Elusive Algorithms

jim@quickthreadprogramming.com

This article on parallel programming will choose one of those elusive algorithms that upon first glance seem to be neither vectorizable nor parallelizable. The intent of this article is not to address the specific algorithm, but rather to provide you with an approach to problems that share similarities with this algorithm. The elusive algorithm for this article is the inclusive scan:

```
float Simple_inclusive_scan(float b[], float a[], int n)
{
  float carry = 0.0f;
  for(int i=0; i < n; ++i)
  {
    b[i] = (carry += a[i]);
  }
  return carry;
}
```

In:	1	2	3	4	5	6	7	8	…
Out:	1	3	6	10	15	21	28	36	…

Where the output is the sum of the prior output (or 0 for first), and the value of the input. This loop has a temporal dependency that, at first inspection, defies both vectorization and parallelization.

For those readers not adventuresome enough to use the vector intrinsic instructions, the last chart in the article addresses parallelization without vectorization as compared to parallelization with vectorization. It is recommended that you read through the vectorization section to show you what is involved.

Let's consider the performance we expect from this loop. Assuming the processor cache line is 64 bytes, it can hold 16 floats. This means that 15 of the fetches of a[] will reside in the L1 cache and the 16'th (the 1st in the cache line) will come from RAM (or from an outer level cache or may be in flight if noticed by the hardware prefetcher). This means that 15 of 16 fetches are very fast, (on the order of 4 clock cycle latency), the addition will take another clock cycle as will the store (15/16ths of the time). If this loop is unrolled, you should, therefore, get close to 6 clock cycles per element for 15 out of 16 elements, while the 16th will take longer.

What about using the AVX __mm256 intrinsic instructions on your host processor or on Xeon Phi the _mm512 intrinsic instructions? They certainly could speed things up. Assume you look at the problem a different way for the AVX attempt. The AVX instruction set has swizzle, permute, shuffle and mask instructions. Can these be put to use? This can be sketched:

Fetch:	1	2	3	4	5	6	7	8	…
Swz+prm		1	1	1	1	1	1	1	…
add	1	3	4	5	6	7	8	9	

Swz+prm		2	2	2	2	2	2	…
add	1	3	6	7	8	9	10	11
Swz+prm			3	3	3	3	3	…
add	1	3	6	10	11	12	13	14
Swz+prm				4	4	4	4	…
add	1	3	6	10	15	16	17	18
Swz+prm					5	5	5	…
add	1	3	6	10	15	21	22	23
Swz+prm						6	6	…
add	1	3	6	10	15	21	28	29
Swz+prm							7	
add	1	3	6	10	15	21	28	36
Out:	1	3	6	10	15	21	28	36 …

This appears to require 16 instructions, however, 3 of the swz/prm steps may require a swizzle and a permute instruction. The sketch would seem to indicate that we need about 23 instructions to get the job done.

Is there a better way to implement this?

The answer is yes.

Fetch:	1	2	3	4	5	6	7	8 …
perm:	1	1	3	3	5	5	7	7 ([0]->[1],[2]->[3]…[6]->[7])
and:	0	1	0	3	0	5	0	7 (remove unwanted)
add:	1	3	3	7	5	11	7	15
perm:	1	3	3	3	5	11	11	11 ([1]->[3:2], [5]->[6:2]
and:	0	0	3	3	0	0	11	11 (remove unwanted)
add:	1	3	6	10	5	11	18	26
perm:	0	0	0	0	1	3	6	10
perm:	0	0	0	0	10	10	10	10
add	1	3	6	10	15	21	28	36
Out:	1	3	6	10	15	21	28	36 …

This implementation requires only approximately 11 instructions; fewer than half the 23 we used in our first attempt

The actual code will require propagating a carry from one vector to the next. The AVX implementation looks like this:

```
float Vector_inclusive_scan(__m256* Bv, __m256* Av, int Nv)
{
  __m256  Zero = _mm256_setzero_ps();
  __m256  Carry = _mm256_setzero_ps();
  __m256 mask_F0F0F0F0 =
_mm256_castsi256_ps(_mm256_set_epi32(0xFFFFFFFF,0x00000000,0xFFFFFFFF,0x00000000,0xFFFFFFFF,0x00000000,0xF
FFFFFFF,0x00000000));
```

```
  __m256 mask_FF00FF00 =
_mm256_castsi256_ps(_mm256_set_epi32(0xFFFFFFFF,0xFFFFFFFF,0x00000000,0x00000000,0xFFFFFFFF,0xFFFFFFFF,0x0
0000000,0x00000000));
  if(Nv-- > 1) // see if more than one vector, post decriment Nv
  {
    __m256 Out = Av[0];  // Out= 8, 7, 6, 5, 4, 3, 2, 1  (AVX register notation High:Low)
    for(int i = 0; i < Nv; ++i)
    {
      __m256 OutNext = Av[i+1]; // prefetch
                // Out= 8, 7, 6, 5, 4, 3, 2, 1  (AVX register notation High:Low)
      __m256 t1 = _mm256_permute_ps(Out,(2<<6)+(2<<4)+(0<<2)+0);  // t1 = 7, 7, 5, 5, 3, 3, 1, 1
      t1 = _mm256_and_ps(t1, mask_F0F0F0F0);            // t1 = 7, 0, 5, 0, 3, 0, 1, 0
      Out = _mm256_add_ps(Out, t1);                     // Out=15, 7,11, 5, 7, 3, 3, 1
      __m256 t2 = _mm256_permute_ps(Out,(1<<6)+(1<<4)+(1<<2)+0);  // t2 =11,11,11, 5, 3, 3, 3, 1
      t2 = _mm256_and_ps(t2,mask_FF00FF00);             // t2 =11,11, 0, 0, 3, 3, 0, 0
      Out = _mm256_add_ps(Out, t2);                     // Out=26,18,11, 5,10, 6, 3, 1
      __m256 t3 = _mm256_permute2f128_ps(Out,Zero,3+(0<<4));// t3 =10, 6, 3, 1, 0, 0, 0, 0
      t3 = _mm256_permute_ps(t3,3+(3<<2)+(3<<4)+(3<<6));    // t1 =10,10,10,10, 0, 0, 0, 0
      Out = _mm256_add_ps(Out,t3);                      // Out=36,28,21,15,10, 6, 3, 1
      Bv[i] = Out = _mm256_add_ps(Out,Carry);           // Out=36,28,21,15,10, 6, 3, 1 + Carry
      Carry = _mm256_permute_ps(Out, 3+(3<<2)+(3<<4)+(3<<6));//Carry=36,36,36,36,10,10,10,10
      Carry = _mm256_permute2f128_ps(Carry,Zero,1+(1<<4));  //Carry=36,36,36,36,36,36,36,36
      Out = OutNext;
    } // for(int i = 0; i < Nv; ++i)
  } // if(Nv-- > 1)
  __m256 Out = Av[Nv]; //last vector (Nv was decremented)   // Out= 8, 7, 6, 5, 4, 3, 2, 1
  __m256 t1 = _mm256_permute_ps(Out,(2<<6)+(2<<4)+(0<<2)+0);// t1 = 7, 7, 5, 5, 3, 3, 1, 1
  t1 = _mm256_and_ps(t1, mask_F0F0F0F0);                    // t1 = 7, 0, 5, 0, 3, 0, 1, 0
  Out = _mm256_add_ps(Out, t1);                             // Out=15, 7,11, 5, 7, 3, 3, 1
  __m256 t2 = _mm256_permute_ps(Out,(1<<6)+(1<<4)+(1<<2)+0);// t2 =11,11,11, 5, 3, 3, 3, 1
  t2 = _mm256_and_ps(t2,mask_FF00FF00);                     // t2 =11,11, 0, 0, 3, 3, 0, 0
  Out = _mm256_add_ps(Out, t2);                             // Out=26,18,11, 5,10, 6, 3, 1
  __m256 t3 = _mm256_permute2f128_ps(Out,Zero,3+(0<<4));    // t3 =10, 6, 3, 1, 0, 0, 0, 0
  t3 = _mm256_permute_ps(t3,3+(3<<2)+(3<<4)+(3<<6));        // t1 =10,10,10,10, 0, 0, 0, 0
  Out = _mm256_add_ps(Out,t3);                              // Out=36,28,21,15,10, 6, 3, 1
  Bv[Nv] = Out = _mm256_add_ps(Out,Carry);                  // Out=36,28,21,15,10, 6, 3, 1 + Carry
  return ((float*)&Bv[Nv])[7];
}
```

The code example above uses the AVX register notation where the lowest element is expressed on the right side of the register. Please note that the above loop additionally performs a pre-fetch of the next input variable while performing the permutations on the data.

Additional, improvements can be made when the AVX-512 instruction set is available, since it provides r maskz instructions, however we don't yet have AVX-512.

The Intel® Xeon Phi™ Coprocessor (codename Knights Corner) strategy is similar to that used on the host's AVX , but with the vector widths being 512 bits holding 16 floats:

```
float Vector_inclusive_scan(__m512* Bv, __m512* Av, int Nv)
{
  __m512  Zero = _mm512_setzero_ps();
  __m512  Carry = _mm512_setzero_ps();
  __mmask16 mask_1010101010101010 = 0b1010101010101010;
  __mmask16 mask_1100110011001100 = 0b1100110011001100;
  __mmask16 mask_1111000011110000 = 0b1111000011110000;
  __mmask16 mask_1111111100000000 = 0b1111111100000000;

  if(Nv-- > 1) // see if more than one vector, post decriment Nv
  {
    __m512 Out = Av[0];// Out= 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
```

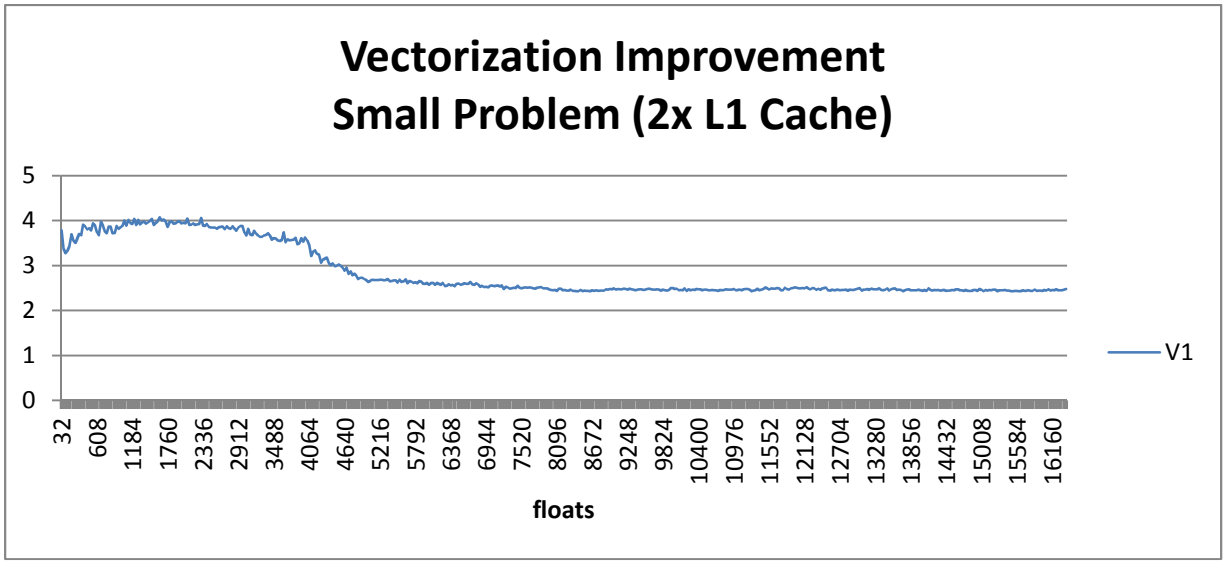
```
  for(int i = 0; i < Nv; ++i)
  {
    __m512 OutNext = Av[i+1]; // prefetch
    __m512 t1;
    // Out= 16, 15, 14, 13, 12, 11, 10,  9,  8,  7,  6,  5,  4,  3,  2,  1
    t1 = _mm512_swizzle_ps(Out,_MM_SWIZ_REG_CDAB);
    // t1 = 15, 16, 13, 14, 11, 12,  9, 10,  7,  8,  5,  6,  3,  4,  1,  2
    Out = _mm512_mask_add_ps(Out,mask_1010101010101010,Out,t1);
    // Out= 31, 15, 27, 13, 23, 11, 19,  9, 15,  7, 11,  5,  7,  3,  3,  1
    t1 = _mm512_swizzle_ps(Out,_MM_SWIZ_REG_BBBB);
    // t1 = 27, 27, 27, 27, 19, 19, 19, 19, 11, 11, 11, 11,  3,  3,  3,  3
    Out = _mm512_mask_add_ps(Out,mask_1100110011001100,Out,t1);
    // Out= 58, 42, 27, 13, 42, 30, 19,  9, 26, 18, 11,  5, 10,  6,  3,  1
    t1 = _mm512_swizzle_ps(Out,_MM_SWIZ_REG_DDDD);
    // t1 = 58, 58, 58, 58, 42, 42, 42, 42, 26, 26, 26, 26, 10, 10, 10, 10
    t1 = _mm512_mask_permute4f128_ps(Zero,mask_1111000011110000,t1,_MM_PERM_CCAA);
    // t1 = 42, 42, 42, 42,  0,  0,  0,  0, 10, 10, 10, 10,  0,  0,  0,  0
    Out = _mm512_add_ps(Out, t1);
    // Out=100, 84, 69, 55, 42, 30, 19,  9, 36, 28, 21, 15, 10,  6,  3,  1
    t1 = _mm512_swizzle_ps(Out,_MM_SWIZ_REG_DDDD);
    // t1 =100,100,100,100, 42, 42, 42, 42, 36, 36, 36, 36, 10, 10, 10, 10
    t1 = _mm512_mask_permute4f128_ps(Zero,mask_1111111100000000,t1,_MM_PERM_BBBB);
    // t1 = 36, 36, 36, 36, 36, 36, 36, 36,  0,  0,  0,  0,  0,  0,  0,  0
    Out = _mm512_add_ps(Out, t1);
    // Out=136,120,105, 91, 78, 66, 55, 45, 36, 28, 21, 15, 10,  6,  3,  1
    Bv[i] = Out = _mm512_add_ps(Out,Carry);
    // Out=Out + Carry
    Carry = _mm512_swizzle_ps(Out,_MM_SWIZ_REG_DDDD);
    // Carry=136,136,136,136, 91, 91, 91, 91, 45, 45, 45, 45, 10, 10, 10, 10
    Carry = _mm512_permute4f128_ps(Carry,_MM_PERM_DDDD);
    // Carry=136,136,136,136,136,136,136,136,136,136,136,136,136,136,136,136
    Out = OutNext;
  } // for(int i = 0; i < Nv; ++i)
} // if(Nv-- > 1)
__m512 Out = Av[Nv]; //last vector (Nv was decremented earlier)
// Out= 16, 15, 14, 13, 12, 11, 10,  9,  8,  7,  6,  5,  4,  3,  2,  1
__m512 t1 = _mm512_swizzle_ps(Out,_MM_SWIZ_REG_CDAB);
// t1 = 15, 16, 13, 14, 11, 12,  9, 10,  7,  8,  5,  6,  3,  4,  1,  2
Out = _mm512_mask_add_ps(Out,mask_1010101010101010,Out,t1);
// Out= 31, 15, 27, 13, 23, 11, 19,  9, 15,  7, 11,  5,  7,  3,  3,  1
t1 = _mm512_swizzle_ps(Out,_MM_SWIZ_REG_BBBB);
// t1 = 27, 27, 27, 27, 19, 19, 19, 19, 11, 11, 11, 11,  3,  3,  3,  3
Out = _mm512_mask_add_ps(Out,mask_1100110011001100,Out,t1);
// Out= 58, 42, 27, 13, 42, 30, 19,  9, 26, 18, 11,  5, 10,  6,  3,  1
t1 = _mm512_swizzle_ps(Out,_MM_SWIZ_REG_DDDD);
// t1 = 58, 58, 58, 58, 42, 42, 42, 42, 26, 26, 26, 26, 10, 10, 10, 10
t1 = _mm512_mask_permute4f128_ps(Zero,mask_1111000011110000,t1,_MM_PERM_CCAA);
// t1 = 42, 42, 42, 42,  0,  0,  0,  0, 10, 10, 10, 10,  0,  0,  0,  0
Out = _mm512_add_ps(Out, t1);
// Out=100, 84, 69, 55, 42, 30, 19,  9, 36, 28, 21, 15, 10,  6,  3,  1
t1 = _mm512_swizzle_ps(Out,_MM_SWIZ_REG_DDDD);
// t1 =100,100,100,100, 42, 42, 42, 42, 36, 36, 36, 36, 10, 10, 10, 10
t1 = _mm512_mask_permute4f128_ps(Zero,mask_1111111100000000,t1,_MM_PERM_BBBB);
// t1 = 36, 36, 36, 36, 36, 36, 36, 36,  0,  0,  0,  0,  0,  0,  0,  0
Out = _mm512_add_ps(Out, t1);
// Out=136,120,105, 91, 78, 66, 55, 45, 36, 28, 21, 15, 10,  6,  3,  1
Bv[Nv] = Out = _mm512_add_ps(Out,Carry);
// Out=Out + Carry
```

```
  return ((float*)&Bv[Nv])[15];
}
```

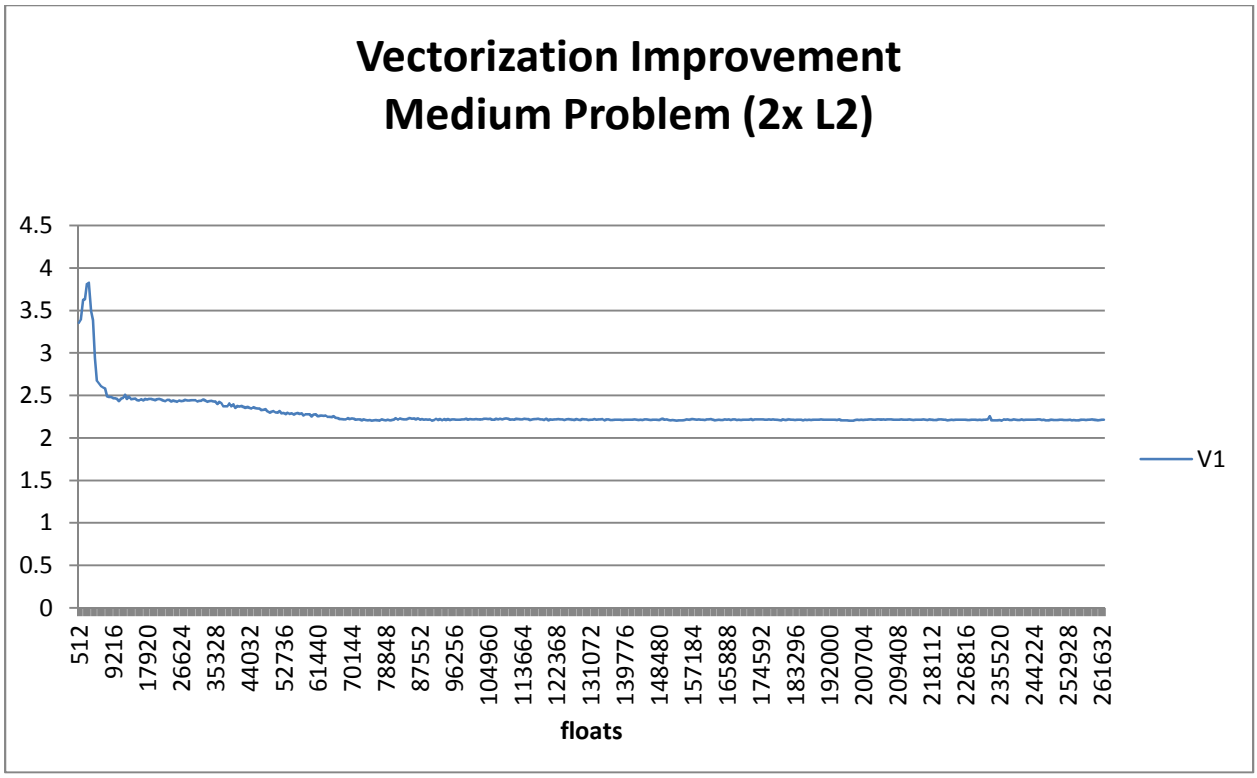
This code produces output of 16 floats in 11 instructions, plus 3 to handle the carry propagation and pre-fetched data (which we omitted from our sketch). The Knights Corner implementation can therefore produce twice as many floats in the same number of instructions as the AVX implementation. Notice that on Knights Corner _mm512 intrinsics we can take advantage of its mask add instruction which eliminates the instruction as used in the AVX version.

We will now examine the performance on the Intel Xeon Phi Coprocessor.

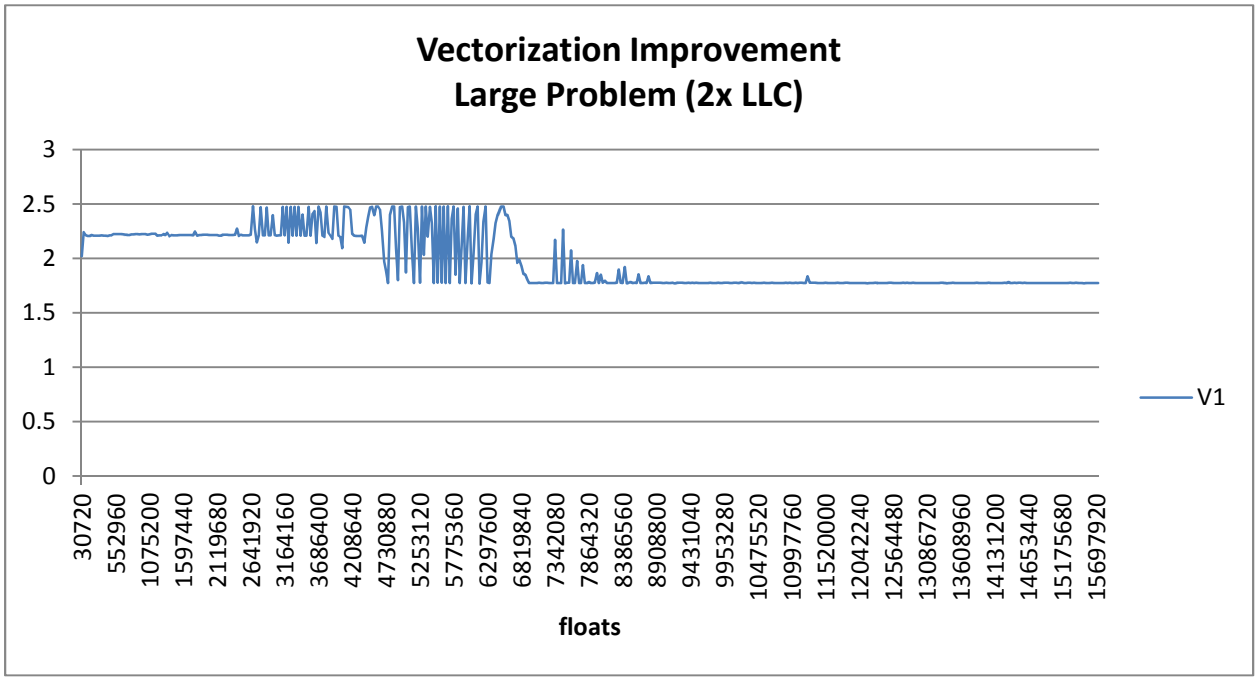
The Intel Xeon Phi coprocessor used to produce the test data is the model 5110P, with 60 cores and 4 hardware threads per core. We ran several different test scenarios with varying numbers of threads per core as well as memory capacity. For a base line metric we will initially look at the single thread performance difference between vectorized and non- vectorized code. These variations were run using capacities of 2x L1 cache, 2x L2 cache, and 2x LLC (on Intel Xeon Phi coprocessor this is the sum of the L2 caches). This provides for a small, medium and large problem analysis. The base line test is to determine the effect of vectorization on a single thread for various problem sizes. We made four test runs and the fastest time was used for both the serial and the vector version. The same input data is used and same output area. The chart shows performance relative to the simple method (standard C coding), so bigger is better All codes are compiled with -O3 optimization.



Vectorization produced between 2.5x and 4x over Simple inclusive scan for small problem size (under 2x L1 cache).



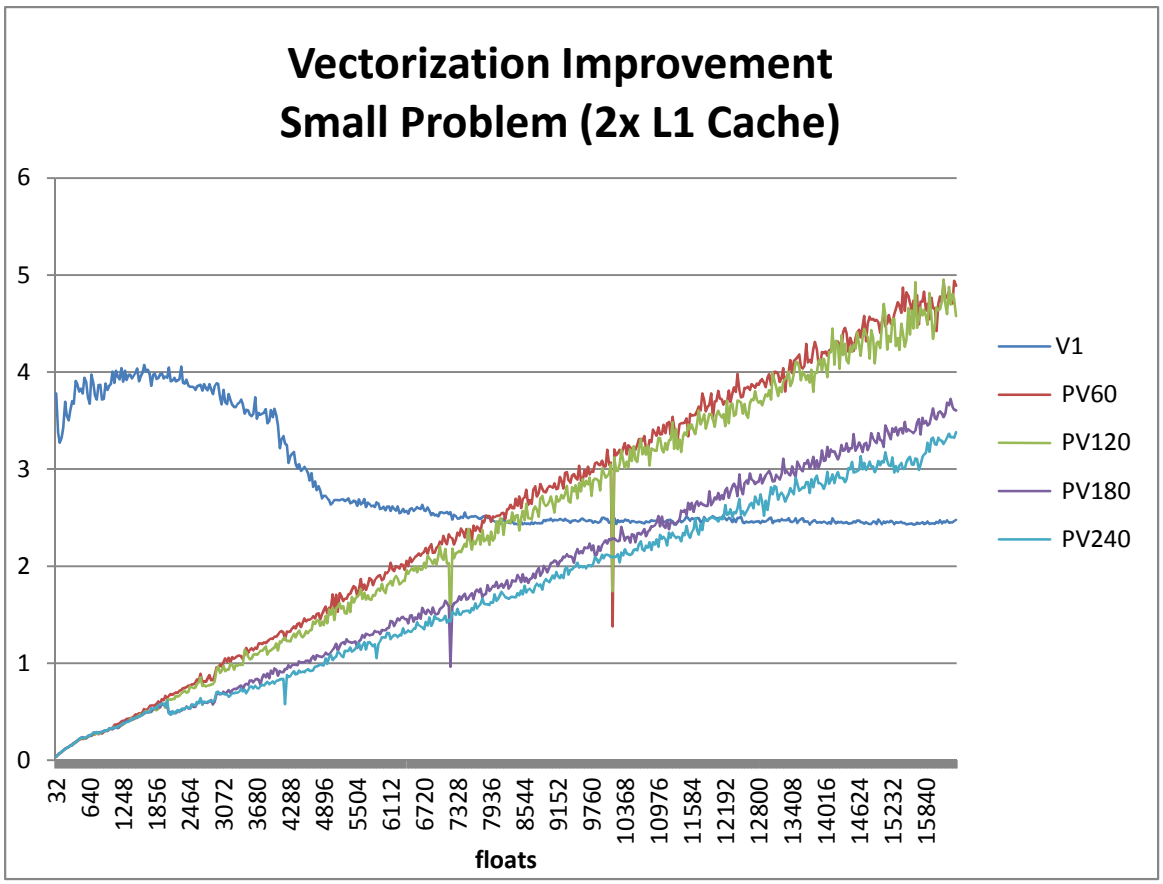
Essentially the same improvement for medium problem as for small problem. Now let's look at the large problem chart:



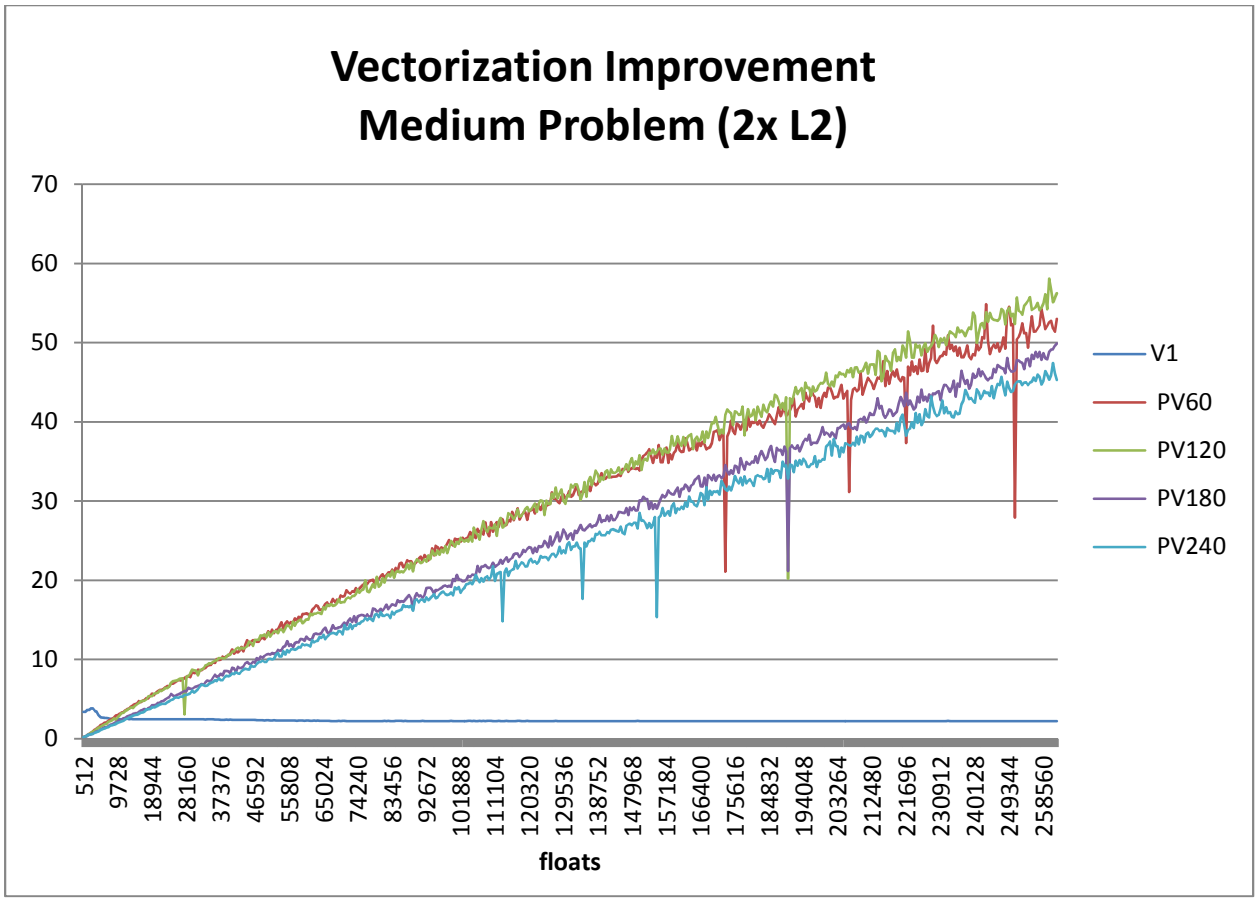
This illustrates that when the data does not fit in the LL cache (~30MB) the memory fetch latencies diminish the gain from 2.5x to about 1.75x that of the Simple serial loop.

Can we do anything for parallelization?

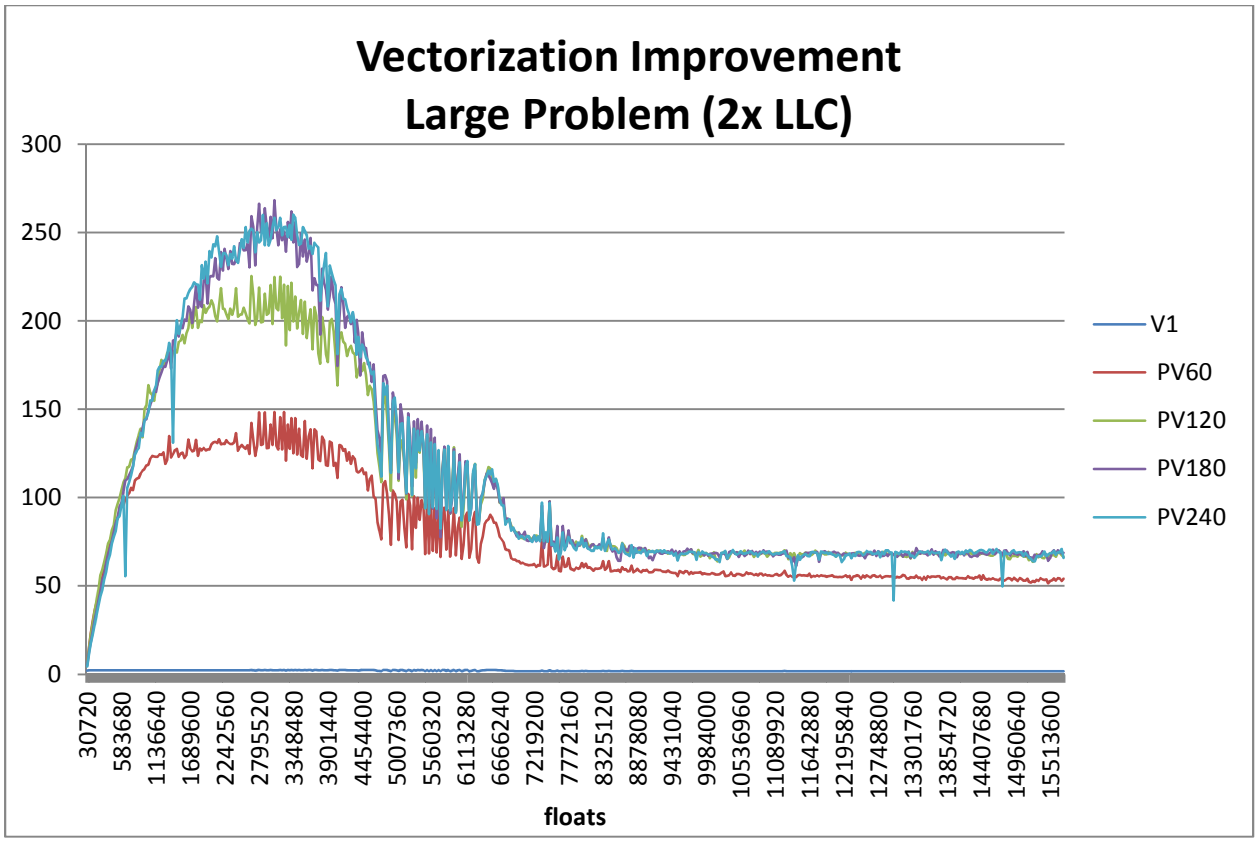
As it turns out, we can, that is if you are willing to violate one of the dogmas you may have learned in your CS class about not handling data more than once (or at least as little as possible). In looking at the vectorized code for the Knights Corner you note that the loop has about 14 instructions. The loop processes the entire data sequentially due to the carry propagation. If you were to chop the input and output data into multiple pieces, each could be calculated separately in parallel, however, this would require an additional pass to add the carry from the first piece to the second through n'th piece, and the carry from the second piece to the third through n'th piece, etc… Instead of each carry being propagated independently, each output can be carry processed once, if each piece sums up the carries from the pieces that come before it. This does add the overhead of processing the same output data twice. However, if this overhead (split by number of threads) is less than the gain of the parallel first pass, then you will experience a net gain in performance. Let's see what happens.



On the small problem size, the crossover is at about 8000 points. Surprisingly 1 thread per core appears to be doing better than 2 threads per core. The 3 and 4 thread runs have significantly lower slopes. Will this hold true as we increase the problem size? Let's look at the medium size problem (2x L2):

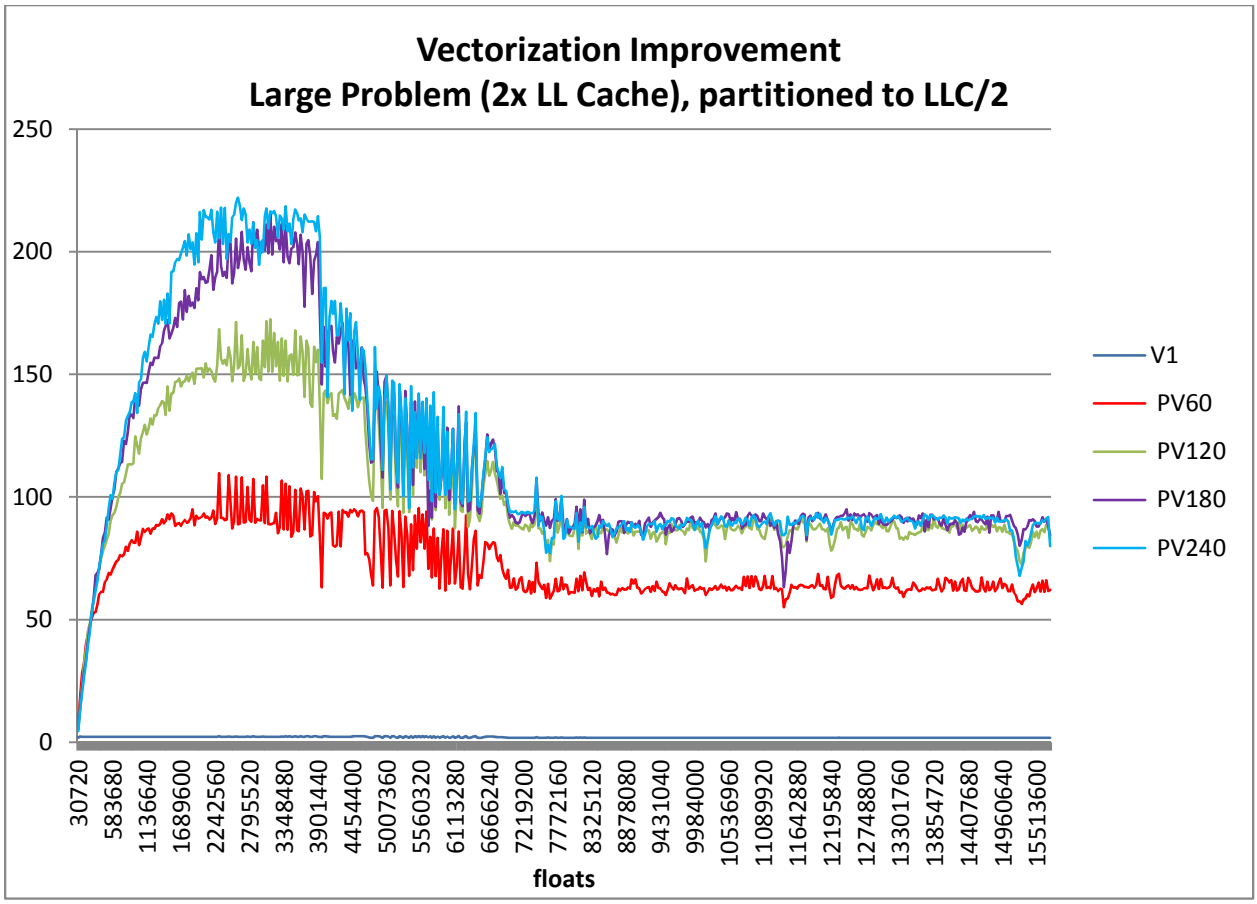


On the medium problem size, the two threads per core run has better performance at the larger problem size. Will this hold true as we increase the problem size? Looking at large problem (2x LLC):



For the larger problem sizes, which are typical for Intel Xeon Phi coprocessor, the use of 180 or 240 threads is optimal when the problem size fits within the Last Level Cache (sum of the L2's on Knights Corner). In the sweet spot, super –linear scaling is observed. In the more realistic problem size range (right side), this scales with the number of cores. As you exceed the sweet spot size, then the benefit diminishes for increased number of threads per core. This exhibits memory bandwidth limitations of the core as opposed to that of the thread to cache bandwidth.

The next step for optimization is to see if the plateau on the right side of the chart above can benefit from partitioning the problem into a size that fits within the LLC. We will investigate what happens when we split problem sizes larger than ½ the size of the LLC in two.



Although the peak performance is less in the sweet spot, let's consider the averages:

Overall averages between non-split and split

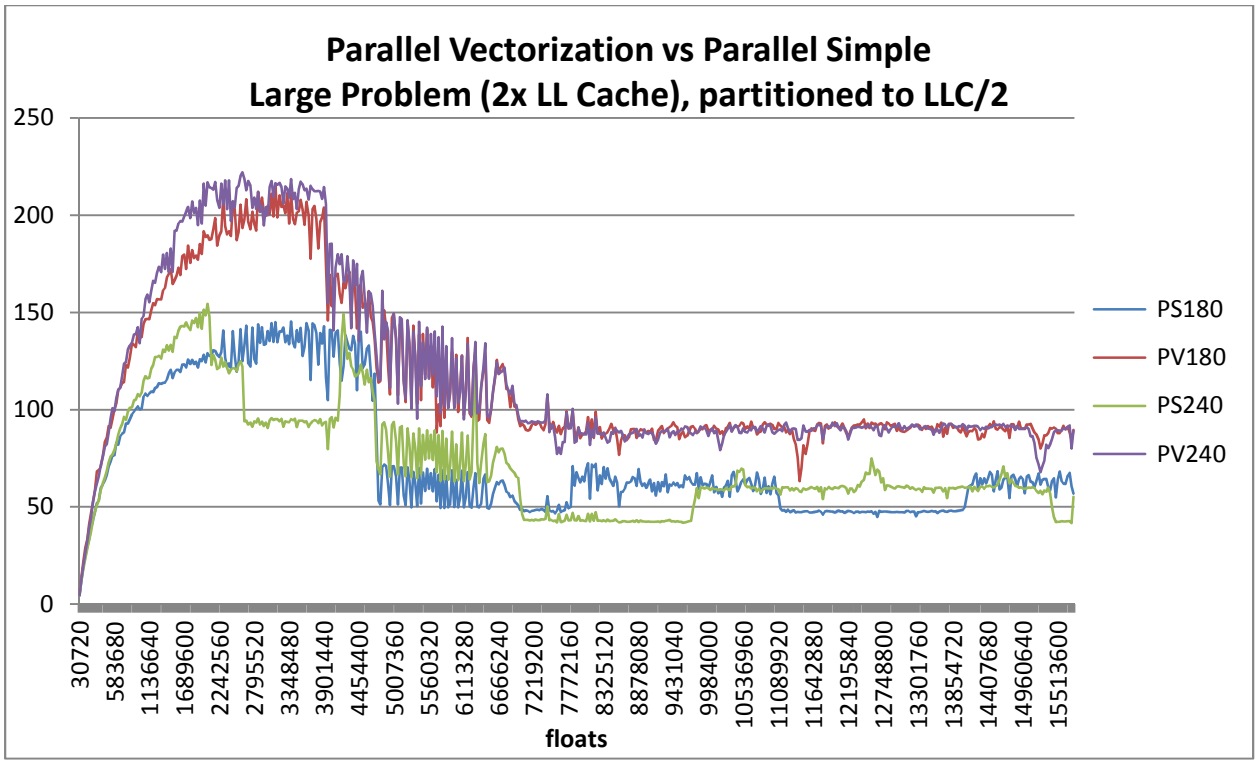
	V1	PV60	PV120	PV180	PV240
Non-Split:	1.980308457	77.88689207	104.2286789	111.289125	111.5467352
Split:	1.970007871	71.82705195	103.6955721	115.107594	118.130153

Averages of sizes larger than LLC:

	V1	PV60	PV120	PV180	PV240
Non-Split:	1.86847558	62.84718011	78.77605028	79.48406685	79.04674365
Split:	1.782080802	62.9868145	86.16692748	89.60063435	89.08126908

Considering that the problem sizes are typically larger than LLC we can conclude that vectorization and parallelization of this Elusive Algorithm yields an 89x improvement over the initial, serial, solution.

For those readers that are too timid to enter into the realm of using intrinsic functions, you can still rely on using the parallelization technique as used for the vectorization code (just call the non-vectorized code). The benefits of parallelization of both vectorized and non-vectorized code (partitioned):



The parallelized simple version does yield a reasonable rate of return whereas the vectorized version is significantly better. At the upper end of the problems size, where the advantage is diminished, the hand vectorized code is about 1.6x faster than the non-vectorized code.

Please be mindful that the performance data reported here is not a generality, merely an observation relating to this specific program. The point of the article is not to fully optimize this algorithm (which is something that the reader should be capable of doing). Rather, the point of the article is to incentivize the reader to re-examine their own “Elusive Algorithms” that they have encountered. Most of these have carry propagation characteristics. When the carry propagation is dependent on even a small compute section of code, it may be beneficial to separate the compute section from the carry propagation (handle the data twice).

To summarize: Elusive Algorithms (to parallelization), typically experience temporal dependencies such as the carry propagation of the inclusive scan. There are many such time-dependent algorithms that you will encounter, or rather have encountered, that seemingly defy vectorization and parallelization attempts. This article is intended to incentivize you, as well as provide guidance, as to how to address your own Elusive Algorithm.

Jim Dempsey