# Justification

When we measured the execution times of ML2D, there were three major hotspots, using about 30% of the runtime each.

1. getAllSquaredDifferences
2. backProjection
3. updateOtherParams

Our goal was to reduce the elapsed time as far as practical using a cluster of multi-core systems.

# The three loops

The three loops all shared the following properties

1. The same basic loop structure (
   * For (every class, every image, every rotation, every translation) sometimes do a calculation based on a few long vectors selected by the loop indices
2. The calculation was very amenable to vector processing
3. The calculation for each the (class, image, rotation, translation) were independent
4. The calculations shared inputs
5. The total amount of data was huge
6. The number of classes and images was large enough that the outer two loops supplied more than enough parallelism for the available hardware

These can be simply described by the following example

for (int c = 0; c < numberClasses; c++)

for (int i = 0; i < numberImages; i++)

for (int r = 0; r < numberRotations; r++)

for (int t = 0; t < numberTranslations; t++)

if (test(c,i,r,t)) calculation(c,i,r,t);

The calculations are slightly more complex versions of

auto classVector = classVectors[c,r];

auto imageVector = imageVectors[i,t];

auto otherVector = otherVectors[c,i];

for (int s = 0; s < numberSamples; s++)

combine(classVector[s], imageVector[s], otherVector[s])

The combine was a simple arithmetic calculation, something like

(classVectorElt – imageVectorElt) \* otherVectorElt

# The approach

It was immediately recognized that the key to optimizing these was to introduce four techniques

1. Use as many cores as possible
2. Use as little L3 and beyond memory traffic as possible
3. Use as many vector units as possible
4. Try to use the data loaded into registers several times before loading more

# Details

## Introducing parallelism

Introducing parallelism was easy. We were expecting tens of classes and thousands of images, so that the product of the number of classes and number of images greatly exceeded the number of available cores.

So simply introducing an omp parallel for added the parallelism

#pragma omp parallel for collapse(2)

for (int c = 0; c < numberClasses; c++)

for (int i = 0; i < numberImages; i++)

The shared inputs were not modified in the loops, so were not a problem

There were some outputs that were the sums of values computed in the combine step, and these were optimized by computing the sum into a temporary, and then using a lock while adding the sum into the final sum of sums.

## Tiling over the samples to reduce L2 miss rates

The next challenge was to reduce the L2 miss rate. Before considering the effect of the if (test(c,i,r,t)), let us examine the slightly easier problem when the calculation is always done.

Because we are doing each (c,i) pair in parallel, we can focus on the inner loops. The input to the

for (int r = 0; r < numberRotations; r++)

for (int t = 0; t < numberTranslations; t++)

auto classVector = classVectors[c,r];

auto imageVector = imageVectors[i,t];

auto otherVector = otherVectors[c,i];

Since the same classVectors[c,r] and otherVector[c,i] are used for all the t loops, it would be very desirable if they stayed in the L1 cache from one iteration to the next. The numberSamples is expected to be about 90,000, so each t loop iteration consumes 90,000 x 3 x bytes per element. The original code uses double as the element type, so this will be about 2MB – far in excess of the 32KB L1 cache or the 256KB of the L2 cache. Since the L2 cache may be shared by two cores, this means that we need to start on a second t loop iteration after approximately 32KB / 3 x 8 = 1000 samples are processed. To achieve this, the loop is rewritten as approximately

for (int sLo = 0; sLo < numberSamples; s += 1000)

for (int r = 0; r < numberRotations; r++)

for (int t = 0; t < numberTranslations; t++)

auto classVector = classVectors[c,r];

auto imageVector = imageVectors[i,t];

auto otherVector = otherVectors[c,i];

for (int s = 0; s < numberSamples; s++)

combine(classVector[s], …)

Some tweaking is required for the combine to work this way, saving its intermediate results from one sLo iteration to the next, but not much.

There are two ways that we could continue this narrative – one is describing loop-unrolling the t loop, showing that the classVector and the otherVector are shared across the unrolled iterations, and thus only have to be fetched once. But this cannot be done in our case, because of the if (test(c,i,r,t)), so instead we will cope with that, and save these loads later using a slight variation on unrolling.

## Coping the condition around the innermost loop

This condition means that only some of the (r,t)pairs are processed by the innermost loop.

To get the benefit of unrolling, we search all the (r,t)pairs and only enter those that must be done into a list. This list can then be processed to get the benefits of unrolling.

ListOfRT list;

for (int r = 0; r < numberRotations; r++)

for (int t = 0; t < numberTranslations; t++)

if (test(c,i,r,t)) list.append(r,t)

This list can then be processed to get the benefits of unrolling. But the obvious approach does not work, because it is not clear that iterations share the same list[e].r

for (int e = 0; e < list.size(); e++)

calculation(c,i,list[e].r,list[e].t);

Instead we preprocess the list, merging elements that share a common list[e].r.

While doing this preprocessing, which builds a plan, we can do more than just combine consecutive elements that share a common t value. This is very desirable, since it saves fetches. In addition we can try to do the (r,t) pairs that are in the cache, rather than just doing the next one.

This selection is implemented using a grid and a table that keeps track of what is still in the cache.

This diagram might help: Assuming that we have just finished the grey squares, and removed their X’s, we have a choice. We should choose to do (3,1)+(3,2)+(3,3) next, benefitting from the current cache contents, rather than doing (2,3)+(2,4) which may eject the current cache contents without reusing them.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Rot\Trans | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 0 |  |  |  |  |  |  |  |
| 1 |  |  |  |  |  |  |  |
| 2 |  |  |  | x | x |  |  |
| 3 |  | x | x | x |  |  |  |
| 4 |  |  |  | x |  |  |  |
| 5 |  |  |  |  | x |  |  |
| 6 |  |  |  |  |  |  |  |

There is a compromise between finding the best pattern – which takes time – and doing a less optimal pattern. The current code is just a crude approximation.

At the end of this processing, there is a list of “steps”, where each step is one of the following

1. A set of 9 to do – 3 rotations which must be combined with 3 translations
2. A single rotation that must be combined with anything from 2 to 5 translations
3. A single rotation and translation that must be combined.

Each of these cases is special cased, to eliminate redundant loads. For example, the single rotation triple translation case is coded as

auto classVector\_r0 = classVectors[c,r0];

auto imageVector\_t0 = imageVectors[i,t0];

auto imageVector\_t1 = imageVectors[i,t1];

auto imageVector\_t2 = imageVectors[i,t2];

auto otherVector = otherVectors[c,i];

for (int s = 0; s < numberSamples; s++) {

combine(classVector\_r0[s], imageVector\_t0[s], otherVector[s]);

combine(classVector\_r0[s], imageVector\_t1[s], otherVector[s]);

combine(classVector\_r0[s], imageVector\_t2[s], otherVector[s]);

}

The single rotation and single translation case is similarly placed in a loop with several others, which allows some overlapping in the hardware, and the sharing of the fetch of the otherVector.

## Vector processing the innermost loop

The Intel compilers have no problem vectorising the innermost loops, even the 3x3 version. In that one case, there is not enough vector registers for the complex combine in real code, unless the AVX512 instruction set is available.

## Using float instead of double

While the above vectors are kept by the rest of the code in double, these kernels can get adequate results when the values have been cast to float. Since this allows two times as many values to be kept in cache, and since the arithmetic operations are faster on float, it is worth converting the values to float before feeding them into the above code in all cases except when the test(c,i,r,t)is so rare that the vectors are only processed once.

In the code, this is implemented by a Capture class. Each vector is converted from the original double to float just before it is needed for the first time, so that the result will be in cache when needed.

This conversion if not done for vectors that are used exactly once, unless it is to combine them with another vector that is used more than once.

## Eliminating one last multiply

There is one last optimization done in the code.

The loops are applying combine to complex numbers rather than to real numbers, and the formula is

sum += ( (a.r – b.r)\*\*2 + (a.i – b.i)\*\*2 ) \* w

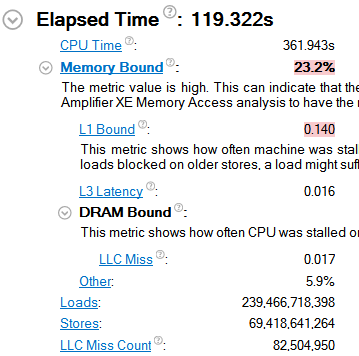
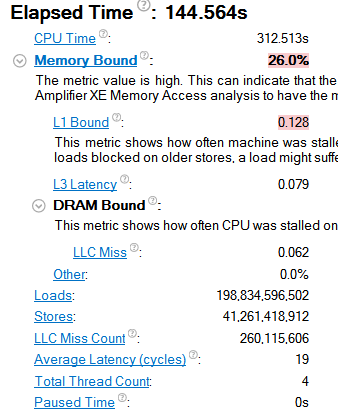
Since the various values are repeatedly used, it is worthwhile replacing a.r with aTimesSqrtW.r by capturing the float(a\*sqrt(w)) instead of just a. This formula can then be replaced with

sum += ( (aTimesSqrtW.r – bTimesSqrtW.r)\*\*2 + … )

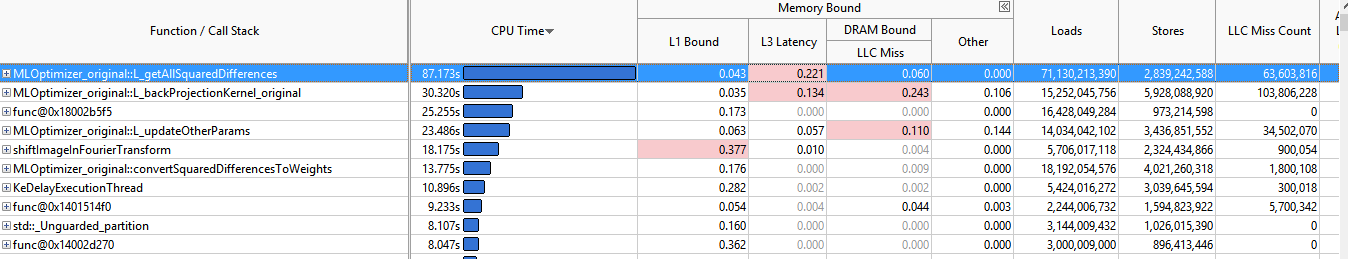
# Summary

All these techniques, combined, have changed the L1 cache miss rates, which may enable the multiple cores of a Xeon Phi to all be running at full speed when executing these loops.

Before After



Before



After

