Experiments with WRF on Intel® Many Integrated Core (Intel MIC) Architecture

Larry Meadows

Intel Corporation, Hillsboro OR, USA
lawrence.f.meadows@intel.com

Abstract. WRF is a well-known weather code with a hybrid OpenMP and MPI implementation. This paper investigates the performance of WRF on heterogeneous hardware consisting of Intel® Xeon® processors and Intel MIC Architecture co-processors, using offload, OpenMP, and MPI.

1 Introduction

The Intel Many Integrated Core (MIC) architecture[1] was originally announced in May of 2010. The Knights Ferry design and development kit will be followed with the Knights Corner product. The Intel MIC architecture combines many Intel Architecture CPU cores on a single chip. This architecture is very different from the GPGPU architectures in that it can execute a full operating system and entire programs, rather than just kernels. In particular, it is possible to run one or more ranks of an MPI program on an Intel MIC chip.

The Weather Research and Forecasting (WRF) Model[2] is a widely respected weather prediction system developed by a collaborative partnership among NCAR, NOAA, and several other agencies. A version of WRF was included in the retired SPEC HPC2002 benchmark suite and is included in the SPEC MPI2007 benchmark suite. WRF has a hybrid MPI and OpenMP parallel model, and comes with benchmark data sets that represent real problems and include verification tests, making it an excellent code for studying and tuning performance on heterogeneous parallel systems.

Section 2 describes the Intel MIC hardware architecture. Section 3 describes the Intel OpenMP and MPI implementations for Intel MIC architecture. Section 4 describes WRF, its parallel model, and the benchmark data set. Section 5 presents results for compiler offload. Section 6 presents results for hybrid MPI+OpenMP runs on heterogeneous hardware.

2 Intel MIC Architecture

2.1 Hardware Architecture

The results in this paper were obtained on a Knights Ferry (KNF) PCI-Express card. The KNF processor consists of 32 in-order cores running at 1200MHz,

B.M. Chapman et al. (Eds.): IWOMP 2012, LNCS 7312, pp. 130–139, 2012.
© Springer-Verlag Berlin Heidelberg 2012

each with four hardware thread contexts. Each core has 256KiB of shared L2 cache, 32KiB of L1 data cache, and 32Kib of L1 instruction cache. The cores are interconnected to each other and to memory controllers by a ring bus. All caches are coherent with each other and with main memory. Each core has a 512-bit vector floating point unit that is able to operate on 16 single precision floating point values per cycle.

The hardware thread contexts each have their own set of scalar Intel64 registers as well as 512-bit vector floating point registers. The four hardware threads help to cover latency as is usual in SMT architectures.

Fig. 1 is a block diagram of the Intel MIC architecture.

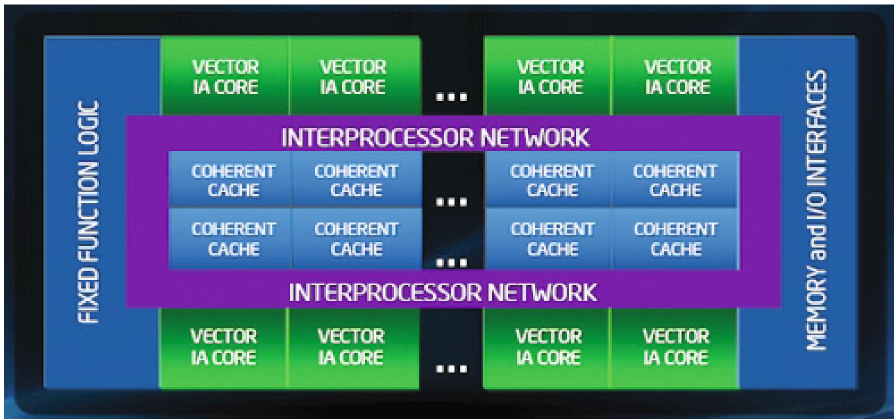

Fig. 1. Intel MIC Architecture Block Diagram

KNF is limited to 2GiB of GDDR5 memory.

Multiple KNF cards can communicate with each other and with the host over the PCI express bus. A DMA engine supports asynchronous data movement between cards and between a card and the host.

The host hardware used in this paper is a dual socket Intel Xeon X5680 running at 3.33GHz. Each socket is a 6 core part, and each core has two hardware thread contexts, for a total of 24 hardware thread contexts. The system has 24GiB of memory and runs Red Hat EL6.

2.2 System Software Architecture

KNF runs a version of the Linux kernel with a Busybox[3]environment. The device has a ramdisk to hold the kernel and the command environment. A virtual ethernet driver and NFS support is provided, so host file systems can be mounted on the card, or native executables and shared objects can be copied to the ramdisk.

Both the host kernel and the device kernel support DMA through a kernel driver. DMAs can be initiated from either the host or the device and can be

reads or writes. The hardware DMA engine requires 64-byte alignment of source and destination addresses. It is also possible to map remote memory across the PCI-Express bus so that writes from one side are visible to the other side.

For memory to be mapped or DMAd, the physical memory must first be registered (and the virtual addresses pinned) so that it isn't paged out. Registration is a relatively expensive operation, and there is a limit to the amount of memory that can be registered, so careful management of memory registration is required.

3 Software Stack Implementation

The work described here uses three major pieces of software: OpenMP, MPI, and compiler-assisted offload.

3.1 OpenMP

Intel's OpenMP implementation fully supports the OpenMP 3.1 specification[4]. OpenMP annotations are recognized by the compiler, which then generates code and calls to the OpenMP runtime library to realize the OpenMP parallelism[5].

Since Intel MIC is an implementation of Intel64 Architecture, it is able to use substantially the same compiler and runtime as Xeon. There are two differences between Intel MIC and Intel Xeon that affect the runtime implementation or the optimal use of OpenMP: thread yielding and affinity.

The Intel MIC core is a 4-thread implementation of simultaneous multithreading (SMT)[6]. When a thread is stalled the core executes another thread that is able to execute. The OpenMP runtime often uses busy wait loops for synchronization. It is important that the thread executing the busy wait loop yield the core so that other threads on the same core can execute. Intel MIC provides the `delay` instruction for this purpose. The `delay` instruction causes the hardware thread picker to advance to the next thread on the core. This instruction takes an integer argument giving the number of cycles for which the thread should be skipped.

Each hardware thread on Intel MIC is represented by an OS CPU number. By default, the OS can schedule OpenMP threads on any available CPU, and can change the CPU on which an OpenMP thread is executed at any time. It is often useful to restrict the sets of CPUs on which an OpenMP thread can execute, and also to place particular OpenMP threads on the same core as other threads.

Intel provides the `KMP_AFFINITY` environment variable (as well as an API) for this purpose[7]. OpenMP threads can be bound to individual OS CPUs or to cores, and can be placed according to a policy, or explicitly placed on a given core or OS CPU. The policies are called `compact`, to place threads in order on cores; `scatter`, to place threads round-robin on cores; and `balanced`, to place threads in order on cores but to use as many cores as possible.

3.2 MPI

One of the major differences between Intel MIC and GPGPU is that Intel MIC supports a full OS environment. Entire programs, not just kernels, can execute on Intel MIC. Intel provides an implementation of MPI-2[8] based on MPICH2[9] that supports MPI communication between process on the devices and processes on the host, and will support clusters of such nodes. The implementation uses a combination of mapped memory and DMA for host-device communication, and shared memory for intra-device and intra-host communication.

3.3 Compiler Offload

Intel MIC can be used in two ways: as a platform for native execution, and as an offload platform to run portions of a computation. The latter is the way in which GPGPUs are used as accelerators today. Intel's compiler provides the ability to offload essentially arbitrary sections of code to the device from a program executing on the host using directives (in Fortran) and pragmas (in C/C++). The offload directives typically specify the data to be transferred between the host and the device and delimit the code to be executed on the device. The actual code that is executed on the device can use OpenMP directives or any other model of parallelism, make system calls, do I/O, and anything else a native program can do on the device.

Section 5 has an example of offloading a computation from WRF onto the device (in Fortran).

Offload directives exist for Fortran, C, and C++. The remainder of this section describes some of the directives for Fortran.

It is possible to offload an OpenMP parallel region to the device with the `!DEC$ OMP OFFLOAD` directive followed by an OpenMP parallel construct. This results in an offload region consisting of the body of the parallel construct:

```
!DEC$ OMP OFFLOAD
!OMP PARALLEL
...
!$OMP END PARALLEL
```

The combined parallel workshare constructs may also be used. It is also possible to offload a block of code or a call statement.

Attributes following the offload directive tell the compiler which variables are input to the device (`IN`), output from the device (`OUT`), or both input and output (`INOUT`). The compiler generates code to transfer the input data before executing the offload region, execute the offload region, and transfer the data back after executing the offload region. The compiler attempts to automatically transfer data that is used in the offload region, but it is often necessary and/or more efficient to explicitly specify the data to be transferred. There are also mechanisms to transfer data asynchronously and to start an offload region and later wait for its completion.

It is also possible to place persistent data, for example, Fortran common blocks, on both the device and the host. Such data can be updated independently

by both the device and the host, and can also be transferred to and from the host with the appropriate offload attributes.

Finally, if a subprogram is called from within an offload region, it is necessary to annotate the subprogram definition so that a device version of the subprogram is created. This is done with the `!DEC$ ATTRIBUTES OFFLOAD: MIC::` *`subprogram-name`* directive.

When an OpenMP offload region is created on the device, it is completely independent from any OpenMP region on the host. The number of threads, thread affinity, and any other attributes of the region are determined from device-specific environment variables or OpenMP API calls. However, OpenMP thread teams persist from one offload region to another on the device, so any thread creation cost is incurred only once.

4 WRF Benchmark

WRF can be run as a single OpenMP process or as multiple MPI+OpenMP processes. The MPI implementation decomposes the domain and exchanges the boundaries of the grid at each timestep. The OpenMP implementation further decomposes the grid into a set of tiles with one tile per OpenMP thread. Each set is computed in parallel using the OpenMP `PARALLEL DO` construct. The implicit barrier at the end of each construct is used for synchronization.

The benchmark data set used in this paper is the single domain 12km Continental U.S. (CONUS) dataset with a simulation time step of 72 seconds over a three hour simulated time period. This benchmark is from the WRF V3 benchmark page[10]. Version 3.0 of the WRF code was used.

The majority of the execution time is spent in the `solve_em` subroutine and in subroutines it calls. This subroutine contains 38 `PARALLEL DO` constructs. Thus there is overhead for starting and ending each construct, as well as some serial time outside of the constructs.

We used Vtune Amplifier XE to profile serial execution of the code on the host. The top 20 functions accounted for 69% of the execution time. The microphysics routine `wsm52d` took 11.6% of the time.

5 WRF Offload

Reference [11] describes work done to offload the wsm5 microphysics routine to an Nvidia accelerator. Since this function takes only 11.6% of the serial time, this work did not result in significant overall performance improvement. However, it is a good way to demonstrate the offload capabilities of different compilers and hardware, so we performed a similar experiment.

The outer loop for this region is show in Fig.2. The NVIDIA offload involved substantial code restructuring to better match the device's characteristics. Since Intel MIC is a general purpose CPU architecture, we were able to offload this region by simply placing an appropriate directive identifying the offload region and data to be transferred before the outermost DO loop, as seen below:

```
DO j=jts,jte
   DO k=kts,kte
   DO i=its,ite
      t(i,k)=th(i,k,j)*pii(i,k,j)
      qci(i,k,1) = qc(i,k,j)
      qci(i,k,2) = qi(i,k,j)
      qrs(i,k,1) = qr(i,k,j)
      qrs(i,k,2) = qs(i,k,j)
   ENDDO
   ENDDO
   CALL wsm52D(t, q(ims,kms,j), qci, qrs                    &
              ,den(ims,kms,j)                               &
              ,p(ims,kms,j), delz(ims,kms,j)                &
              ,delt,g, cpd, cpv, rd, rv, t0c                &
              ,ep1, ep2, qmin                               &
              ,XLS, XLV0, XLF0, den0, denr                  &
              ,cliq,cice,psat                               &
              ,j                                            &
              ,rain(ims,j),rainncv(ims,j)                   &
              ,sr(ims,j)                                    &
              ,ids,ide, jds,jde, kds,kde                    &
              ,ims,ime, jms,jme, kms,kme                    &
              ,its,ite, jts,jte, kts,kte                    &
              ,snow(ims,j),snowncv(ims,j)                   &
                                                           )
   DO K=kts,kte
   DO I=its,ite
      th(i,k,j)=t(i,k)/pii(i,k,j)
      qc(i,k,j) = qci(i,k,1)
      qi(i,k,j) = qci(i,k,2)
      qr(i,k,j) = qrs(i,k,1)
      qs(i,k,j) = qrs(i,k,2)
   ENDDO
   ENDDO
ENDDO
```

Fig. 2. wsm5 outer loop

```
!dec$ omp offload target(mic:0) in(delt,g,cpd,cpv,t0c, &
!dec$& den0,rd,rv,ep1,ep2,qmin,XLS,XLV0,XLF0,cliq,      &
!dec$& cice,psat,denr,jts,jte,kts,kte,its,ite,ims,      &
!dec$& kms,ids,                                         &
!dec$& ide,jds,jde,kds,kde,ime,jms,jme,kme)             &
!dec$& in(t,qci,qrs) in(delz,p,den,pii)                 &
!dec$& inout(snowncv,rainncv)                           &
!dec$& inout(qs,qr,qi,qc,th,q,snow,rain)                &
!$omp parallel do private(i,j,k,t,qci,qrs)
```

This tells the compiler which variables to transfer in to, out from, or both in to and out from the card, and further to create an OpenMP `PARALLEL DO` to run on the card. The compiler creates variable descriptors and a function (encapsulated in a shared object). When the first offload region is encountered, the offload runtime on the card loads the shared object into memory; then the offload runtime on the host and the card cooperate to exchange the data and call the offload function.

It is also necessary to ensure that persistent data on the card is properly initialized. The routine `wsm5init` must be called on the card. Finally, the routine `rgmma` is called from `wsm52D` and the compiler must be told that a device version of that routine is required. The following additional directives, and one additional subroutine call, were required in `module_mp_wsm5.F` and `module_physics_init.F`:

```
module_mp_wsm5.F:
!dec$ attributes offload: mic:: wsm52D
!dec$ attributes offload: mic:: rgmma
!dec$ attributes offload: wsm5init
module_phsics_init.F:
!dec$ offload target(mic:0) &
!dec$ in(rhoair0,rhowater,rhosnow,cliq,cpv,allowed_to_read)
      call wsm5init(rhoair0,rhowater,rhosnow,cliq,cpv, &
                    allowed_to_read)
```

Offloading wsm5 to the Intel MIC card decreased the time spent in wsm5 by a factor of 4.4. This includes the time for data transfer to and from the card. This speedup is comparable to the speedup quoted in [11].

It does not appear that WRF has any other obvious offload opportunities, at least on this benchmark data set. The other high profile routines are not as compute intensive relative to the amount of data that would have to be transferred. Conceivably the offload could be performed at the level of the OpenMP parallel regions in `solve_em`, but this would involve moving almost all the data to the card and would require a substantial number of offload directives. Therefore, we took advantage of the general purpose abilities of Intel MIC and used MPI to move an entire subdomain of the model to the card. This is described in the next section.

6 MPI Implementation

WRF supports parallelism at two levels. The problem can be decomposed into MPI ranks (processes), and then each rank can use OpenMP for parallelism within the process. This made it easy to run part of the benchmark on the MIC card and part of the benchmark on the host.

Normally we would have started by running a single process on the MIC card, tuning for OpenMP and serial performance, and then adding MPI. However, the benchmark data set is too large to fit in the 2GiB memory of the KNF card. Measurements on the host indicate that the benchmark requires more than 3GiB of memory.

The bulk of the WRF data goes on the main thread's stack. Each OpenMP thread also requires a stack. Since Intel MIC uses lots of threads, the per-thread stack size becomes significant. After experimentation we determined that one MPI rank would fit on the card if the problem was decomposed into four MPI ranks. This resulted in a main thread stack size of 450MiB and per-thread stack sizes of 7MiB for a total of 898MiB on 64 threads. Together with the code size, other internal memory usage, and the ramdisk holding the OS and the images, and memory usage by the kernel and other processes, this resulted in 100% memory usage on the card.

Running 4 ranks, with three on the host and one on the card, resulted in a simulation time speedup of 3.29x over the serial code. The remainder of this section analyzes the various bottlenecks.

6.1 Timing Model

The OpenMP regions all have implicit barriers at the end. There are no MPI calls in the OpenMP regions. The MPI exchanges also result in implicit synchronization. Each simulation timestep consists of a number of OpenMP regions with MPI exchanges and some serial code in between. So the timing model is relatively simple: $T_{step} = T_{omp} + T_{mpi} + T_{serial}$. T_{serial} includes both serial computation time and serial OpenMP overhead (such as loop setup and fork-join overhead). T_{omp} can be further divided into true parallel time and load imbalance: $T_{omp} = T_{par} + T_{imb}$. Load imbalance occurs when one thread has less work than another thread, and shows up as more time spent by a thread in the implicit barrier.

6.2 Timing Measurement and Results

Vtune has two different kinds of profiling collectors. The software stack sampling collector uses posix timers to generate periodic interrupts to each thread. It records the stack at each sample and then provides a breakdown by callstack. This collector was used on the host to collect the initial profile mentioned in Section 4. The second collector uses hardware Performance Monitoring Unit (PMU) events. The events are programmed to interrupt after a certain count threshold is reached. This collector does not provide a callstack but it does

provide the IP of the instruction that was executing when the interrupt occured. The software stack sampling collector is not yet available on Intel MIC so we used the PMU collector. We used the hardware event CPU_CLK_UNHALTED which increments on every clock cycle with an overflow threshold of 2,000,000, which results in an effective sample rate of 600 HZ. Using the IP associated with the sample we can get a statistical profile of the application.

The OpenMP runtime library is instrumented to record the entry and exit to each parallel region. The entry is recorded on each thread, and the exit is recorded by the main thread after the parallel region completes (immediately after the master thread returns from the barrier).

Both the runtime instrumentation points and the PMU samples use the high-resolution Time Stamp Counter (TSC) available on the Intel MIC. The TSCs are synchronized between the cores and their resolution is the clock frequency; furthermore the instruction (RDTSC) that is used to read the TSC has very low overhead.

Thus we have two data sources that we can accurately correlate: the beginning and end of each parallel region, and the samples themselves. This gives us estimates of T_{omp} and of $T_{mpi} + T_{serial}$ as follows: when a sample for a thread falls within a parallel region for that thread, it is labeled as T_{omp}; when it falls outside a parallel region, it is labeled as T_{serial}.

We can further segregate the T_{serial} samples by looking at the shared object (module) in which the IP for the sample resides. For example, the OpenMP runtime and the MPI runtime are implemented as shared objects; further the WRF code itself is a separate module (the main program `wrf.exe`).

We tried to measure MPI time directly using mpiP[12]. The tool worked, but it increased execution time by more than 10%, so the results were inconclusive. Ideally we could get T_{mpi} from the samples that fall into the MPI module; however, much of the MPI time is spent doing DMA in the kernel.

The following table shows T_{par} and T_{imb} statistics per thread, and T_{omp}, T_{serial}, and T_{mpi} (estimated) as percentages of the total simulation time.

	Average	Stddev	Overall
T_{par}	45.17	7.97	
T_{imb}	17.03	7.35	
T_{omp}			64.64
T_{serial}			7.70
T_{mpi}			27.56

7 Conclusions and Future Work

This paper describes the results of running one of the standard WRF benchmarks on Intel MIC KNF hardware using both compiler offload and heterogeneous MPI. A very low overhead method of determining OpenMP load imbalance is presented. The design and development KNF kit exhibits respectable performance.

Some of the methods used in this paper provide implementation ideas for future tools. Future areas for investigation include finer-grained characterization of OpenMP and MPI overheads and analysis of core-level performance on specific WRF subroutines.

References

1. Intel Corporation, http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html
2. NCAR, et al.: The Weather Research and Forecasting Model, http://www.wrf-model.org
3. http://busybox.net/about.html
4. The OpenMP ARB, Inc., OpenMP Application Program Interface Version 3.1, http://www.openmp.org
5. Tian, X., et al.: Intel OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance. Intel Technology Journal 6(1) (February 2002)
6. Eggers, S.J., Emer, J.S., Levy, H.M., Lo, J.L., Stamm, R.L., Tullsen, D.M.: Simultaneous Multithreading: A Platform for Next-Generation Processors. IEEE Micro, 12–19 (October 1997)
7. Intel Corporation, Intel® Fortran Compiler XE 12.1 User and Reference Guides, Document number 323276-121US
8. Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, Version 2.2, http://www.mpi-forum.org
9. Argonne National Laboratories, http://www.mcs.anl.gov/research/projects/mpich2/
10. WRF WG2, WRF V3 Parallel Benchmark Page, http://www.mmm.ucar.edu/wrf/WG2/benchv3/
11. Wolfe, M., Toepfer, C.: PGI Insider, The PGI Accelerator Programming Model on NVIDIA GPUS Part 3: Porting WRF (October 2009)
12. Vetter, J., Chambreau, C.: mpiP: Lightweight, Scalable MPI Profiling, Version 3.3, June 23 (2011), http://mpip.sourceforge.net