Superscalar programming 101 (Matrix Multiply)

(part 3)

In the previous article we have seen that by reorganizing the loops and with use of temporary array we can observe a performance gain with SSE small vector optimizations (compiler does this) but a larger gain came from better cache utilization due to the layout change and array access order. The improvements pushed us into a memory bandwidth limitation whereby the Serial method now outperforms the Parallel method (of the Serial method).

The memory bandwidth limitation is an important factor to consider and warrants a change in programming strategy: In order to attain additional performance gains we are going to attack the problem from the perspective of trying to keep the data access patterns to the closest cache level.

But how are we going to do this?

On a system with Hyper Threading, such as the Core i7 920, the Hyper Threads within a single core share the 256KB L2 cache and, depending in internal architecture, share the 32KB L1 data cache. While all cores within the socket share the 8MB L3 cache (6MB or 12MB on other processor models).

The Cilk++ method uses the common practice of “divide and concur” to partition (or tile) the matrix into smaller working sets that nicely fit into the cache available to a thread. This is a good starting point. What we need to look at next is how to coordinate the activity *between* caches within the processor(s) of the system.

While you can tile using nested loops, consider this: As you reduce the tile size, you also increase the number of interactions with the thread scheduler (entering and exiting the inner most loop). Thread scheduling is not cheap. In the first chart, the overhead break even occurred at 50 x 50 tile size. Is there a different way to program such that you can use 2x2 or 2x1 or 1x2 tiles and attain superior performance?

The answer to this is: Yes

Targeting tasks to specific threads, or grouping of threads is difficult to do effectively using most threading tool kits (e.g. MS Parallel Collections, OpenMP, or Cilk++). Cache level task grouping is a built-in design feature of QuickThread. Example:

```
// divide the iteration space across each multi-core processor
// L3$ specifies by L3 cache
//          .OR.
// within processor (Socket) for processors without L3 cache
parallel_for( OneEach_L3$, intptr_t(0), size,
    [&](intptr_t iBeginL3, intptr_t iEndL3)
    {
        // divide our L3 iteration space by L2 within this threads L3
        parallel_for( OneEach_Within_L3$ + L2$, iBeginL3, iEndL3,
            [&](intptr_t iBeginL2, intptr_t iEndL2)
            {
                // Here we are running as the Master thread of a
                // 2 team member team (or 1 in the event of older
                // processor)
                //
                // Now bring in our other team member(s)
```

```
                // form team of threads sharing this threads L2 cache
                parallel_distribute( L2$,
                    [&](intptr_t iTMinL2, intptr_t nTMinL2)
                    {
                        // ... (Do Work)
                    } // [&](intptr_t iTMinL2, intptr_t nTMinL2)
                ); // parallel_distribute( L2$,
            } // [&](intptr_t iBeginL2, intptr_t iEndL2)
        ); // parallel_for( OneEach_Within_L3$ + L2$, iBeginL3, iEndL3,
    } // [&](intptr_t iBeginL3, intptr_t iEndL3)
); // parallel_for( OneEach_L3$, intptr_t(0), size,
```

The outer loop is a per socket division, the next deeper loop is a per L2 cache, and the inner layer is a n-Way split by the threads sharing L2 (2 threads on Core i7 920, 2 threads on Q6600)

The technique is to use the **parallel_distribute** as the inner most division, then use the loop partitioning of the outer loops within a state machine loop placed inside the **parallel_distribute** level. Say what?

The parallel_for in QuickThread performs the task to thread set selection and partitioning of the iteration space, but specifically does not drive the iteration. Due to this feature of QuickThread, the iteration domain can be passed deeper into the loop nest levels. The second loop does the same thing (pass iteration space to inner most parallel_distribute. Now as to why this is important.

This alternate technique creates the environment for an intelligent swarm of threads performing a coordinated attack of the problem. These threads know which threads share the nearest cache, which threads share each of the largest cache(s), and which threads are not sharing caches with the current thread. This identification is implicit by the sub range of the outer two loops and by the team member number (`iTMinL2`) within the **parallel_distribute**.

Adding an additional outer layer loop per NUMA node could easily be handled using:

```
parallel_for( OneEach_M0$, intptr_t(0), size,
  ...
```

However, for this matrix multiplication, this would not provide any additional benefit unless your system has an L4 cache external to the processor and which is also shared amongst processors sharing the same NUMA node. NUMA aware issues are best handled in the memory allocation routines which can be placed within the L3 cache distribution layer of the above loop structure.

The interior state machine loop will partition (tile) the output array:

Socket (L3 cache)
Core Pair/HT Siblings (L2 cache/L1 cache)
Amongst Core Pair/HT Siblings (L2 cache/L1 cache)

To accomplish this, we segment the matrix into 2x2 tiles with each 2x2 tile being serviced by a 2 thread team. The 2 thread team I will call a “Tag Team” (as in Tag Team Wrestlers). On the Core i7 we will have 4 tag teams, one for each core, and the two threads for each team being Hyper Thread siblings within the same core. On Q6600 the tag team becomes the two threads sharing the same L2 (Q6600 has 2 L2 caches, each shared by 2 cores).

If you ever watch TV wrestling, there is a type of wrestling format where each team consists of two team members. Each team is *supposed to* have one member in the wrestling ring (square) at any one time, and they must tag their team mate when they wish to let them have a go at their

opponent. I say *supposed to* in italics since more often than not, the tag exchange usually ends up with the team doing the tag cheating by having two wrestlers in the ring at one time. And in these situations they completely overwhelm their opponent. I am going to show you how to do the equivalent of this using cache directed thread scheduling as available with QuickThread.

The output matrix is divided into 2x2 tiles. The thread teams are derived from the thread pool into 2 thread teams, each thread sharing the closest cache level possible. On Core i7 920 these are Hyper Thread siblings within each core, on Intel Q6600 these are two cores sharing same L2 cache, on AMD Opteron 270 these are two cores within same processor on same NUMA node, etc… An optimization strategy will be employed whereby we will concurrently perform the transposition and DOT products of some of the cells. Then perform the DOT products on the remaining cells.

The Hyper Thread siblings within one core have two integer execution paths, but only one floating point execution path. We code to take advantage of this by having one thread of the tag team, called the Master thread, perform the transpositions using the integer execution path while the other Hyper Thread sibling, called the Slave thread, is performing the DOT product of the lower 1x2 portion of the 2x2 tile concurrently with the transposition (shortly following the transposition of each cache line). Upon completion of the transposition, the Master thread will then begin the DOT product of the upper left cell of 2x2 tile, and is joined shortly thereafter by the Slave thread performing the DOT product on the upper right cell. For the cells handled in this manner (those performing the transposition), cost of the DOT product of half these cells is essentially free, since it occurs during memory latency of the transposition. For the DOT product of the other half of this 2x2 cell, the row and transposed column of one of these output cells is fully cached (potentially all in L1 cache) with the other output cell having the transposed column residing in L1 cache and leaving the row of this output un-cached.

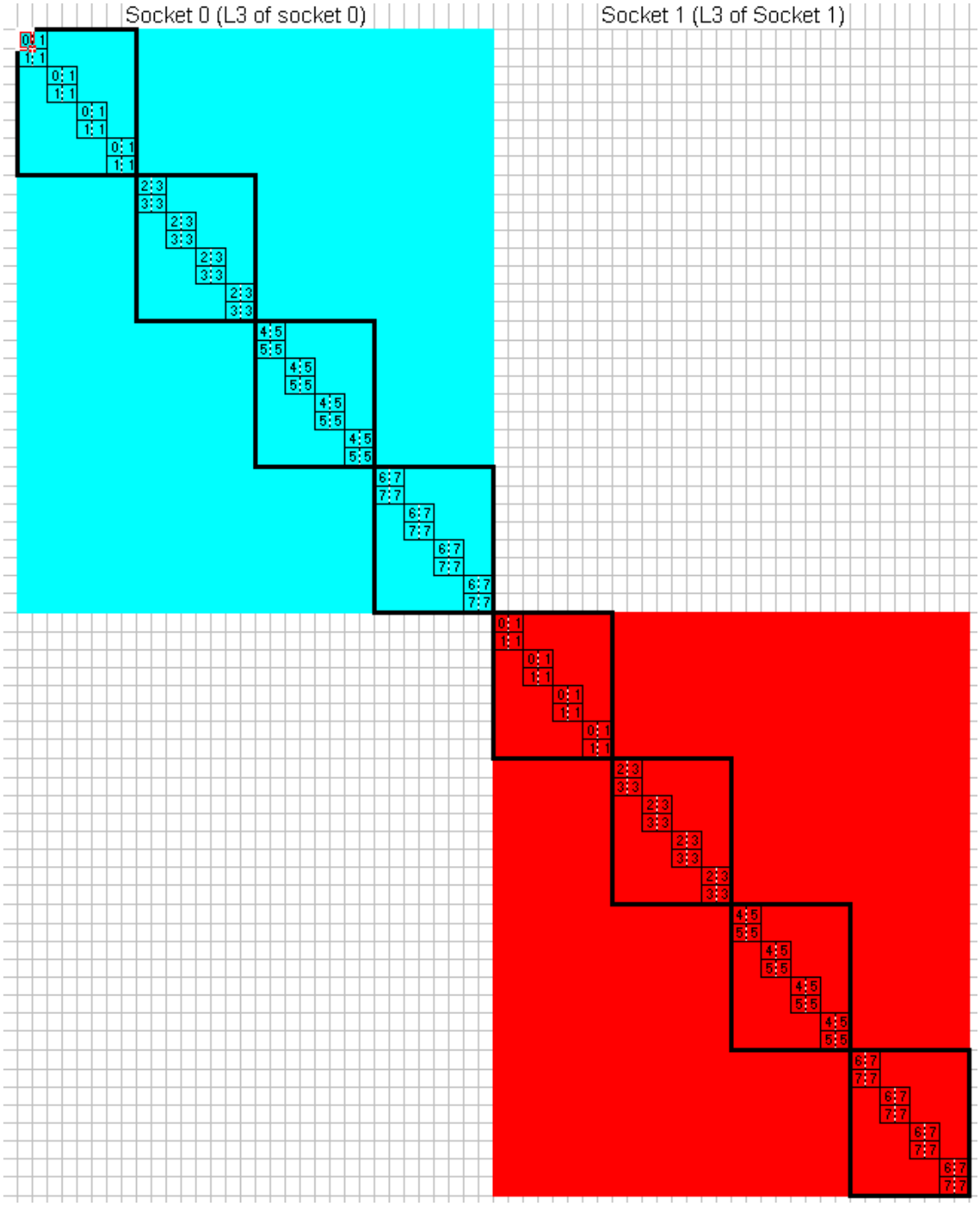
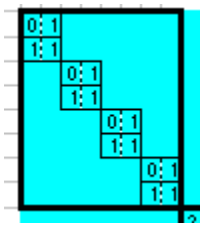


Fig 10

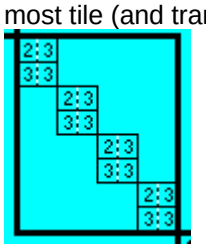
Figure 10 shows the initial Per Socket tile work assignment distribution (color backgrounds, an iterative per processor thread team (heavy outline), and 2x2 cell tile lightest outline.

The 2x2 tiles that lie on the diagonal require additional work for the transposition of the m2 array into m2t array. The master thread (even number of 2 member team) performs the transposition (memory access intensive) using the integer execution unit of the HT core, while the other team member performs the DOT product of 3 of the cells in the 2x2 tile using the floating point execution unit of the HT core. These DOT products are performed concurrent with the

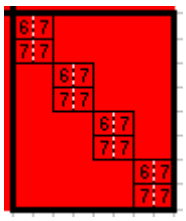
transposition by means of snooping on the progress of the transposition made by the master thread of the tag team. The master thread of the 2 member team then performs the final DOT product. Some experimentation yet needs to be done to see if the Slave thread of the 2 member team ought to perform all 4 DOT products concurrent with the transposition. See below for the {transpose, 1x2},{1x1, 1x1} method:



The first two member thread team works on the upper left most 2x2 tile (and transposition) of its designated zone (shown above). Concurrent with this, the second team works on the upper left most tile (and transposition) of its designated zone:

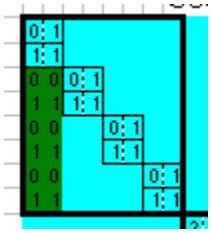


And so on to the last socket, last team working on the upper left most 2x2 tile (and transposition) of its designated zone:



The completion of these diagonal tiles are not signified by the end of a parallel construct. Instead, the completion is signified by writing a completion status into a mailbox. This completion will be asynchronous with the activities occurring by the other threads. And have the “overhead” induced by a non-Interlocked write to a shared memory mailbox.

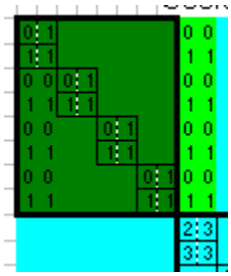
Once the upper left most tile of the diagonal is completed, the two member team consults a flag indicating if there is a work starved thread (early-on this flag will indicate no work starved threads. When there are no work starved threads, the two member team works on the 2x2 tiles in the column in same column of the diagonal tile just completed:



Signified by the green column above. However, each thread computes a 1x2 double DOT within the 2x2 tile. When that 2x2 tile is complete, the two member team consults a flag indicating if there is a work starved thread, if no work starved thread, the team continues down the column, if work starved, it progresses down the diagonal.

The goal of working down the column, just after transposition is the two columns just transposed are most likely to be residing hot in cache (L1, L2 or L3).

When a 2 member thread team completes its diagonal, and the column cells above/below its diagonals (within its two member team zone), The thread team then consults the transposition completed status of the other thread teams within its L3 cache (by way of mailbox)

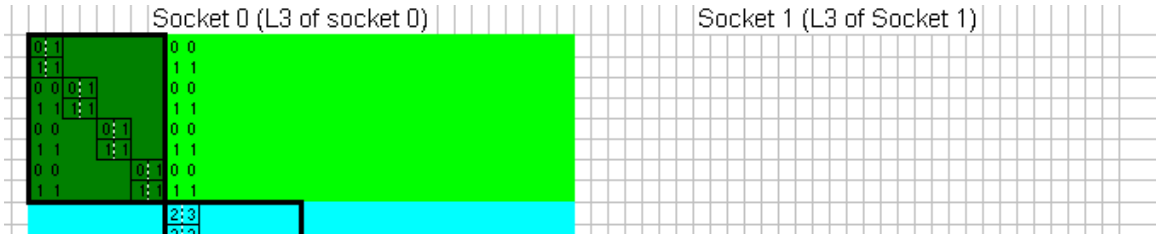


The light green column, is processed by the first team (0/1 in socket 0), after its work has complete, and after second team (2/3) has completed the upper left most diagonal. Which statistically will be done by the time the mailbox is consulted (after 4 transpose with DOTs, plus 12 double DOTs time delay).

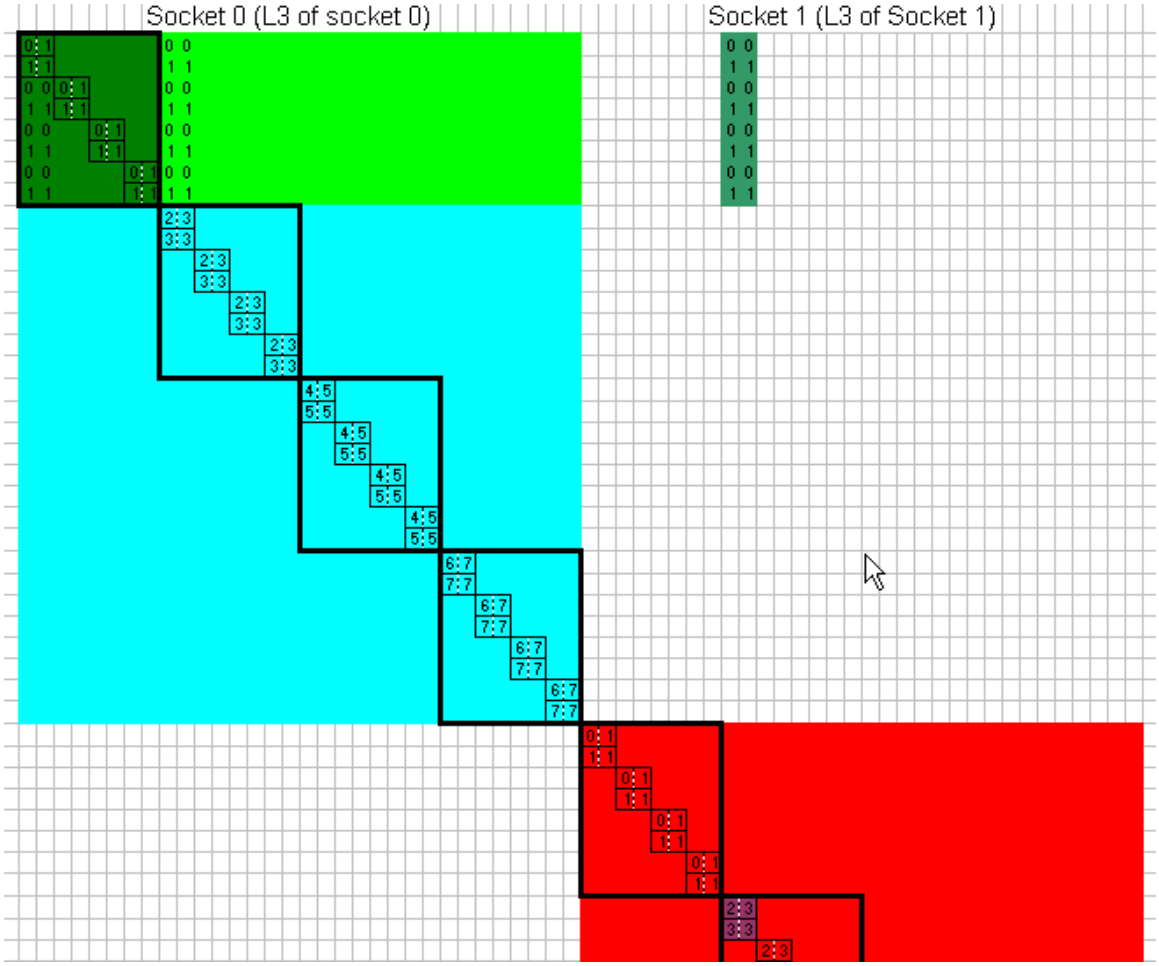
Each thread team does the same. As the thread team progresses over/under the columns of the diagonals of the teams sharing its L3 cache, if it finds an uncompleted designated column within its L3, it posts a “work starved” thread notification such that its L3 cache sharing teams can interrupt column processing and advance to next diagonal processing prior to completing its current column.

Each team, with exception for work starved thread indication, works independent of the other thread teams within its socket.

This iterative process continues until a thread team completes all the designated work within its socket tile:

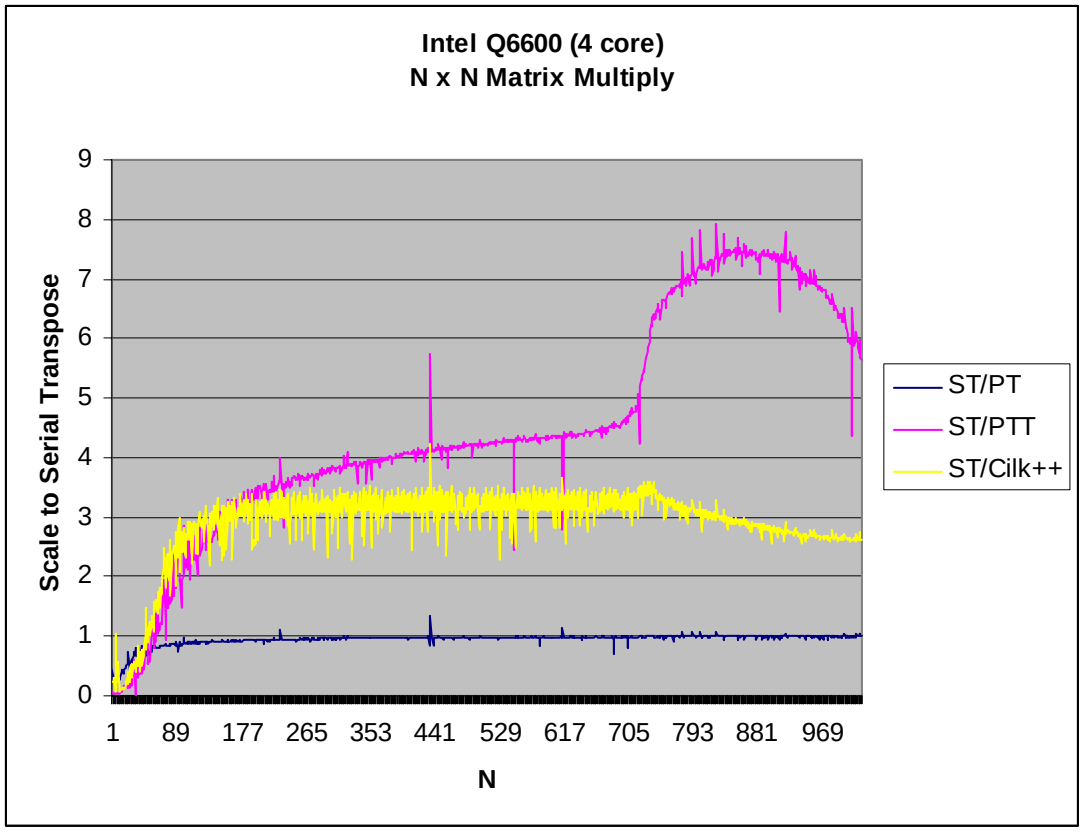


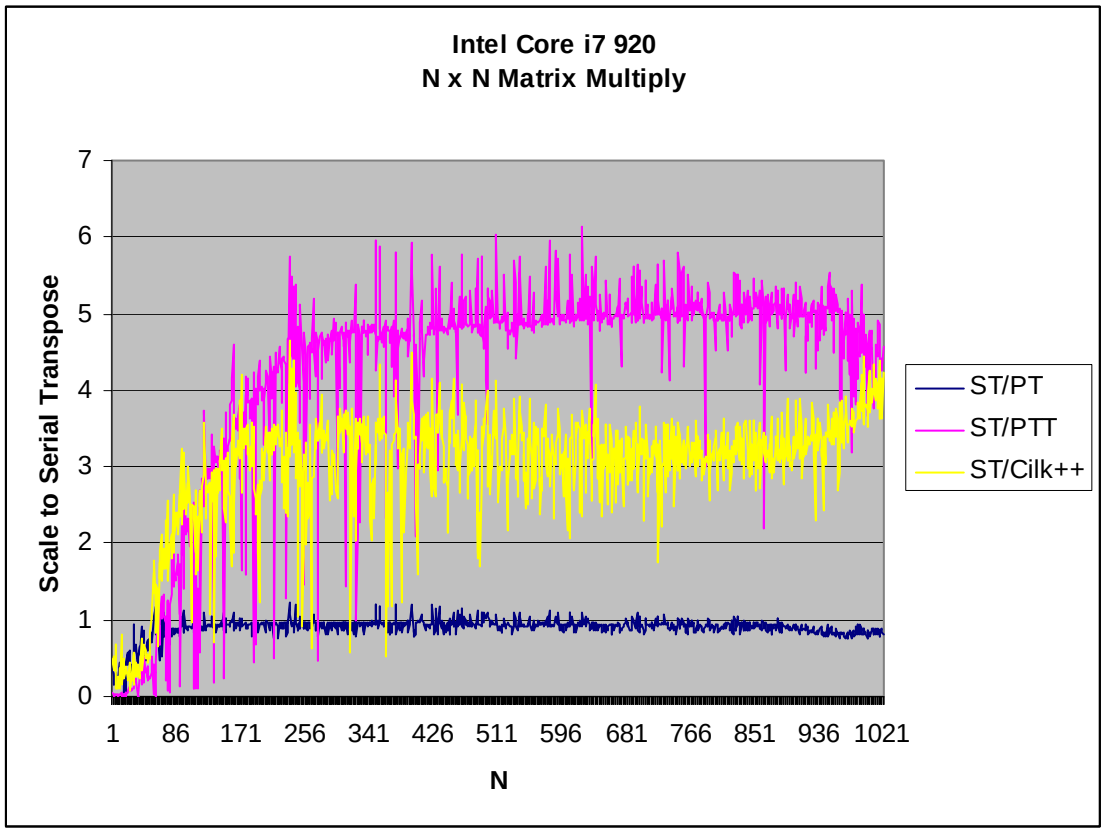
At this point, it now consults the diagonal completion status mailboxes for the diagonals of the other sockets. Being that this is a state machine instead of nested inner loops, the diagonal completion of the other sockets can occur in any order, but will tend to occur on diagonal 2x2 tiles in ascending order within each 2 member team, in each additional socket. As indicate by separated green bar, to right of Socket 0 team 0/1 zone in Socket 1 zone above completed diagonal for second team in red zone (socket 1) below.



As an additional optimization, on systems with NUMA capability, the rows of each array (m1, transpose m2 and output array) are segregated with NUMA considerations. In the 2 socket system, the upper half of the rows (turquoise/green) are in socket 0 locality, and the lower half (red) are socket 1 locality.

Now let’s produce the charts and check the results data:


Intel Q6600 (4 core)
N x N Matrix Multiply
Scale to Serial Transpose
0
1
2
3
4
5
6
7
8
9
1
89
177
265
353
441
529
617
705
793
881
969
N
ST/PT
ST/PTT
ST/Cilk++


Intel Core i7 920
N x N Matrix Multiply
Scale to Serial Transpose
0
1
2
3
4
5
6
7
1
86
171
256
341
426
511
596
681
766
851
936
1021
N
ST/PT
ST/PTT
ST/Cilk++

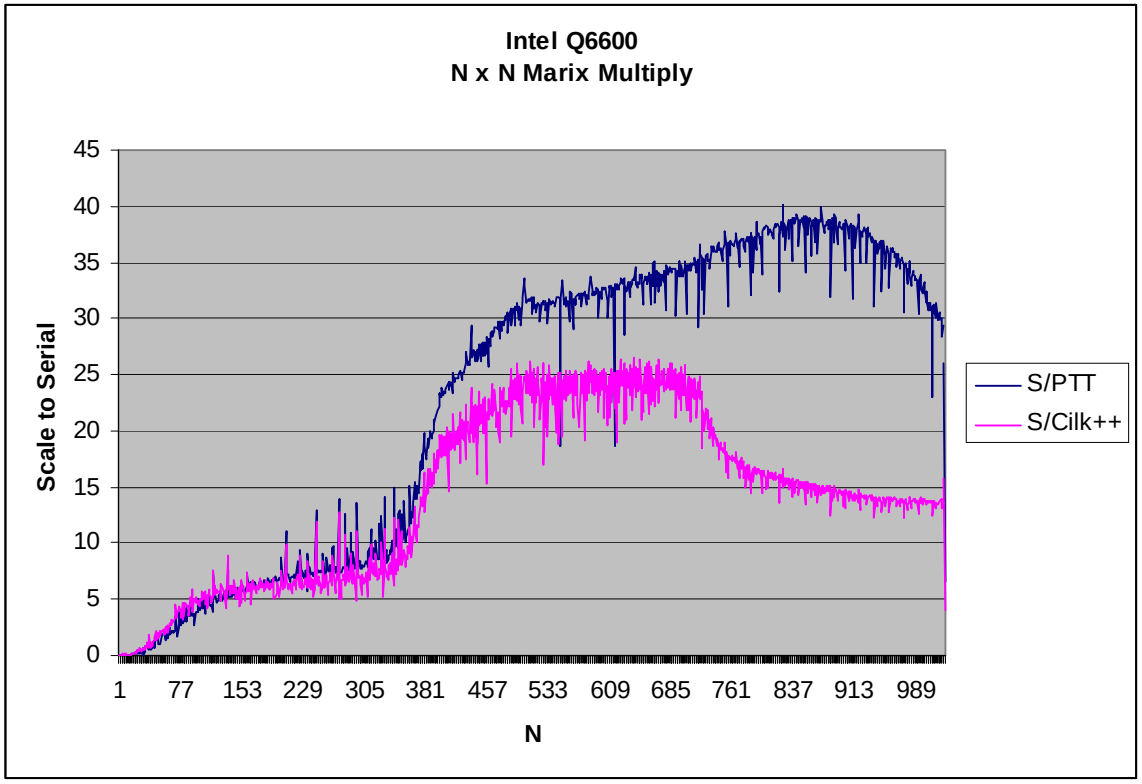
The average scale factors (compared to Serial Transpose) for N = 128 : 1024 are:

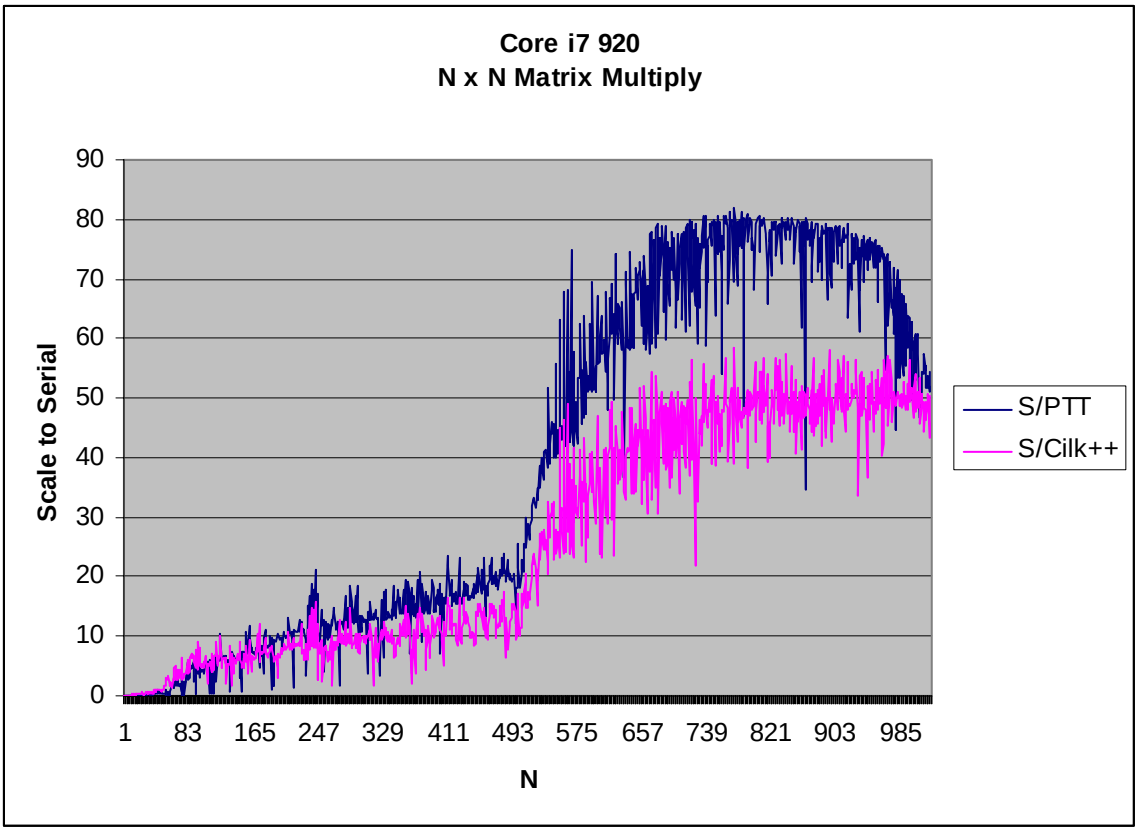
Q6600 4.96x for Parallel Transpose Tag Team and 3.06x for Cilk++
Core i7 920 4.69x for Parallel Transpose Tag Team and 3.20x for Cilk++

The peak values for selected section, ~880 of Q6600 shows ~ 7.5x for Parallel Tag Team vs 3x for Cilk++ method.

The Parallel Tag Team definitely has the advantage over the Cilk++ method (at least in this N range).

To inflate the figures, let's compare back to original Serial method without Transpose





We can now observe that at selected points in the chart we have attained a 38x improvement over Serial on Q6600 and near 80x improvement in scaling over the Serial on Core i7 920.

The complete description of Parallel Transposition Tag Team can be found in the sample code for MatrixMultiply.cpp included in the QuickThread demo kit. www.quickthreadprogramming.com

Can this be improved upon…

I will have to say, yes it can.

Early on in this blog post I mentioned: ”save intrinsic small vector functions for last”.

Internal to the MatrixMultiply, the inner most loop performs a DOT product. All the programming variations are using an inline function to perform the DOT product. The QuickThread Parallel Tag Team method (PTT) uses an additional variation on this DOT function to produce two DOT products at the same time. Each thread of the 2 thread team working on a 2x2 tile, produce results for half of this tile (1x2). Performing the two DOT products concurrently within each thread improves cache hit probability on the larger matrix sizes.

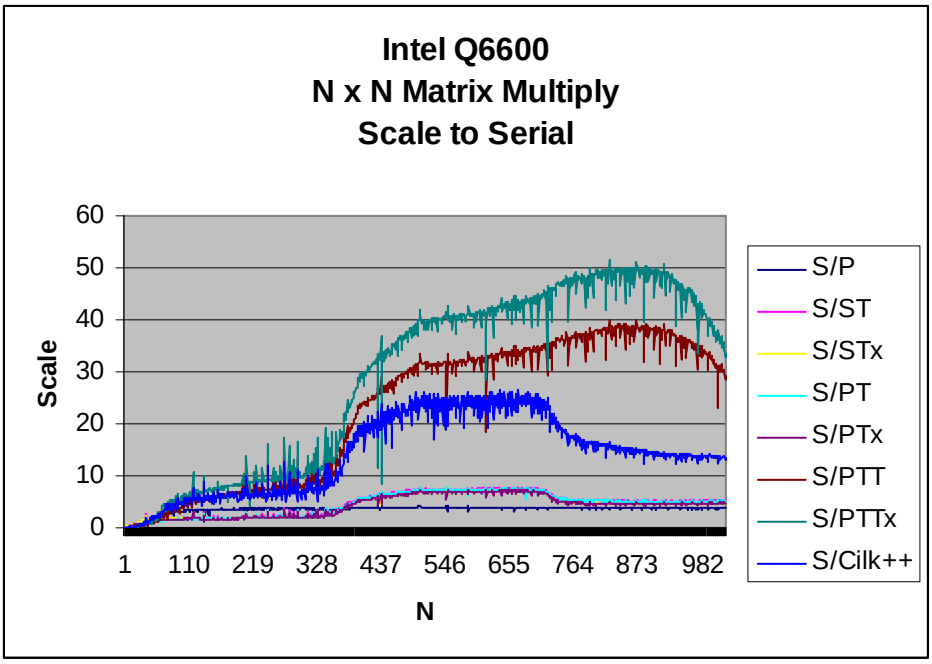
Below are the two DOT functions, the two functions modified to use xmm intrinsics (I am not very good at using xmm intrinsics, you may be able to do better), and two selector functions that call the appropriate DOT within the test program.

```
// compute DOT product of two vectors, returns result as double
// used in QuickThread variations
double DOT(double v1[], double v2[], intptr_t size)
{
```

```
    double temp = 0.0;
    for(intptr_t i = 0; i < size; i++)
    {
       temp += v1[i] * v2[i];
    }
      return temp;
}

double xmmDOT(double v1[], double v2[], intptr_t size)
{
      // __declspec(align(16)) not working reliably for me
      double temp[4];
      intptr_t alignedTemp = (((intptr_t)&temp[0]) & 8) >> 3;

      __m128d     _temp = _mm_set_pd(0.0, 0.0);
      __m128d *_v1 = (__m128d *)v1;
      __m128d *_v2 = (__m128d *)v2;

      intptr_t    halfSize = size / 2;

      for(intptr_t i = 0; i < halfSize; i++)
      {
         _temp = _mm_add_pd(_temp, _mm_mul_pd(_v1[i], _v2[i]));
      }
      // fix code to remove temp[4] array
      _mm_store_pd( &temp[alignedTemp], _temp);
      if(size & 1)
            temp[alignedTemp] += v1[size-1] * v2[size-1];

      return temp[alignedTemp] + temp[alignedTemp+1];
}

// compute two DOT products at once
// effectively
// r[0] = DOT(v1, v2, size);
// r[1] = DOT(v1, v3, size);
// except running both results at the same time
void DOTDOT(double v1[], double v2[], double v3[], double r[2],
intptr_t size)
{
      double      temp[2];
      temp[0] = 0.0;
      temp[1] = 0.0;
      for(int i=0; i < size; ++i)
      {
            temp[0] += v1[i] * v2[i];
            temp[1] += v1[i] * v3[i];
      } // for(int i=0; i < size; ++i)
      r[0] = temp[0];
      r[1] = temp[1];
}

// compute two DOT products at once
// effectively
// r[0] = DOT(v1, v2, size);
// r[1] = DOT(v1, v3, size);
// except running both results at the same time
```

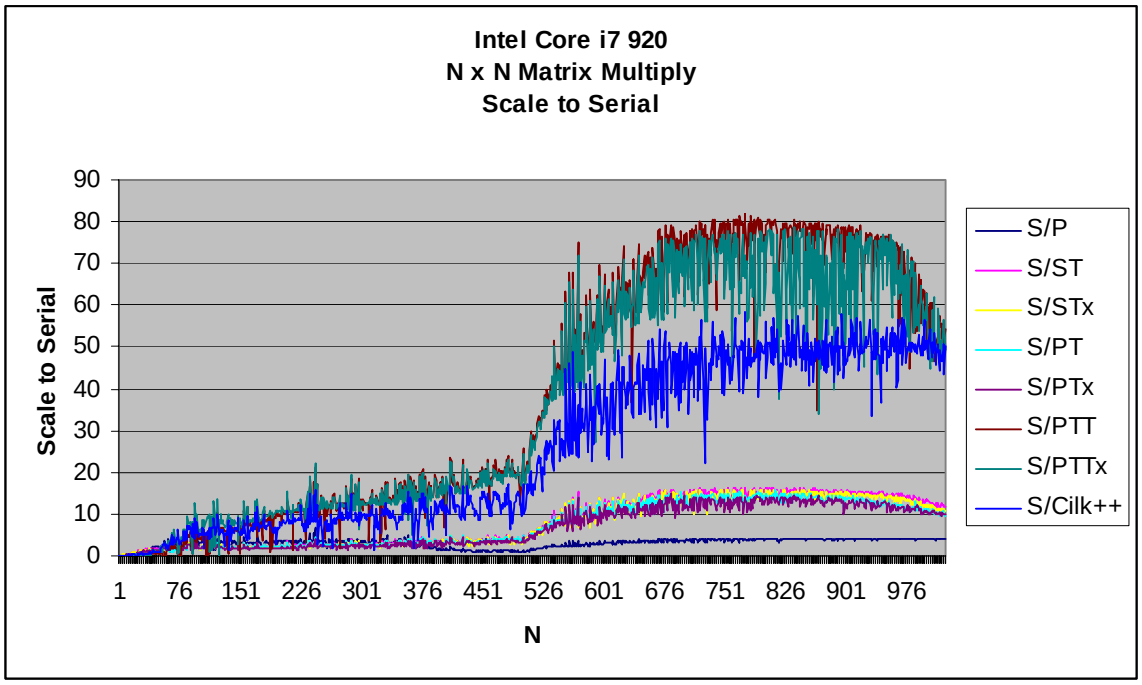
```
void xmmDOTDOT(double v1[], double v2[], double v3[], double r[2],
intptr_t size)
{
      // __declspec(align(16)) not working reliably for me
      double      temp[6];
      intptr_t alignedTemp = (((intptr_t)&temp[0]) & 8) >> 3;
      __m128d     _temp0 = _mm_set_pd(0.0, 0.0);
      __m128d     _temp1 = _mm_set_pd(0.0, 0.0);
      __m128d *_v1 = (__m128d *)v1;
      __m128d *_v2 = (__m128d *)v2;
      __m128d *_v3 = (__m128d *)v3;

      intptr_t    halfSize = size / 2;

      for(intptr_t i = 0; i < halfSize; i++)
      {
         _temp0 = _mm_add_pd(_temp0, _mm_mul_pd(_v1[i], _v2[i]));
         _temp1 = _mm_add_pd(_temp1, _mm_mul_pd(_v1[i], _v3[i]));
      }
      _mm_store_pd( &temp[alignedTemp], _temp0);
      _mm_store_pd( &temp[alignedTemp+2], _temp1);
      if(size & 1)
      {
            temp[alignedTemp] += v1[size-1] * v2[size-1];
            temp[alignedTemp+2] += v1[size-1] * v3[size-1];
      }

      r[0] =  temp[alignedTemp] + temp[alignedTemp+1];
      r[1] =  temp[alignedTemp+2] + temp[alignedTemp+3];
}

bool UseXMM = false;

double doDOT(double v1[], double v2[], intptr_t size)
{
      if(UseXMM)
            return xmmDOT(v1, v2, size);
      return DOT(v1, v2, size);
}

void doDOTDOT(double v1[], double v2[], double v3[], double r[2],
intptr_t size)
{
      if(UseXMM)
            xmmDOTDOT(v1, v2, v3, r, size);
      else
            DOTDOT(v1, v2, v3, r, size);
}
```

As stated in the comments, I've experience an alignment issue with the compiler directive and had to resolve this with a little tweak of the code as a work around. The incorporation of the xmm intrinsic functions into these two routines were relatively easy (for an inexperienced xmm programmer like myself). Let's look at the results:



On the Q6600 (4 core no HT) the S/PTTx with the xmm intrinsic functions clearly exhibited a benefit, approximately 30% improvement. The QuickThread Parallel Tag Team “x” method preponderantly uses the xmmDOTDOT (double DOT function). However, the S/STx, using the xmmDOT (single DOT) function performs slightly worse than the C++ code without the intrinsic functions. This would indicate the compiler optimizations were better than the hand optimizations of an inexperienced intrinsic programmer (not unusual).

Looking at the Core i7 920 we see a different picture:



This processor, with HT, experienced a detriment in performance (-12%) in using the hand written xmm helper functions. Note, the executable was the same for both systems.

For you as a vendor of a program for use on various systems, this is an important piece of information. Knowing the platform specific behavior means you can query the system at program startup and then change the selection of which variant of the code to use. Typically you would involve selecting a functor (function pointer) in the code as opposed to using a selection function.

An alternate approach for unknown behavior is to use a heuristics approach. The application would select each variant of your code on the first few calls and measure the time of execution for each method, then after enough samples are taken, select the best performing variation of the functions and insert the appropriate functor into the dispatch pointer.

Can this be improved upon…

I will have to say, yes it can.

Note the chart lines are rather “noisy”. Additional tuning can improve the harmony and thus move the trend line upwards. This amounts to an estimated additional 15% over the current Parallel Transpose Tag Team method. (xmmDOTDOT on non-HT systems, DOTDOT on HT systems)

15% is usually not worth going after, however, note that both Parallel Tag Team method and the Cilk++ method appear to be dropping off at 1024. Additional tests should be run with larger matrixes, and more importantly on multi-socket systems. When I have an opportunity to collect such data, I will be in the position to publish updated information regarding this performance test.

Is there a superior technique that can improve upon this?
History shows, the answer to this is yes.

In Part-4 we will examine issues relating to multi-socket systems.

Jim Dempsey
jim@quickthreadprogramming.com
www.quickthreadprogramming.com