Benchmarking the Intel® Xeon Phi™ Coprocessor

Infrared Processing and Analysis Center, Caltech

F. Masci, 09/04/2013

1. Summary

This document summarizes our first experience with the Intel Xeon Phi. This is a coprocessor that uses Intel's Many Integrated Core (MIC) architecture to speed up highly parallel processes involving intensive numerical computations. The MIC coprocessor communicates with a regular Intel Xeon ("host") processor through its operating system. The Xeon Phi coprocessor is sometimes referred to as an "accelerator". In a nutshell, the Xeon Phi consists of 60 1.052 GHz cores each capable of executing four concurrent threads and delivering one teraflop of performance. For comparison, the host processor consists of 16 2.6 GHz cores, with two admissible threads per core. More details on the MIC hardware can be obtained from the references in Section 8.

Rather than present yet another guide on how to efficiently program for the Xeon Phi, our goal is to explore whether (and when) there are advantages in using the Xeon Phi for the processing of astronomical data, e.g., as in a production pipeline. Such pipelines are common-use at the Infrared Processing and Analysis Center (IPAC) and range from the instrumental calibration of raw image data, astrometry, image co-addition, source extraction, photometry, and source association. We also outline some lessons learned to assist future developers. **Note:** the findings and opinions reported here are exclusively the author's and do not reflect those of Intel or of any individual.

IPAC has recently acquired a single Xeon Phi card for preliminary benchmarking. We find in general that all existing heritage software based on C/C++/Fortran code can be made to run natively on the Xeon Phi with no recoding. However, whether it will run optimally to fully exploit the MIC architecture is a different question entirely. The answer is usually *no*. Even software that has been extensively multithreaded to utilize a multicore processor isn't guaranteed to run faster on the Xeon Phi than on a regular Intel Xeon machine. In fact, depending on memory and/or disk I/O usage, it can be much slower.

The key is to make efficient use of the Xeon Phi MIC architecture. This is not designed to handle jobs that are memory (primarily RAM) intensive. It is designed to utilize wide vector instruction units for floating point arithmetic (see below for details). Therefore, the types of problems the Xeon Phi is well suited for are intensive numerical computations with a low memory bandwidth. Additionally, the computations need to use one of the highly optimized vector math libraries that were implemented using assembly language constructs tuned specifically for the Xeon Phi architecture. Knowing this programming model beforehand can assist a developer to design software such that segments with intense numerical calculations can be offloaded to the Xeon Phi to be accelerated. The host processor then does most of the data I/O and memory management.

This document is organized as follows:

2. Benchmarks conducted

To demonstrate that our Xeon Phi card performs according to the vendor's specifications and to explore its performance relative to the Xeon host, we ran three separate experiments:

1. A preliminary "naïve" test that multiplies two large matrices using explicit looping of matrix elements where loops were multithreaded using OpenMP pragmas. See Section 4. This makes heavy use of single-precision floating point arithmetic. A similar version with computations in double-precision yielded similar results. This code was compiled to run in "native-MIC" mode (see Section 3 for definition), "MIC-offload" mode, and "Host-exclusive" mode.

2. A test that also multiplies two large matrices but now using the highly (Phi-)optimized Math Kernel Library (MKL) to perform the matrix calculations. This also uses OpenMP to assist with multithreading. See Section 5. This code was compiled to run in "native-MIC mode" (see Section 3 for definition), "MIC-offload" mode, and "Host-exclusive" mode.

3. A test that uses the multithreaded image co-addition module from the "Image Co-addition with Optional Resolution Enhancement" (ICORE) tool. See Section 6. Documentation and download information is available at: *http://web.ipac.caltech.edu/staff/fmasci/home/icore.html* This code was compiled to run *only* in "native-MIC" and Host-exclusive modes (see Section 3).

3. Programming philosophy, execution modes, and some lessons learned

The literature and various online documents broadly outline some "best practices" when programming for the Xeon Phi. But it's not all complete, nor collected in one place, or detailed enough to use in a practical sense, especially for a novice. Here's a summary of the programming models as well as some not-so-obvious ones that were (re)discovered from experimentation.

- The GNU compilers (e.g., *gcc*) are not suitable for compiling code for the Xeon Phi (for both native execution and offloading from the host). The Intel Compilers (e.g., *icc*) have the requisite optimization flags for code to execute efficiently on the Xeon Phi. This was discovered the hard way.

- The Xeon Phi is intended for highly parallelized numerical computations whose runtime scales linearly up to some maximum number threads that can be run concurrently on any processor, whether it's a regular Intel Xeon (host) or a Phi. For the Xeon Phi, the goal is to observe perfect linear scaling up to 120 threads (2 threads per core) before using additional Phi cards in a cluster setup. It is recommended that this scalability be demonstrated first on regular host processors before utilizing the Xeon Phi.

- Code that does heavy numerical work should also have a low memory bandwidth. Local CPU caches should be used as much as possible if there is frequent access to data, not the main RAM. This is referred to as maintaining "locality of reference". It is advised to keep all memory access within the L2 cache if possible. L2 is about 25 MB over all 60 cores on the Xeon Phi. I.e., there's little wiggle room – one of the challenges of optimizing code for the Xeon Phi.

- Make efficient use of the 64 byte-wide vector units (as opposed to 32 bytes on regular Xeon CPUs). I.e., 16 floats (or 8 doubles) can fit in the registers and be operated on simultaneously. The Single Instruction Multiple Data (SIMD) instruction set used for vector units on the Xeon Phi is *not* SSE compliant (Streaming SIMD Extension). This is applicable to Intel Xeon E5 processors only. Some time was spent using SSE-optimized compilation flags for the Xeon Phi that actually lead to a degradation in performance. The key is to use no SSE-specific flags when compiling for the Xeon Phi.

- If you are compiling to run natively on the Xeon Phi, you can get better performance if memory is allocated such that it's aligned on 64 byte cache-line boundaries. This can be achieved using the `posix_memalign()` or `malloc_aligned()` functions. For even better performance, you can make the array size a multiple of 16 for single-precision floating point, or 8 for double-precision, otherwise the unused registers will be internally masked and some overhead is incurred.

- Allow loops in the code to be auto-vectorized (unrolled) by the compiler. This optimizes the processing for the SIMD vector architecture on each *individual* core. At times, the *icc* compiler can be stubborn and refuse to auto-vectorize a loop because it thinks there's some variable dependence. You can use "#pragma simd" to force auto-vectorization by the compiler provided you're certain it's safe to unroll the loop in question.

- The compiler-assisted auto-vectorization step mentioned above is usually not enough to fully optimize code for the Xeon Phi. It's a necessary step for the SIMD 64-byte vector processing on each core, but more work on explicit multithreading to utilize all cores is needed if you have loops operating on large arrays of data with no dependencies between the variables. The latter can be achieved using either the *openMP* library via "omp pragmas" or via the *pthreads* library directly.

- Use the highly optimized vector/matrix functions in Intel's Math Kernel Library (MKL) or lower level Vector Math library (VML). These libraries use optimized assembly language that takes advantage of the SIMD vector instruction set for the Xeon Phi. So far, I've only been able to run things faster on the Xeon Phi (than on the host) when using MKL routines. In the end, all arithmetic operations on data arrays can be performed using vector/matrix operations so it is advised to use these whenever possible. One finding with MKL (see benchmarking below) is that computations run more efficiently for relatively large input vectors and matrices. This establishes the power of the Xeon Phi. More threads are automatically spawned for big jobs. It's analogous to a jet engine receiving more oxygen the faster it goes. At high speed, it performs more efficiently in terms of fuel consumption.

We explored two compilation/execution modes while testing on the Xeon Phi:

1. "Native-MIC" mode where a fully-native Xeon Phi binary is compiled for execution on the MIC O/S directly. This is a good choice when the code has low data I/O and/or memory needs and consists primarily of intensive numerical computations.

2. "MIC-offload" mode where a heterogeneous binary is compiled to run on the host CPU with selected code segments offloaded to the Xeon Phi during runtime. The code segments intended for execution on the Xeon Phi are designated using offload pragmas targeted for the MIC. The advantage here is that the host processor can manage any high data/memory I/O needs while segments involving heavy numerical work can be "accelerated" on the Xeon Phi. Results are then copied back to the host file system with little overhead.

4. Preliminary “naïve” testing

We first conducted a simple test that multiplies two matrices with 3000 × 3000 elements each. We call this test “naïve” since it’s programmed using nested loops over each matrix element and carrying out the arithmetic as we go along, i.e., as the textbooks say when multiplying matrices. The multiplication operation was repeated 10 times and the performance metrics were averaged at the end. We have also parallelized the loop computations using OpenMP pragmas (see code in Section 9.a.). We were suspicious from the outset that this approach was non-optimal for the Xeon Phi, but we wanted to see where this would take us. Such code is typical of heritage software lying around for performing matrix computations (without the multithreading of course).

Test code “test1.c” (Section 9.a.) was compiled to run in “native-MIC” mode, “MIC-offload” mode, and “Host-exclusive” mode (see Section 3 for definitions). Figure 1 shows the number of Floating point Operations per second, or FLOPs/sec versus the number of threads spawned. All computations used single-precision floating point. Only the native-MIC (red) and Host-exclusive (blue) runs are shown. We found that the MIC-offload mode performed ~10% worse (lower flops) than the native-MIC run. This is probably due to the additional overhead in transferring (offloading) the large data arrays from the host’s main memory to the Xeon Phi.

Figure 1 shows that the maximum achievable performance on the Xeon Phi (~120 GFlops/sec) is comparable to that on the host (within the measured variance). This is contrary to expectations reported by the vendor and others in the literature.

The theoretical performance one expects from the Xeon Phi is:

$$\begin{aligned}\#\text{GFlops/sec} &= 16[\#\text{single-precision floats in SIMD vector unit}] \times 2[\text{FMA}] \times 1.052[\text{GHz}] \times 60[\text{cores}] \\ &= 2019.84 \text{ GFlops/sec} \\ &\sim 2 \text{ TFlops/sec.}\end{aligned}$$

While the theoretical performance one expects from the Xeon host processor is:

$$\begin{aligned}\#\text{GFlops/sec} &= 8[\#\text{single-precision floats in SIMD vector unit}] \times 2[\text{FMA}] \times 2.599[\text{GHz}] \times 16[\text{cores}] \\ &= 665.34 \text{ GFlops/sec} \\ &\sim 0.66 \text{ TFlops/sec.}\end{aligned}$$

FMA represents the number of arithmetic operations in a “Fused-Multiply Add” instruction. That is, these Intel processors can execute a “multiply and add” (really two separate operations) as a single instruction, at the same clock rate.

Therefore, we expect the Xeon Phi to outperform the Xeon host by a factor of ~ 3. This is nowhere seen in our first “naïve” test and we were rather dumbstruck. Many other “simple” codes were also tested (with explicit threading over all loops), but all performed equally as well and sometimes worse on the Xeon Phi than on the host. The solution was revealed by consulting previous benchmarks in the literature and investigating how floating-pointing arithmetic operations can be performed more optimally on the Xeon Phi. Obviously, auto-vectorization and explicit threading of loops is not enough.

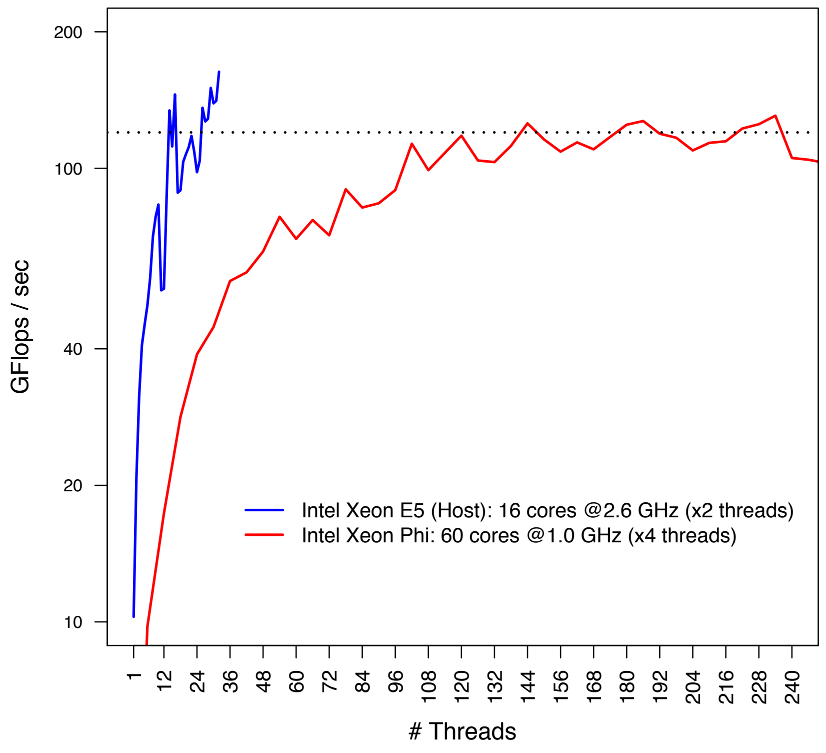

Figure 1: Single-precision floating-point performance for the simple "naïve" matrix multiplication test code (test1.c) involving two 3000 × 3000 matrices. Blue curve is for the "Host-exclusive" runs and red curve is for the Xeon Phi runs using "native-MIC" mode. Dotted line is at 120 GFlops/sec.

5. Reaching One Teraflop performance

We decided to replace the "naïve" multiplication step in test1.c (Section 4) with the `sgemm()` matrix multiplication function from Intel's Math Kernel Library (MKL). As outlined in Section 3, this library uses optimized assembly language constructs to take advantage of the SIMD vector instruction set for Xeon Phi coprocessors. Results for test code "test2.c" (Section 9.b.) are shown in Figure 2. The long sought-after one TFlops/sec performance metric is now achieved, however, this is still about a factor of two below the theoretical prediction of ~ 2 TFlops/sec (see Section 4). Benchmarks advertised by the vendor and others peak at ~ 1.6 - 1.8 TFlops/sec, but no guidance is given on the set-up or circumstances under which these are achieved. Furthermore, relative to the host processor, we find the Xeon Phi performs ~ 2.5 better! This isn't too far off from the theoretical prediction of ~ 3 reported in Section 4. It appears the use of MKL routines for floating-point arithmetic is pivotal in exploiting the Xeon Phi.

As a further test, we fixed the number of threads to the maximum possible on the Xeon Phi (240) and the host processor (32) and explored the performance as a function of matrix size using test code test2.c. Figure 3 shows the result. It appears that we reach maximum performance when the matrix size exceeds 2500 × 2500 elements. That is, both the Xeon host and Phi perform best (with better throughput and efficiency) when the number of computations is large. This shows the Xeon Phi is best suited for large/heavy numerical problems and not worth the effort for small ones where data/memory I/O is likely to dominate (relatively speaking).

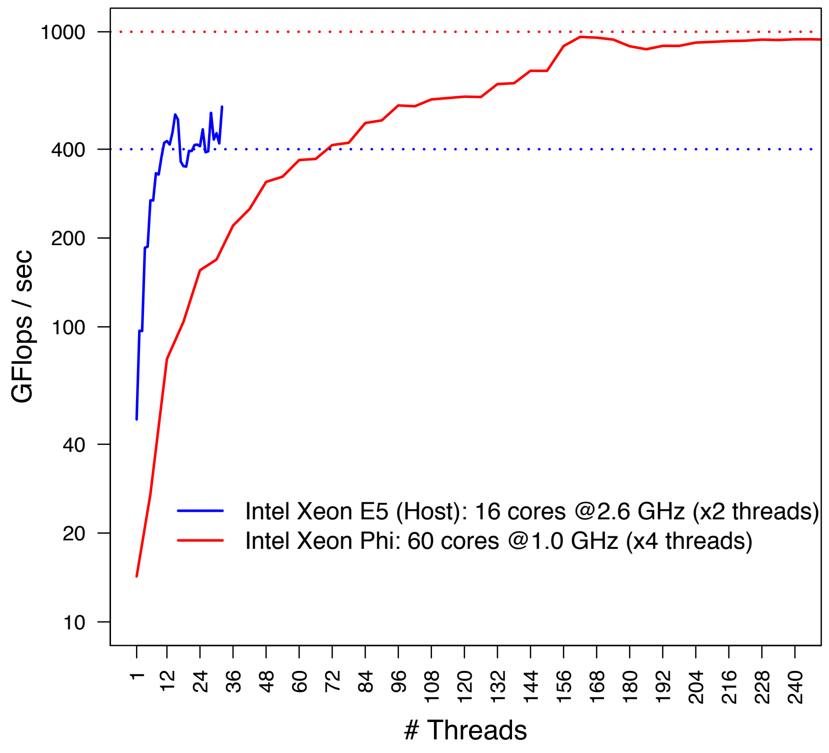

Figure 2: Single-precision floating-point performance for our optimized matrix multiplication test code (test2.c) involving two 3000 × 3000 matrices. Blue curve is for the "Host-exclusive" runs and red curve is for the Xeon Phi runs using "native-MIC" mode. Horizontal dotted lines (at 400 and 1000 GFlops/sec) are to guide the eye.

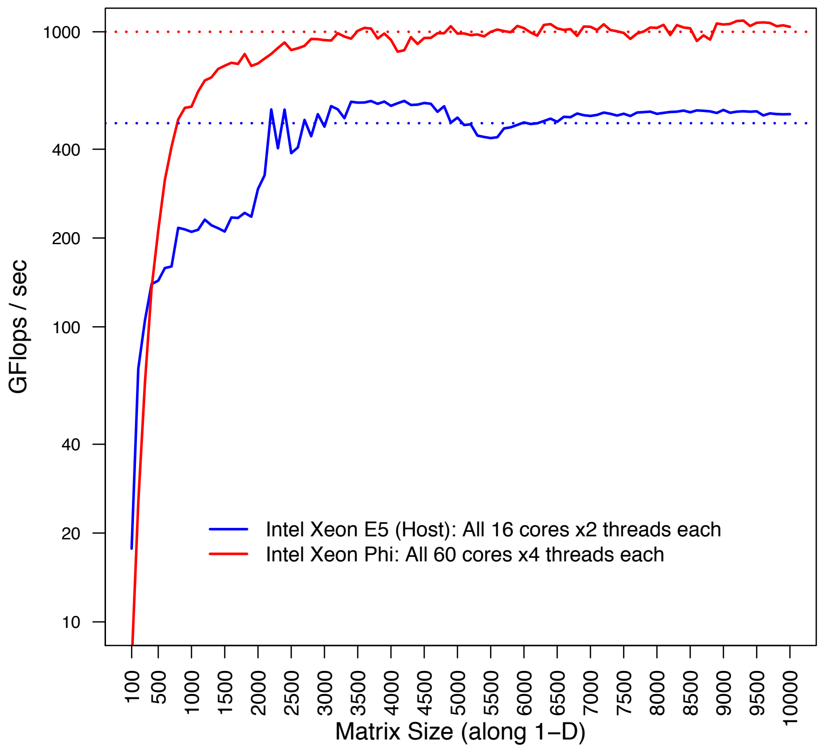

Figure 3: Single-precision floating-point performance for our optimized matrix multiplication test code (test2.c) as a function of matrix size. Blue curve is for the "Host-exclusive" runs and red curve is for the Xeon Phi runs using "native-MIC" mode. The number of concurrent threads used was fixed at the maximum values of 32 and 240 for the host and Xeon Phi respectively.

6. Testing the ICORE image co-addition module

We also conducted preliminary experiments with the ICORE tool – "Image Co-addition with Optional Resolution Enhancement" (*http://web.ipac.caltech.edu/staff/fmasci/home/icore.html*). This tool consists of a set of modules to perform image overlap-offset corrections, photometric-gain corrections, interpolation, outlier masking, co-addition, and optional resolution enhancement. The input images are supplied as files in FITS format. All the modules are multithreaded to take advantage of parallelization on multicore processors. We recompiled the co-addition module *awaico* and all dependent libraries (*libcfitsio*, *libwcs*, *libtwoplane*) from ICORE to run natively on the Xeon Phi ("native-MIC" mode), as well as on the host processor ("Host-exclusive" mode). Due to the recoding effort needed to generate an executable in "MIC-offload" mode, we did not explore this mode at this time (however, see discussion below).

Our test consisted of co-adding 29 overlapping intensity images (with accompanying mask and uncertainty images) from the WISE mission onto a 0.4° × 0.4° footprint. This footprint was centered on the IC342 galaxy. Each input image consists of ~ 1016 × 1016 pixels. Runtimes for the host processor and Xeon Phi (in *native*-MIC mode) are shown in Figure 4. Clearly, the host processor outperforms the native-MIC execution by about a factor of 4. This is somewhat expected since native-MIC mode execution is not optimal for this type of application. The co-addition module contains a number of disk and memory I/O steps interspersed with numerical computations (e.g., sky-to-pixel mappings, resampling, and interpolations). As mentioned above, the Xeon Phi alone cannot handle heavy data I/O.

6.a. Future Upgrades

A worthwhile exercise would be to recode the co-addition module (and other modules) such that only the compute-heavy steps are offloaded to the Xeon Phi (MIC-offload mode). Even more, we can make extensive use of the trig and vector processing functions from Intel's Math Kernel Library (MKL) – all optimized for the Xeon Phi. In the end, we expect the compute-heavy steps alone to be significantly accelerated, by perhaps a factor of 2.5 (analogous to Figures 2 and 3). Note however that the data I/O overhead (on the host) will always remain. We expect overall runtimes (blue curve in Figure 4) to drop by >~ 30%. This is a work in progress.

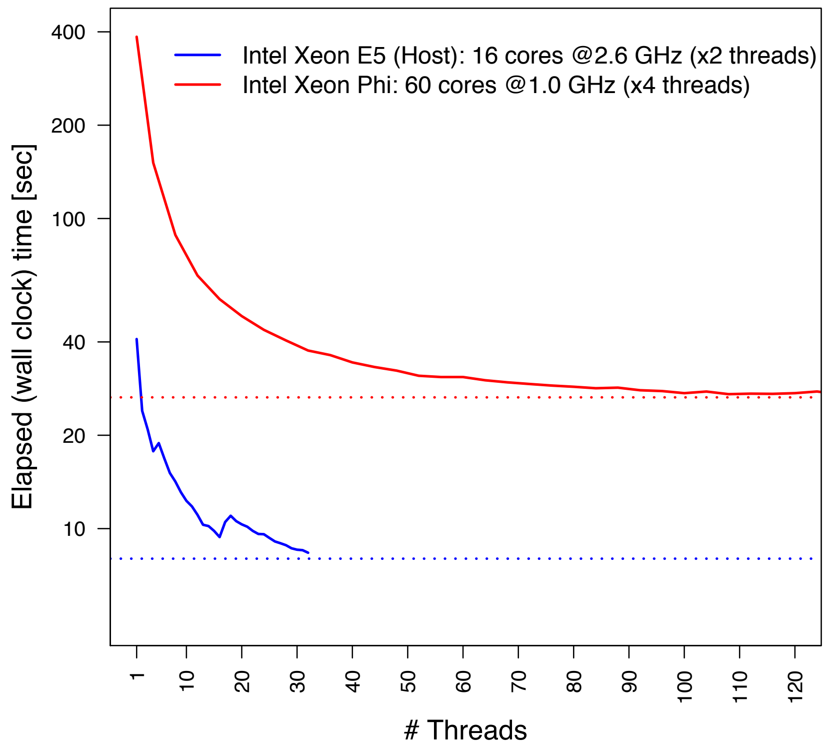

Figure 4: Runtime as a function of number of threads for the ICORE image co-addition module executed on the Host processor (blue curve) and the Xeon Phi (red curve; in native-MIC mode).

In the course of testing, we made extensive use of the Coprocessor Offload Infrastructure (COI) tools. These enable one to submit native-MIC compiled binaries to the Xeon Phi (via *micnativeloadex*) as well as monitor them in real time (via *micsmc*). Figure 5 shows an example snapshot of the monitoring tool.

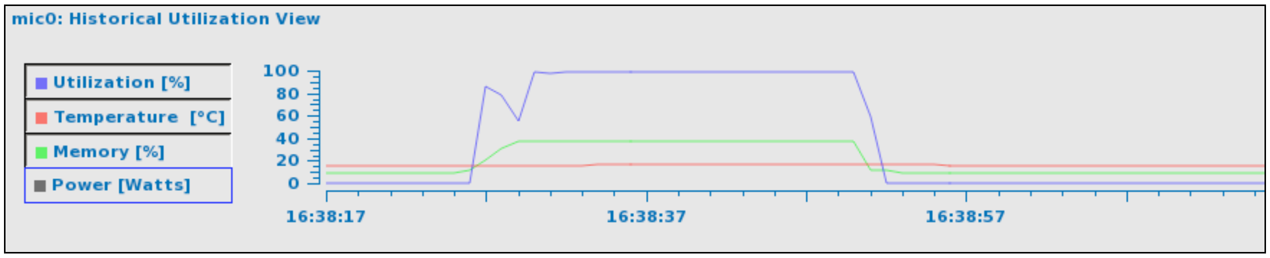

Figure 5: Example of the Xeon Phi monitoring tool (*micsmc*) with all 60 cores being utilized.

7. Conclusions and words of wisdom

While most heritage software will run on the Xeon Phi with little effort, there's no guarantee it will run optimally to fully exploit its architecture, even if the segments that carry out intensive numerical computations have been highly parallelized. Some recoding using offload pragmas targeted for the MIC and use of vector/matrix libraries optimized for numerical work is inevitable. This could be expensive for existing codes that are not structured in a manner to trivially separate the heavy data/memory I/O steps from the pure computational ones.

Approximately one teraflop can be obtained on the Xeon Phi when using optimized vector-math libraries. Even then, we find this is only a factor of 2 to 2.5 times greater than that achieved on regular Intel Xeon E5 processors with 16 2.6 GHz cores. Is a factor of 2 to 2.5 improvement really worth the human labor to recode existing "numerical-heavy" software for optimal execution on Xeon Phi cards? This depends on the developer's experience with the software and its complexity, but more importantly, on the type of software being run. As it stands, much of the software executed in IPAC's mission-production pipelines (*Spitzer*, WISE, and PTF for example) are primarily data I/O limited and at the mercy of fileservers cooperating and responding in sync with the CPUs.

However, this doesn't mean there's no place for Xeon Phi cards at IPAC. Future developers should use the Xeon Phi when it makes sense to do so. Existing heritage code can be judiciously recycled with an eye for "accelerating" heavy computational steps with the Xeon Phi. There's always a gain, but as mentioned above, the factor of 2 to 2.5 improvement over regular processors won't buy you much if most of the time is spent moving data off/onto disks and/or in/out of RAM. The compute-heavy steps (until now) are usually a small fraction (<~ 30%) of pipeline processing budgets for astronomical applications that go from raw-image data to source-catalogs. Regular Intel Xeon processors are getting faster and more efficient at managing memory. Nonetheless, there will be scientists who would benefit from the Xeon Phi by running customized code involving heavy numerical work, e.g., N-body simulations, gravitational lensing shear calculations, or radiative transfer models.

8. References and further reading

The following links and documents contain good examples and guidelines for a novice when programming for the Intel Xeon Phi.

- http://software.intel.com/sites/default/files/article/335818/intel-xeon-phi-coprocessor-quick-start-developers-guide.pdf
- http://software.intel.com/en-us/articles/building-a-native-application-for-intel-xeon-phi-coprocessors
- http://www.drdobbs.com/parallel/programming-intels-xeon-phi-a-jumpstart/240144160
- http://www.prace-project.eu/IMG/pdf/Best-Practice-Guide-Intel-Xeon-Phi.pdf
- http://software.intel.com/sites/default/files/article/366893/offload-runtime-for-the-intelr-xeon-phitm-coprocessor.pdf
- http://research.colfaxinternational.com/file.axd?file=2013%2F5%2FColfax_Static_Libraries_Xeon_Phi.pdf
- https://hpcforge.org/plugins/mediawiki/wiki/pracewp8/images/6/68/XeonPhi.pdf
- http://software.intel.com/en-us/articles/getting-started-with-openmp
- http://d3f8ykwhia686p.cloudfront.net/1live/intel/An_Introduction_to_Vectorization_with_Intel_Compiler_021712.pdf

9. Appendices

Below are the C codes used in the "simple" benchmarking tests presented in Sections 4 and 5. These were written by combining various code snippets (with much experimentation) from the links in Section 8. Further below are the compilation, environment variables and execution command lines used.

9.a. test1.c: code used to generate Figure 1 (Section 4)

```
#ifndef MIC_DEV
#define MIC_DEV 0
#endif

#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <mkl.h> /* needed for the dsecnd() timing function. */
#include <math.h>

/* Program test1.c: multiply two matrices using explicit looping of elements. */

/*----------------------------------------------------------------------*/
/* Simple "naive" method to multiply two square matrices A and B
   to generate matrix C. */

void myMult(int size,
            float (* restrict A)[size],
            float (* restrict B)[size],
            float (* restrict C)[size])
{
  #pragma offload target(mic:MIC_DEV) in(A:length(size*size)) \
                                      in( B:length(size*size)) \
                                      out(C:length(size*size))
  {
    /* Initialize the C matrix with zeroes. */

    #pragma omp parallel for default(none) shared(C,size)
    for(int i = 0; i < size; ++i)
      for(int j = 0; j < size; ++j)
        C[i][j] = 0.f;

    /* Compute matrix multiplication. */

    #pragma omp parallel for default(none) shared(A,B,C,size)
    for(int i = 0; i < size; ++i)
      for(int k = 0; k < size; ++k)
        for(int j = 0; j < size; ++j)
          C[i][j] += A[i][k] * B[k][j];
  }
}

/*----------------------------------------------------------------------*/
/* Read input parameters; set-up dummy input data; multiply matrices using
   the myMult() function above; average repeated runs. */

int main(int argc, char *argv[])
{
  if(argc != 4) {
    fprintf(stderr,"Use: %s size nThreads nIter\n",argv[0]);
    return -1;
  }

  int i,j,nt;
  int size=atoi(argv[1]);
  int nThreads=atoi(argv[2]);
  int nIter=atoi(argv[3]);

  omp_set_num_threads(nThreads);

  /* when compiled in "mic-offload" mode, this memory gets allocated on host,
     when compiled in "mic-native" mode, it gets allocated on mic. */

  float (*restrict A)[size] = malloc(sizeof(float)*size*size);
  float (*restrict B)[size] = malloc(sizeof(float)*size*size);
  float (*restrict C)[size] = malloc(sizeof(float)*size*size);

  /* this first pragma is just to get the actual #threads used
```

```
     (sanity check). */

  #pragma omp parallel
  {
    nt = omp_get_num_threads();

    /* Fill the A and B arrays with dummy test data. */
    #pragma omp parallel for default(none) shared(A,B,size) private(i,j)
    for(i = 0; i < size; ++i) {
      for(j = 0; j < size; ++j) {
        A[i][j] = (float)i + j;
        B[i][j] = (float)i - j;
      }
    }
  }

  /* warm up run to overcome setup overhead in benchmark runs below. */

  myMult(size, A,B,C);

  double aveTime,minTime=1e6,maxTime=0.;

  /* run the matrix multiplication function nIter times and compute
     average runtime. */

  for(i=0; i < nIter; i++) {
    double startTime = dsecnd();
    myMult(size, A,B,C);
    double endTime = dsecnd();
    double runtime = endTime-startTime;
    maxTime=(maxTime > runtime)?maxTime:runtime;
    minTime=(minTime < runtime)?minTime:runtime;
    aveTime += runtime;
  }
  aveTime /= nIter;

  printf("%s nThreads %d matrix %d maxRT %g minRT %g aveRT %g GFlop/s %g\n",
         argv[0],nt,size,maxTime,minTime,aveTime, 2e-9*size*size*size/aveTime);

  free(A);
  free(B);
  free(C);

  return 0;
}
```

9.b. test2.c: Code used to generate Figures 2 and 3 (Section 5)

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <mkl.h>
#include <math.h>

/* Program test2.c: multiply two matrices using a highly thread-optimized
   routine from the Intel Math Kernel Library (MKL). */

/*----------------------------------------------------------------------*/
/* Multiply two square matrices A and B to generate matrix C using the
   optimized sgemm() routine (for single precision floating point) from MKL. */

float fastMult(int size,
               float (* restrict A)[size],
               float (* restrict B)[size],
               float (* restrict C)[size],
               int nIter)
{
  float (*restrict At)[size] = malloc(sizeof(float)*size*size);
  float (*restrict Bt)[size] = malloc(sizeof(float)*size*size);
  float (*restrict Ct)[size] = malloc(sizeof(float)*size*size);

  /* transpose input matrices to get better sgemm() performance. */

  #pragma omp parallel for
  for(int i=0; i < size; i++)
    for(int j=0; j < size; j++) {
      At[i][j] = A[j][i];
      Bt[i][j] = B[j][i];
    }

  /* scaling factors needed for sgemm(). */

  float alpha = 1.0f;
  float beta = 0.0f;

  /* warm up run to overcome setup overhead in benchmark runs below. */

  sgemm("N", "N", &size, &size, &size, &alpha,
        (float *)At, &size, (float *)Bt, &size, &beta, (float *) Ct, &size);

  double StartTime=dsecnd();

  for(int i=0; i < nIter; i++)
    sgemm("N", "N", &size, &size, &size, &alpha,
          (float *)At, &size, (float *)Bt, &size, &beta, (float *) Ct, &size);

  double EndTime=dsecnd();

  float tottime = EndTime - StartTime;
  float avgtime = tottime / nIter;
  printf("tot runtime = %f sec\n", tottime);
  printf("avg runtime per vec. mult. = %f sec\n", avgtime);
  float GFlops = (2e-9*size*size*size)/avgtime;

  free(At);
  free(Bt);
  free(Ct);

  return ( GFlops );
}

/*---------------------------------------------------------------------*/
/* Read input parameters; set-up dummy input data; multiply matrices using
   the fastMult() function above; average repeated runs therein. */

int main(int argc, char *argv[])
{
  if(argc != 4) {
    fprintf(stderr,"Use: %s size nThreads nIter\n",argv[0]);
    return -1;
  }

  int i,j,nt;
  int size=atoi(argv[1]);
  int nThreads=atoi(argv[2]);
  int nIter=atoi(argv[3]);

  omp_set_num_threads(nThreads);
  mkl_set_num_threads(nThreads);

  /* when compiled in "mic-offload" mode, this memory gets allocated on host,
     when compiled in "mic-native" mode, it gets allocated on mic. */

  float (*restrict A)[size] = malloc(sizeof(float)*size*size);
  float (*restrict B)[size] = malloc(sizeof(float)*size*size);
  float (*restrict C)[size] = malloc(sizeof(float)*size*size);

  /* this first pragma is just to get the actual #threads used
     (sanity check). */

  #pragma omp parallel
  {
    nt = omp_get_num_threads();

    /* Fill the A and B arrays with dummy test data. */
    #pragma omp parallel for default(none) shared(A,B,size) private(i,j)
    for(i = 0; i < size; ++i) {
      for(j = 0; j < size; ++j) {
        A[i][j] = (float)i + j;
        B[i][j] = (float)i - j;
      }
    }
  }

  /* run the matrix multiplication function nIter times and average
     runs therein. */

  float Gflop = fastMult(size,A,B,C,nIter);
  printf("size = %d x %d; nThreads = %d; #GFlop/s = %g\n",
         size, size, nt, Gflop);

  free(A);
  free(B);
  free(C);

  return 0;
}
```

9.c. Compilation and runtime environment for codes above

All environment variables and example command lines below were set/executed directly on the Host (Xeon E5) processor. The environment variables containing "MKL" and compiler option "–mkl" can be omitted for the "naïve" test code of Section 9.a. Furthermore, for the code in Section 9.b., the `MKL_MIC_ENABLE` environment variable was used to control the offloading of MKL-specific routines to the Xeon Phi. This avoids explicit use of offload "target(mic)" pragmas for the MKL routines.

Depending on the execution mode, three different binaries were generated for each test?.c code above: *test_mic*, *test_mic_offload*, *test_host*. The input arguments are *"size of square matrix along one dimension"; "number of concurrent threads"; "number of iterations for runtime averaging"*.

Full native execution on the Xeon Phi:

```
source /opt/intel/bin/compilervars.csh intel64

icc -mkl -O3 -mmic -openmp -L /opt/intel/lib/mic -Wno-unknown-pragmas -std=c99
-vec-report2 -liomp5  -o test_mic test.c

setenv SINK_LD_LIBRARY_PATH
/opt/intel/composer_xe_2013/lib/mic:/opt/intel/mkl/lib/mic;
setenv MKL_MIC_ENABLE 1;
setenv MIC_ENV_PREFIX MIC;
setenv MIC_KMP_AFFINITY "granularity=thread,balanced";
setenv MIC_USE_2MB_BUFFERS 32K
setenv MIC_MKL_DYNAMIC false

/opt/intel/mic/bin/micnativeloadex ./test_mic -a "3000 240 10";
```

Heterogeneous binary with offloading of selective code to the Xeon Phi:

```
source /opt/intel/bin/compilervars.csh intel64

icc -offload-option,mic,compiler,"-mP2OPT_hpo_vec_check_dp_trip=F -fimf-
precision=low -fimf-domain-exclusion=15  -opt-report 1" -
mP2OPT_hlo_pref_issue_second_level_prefetch=F -
mP2OPT_hlo_pref_issue_first_level_prefetch=F -vec-report2  -O3  -openmp -
intel-extensions -opt-report-phase:offload -openmp-report -mkl -Wno-unknown-
pragmas -std=c99  -o test_mic_offload test.c

setenv MKL_MIC_ENABLE 1;
setenv MIC_ENV_PREFIX MIC;
setenv MIC_KMP_AFFINITY "granularity=thread,balanced";
setenv MIC_USE_2MB_BUFFERS 32K
setenv MIC_MKL_DYNAMIC false;
setenv KMP_AFFINITY "granularity=thread,scatter";

./test_mic_offload 3000 240 10
```

Host-only execution (no use of the Xeon Phi):

```
source /opt/intel/bin/compilervars.csh intel64

icc -xhost -mkl -O3 -no-offload -openmp -Wno-unknown-pragmas -std=c99 -vec-
report2 -o test_host test.c

setenv MKL_MIC_ENABLE 0;
setenv KMP_AFFINITY "granularity=thread,scatter";
setenv USE_2MB_BUFFERS 32K;
setenv MKL_DYNAMIC false;

./test_host 3000 32 10
```