# BONE ANISOTROPY MAPPING

*Jarunan Panyasantisuk, Joao Rivera, Rajan Gill, Ryan Cherifa*

Department of Computer Science
ETH Zürich
Zürich, Switzerland

## ABSTRACT

In this paper, we optimized an implementation for bone anisotropy mapping to obtain the microstructure orientation of a low resolution image of a bone from a high resolution image. This process includes three main components that we optimized, namely region extraction, the mean intercept length and ellipsoid fitting. For the latter two, we achieve a maximum performance improvement of 6.4x and 3.3x compared to the initial implementation. The maximum performance improvement for the overall application is 3.9x.

## 1. INTRODUCTION

Bone fabric anisotropy or microstructure orientation was recently included in finite element (FE) models to improve the accuracy in predicting bone stiffness and strength [1]. To save computing cost, an FE model of bone is generated from a clinical computer tomography (CT) scanned image with low resolution (1-3mm). However, the bone microstructure details can be obtained only by high resolution peripheral CT with the resolution of 60-82 $\mu$m. Therefore, bone anisotropy mapping methodology is required to map the bone microstructure orientation from the high resolution image onto the low resolution image.

Bone anisotropy mapping methodology includes coordinates mapping between a low and a high-resolution images, region extraction (RE), the mean intercept length (MIL) method for quantification of the microstructure orientation, ellipsoid fitting (EF) of MIL and eigendecomposition to obtain the major direction of the microstructure.

**Motivation.** In the recent study, bone anisotropy mapping algorithms are performed for all low resolution image voxels and for all pairs of low and high resolution images [1]. This preprocessing step consumes a significant amount of computing time. Moreover, researchers expect larger dataset to create a more general FE models of bone. Therefore, the performance of these algorithms are crucial. Two software packages for image processing which include MIL calculation are Medtool, a commercialized PYTHON package, and BoneJ, an open-source JAVA plugin for ImageJ. The external packages needed to be integrated to the computation pipeline and optimization is not straightforward.

In this paper, we present an integrated and optimized methodology of bone anisotropy mapping. At our best knowledge, this is the first paper to study and optimize the performance of MIL depending on the extracted region size.

## 2. BACKGROUND

The methodology is shown in Fig. 1. Coordinates from a low resolution image were mapped to its own high resolution image. Then, a sphere region is extracted and centered at the mapped coordinate in the high resolution image.

Subsequently, the anisotropy of the extracted bone region is quantified by using MIL method which imposed direction vectors on the regions. The mean length of each vector is the sum of the length inside the bone region divided by the number of intercepts which intersect with bone/non-bone transition.

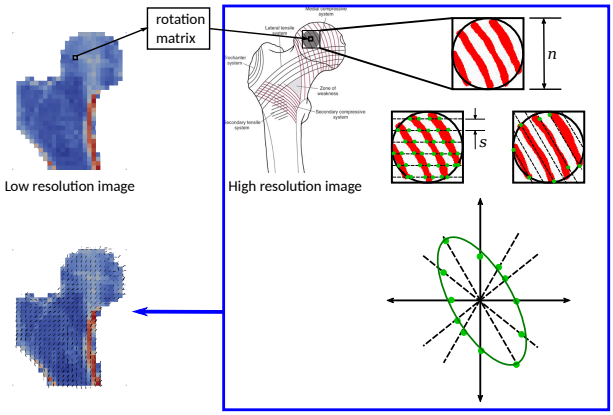


**Fig. 1**. Bone anisotropy mapping methodology includes coordinate mapping from low to high resolution image, RE from the high resolution image, MIL calculation, EF to obtain a fabric tensor and eigendecomposition of the fabric tensor from which the major eigenvector can be visualized. $n$ is the extracted region dimension in one direction. $s$ is the stride between parallel direction vectors.

The MIL values can be plotted as a cloud of points in the direction vector space and an ellipsoid can be fitted to obtain a representative two dimensional tensor, for which

three eigenvalues and three eigenvectors are calculated. The eigenvector associated with the minimum eigenvalue is the major direction of that bone region.

**Region extraction (RE).** This algorithm applies a sphere mask on the high resolution image and copies the extraction region to a separate array. A multiplication was performed for each image voxel.

**Mean Intercept Length (MIL) method.** The mean intercept length of a vector $v$ is expressed as

$$MIL(v) = \frac{h(v)}{I_{count}(v)}, \tag{1}$$

where $h(v)$ is the summation of the bone intensities traversed by all the rays formed by direction vector $v$, and $I_{count}(v)$ is the number of intercepts of the vector $v$. Each ray is separated by a stride $s = 2$. Thus, each direction vector traverses $\frac{n^3}{s^2}$ voxels for $n$ even. We consider only $N_{vec} = 13$ direction vectors.

As an example, consider the (2-D) extracted region in Fig 1 with horizontal vector $\vec{v} = (1, 0, 0)$. In this region, $I_{count}(v)$ is the number of green dots, whereas $h(v)$ is calculated summing up all voxels traversed by the doted lines.

The algorithm includes floating point additions to calculate $h(v)$ and comparisons to detect interceptions. Additions performed to calculate $I_{count}(v)$ are not considered in the cost analysis because those are performed over integers. In addition, one division is performed per direction vector. The total cost of the MIL algorithm per extracted region is therefore $C(n) = N_{vec}(\frac{2n^3}{s^2} + 1)$.

**Ellipsoid fitting (EF).** The ellipsoid fitting problem can be formalized as follows:

$$\text{minimize}_{Q \in \mathbb{R}^{3x3}} \sum_{i=1}^{N_{vec}} (p_i^T Q p_i - 1)^2 \tag{2}$$

where $p_i = MIL(v_i)v_i/||v_i||_2$, and $v_i$ is the $i$th direction vector. The above corresponds to a least squares regression problem. The algorithm we choose to solve this is gradient decent with a backtracking line search [2] for finding the step size. Both the outer gradient descent and the inner line search are iterative algorithms and terminate depending on the input data-set based on an error criteria. The next iteration of the gradient descent cannot proceed before the current iteration is completed. The total number of flops, per call of the ellipsoid fitting algorithm, is $C(k_{\text{g iters}}, k_{\text{bt iters}}, N_{vec}) = (51N_{vec} + k_{\text{bt,iters}}23N_{vec})k_{\text{g,iters}}$ flops, where $k_{\text{bt,iters}}$ is the number of back tracking line search iterations and $k_{\text{g,iters}}$ is the number of outer gradient descent iterations. Note that the cost does not depend on $n$. The iteration counts differ depending on the input points $p_i$. Since the flop count is not deterministic, the code is instrumented to get the exact flop count.

**Table 1**. Cost analysis per low-resolution bone voxel.

| | $N_{add}$ | $N_{mul}$ | $N_{div}$ | $N_{cmp}$ | Total |
|---|---|---|---|---|---|
| RE | - | $n^3$ | - | - | $n^3$ |
| MIL | $\frac{13n^3}{4}$ | - | 13 | $\frac{13n^3}{4}$ | $13(\frac{n^3}{2} + 1)$ |
| EF | $O(1)$ | $O(1)$ | - | - | $O(1)$ |

From the baseline implementation, RE, MIL and EF consume approximately 20, 72, and 8 percent of the overall computing time, respectively. Therefore, we focused on these three algorithms for the optimization. The run time was measured by using time stamp counter (TSC). We define our cost measure as the number of floating point operations (flops) performed by the algorithms. The cost analysis for each algorithm is shown in Table 1, where $n$ is the size of the region extracted during RE. Note that each algorithm is performed once per bone voxel in the low resolution image.

## 3. METHODS

This section explains the baseline implementation and optimization strategies of each algorithm.

### 3.1. RE optimizations

Since we are writing to an array (the extracted region), there may be aliasing. Assuming the compiler doesn't do a runtime aliasing check, a scalar optimization includes loop unrolling and scalar replacement. We can also obtain hard bounds on the performance: ignoring if-checks (the region may need to be truncated if it exceeds the high resolution image), we have two loads, one mult, and one store. Therefore, the peak performance based on this is 1 flop per cycle. Furthermore, for small n, assuming cold cache, the intensity is 1/3 flops/double (per mult, 2 loads + 1 write, which we only count as a read into cache) and for large n the intensity is 1/4 flops/double (2 loads + 1 load for the write, then we expect this written value to be evicted back into ram thus + 1 write). Since $\beta_{RAM} = 2$ doubles/cycle. the peak performance for large $n$ is 0.5 flops/double. We additionally perform SIMD vectorization to utilize vectorized loads and stores.

### 3.2. MIL optimizations

The following C-like pseudo code summarizes the baseline implementation of MIL algorithm. As can be seen, the baseline implementation iterates through all rays of all 13 direction vectors (`for` loop in lines 3-4). For each ray, we first get its starting voxel to begin iterating from (line 5). Finally, the inner loop iterates through the whole ray and computes $h(v)$ and $I_{count}(v)$. The only floating point operations in this loop are the addition and comparison in lines 8-9.

```
1  void mil_base(double* region, int n, double* mil):
2    double h[13] = 0; int I_cnt[13] = 0;
3    for (int v = 0; v < 13; ++v)
4      for every ray r of v  do
5        {k,j,i} = get_start_of_ray(r);
6        int prev = region[k][j][i];
7        while ( {k,j,i} < n && {k,j,i} > 0 )
8          h[v] += region[k][j][i];
9          curr = region[k][j][i] > 0.5; // bone?
10         I_cnt[v] += curr ^ prev; // interception
11         prev = curr;
12         {k,j,i} = next_voxel_indices({k,j,i}, v);
13     mil[v] = h[v] / C[v];
```

**Improving ILP.** The first performance bottleneck in the baseline implementation is due to the non-associativity of floating point additions in line 8. This creates inter-loop dependencies that limit the instruction level parallelism (ILP). Assuming a latency of floating point additions of 4 cycles, an upper bound on the performance due to this dependency is 0.5 flops/cycle (1 `add` + 1 `cmp` every 4 cycles).

To improve ILP, we first unrolled the loop in line 4, to get four different rays of the same vector. We created an accumulator for each ray; thus, now we can perform 4 additions and 4 comparisons in parallel. Note that the rays for each accumulator should be of the same length to avoid handling leftovers at the end. For 1-D direction vectors, e.g. (1,0,0), all rays have the same length (equal to $n$); however, this is not the case for 2-D and 3-D vectors. Thus, we split the implementation for 1-D, 2-D and 3-D vectors to select a suitable set of rays of the same length for each case. We choose rays close to each other to leverage spacial locality.

**Intensity analysis.** The second performance blocker is memory accesses. Assume that the extracted region is much larger than the size of the last level cache, i.e. $8n^3 >> N_{L3}$. For the first horizontal direction vector $v_1 = (1, 0, 0)$, a lower bound for the data read from memory is $Q_1(n) \geq \frac{n^3}{4}$ doubles. Recall that rays are spaced by a stride $s = 2$; thus, not all data in the region is accessed by a vector. The bound of $Q_1(n)$ is close to tight for the horizontal vector because data is accessed sequentially for each ray. Thus, when a (compulsory) cache miss occurs, the data in the whole cache block that is brought to cache will be used in the following iterations taking advantage of spacial locality (except for the first or last cache block in the ray in case of misalignment).

The second direction vector, $v_2 = (0, 1, 0)$, traverses the extracted region vertically. There is no reuse of data from the previous vector since we assumed a large $n$. Further, note that for the vertical vector, at most half of the data in a cache block will be used due to the stride of $s = 2$ for consecutive rays. Since unused data in a cache line still have to be read from memory, a lower bound for the data read for this vector is $Q_2(n) \geq \frac{n^3}{2}$ doubles. The remaining vectors will behave similarly as the vertical vector. Hence, a lower bound for the total amount of data read from memory is $Q(n) \geq \frac{n^3}{4} + \frac{12n^3}{2} = 6.25n^3$. Recall from the cost analysis of the previous section that the total number of flops performed by MIL is $W(n) = 13(\frac{n^3}{2} + 1) \approx 6.5n^3$. An upper bound on the operation intensity of MIL for a large $n$ is therefore $I(n) \leq \frac{6.5n^3}{6.25n^3} = 1.04$ flops/double. We conclude that the implementation of the algorithm is memory bound when the data does not longer fit in the cache.

**Blocking.** To improve the operational intensity and cache locality, we applied blocking. The idea behind blocking is that we can improve on data reuse by partially calculating $h(v)$ and $I_{count}(v)$ in a cube-block of size $N_B^3$ that fits in L1 cache. Since the blocked data will still be in the cache for the next vector, data reuse will be improved. Afterwards, we repeat this process for the next block until all the extracted region is covered. For simplicity, we assumed that the region size $n$ is divisible by the block size $N_B$. Fig 2 depicts a block of size $N_B$ in an extracted region of size $n$.

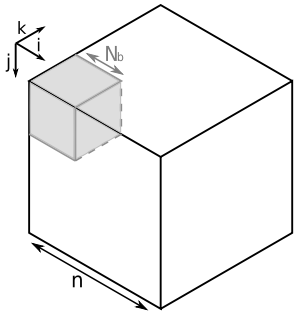


**Fig. 2**. Example of one block of size $N_B$ inside a extracted region of size $n$.

There are two things to consider when implementing blocking for MIL. First, note that we are accessing voxels in the extracted region that are traversed by the rays of all vectors. To this end, the implementation first determines the start of a ray in the limits of the region to begin iterating from. When blocking, we have to pay special attention to choose the right starting point for each partial ray processed by each block. To illustrate this, see Fig 3.a. The highlighted blue voxels are the starting points for each block for the 2-D direction vector (1,1,0). As can be seen,blocks B1 and B2 does not have the same pattern for the starting points. This means that we would have to determine the start pattern for each block to implement blocking. We therefore prefer to have the same starting patter within all blocks. This is easy to achieve for 1-D and 2-D vectors by simply choosing $N_B$ multiple of the stride $s = 2$. On the other hand, to achieve homogeneity between blocks when processing 3-D vectors, we had to change the starting pattern for these vectors as shown in Fig 3.b. The second aspect to consider when blocking is the correct initialization of `prev` variable (line 6 in the baseline code) using the voxel before the start of the partial ray of the block. This is necessary to

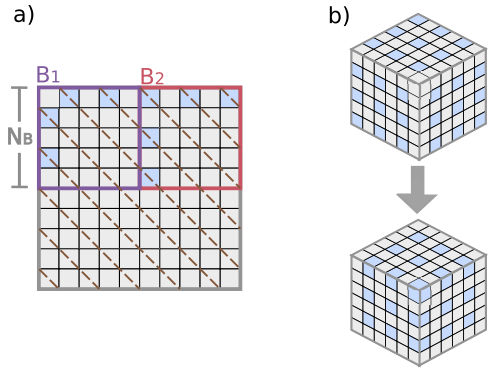


**Fig. 3**. a) Blocks with different starting pattern for rays of direction vector (1,1,0). b) Change in starting pattern for 3-D direction vectors to allow blocking.

detect an interception at the start of a block.

Assuming that the block fits completely in L1 cache, a lower bound for the data read from memory for a block is $Q_B(n) \geq N_B^3$ doubles. Ignoring conflict misses, this lower bound is tight. Since there are $\frac{n^3}{N_B^3}$ blocks, the total amount of data read from memory is $Q(n) \geq N_B^3 \cdot \frac{n^3}{N_B^3} = n^3$ doubles. An upper bound on the operational intensity is therefore $I(n) \leq \frac{6.5n^3}{n^3} = 6.5$ flops/double. Thus, the implementation now becomes *compute bound*. For the final implementation, we choose $N_B = 16$.

**SIMD vectorization.** After using blocking to improve locality, we vectorized the code using SIMD instructions. Recall that four accumulators were used to improve ILP in the first optimization. Since the data is in double precision, we can store the accumulators in a SIMD AVX register and apply vector instructions. We also used loop unrolling to expose additional rays that can be handled in a vector, therefore improving ILP. Similar to the scalar version, the implementation for 1-D, 2-D and 3-D direction vectors are treated separately to choose suitable rays for each case. In summary, in most cases we are procession 8 rays simultaneously, stored in two SIMD register. The only exception is 3-D vectors in which processing 12 rays simultaneously in three SIMD registers was a more suitable option. Fig 4 shows the vectorized diagram for the main operations used in MIL. Note that 3-D vectors have more and shorter rays than 2-D or 1-D vectors. Thus, there is more overhead when determining the next set of rays to be processed.

### 3.3. Ellipsoid fitting optimizations

**Intensity analysis.** In the context of the overall algorithm, $N_{vec}$ is fixed to 13. Thus, the 13 points is small enough for the points $p_i$ to all fit in L1 cache. Whether we assume cold or warm cache, the operational intensity is very high

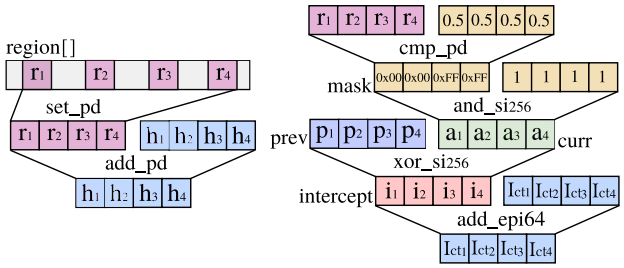


**Fig. 4**. SIMD vectorization of four rays in MIL.

(infinity if we assume warm cache). However, we choose to investigate what happens for varying $N_{vec}$ out of curiosity.

If the working set (i.e. all the points $p_i$) don't fit in L1, then each time we compute the gradient, and every iteration of the back tracking line search, we have to reload the points into L1 cache, and thus $Q \geq (3N_{vec} + k_{\text{bt,iters}} 3N_{vec}) k_{\text{g,iters}}$ [doubles]. Thus,

$$I(N_{vec}) \leq \frac{(51N_{vec} + k_{\text{bt,iters}} 23N_{vec}) k_{\text{g,iters}}}{(3N_{vec} + k_{\text{bt,iters}} 3N_{vec}) k_{\text{g,iters}}} \approx 8 \sim O(1)$$

The intensity is still high enough to be compute bound.

**Scalar optimization.** Loop unrolling and breaking up dependencies to increase ILP doesn't really help, as the pipeline is already filled and aliasing is not an issue (don't write constantly to one of the input arrays). For example, when computing the gradient or when computing the cost during the line search, a quadratic form is computed for each of the points. Each quadratic form can fill the pipeline as not all computations have serial dependencies. However, one particular optimization that proved effective was removing loop invariant code: in the backtracking line search, a condition is checked which involves comparing the cost of the current best guess of $Q^*$ with that of the previous iteration, and the compiler was not able to move the cost of the previous best guess outside the loop (didn't modify flop count). Doing this hits the maximum achievable flop count of 3, which is derived using the instruction mix.

**SIMD vectorization.** Each of the points $p_i$ contribute to the gradient as well as to the cost (as a quadratic form, as mentioned earlier). Thus, a vectorization strategy is to vectorize across the points to simultaneously compute the quadratic forms for four points at a time, and then sum the results at the end. Doing so involves shuffling the input array of points, which is in the format

$$[x_0, y_0, z_0, x_1, y_1, z_1, x_2, y_2, ...]$$

to a format suitable for vectorization

$$[x_0, x_1, x_2, ...] \quad [y_0, y_1, y_2, ...] \quad [z_0, z_1, z_2, ...].$$

This is similar to the complex conversion problem of homework 3, except for 3D rather than 2D. Now every time the gradient or cost needs to be computed, the above shuffled loads take place to fill up 4 points worth of data before the flops can be computed using SIMD instructions. However, an additional optimization is to pre-shuffle the points before commencing with the gradient decent algorithm and store the shuffled data into memory. Subsequently, during the gradient decent computations, only loads from the already shuffled memory is done. This approach makes sense if the passes through the working set is at least larger than 1, which is the case (i.e. $k_{\text{bt,iters}} + k_{\text{g,iters}} > 1$).

## 4. EXPERIMENTAL RESULTS

This section presents the performance results for each of the discussed algorithms.

**RE.** *Experimental setup*. The tests were performed on a Intel i7 6820HQ (Skylake) with TurboBoost disabled, compiled with `gcc -O3 -fno-tree-vectorize -mavx2`. The L1, L2 and L3 cache size are 32KB, 256KB, and 8MB, respectively. The sphere region size is ranged from $n = 8$ to 128.

*Results* As seen in Fig. 5 (a), the loop unrolling and scalar replacement barely improved the performance for the region extraction algorithms for small $n$: the compiler might already perform well with a real-time alias check. When the extracted region size fits in the cache, recall that the performance was bounded to 1 flops/cycle (scalar case) due to the instruction combinations. On the other hand, when the extracted region size was too large for the cache, the performance was bound to 0.5 flops/cycle due to the memory. We typically operate over large $n$. In this case, we can only hit 0.2 flops/cycle. This could happen if we are not getting the full bandwidth from ram. Using vectorized loads we expected the performance to improve, but it only had an effect for small $n$.

**MIL results.** *Experimental setup*. All experiments in this section were performed on an Intel Core i9-8950HK The L1, L2 and L3 cache size were 32KB, 256KB, and 8MB, respectively.), @2.90GHz with TurboBoost disabled using gcc 7.3.0 with flags `-O3 -fno-vectorize-tree -mavx2`. The L1, L2 and L3 cache size are 32KB, 256KB, and 12MB, respectively. The region size is ranged from $n = 16$ to 400.

*Results*. Fig 5 (b) shows the performance plot for MIL for four different configurations, namely the *baseline*, *scalar non-blocking*, *scalar with blocking* and *SIMD with blocking*. As can be seen, the performance of *scalar non-blocking* (blue line) already improves the performance by around 2x for small sizes compared to the baseline. This is mainly due to an improvement in ILP. However, performance drops considerably when the data does not longer fits in L3 cache. As expected *scalar with blocking* (green line) improves the performance by 3x for large sizes by reducing the number

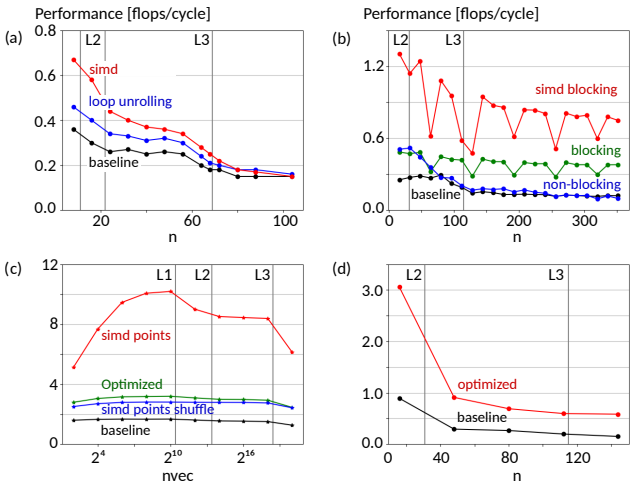


**Fig. 5**. Performance of (a) RE, (b) MIL, (c) EF and (d) overall.

of cache misses. Finally, *SIMD with blocking* (red line) achieves a maximum performance gain of 6.4x compared to the baseline when the data does not fit cache. For a small $n$, we observe a maximum performance of 1.3 flops/cycle. Since MIL doesn't use FMA instructions, the peak performance is 8 flops/cycle. Thus, we are only getting at most 16.25% of the peak performance. We attribute this low percentage of peak performance to the extra non floating point instructions performed, as well as the overhead of changing to another set of rays specially for 2-D and 3-D vectors.

*Conflict Misses*. For block size $N_b = 16$, the 16x16x16 block barely fits in L1 cache, assuming full associativity. Thus, as we walk across this block over all direction vectors, there will only be compulsory misses on the very first pass of the block. However, the results in Fig 5 (b) show that for $n = 64x$, where $x = 0, 1, 2, ...$, there is a drop in performance in the blocking case. This reduced performance can be attributed to the increase in cache misses due to conflicts, as L1 cache has 64 sets, and is 8 way associative, with each cache block holding 8 doubles. When $n = 64x$, it can be shown that for fixed $i, j$, the indexes in the set

$$\{(i, j, k) | k = 0, 1, 2, ...\}$$

map to the same cache set. Similarly, for fixed $i, k$ and for some index $j_0$, the indexes in the set

$$\left\{ (i,j,k) \;\middle|\; j = j_0 + \begin{cases} 0, 7, & x = 1, 3, 5, .. \\ 0, 3, .. & x = 2, 6, 10, .. \\ 0, 2, .. & x = 4, 12, 20, .. \\ 0, 1, .. & x = 8, 16, 24, .. \end{cases} \right\}$$

also map to the same cache set. This is shown in Fig 6 for $x = 1$. From the figure, we can see that some cache lines

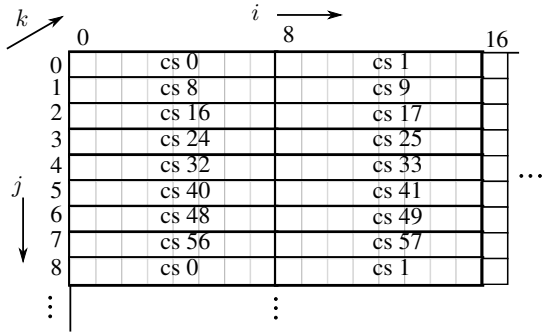


**Fig. 6**. Cache set mappings for the first 16x16x16 block of a 64x64x64 region (all units are in doubles). Only a single plane is shown (fixed $k$), but the image is the same for all $k$. Note that a single cache block is 8 doubles. The numbers represent the cache set index for the associated cache block.

are not being used for a single 16x16x16 block. Furthermore, the cache set 0 is used to hold 2 cache blocks of data per $k$, thus cache set 0 is mapped to a total of 32 times over all $k$, which is greater than 8. Thus in this case, we effectively only utilize 25% of the L1 cache. As a consequence, after one pass of the block for one direction vector, when we start another pass of the same block for the next direction vector, there is no reuse from the previous pass due to conflict misses. However, when $n$ is e.g. 65, the cache set mapping becomes much more diverse and thus the conflict misses are mitigated. To summarize, since the block is not stored contiguously, the strides that we take is associated with $n$, and so in this case for $n$ a multiple of 64 presents a particularly nasty scenario for cache conflicts. In practice, a simple solution is to ensure that $n$ is not a multiple of 64.

**EF.** *Experimental setup* The setup for the overall performance is the same as RE.

*Results* The results can be seen in Fig. 5 (c). The *baseline* already performs quite well due to the lack of dependencies and aliasing. The *Optimized* version removes loop-invariant code and we can hit the peak from the instruction mix. The 'SIMD points' shuffles the points once during an initializing before proceeding with the algorithm, and this yields the largest performance gain; not doing this initializing corresponds to the *SIMD points shuffle* curve. As mentioned, we operate at $N_{vec} = 13$, and thus the performance gain with respect the baseline is roughly 3.3x.

**Overall performance.** *Experimental setup* This experiment shows the performance of the bone anisotropy estimation algorithm from end-to-end combining RE, MIL and EF and was performed on Intel i9-8950HK CPU @ 2.90GHz (Coffe Lake) without TurboBoost. The L1, L2 and L3 cache size were 32KB, 256KB, and 12MB, respectively. The highest optimization flag was used without vectorization (`-O3 -fno-vectorize-tree`). The sphere region size were ranged from $n = 16$ to 144 by steps of 32.

**Table 2**. Number of cycles

| | 16 | 80 | 144 |
|---|---|---|---|
| baseline | $1.025e+09$ | $7.245e+10$ | $7.197e+11$ |
| optimized | $2.994e+08$ | $2.790e+10$ | $1.860e+11$ |

*Results* The optimisation version of bone anisotropy estimation improves the performance by a factor of approximately $\times 4$, as shown in figure 5 (d), compare to the baseline due to cache optimization with blocking. Additionally, a comparison of number of cycles is given in Table 2.

## 5. CONCLUSIONS

In this paper, the bone anisotropy mapping was integrated and optimized. From a baseline implementation, we identified three parts to optimize, namely region extraction (RE), MIL and ellipsoid fitting (EF), which take roughly 20, 72, and 8 percent of the overall runtime respectively. In particular, RE is memory bound; however, SIMD instructions improve the performance when considering warm cache. For MIL, we achieved a maximum improvement of 6.4x compared to the baseline when optimizing using blocking and SIMD vectorization. The results showed a significant decrease in performance when the region is multiple of 64 due to conflict misses on L1 cache. We showed that only 25% of the cache is being effectively used for the case when $n = 64$. For EF, an important optimization was to pre-shuffle the input points before starting with the gradient decent algorithm to make the input suitable for vectorization. The performance after optimizations when $N_B = 13$ for ellipsoid fitting was roughly 6.6 flops/cycle, which corresponds to a performance gain of 3.3x compared to the baseline. For other values of $N_B$ we achieved a maximum performance of roughly 10 flops/cycle. Finally, the overall performance achieved is 3.9x.

As future work, we can reduce conflict misses in MIL when the regions size is multiple of 64 by padding each dimension to increase its size. Also, we could merge some of the building blocks to get additional improvements. For example, we could add some of the computations from MIL (or EF) to RE in order to improve its operational intensity.

## 6. REFERENCES

[1] J. Panyasantisuk et al., "Mapping anisotropy improves qct-based finite element estimation of hip strength in pooled stance and side-fall load configurations," *Medical Engineering and Physics*, vol. 59, pp. 36–42, September 2018.

[2] Stephen Boyd and Lieven Vandenberghe, *Convex optimization*, Cambridge university press, 2004.