SOLUTION NOTES

Comparative Architectures 2002 Paper 7 Question 3 (IAP)

(*a*) The above algorithm requires that each of the locations of A,B and C are read N times. Ideally, we'd like all of the accesses to the same location to occur in quick succession so that the temporal locality results in L1 cache hits. Furthermore, we'd like the inner-most loops to to be accessing the arrays using unit stride to make use of spatial locality.

When all three matrices fit in the L1 cache this code performs reasonably OK, except for missing out on some of the implicit prefetch that would have been provided through spatial locality.

When presented with large matrices, this code goes badly wrong. Firstly, there will be little benefit from temporal locality. Although each element of B will only be accessed only once, elements from A and C will have to be fetched into the cache N times since they will almost certainly be displaced before they are needed again (capacity misses).

Secondly, this code does not make good use of spatial locality though unit stride of the inner most loop. Each cache line is likely to contain several elements (e.g. 8 64 bit double precision 64 bit FP values in a 64 byte cache line), so significant benefit is squandered. Furthermore, the non-unit stride accesses may well cause the TLB to thrash.

(*b*)

```
for(k=0;k<N;++k){
  for(j=0;j<N;++j){
    for(i=0;i<N;++i){
C[k][i] = C[k][i] + ( A[k][j] * B[j][i] );
    }
  }
}
```

This loop order means that the accesses to both B and C in the inner loop are unit stride, so we will benefit from spatial locality implicit prefetch. A[k][j] remains constant during the inner loop but there will also be some benefit from spatial locality due to j being the middle loop.

(*c*) For large matrices, the above algorithm gets little benefit from temporal locality since data is displaced before it is reused. The algorithm below computes the multiply in blocks of size b.

Ignoring associativity misses the algorithm avoids unnecessary cache misses provided the following can all be held in the cache simultaneously: one row of b elements of C, one cache line from A, and a bxb tile of B. The blocking factor is chosen to enable this.

Due to the limited associativity of most modern L1 caches it may be necessary to pick a smaller block size.

```
for (kb=0;kb<N;kb+=b){
   for (jb=0;jb<N;jb+=b){
     for (ib=0;ib<N;ib+=b){
for(k=kb;k<kb+b;++k){
  for(j=jb;j<jb+b;++j){
    for(i=ib;i<ib+b;++i){
      C[k][i] = C[k][i] + ( A[k][j] * B[j][i] );
    }
  }
}
     }
   }
 }
```

The above algorithm could further be improved by adding another level of blocking e.g. to implement a smaller block size that can be computed holding all the values in registers.

(*d*) Since this algorithm is basically cache-limited it is unlikely that any benefit could be achieved though parallelising it on a single processor sharing a single cache. A naive attempt could make matters much worse by causing more cache misses.