[c], 64);
    posAll[c][0:3] += currMov[c][0:3];
    // boundary conditions: cells can not move out of the cube [0,1]^3
    for (d=0; d<3; d++) {
        if (posAll[c][d]<0) {
            posAll[c][d]=0;
        }else if (posAll[c][d]>1) posAll[c][d]=1;
    }
}
```

Changed code line(s): 1, 3-5, 10

#### Optimization Notes

This final part of the code is optimized in the same way as the Phase 1 optimization: OpenMP thread parallelization with SIMD and if clause reduction.

The OpenMP thread parallelization with SIMD is a good change because the schedule is static. Further, we know the size of the loop and the cost of each iteration is the same so the load cost will be equal for all threads. The if n>500 condition helps to reduce the added overhead for the first 500 n values.

The original code always does two checks. The optimized code reduces these if checks by doing one, and when necessary doing the second if check. This reduces the total number of comparisons done.

#### Other Optimizations

Besides the optimizations listed in each section of the code previously, there were other optimizations made that were not explicitly remarked on:

Defining all post-increments as pre-increments. The post-increment makes a copy of the actual variable and then increments it. A pre-increment does not make this copy so it's faster. This is only a gain when the point of incrementation does not affect the result. Memory alignment on all arrays. When the arrays are aligned, the memory operations are faster and the CPU doesn't need to mask or convert the data.

#### Rejected Optimization Techniques

During the development of this final code I experimented with other parallelization and vectorization techniques, such as Intel Cilk Plus and Intel® Threading Building Blocks (Intel® TBB), as well as some mathematic thread-optimized functions from the Intel MKL. Using Intel Cilk functions made the code slow, though this might be because I didn't fully understand how to implement Intel Cilk functions.

Since the OpenMP functions performed well without problems, I did not do additional testing with the Intel TBB functions, so it is possible that Intel TBB could improve performance as well.

Another possibility was using Intel MKL functions, which perform well when they work with large data. While the code moves large parts of data, the instantaneous data volume is low. Because of the low instantaneous data volume (and possibly other reasons) the Intel MKL functions were not a good option for optimization.

#### About the Author



Á

Daniel Veá Falguera is an Electronic Systems Engineering student and entrepreneur with interests and knowledge about electronics and computer programming.

#### Links

Most of the optimizations done in this code are based on this Intel Xeon Phi coprocessor article:

<https://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-1-optimization>

Other similar commented options and solutions were provided at the Intel Modern Code for Parallel Architectures forums:

<https://software.intel.com/en-us/forums/intel-moderncode-for-parallel-architectures>

Intel® MKL: <https://software.intel.com/en-us/mkl-reference-manual-for-c>

Intel® TBB: <https://www.threadingbuildingblocks.org/>

Intel® Cilk Plus: <https://www.cilkplus.org/>


OpenMP\*: <http://openmp.org/wp/>

Â

For more complete information about compiler optimizations, see our [Optimization Notice](#).

<p>Categories:</p> <p><a href="#">Academic</a> <a href="#">Code Modernization</a> <a href="#">Intel® Many Integrated Core Architecture</a> <a href="#">Threading</a> <a href="#">Vectorization</a> <a href="#">Intel® Advanced Vector Extensions</a> <a href="#">OpenMP*</a> <a href="#">C/C++</a> <a href="#">Server</a> <a href="#">Developers</a> <a href="#">Professors</a> <a href="#">Students</a> <a href="#">Linux*</a> <a href="#">Intermediate</a></p>
<p>Tags:</p> <p><a href="#">Intel® Modern Code Developer Challenge</a></p>

[Server](#) [Forum](#)

 [Disclaimer](#)

Using this translation widget will provide you with a machine translation of the original content. The machine translation is provided for informational purposes only; it should not be relied upon as complete or accurate.

**Add a Comment**

[Top](#)

(For technical discussions visit our [developer forums](#). For site or software product issues [contact support](#).)

Please [sign in](#) to add a comment. Not a member? [Join today](#)  
[Support](#) [Terms of Use](#) [\\*Trademarks](#) [Privacy](#) [Cookies](#) [Publications](#)  
[Intel® Developer Zone Newsletter](#) [Intel® Parallel Universe Magazine](#)

Look for us on:

[Facebook](#) [Twitter](#) [Google+](#) [LinkedIn](#) [YouTube](#)  
[English](#) [ç@€â½“ä,æ-ž](#) [FranÅ§sais](#) [Ð ŃfŃŃÐ°Ð,Ð'](#) [EspaÃ±ol](#) [PortuguÃs](#) [Deutsch](#)



Rate Us



You are viewing a mobilized version of this site...  
[View original page here](#)

Mobilized by  Mowser