VRE for regional Interdisciplinary communities in Southeast Europe and the Eastern Mediterranean

Software techniques for optimization for the Intel Xeon Phi coprocessors

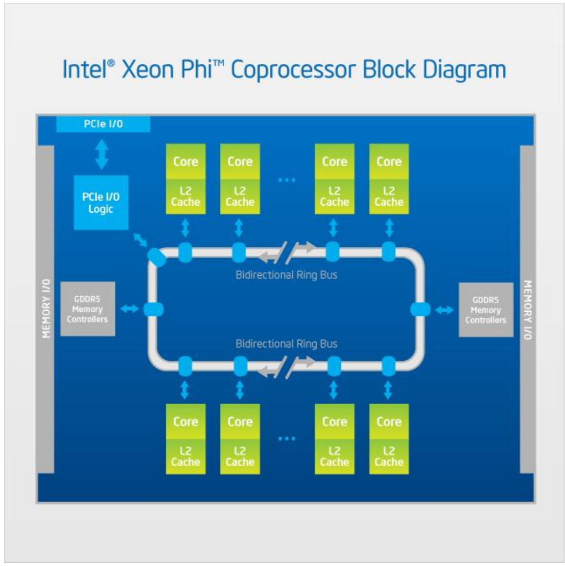
Emanouil Atanassov
Institute of Information and Communication Technologies - BAS

Outline

- About the Xeon Phi coprocessor
- Software configuration at the coprocessor
- Cross-compiling with development tools
- Using vtune etc. – to be skipped
- How to use profiling information
- 3 types of using the coprocessor
- Environmental variables that control execution
- Using MKL automatic offload with the coprocessor

The Xeon Phi coprocessor

- The Xeon Phi coprocessor is in our system an add-on card that is plugged-in into something that looks like a regular server, starts-up its own OS image and can communicate using Ethernet or Infiniband.
- Our version is Xeon Phi 7120P. It has 61 physical cores, 16 GB RAM.
- Frequency 1232.263 Mhz.
- Each core can run 4 independent hardware threads of execution.
- Typically one core is reserved for the OS, but this is not enforced.
- Has vector unit for floating point, where 8 double precision numbers or 16 float point numbers can be processed in one instruction. This is the main advantage of having Xeon Phi.



Types of application execution - offload

- Offload mode – application is started on the normal CPU, some subroutines are executed actually on the Xeon Phi.
- More than one card may be used.
- Programmers can control the offloading
- Users can also control the offloading, for example via environment variables.
- Cheap way to introduce execution on Xeon Phi only for some parts of otherwise complex codes.
- Not future-proof

Types of application execution - native

- Native mode – application is executed only on the Xeon Phi.
- Usually application is cross-compiled on the server, then via ssh is launched on the Xeon Phi.
- When using configure, add:

  --host=x86_64-k1om-linux
- One should control LD_LIBRARY_PATH to make sure all libraries are available.
- Same directories like /home, /opt/intel, etc. are available in Xeon Phi.
- Architecture is obviously different
- Some instructions from the x86_64 instruction set are missing on Xeon Phi, therefore some assembly code will not compile or execute properly.

Types of application execution – symmetric mode

- In symmetric mode the application is using both the CPU and the Xeon Phi.
- In our case – 2 CPUs and 2 Xeon Phi coprocessors.
- Usually MPI is used to launch different applications – one for the CPUs and the other for the Xeon Phi.
- The main server is a NUMA machine, although it is with shared memory.
- Memory is allocated on "first touch" – if one of the two CPUs first touches virtual memory region, this region is allocated from memory that is physically close to this CPU.
- It is logical to have at least two MPI processes and then to use OpenMP for each process with 8 or 16 threads. With hyperthreading – 2x16 threads in total, without hyperthreading – 2x8 threads.
- Communication between the MPI processes happens over infiniband

Types of application execution

- The symmetric mode is the most advanced. However, it requires two executables to be created and load-balancing between them to be performed – it is complicated.
- The native mode can be a first step to symmetric mode. The result may be good enough.
- The offload mode is “cheap” in the sense that a complex application which depends on outside libraries that are not always available on the Xeon Phi can still use the floating point power of Xeon Phi in some routines.

How to use the advantages of Xeon Phi

- Using libraries that make use of its floating point capabilities
  - MKL
  - Others
- Developing software with automatic or manual vectorization
  - The Intel Compilers may be able to use the vector capabilities of Xeon Phi
  - Developers may use hints or directives to help the compiler
  - The GCC compiler is not able to use the vector capabilities of Xeon Phi currently
- Using applications that are optimized for Xeon Phi –
  - Nothing special to be done in this case – just run the application

Using MKL on Xeon Phi

- Some MKL routines are specifically optimized for Xeon Phi
- When compiling for native execution on Xeon Phi, this routines are automatically used.
- Automatic offload happens when problem size is big enough
- It includes the following routines:
  - ?GEMM, ?SYMM, ?TRMM, and ?TRSM
  - LU, QR, Cholesky factorizations
  - Can be enabled or disables:
    - rc = mkl_mic_enable( ) – in the code
    - MKL_MIC_ENABLE=1 – via environment variable (in bash shell).
- If OFFLOAD_REPORT is on, users can use the function mkl_mic_set_offload_report() to dynamically turn on/off reporting to understand what is happening.
- MKL_MIC_DISABLE_HOST_FALLBACK=1 - disable the automatic fallback to the host - find bugs.
- OMP_NUM_THREADS becomes MIC_OMP_NUM_THREADS, KMP_AFFINITY becomes MIC_KMP_AFFINITY for the MIC

Using MKL on Xeon Phi

- MKL is highly optimized. It contains routines for:
- Basic Linear Algebra Subprograms (BLAS):
- Sparse BLAS Level 1, 2, and 3 (basic operations on sparse vectors and matrices)
- LAPACK routines for solving systems of linear equations
- LAPACK routines for solving least squares problems, eigenvalue and singular value problems, and Sylvester's equations
- ScaLAPACK computational, driver and auxiliary routines
- PBLAS routines for distributed vector, matrix-vector, and matrix-matrix operation
- Direct and Iterative Sparse Solver routines
- Vector Mathematics (VM) functions for computing mathematical functions on vectors
- Vector Statistics (VS) functions for generating vectors of pseudorandom numbers
- General Fast Fourier Transform (FFT) Functions
- Cluster FFT functions
- Tools for solving partial differential equations
- Optimization Solver routines for solving nonlinear least squares problems through the Trust-Region (TR) algorithms and computing Jacobi matrix by central differences

Using MKL on Xeon Phi

- If an application fails to run on the MIC because of not found dynamic libraries, fix LD_LIBRARY_PATH
- Use ldd to find which libraries are not found and look for them.
- Sometimes static linking makes faster executables, but beware that application may fail to run after the glibc library on the system is upgraded.
- In general, some libraries are more difficult to be found for static linking (.a suffix vs .so suffix).

Specifics of Xeon Phi

- Using Xeon Phi instead of just CPU is useful if its floating points capabilities are used.
- Xeon Phi can not achieve its maximum if only one hardware thread per core is used
- That is why it is normal to use more threads (or MPI processes) on the MIC.
- One can try with 60, 61, 120, 122, 180, 183, 240, 244 to see which is fastest.
- For OpenMP programs – use OMP_NUM_THREADS.
- For MKL offload – use MIC_OMP_NUM_THREADS
- export MIC_ENV_PREFIX=MIC
- export MIC_OMP_NUM_THREADS=60
- When combining MPI and OpenMP, use

    –genv MIC_OMP_NUM_THREADS

instead

Environmental variables controlling MPI execution

- Many variables control MPI communications at execution time.
- Use I_MPI_DEBUG to display some debugging info if program not starting for example
- Use I_MPI_STATS to collect statistics.
- See which routines are most used.
- I_MPI_FABRICS=shm:dapl – if it is not the default, it can be better
- I_MPI_FABRICS=shm:ofa – usually slower.
- I_MPI_ADJUST_REDUCE=2 – select second algorithm for reduce.
- I_MPI_DAPL_SCALABLE_PROGRESS=1 – can be better or not
- I_MPI_DAPL_US=enable – can be better or not
- I_MPI_FALLBACK=disable - find bugs in configuration, otherwise program runs slow

Compilation of hybrid applications

- Hybrid applications that use both MPI and OpenMP can be compiled only if the proper options are used and the MPI library supports it.
- MPI library from Intel does support that mode.
- Version of openmpi that presumably supports this mode is also available.
- Version of openmpi without the overhead of such support will be faster if no OpenMP is used.
- To use the multithreaded version of the MPI library, load with option release_mt
- To compile properly, add option –mt_mpi to mpiicc (or mpiifort).
- When starting, use instead of MPI_INIT
- int required_level=MPI_THREAD_SERIALIZED;
- int provided_level;
- MPI_Init_thread(&argc, &argv, required, &provided);
- Always check if provided_level==required_level, program may not fail immediately and the bug will be difficult to understand.

Conclusions

- Use the path of least resistance
- Many options are available even for non-developers to improve application performance
- Little testing can give lots of speed improvement.

- Launch job log on to MIC, see what is there
- top
- cat /proc/cpuinfo
- ifconfig
- mount

- Hello MIC program – native
- MKL example Monte Carlo – simple - job
- MKL example Monte Carlo – complicated – OpenMP + MPI on MIC
- MKL example Monte Carlo – complicated symmetric - combine HOST + MIC
- start vtune – demo

OpenMP example

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define CHUNKSIZE   10
#define N       100
int main (int argc, char *argv[]){
int nthreads, tid, i, chunk;
float a[N], b[N], c[N];
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;
#pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
  {
  tid = omp_get_thread_num();
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  #pragma omp for schedule(dynamic,chunk)
  for (i=0; i<N; i++)
    {
    c[i] = a[i] + b[i];
    printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
    }
  } /* end of parallel section */
}
```

OpenMP example

- Copy example omp_test.sh and omp_test.c
- icc -qopenmp omp_test.c -o omp_test.out
- OMP_THREAD_NUM=12 ./omp_test.out
- Can be run on the head node
- Now try for the mic:
- Copy example omp_test.sh and omp_test.c
- Compile and submit
- icc -qopenmp –mmic omp_test.c -o omp_test.out
- qsub –q edu omp_test.sh
- Note the LD_LIBRARY_PATH
- Set OMP_NUM_THREADS=122 and run again.

OpenMP using mkl

```
#include <stdio.h>
#include <omp.h>
#include "mkl_vsl.h"
#define BLOCK 100
#define ITER 1000
int main(){
double s=0.;
#pragma omp parallel default(none) reduction(+:s)
{
  double buff[BLOCK];
VSLStreamStatePtr stream; // Select type of VSLStreamStatePtr stream;
int seed_val=omp_get_thread_num();
vslNewStream(&stream, VSL_BRNG_WH, (int)seed_val);
int i;
for ( i=0; i<ITER; i++ ) {
     if (i % omp_get_num_threads() == omp_get_thread_num()){
          vdRngGaussian (VSL_RNG_METHOD_GAUSSIAN_ICDF, stream, BLOCK, buff, 5, 2);
          for (int j=0;j<BLOCK;j++){
               s += buff[j];
          }
     }
}
s=s/ITER/BLOCK;
vslDeleteStream( &stream );
}
  /* Printing results */
  printf( "Sample mean of normal distribution = %f\n", s );
  return 0;
}
```

MPI + OpenMP + MKL

- mpiicc -mt_mpi -mkl -qopenmp mpi_test.c
- ^^^^^^^
- Do not forget
- Load the release_mt version of MPI_LIBRARY