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 [**?**, **?**]. 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 [**?**]. 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 divided 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.

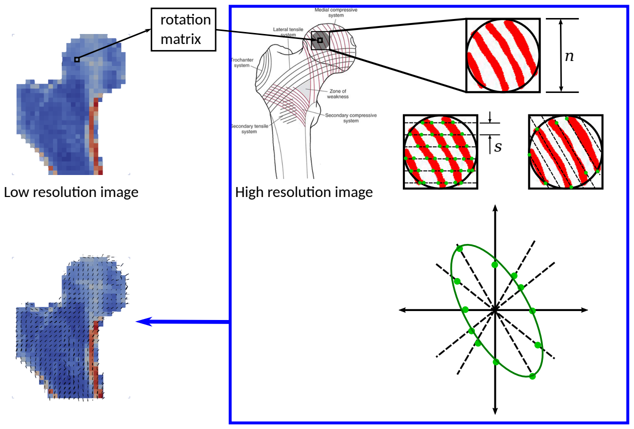

Fig. 1. Bone anisotropy mapping methodology includes coordinate mapping from low to high resolution image, RE from the high resolution image, MIL calculation, EF to obtain a fabric tensor and eigendecomposition of the fabric tensor from which the major eigenvector can be visualized. n is the extracted 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

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

where $h(v)$ is the summation of the bone intensities "touched" by all the rays formed by direction vector v, and $I_{count}(v)$ 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, $I_{count}(v)$ 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 $I_{count}(v)$ 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 $C(n) = N_{vec}(\frac{2n^3}{s^2} + 1)$. Table 1 provides a detailed breakdown of the cost analysis.

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 ith 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 [?] for finding the step size. Both the outer gradient descent and the inner line search are iterative algorithms and terminates 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.

From the basic implementation, RE calculation, 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 non-zero voxel in the low resolution image. (This can be omitted) Thus, the cost of the whole program is $N_{nz}(n^3 + 13(n^3/2 + 1) + O(1))$ where N_{nz} is the number of non-zero voxels in the low resolution image which represent the bone voxel. To get the exact flop count, the $N_{nz}O(1)$ term, which comes from EF, is computed through instrumentation.

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	$O(1)$	$O(1)$	0	0	$O(1)$

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:

```
void mil_base(double* region, int n, double* mil){
  double h[13] = 0; int C[13] = 0;
  // Core of MIL
  for (int v = 0; v < 13; ++v)
    for every ray r of v  do
      {k,j,i} = get_start_of_ray(r);
      int prev = region[k][j][i];
      while ( {k,j,i} < n && {k,j,i} > 0 )
        h[v] += region[k][j][i];
        curr = region[k][j][i] > 0.5; // bone?
        C[v] += curr ^ prev; // interception
        prev = curr;
        {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 $I_{count}(v)$. Note that $I_{count}(v)$ 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 endy. 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. Also we choose, rays close to each other to leverage 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 $I_{count}(v)$ 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 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 as shown in figure 2.

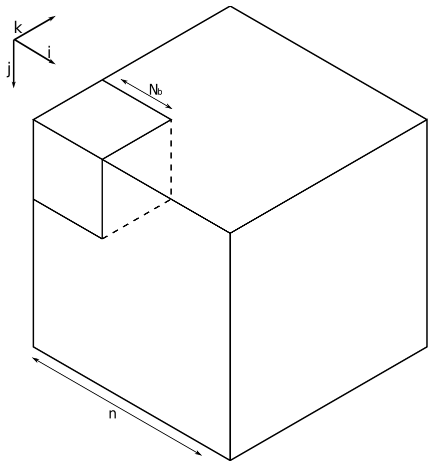

Fig. 2. Example of one block of size N_b inside a extracted region of size n.

There are two things to consider when implementing blocking for MIL. First, note that we are accessing 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, see Figure 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.

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 for these vectors as shown in Figure 3.b.

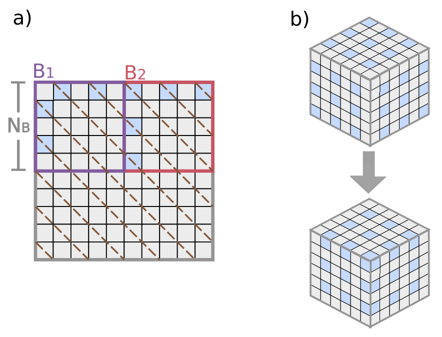

Fig. 3. a) Blocks with different starting pattern for rays of direction vector (1,1,0). b) Change in starting pattern for 3-D direction vectors to allow blocking.

The second aspect to consider when blocking is the correct initialization of `prev` variable (line 7 in the baseline code) using the voxel before the start of the partial ray of the block. This is necessary to 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 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 4 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 because 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.

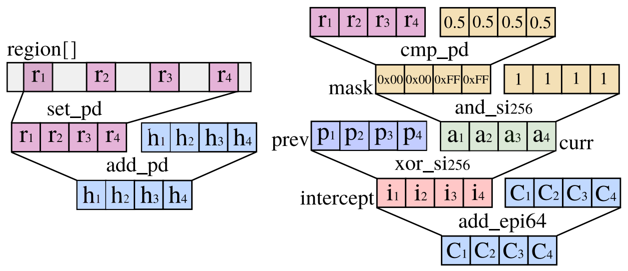

Fig. 4. SIMD vectorization of four rays in MIL.

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 going to be very high (infinity if we assume warm cache). However, we choose to investigate what happens for varying N_{vec} out of curiosity. n in plot should be Nvec

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,

$$I(N_{vec}) \leq \frac{(51N_{vec} + k_{\text{bt,iters}} 23N_{vec}) k_{\text{g,iters}}}{(3N_{vec} + k_{\text{bt,iters}} 3N_{vec}) k_{\text{g,iters}}} \tag{3}$$

$$= \frac{51N_{vec} + k_{\text{bt,iters}} 23N_{vec}}{3N_{vec} + k_{\text{bt,iters}} 3N_{vec}} \approx 8 \tag{4}$$

$$\sim O(1) \tag{5}$$

In this case, 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 with some computations 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 best guess, 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

$$[x_0, y_0, z_0, x_1, y_1, z_1, x_2, y_2, ...] \tag{6}$$

to a format suitable for vectorization

$$[x_0, x_1, x_2, ...] \tag{7}$$
$$[y_0, y_1, y_2, ...] \tag{8}$$
$$[z_0, z_1, z_2, ...] \tag{9}$$

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 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. 5, 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.

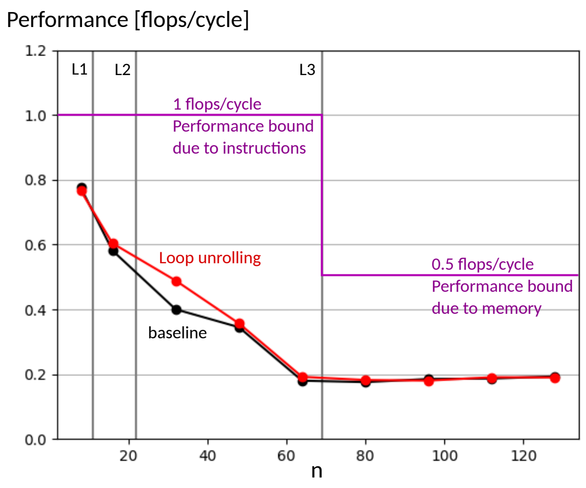

Fig. 5. Performance of RE

should explain how the 0.5 bound was derived

4.1. MIL results

Experimental setup. All experiments in this section were performed on an Intel Xeon E-2176M machine (Coffee Lake) with TurboBoost disabled using gcc 7.3.0 compiler with flags `-O3 -std=c++11 -mavx2`, and under Ubuntu 18.04. The sphere region size were ranged from $n = 16$ to 400 for each dimension.

Results. correct speedup numbers after updating the plot. Figure 6 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 as discussed in the previous section. However, performance drops considerably when the data does not longer fits in L3 cache because this implementation is *memory bound*. As expected *scalar with blocking* (green line) implementation improves the performance by 4x for large sizes by reducing the number of cache misses. Finally, the *SIMD with blocking* (red line) achieves a maximum performance gain of 8x compared to the baseline when the data does not fit cache. For a small n, we observe a maximum performance of 1.4 flops/cycle. Since MIL only uses floating point additions and comparisons, the peak performance of the machine for these operations is 8 flops/cycle. Thus, we are only getting at most 17.5% of the peak performance.

We attribute this low percentage of peak performance to the extra non floating point instructions performed in the core of MIL, as well as the overhead of changing to another set of rays, the latter is specially true for 2-D and 3-D direction vectors where there are more and shorter rays. Observe that for n multiple of 64, the performance drops significantly. We believe this is due to conflict misses as explained in the following subsection.

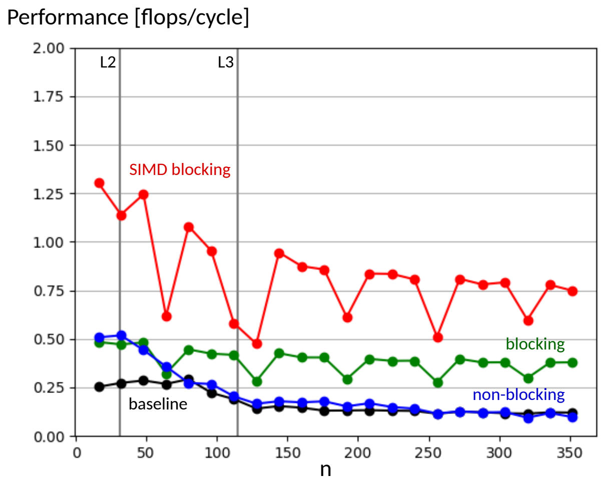

Fig. 6. Performance of MIL

Conflict Misses When the block size $N_b = 16$, then 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 Figure 6 shows that for $n = 64x$, where $x = 0, 1, 2, ...$, we see a drop in performance, most notably in the blocking case. This reduced performance can be attributed to the increase in cache misses due to conflicts, as L1 cache has limited associativity: 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

$$\{(i,j,k) | k = 0,1,2,...\} \tag{10}$$

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

$$\left\{ (i,j,k) \middle| j = j_0 + \begin{cases} 0,7, & x = 1,3,5,.. \\ 0,3,.. & x = 2,6,10,.. \\ 0,2,.. & x = 4,12,20,.. \\ 0,1,.. & x = 8,16,24,.. \end{cases} \right\} \tag{11}$$

also map to the same cache set. This is visualized in Figure 7 for the $x = 1$ case. From the figure, we can see that some cache lines are not being used. 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 become 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 set conflicts. In practice, a simple solution is to ensure that n is not a multiple of 64.

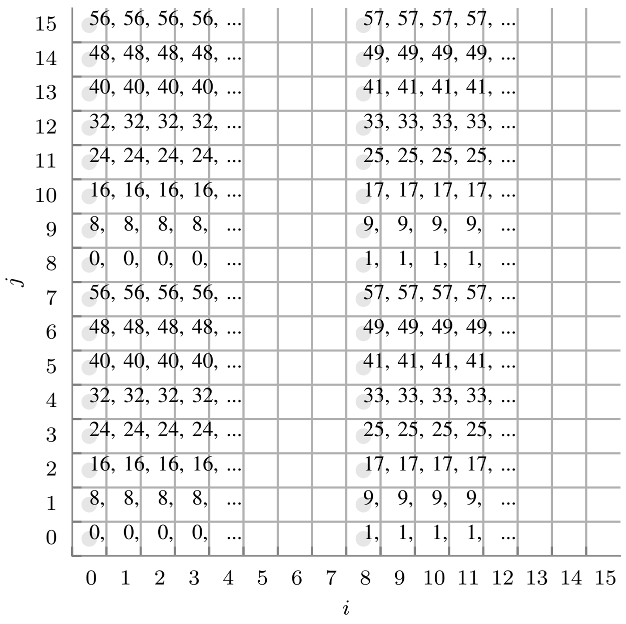

Fig. 7. Cache set mappings for the first 16x16x16 block of a 64x64x64 region (all units are in doubles). Note that a single cache block is 8 doubles. The grey circle represents the start of a cache block (memory is contiguous along i). The numbers represent the cache set numbers for the associated cache-block.

EF. *Experimental setup*

Results

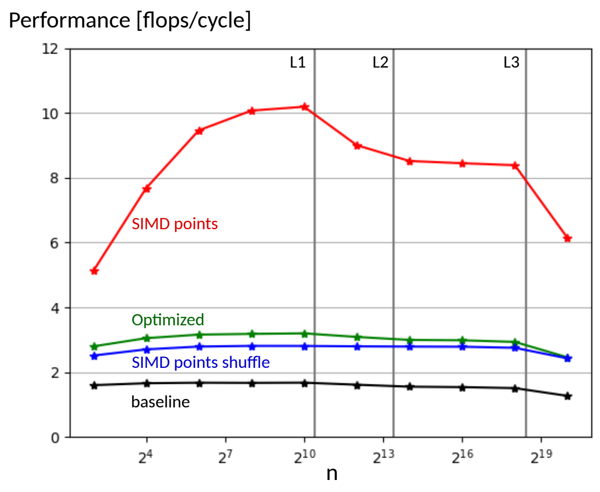

Fig. 8. Performance of EF

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.