BONE ANISOTROPY MAPPING

Jarunan Panyasantisuk, Joao Rivera, Rajan Gill, Ryan Cherifa

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

Todo

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, 2]. 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 μm. Therefore, bone anisotropy mapping methodology was 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 (n=71) of low and high resolution images [3]. 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 are presenting 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 calculation 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 devided by the number of intercepts which intersect with bone/non-bone transition. The MIL values can be plots 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.

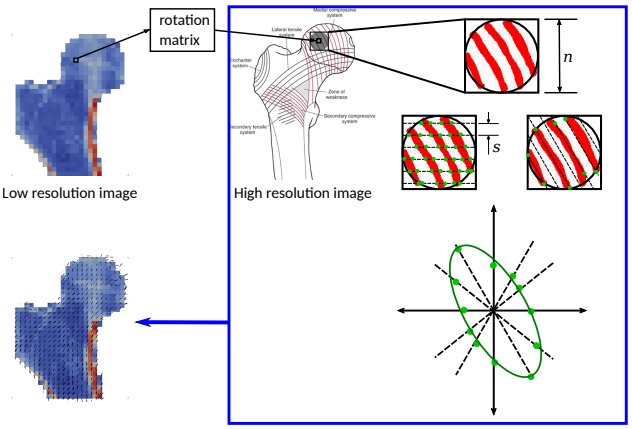

Fig. 1. Bone anisotropy mapping methodology includes coordinate mapping from low to high resolution image, RE from the high resolution image, MIL calculation, EF to obtain a fabric tensor and eigendecomposition of the fabric tensor from which the major eigenvector can be visualized. n is the extracted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

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 can be expressed as

$$MIL(v) = \frac{h(v)}{C(v)}, \quad (1)$$

where $h(v)$ is the summation of the bone intensities "touched" by all the rays formed by direction vector v, and $C(v)$ is the number of intercepts of the vector v. In general, each ray is separated by a stride $s = 2$. Thus, each direction vector touches only $\frac{n^3}{s^2}$ voxels in the region 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, $C(v)$ is the number of green dots, whereas $h(v)$ is calculated summing up all voxels touched by the horizontal doted lines.

The algorithm includes floating point additions to calculate $h(v)$ and comparisons to detect interceptions. Note that additions performed to calculate $C(v)$ 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 $C(n) = N_{vec}(\frac{2n^3}{s^2} + 1)$. Table 1 provides a detailed breakdown of the cost analysis.

Ellipsoid fitting (EF). Todo

From the basic implementation, MIL calculation, EF and RE consume approximately 80%, 15% and 5 % 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 non-zero voxel in the low resolution image. (This can be omitted) Thus, the cost of the whole program is $total \cdot N_{nz}$, where N_{nz} is the number of non-zero voxels in the low resolution image which represent the bone voxel.

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					

3. METHODS

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

3.1. RE optimization

The baseline implementation loops over all voxels in the sphere mask with the size of n^3. For each sphere mask voxel, if a corresponding voxel in the high resolution image is found, the extracted region voxel is set as the multiplication of the sphere mask voxel and the high resolution image voxel. Therefore, loop unrolling with four parallel multiplications and scalar replacement were applied to speed up the computation and avoid aliasing.

3.2. MIL optimizations

The following C-like pseudo code summarizes the baseline implementation of MIL algorithm:

```
1  void mil_base(double* region, int n, double* mil){
2    double h[13] = 0; int C[13] = 0;
3    // Core of MIL
4    for (int v = 0; v < 13; ++v)
5      for every ray r of v  do
6        {k,j,i} = get_start_of_ray(r);
7        int prev = region[k][j][i];
8        while ( {k,j,i} < n && {k,j,i} > 0 )
9          h[v] += region[k][j][i];
10         curr = region[k][j][i] > 0.5; // bone?
11         C[v] += curr ^ prev; // interception
12         prev = curr;
13         {k,j,i} = next_voxel_indices({k,j,i}, v);
14     mil[v] = h[v] / C[v];
15   }
```

As can be seen, the baseline implementation iterates through all rays of all 13 direction vectors (see the `for` loop in lines 4-5). For each ray, we first get its starting voxel to begin iterating from (line 6). Finally, the inner loop iterates through the whole ray and computes $h(v)$ and $C(v)$. Note that $C(v)$ is an integer array; thus, the only floating point operations in this loop are the addition and comparison in lines 9-10.

Improving ILP. The first performance bottleneck in the baseline implementation is due to the non-associativity of floating point additions in line 9. This creates inter-loop dependencies between addition operations that limit the instruction level parallelism (ILP) of the implementation. 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).

In order to improve ILP, we used accumulators with loop-unrolling. We first unrolled the loop in line 5, 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-Dimensional 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 direction vectors in order to select a suitable set of rays of the same length for the inner most loop in each case. If space, explain that the set of rays where chosen close to each other on the x dimension to try to exploit some spacial locality.

Intensity analysis. The second performance blocker are 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 (mandatory) cache miss occurs, the data in the whole cache line that is brought to cache will be used in the following iterations taking advantage of spacial locality (except for the first or last cache line in the ray in case of misalignment).

The second direction vector, $v_2 = (0, 1, 0)$, iterates through the extracted region vertically. Since we assumed a very large data-set, there is no reuse of data from the previous vector. Further, note that for the vertical vector, only half of the data in a cache line will be used in the best case 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 the vertical 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. In order to improve the operational intensity and cache locality, the next optimization applied was blocking. The idea behind blocking is that we can improve on data reuse by partially calculating $h(v)$ and $C(v)$ 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 improve by taking advantage of spacial and temporal locality. 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.

There are two things to consider when implementing blocking for MIL. First, note that we are accessing the voxels in the extracted region that are touched by the rays of all direction 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 points for the partial rays processed by each block. To illustrate this, consider Figure ??. The highlighted voxels are the starting points for each block. As can be seen, in this particular case, not all blocks have the same pattern for the starting points (see for example B1 and B2). This means that we would have to determine the start pattern for each block to implement blocking. Since we favor simplicity, we would like 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 of the baseline implementation as shown in Figure ??.

The second aspect to consider when blocking is the correct initialization of `prev` variable (line 7 of listing ??) using the voxel before the start of the partial ray of the block. This is necessary to detect an interception at the start of the block.

Assuming that the block fits completely in L1 cache, a lower bound for the data read from memory when processing 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 in total, the amount of data read from memory to process the whole region using blocking is bounded by $Q(n) \geq N_B^3 \cdot \frac{n^3}{N_B^3} = n^3$ 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 became *compute bound*. For the final implementation, we choose $N_B = 16$ which is the size that achieves the best performance.

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 an additional rays that can be handled in a vector, therefore improving ILP. Note that we always choose rays with the same length to avoid the overhead of handling leftovers. 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 per loop iteration, 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. Figure 2 shows the vectorized diagram for the main operations used in MIL. It is also worth noting that 3-D direction vectors have inherently more overhead that 2-D or 1-D vectors be-

cause they have more and shorter rays. Thus, there is more overhead when determining the starting points for the next set of rays to be processed.

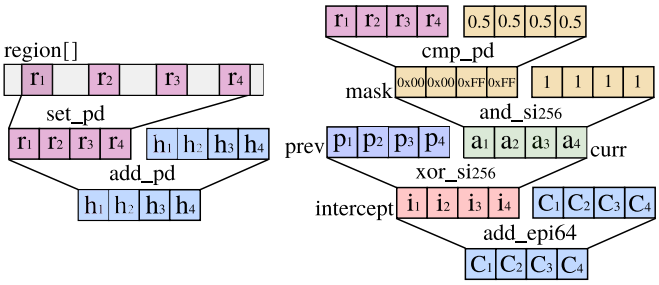

Fig. 2. SIMD vectorization of four rays in MIL.

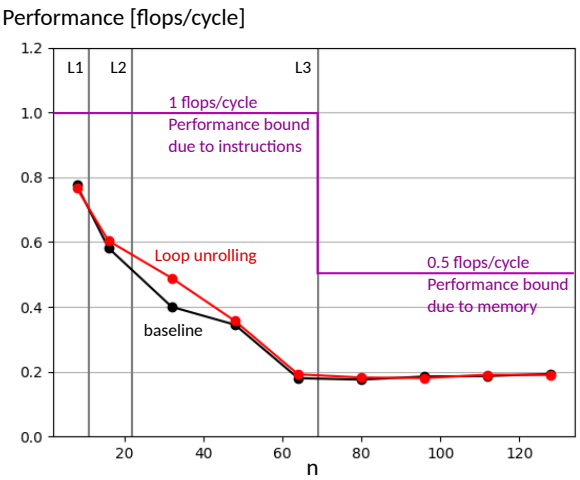

Fig. 3. Performance of RE

3.3. EF

moving loop invariant code, SIMD

4. EXPERIMENTAL RESULTS

The experimental results are presenting here according to the algorithms.

RE. *Experimental setup* The tests were performed on Intel i7 U7600 (Kaby Lake), 2.8 GHz without Turboboost. The L1, L2 and L3 cache size were 32KB, 256KB, and 8MB, respectively. The highest optimization flag was used without vectorization (`-O3 -fno-vectorize-tree`). The sphere region size were ranged from $n = 8^3$ to 128^3.

Results As seen in Fig. 3, the loop unrolling and scalar replacement did not improve the performance for the region extraction algorithms. The compiler might already perform well in alias checking and parallelizing the multiplications. When the extracted region size fitted in the cache, the performance was bound to 1 flops/cylce 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. The results shows that the algoirthm reached around 0.8 flops/cycle in the cache and stayed around 0.2 flops/cycle in the memory.

MIL. *Experimental setup*

Results

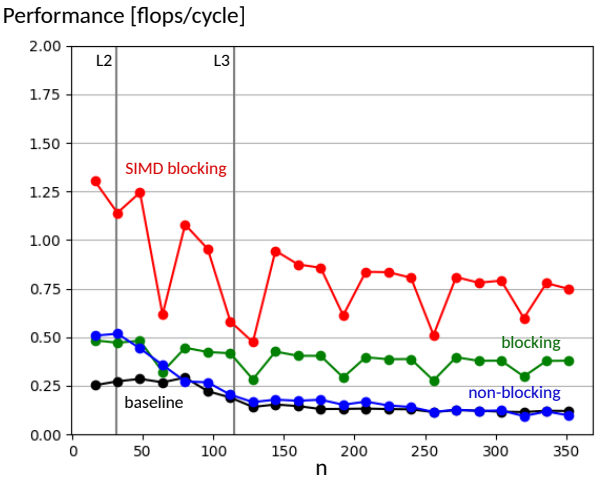

Fig. 4. Performance of MIL

Conflict Misses

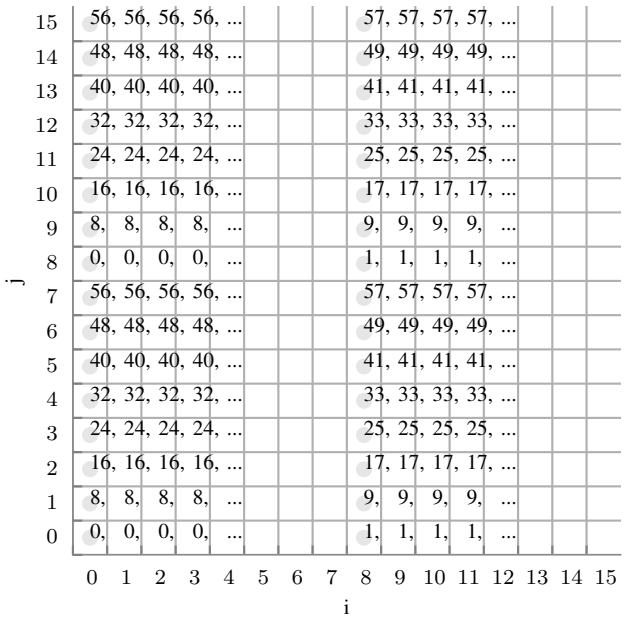

Fig. 5. Cache line mappings for the first 16x16x16 block of a 64x64x64 region (all units are in doubles). Note that a single cache block is 8 doubles. The grey circle represents the start of a cache block. The numbers represent the cache line numbers for the associated i, j entry and for $k = 0, 1, 2, 3, ...$

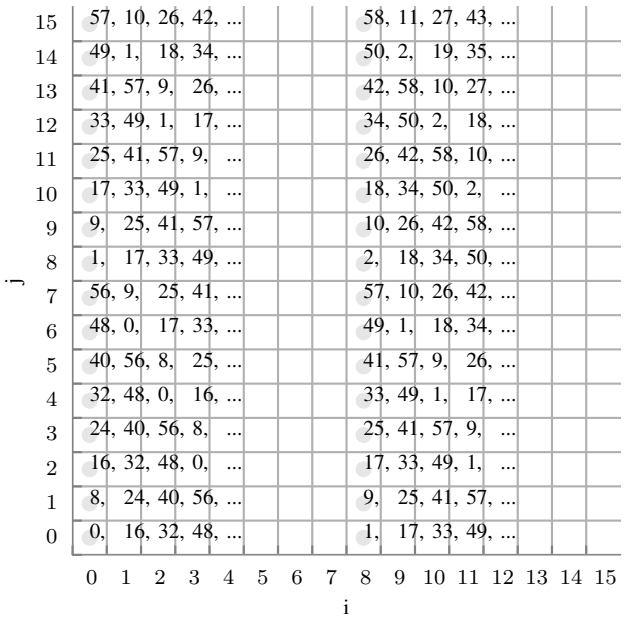

Fig. 6. Cache line mappings for the first 16x16x16 block of a 65x65x65 region (all units are in doubles). Note that a single cache block is 8 doubles. The grey circle represents the start of a cache block. The numbers represent the cache line numbers for the associated i, j entry and for $k = 0, 1, 2, 3, ...$

EF. *Experimental setup*
Results

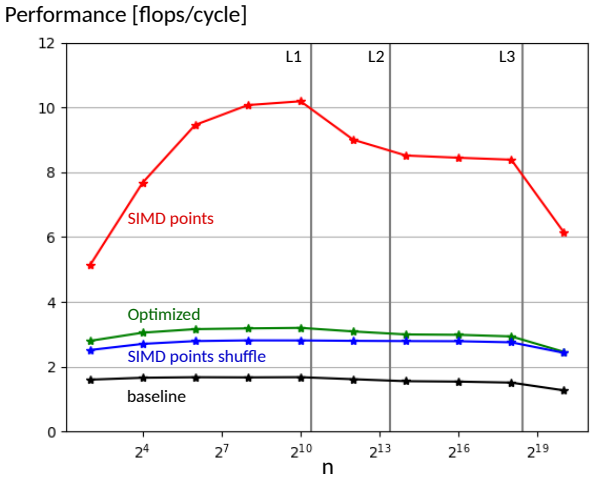

Fig. 7. Performance of EF

Overall performance.

5. CONCLUSIONS

The bone anisotropy mapping was integrated and optimized. MIL calculation consumes the large percentage of the overall computation time, followed by EF and RE.

The performance of RE was bounded by the intructions mix when it was in cache and memory bound when in memory. The loop unrolling did not improve the performance and the compiler already optimized the alias checking in the basic implementation.

MIL calculation

EF

6. FURTHER COMMENTS

Here we provide some further tips.

Further general guidelines.

7. REFERENCES

[1] G. Maquer, S.N. Musy, J. Wandel, T. Gross, and P.K. Zysset, "Bone volume fraction and fabric anisotropy are better determinants of trabecular bone stiffness than other morphological variables," *J Bone Mineral Res*, vol. 30, no. 6, pp. 1000–1008, June 2015.

[2] S.N. Musy, G. Maquer, J. Panyasantisuk, J. Wandel, and P.K. Zysset, "Not only stiffness, but also yield strength of the trabecular structure determined by non-linear fe is best predicted by bone volume fraction and fabric tensor," *J Mech Behav Biomed Mater*, vol. 65, pp. 808–813, January 2017.

[3] J. Panyasantisuk, E. Dall'Ara, M. Pretterklieber, D.H. Pahr, and P.K. Zysset, "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.