Best Practice Guide
Intel Xeon Phi v1.1

Michaela Barth, KTH Sweden

Mikko Byckling, CSC Finland

Nevena Ilieva, NCSA Bulgaria

Sami Saarinen, CSC Finland

Michael Schliephake, KTH Sweden

Volker Weinberg (Editor), LRZ Germany <weinberg@lrz.de>

14-02-2014

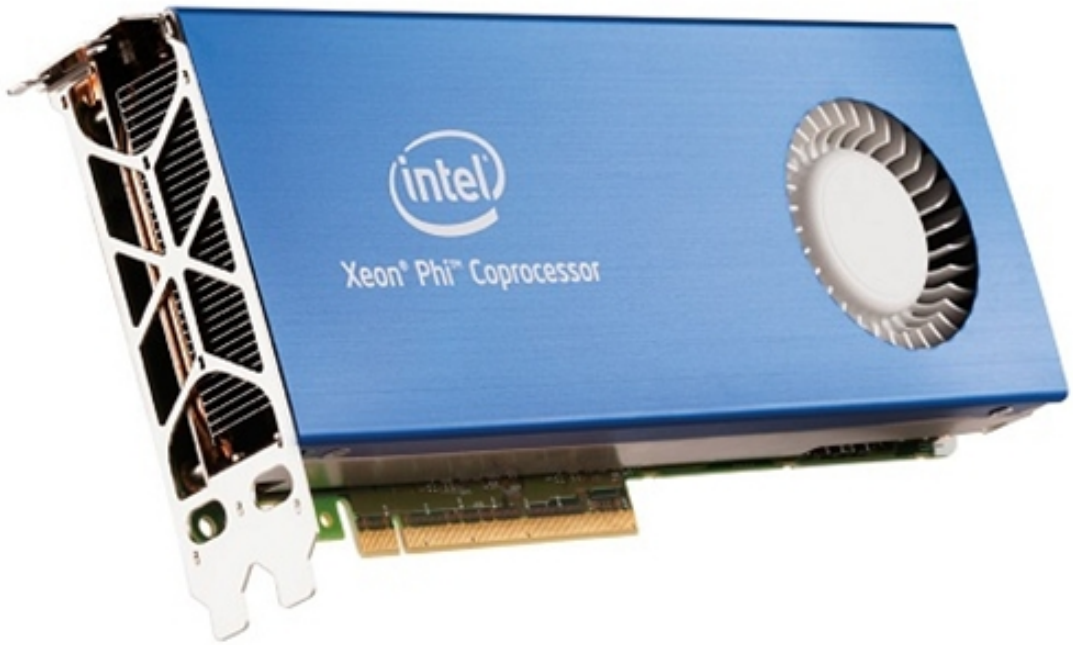
Table of Contents

1. Introduction 4
2. Intel MIC architecture & system overview 4
2.1. The Intel MIC architecture 4
2.1.1. Intel Xeon Phi coprocessor architecture overview 4
2.1.2. The cache hierarchy 6
2.2. Network configuration & system access 7
3. Native compilation 11
4. Intel compiler's offload pragmas 12
4.1. Simple example 12
4.2. Obtaining informations about the offloading 13
4.3. Syntax of pragmas 14
4.4. Recommendations 15
4.4.1. Explicit worksharing 15
4.4.2. Persistent data on the coprocessor 15
4.4.3. Optimizing offloaded code 16
4.5. Intel Cilk Plus parallel extensions 17
5. OpenMP and hybrid 17
5.1. OpenMP 17
5.1.1. OpenMP basics 17
5.1.2. Threading and affinity 17
5.1.3. Loop scheduling 18
5.1.4. Scalability improvement 18
5.2. Hybrid OpenMP/MPI 19
5.2.1. Programming models 19
5.2.2. Threading of the MPI ranks 19
6. MPI 19
6.1. Setting up the MPI environment 19
6.2. MPI programming models 20
6.2.1. Coprocessor-only model 20
6.2.2. Symmetric model 20
6.2.3. Host-only model 21
6.3. Simplifying launching of MPI jobs 21
7. Intel MKL (Math Kernel Library) 21
7.1. MKL usage modes 22
7.1.1. Automatic Offload (AO) 22
7.1.2. Compiler Assisted Offload (CAO) 23
7.1.3. Native Execution 23
7.2. Example code 24
7.3. Intel Math Kernel Library Link Line Advisor 24
8. TBB: Intel Threading Building Blocks 24
8.1. Advantages of TBB 24
8.2. Using TBB natively 25
8.3. Offloading TBB 27
8.4. Examples 27
8.4.1. Exposing parallelism to the processor 27
8.4.2. Vectorization and Cache-Locality 31
8.4.3. Work-stealing versus Work-sharing 31
9. IPP: The Intel Integrated Performance Primitives 31
9.1. Overview of IPP 31
9.2. Using IPP 32
9.2.1. Getting Started 32
9.2.2. Linking of Applications 33
9.3. Multithreading 33
9.4. Links and References 33
10. Further programming models 33

10.1. OpenCL 34
10.2. OpenACC 34
11. Debugging 34
11.1. Native debugging with gdb 34
11.2. Remote debugging with gdb 34
12. Tuning 35
12.1. Single core optimization 35
12.1.1. Memory alignment 35
12.1.2. SIMD optimization 36
12.2. OpenMP optimization 37
12.2.1. OpenMP thread affinity 37
12.2.2. Example: Thread affinity 38
12.2.3. OpenMP thread placement 39
12.2.4. Multiple parallel regions and barriers 40
12.2.5. Example: Multiple parallel regions and barriers 40
12.2.6. False sharing 42
12.2.7. Example: False sharing 42
12.2.8. Memory limitations 44
12.2.9. Example: Memory limitations 44
12.2.10. Nested parallelism 45
12.2.11. Example: Nested parallelism 46
12.2.12. OpenMP load balancing 46
13. Performance analysis tools 47
13.1. Intel performance analysis tools 47
13.2. Scalasca 47
13.2.1. Compilation of Scalasca 47
13.2.2. Usage 47
Further documentation 48

1. Introduction

Figure 1. Intel Xeon Phi coprocessor



This best practice guide provides information about Intel's MIC architecture and programming models for the Intel Xeon Phi coprocessor in order to enable programmers to achieve good performance of their applications. The guide covers a wide range of topics from the description of the hardware of the Intel Xeon Phi coprocessor through information about the basic programming models as well as information about porting programs up to tools and strategies how to analyze and improve the performance of applications.

This initial version of the guide contains contributions from CSC, KTH, LRZ, and NCSA. It also includes several informations from publicly available Intel documents and Intel webinars [11].

In 2013 the first books about programming the Intel Xeon Phi coprocessor [1] [2] [3] have been published. We also recommend a book about structured parallel programming [4]. Useful online documentation about the Intel Xeon Phi coprocessor can be found in Intel's developer zone for Xeon Phi Programming [6] and the Intel Many Integrated Core Architecture User Forum [7]. To get things going quickly have a look on the Intel Xeon Phi Coprocessor Developer's Quick Start Guide [15] and also on the paper [24].

Various experiences with application enabling for Intel Xeon Phi gained within PRACE on the EURORA-cluster at CINECA (Italy) in late 2013 can be found in whitepapers available online at [16].

2. Intel MIC architecture & system overview

2.1. The Intel MIC architecture

2.1.1. Intel Xeon Phi coprocessor architecture overview

The Intel Xeon Phi coprocessor consists of up to 61 cores connected by a high performance on-die bidirectional interconnect. The coprocessor runs a Linux operating system and supports all important Intel development tools, like C/C++ and Fortran compiler, MPI and OpenMP, high performance libraries like MKL, debugger and tracing tools like Intel VTune Amplifier XE. Traditional UNIX tools on the coprocessor are supported via BusyBox, which combines tiny versions of many common UNIX utilities into a single small executable. The coprocessor is connected to an Intel Xeon processor - the "host" - via the PCI Express (PICe) bus. The implementation of a

virtualized TCP/IP stack allows to access the coprocessor like a network node. Summarized information about the hardware architecture can be found in [17]. In the following we cite the most important properties of the MIC architecture from the System Software Developers Guide [18], which includes many details about the MIC architecture:

Core

- the processor core (scalar unit) is an in-order architecture (based on the Intel Pentium processor family)
- fetches and decodes instructions from *four hardware threads*
- supports a 64-bit execution environment, along with Intel Initial Many Core Instructions
- does not support any previous Intel SIMD extensions like MME, SSE, SSE2, SSE3, SSE4.1 SSE4.2 or AVX instructions
- new vector instructions provided by the Intel Xeon Phi coprocessor instruction set utilize a dedicated 512-bit wide vector floating-point unit (VPU) that is provided for each core
- high performance support for reciprocal, square root, power and exponent operations, scatter/gather and streaming store capabilities to achieve higher effective memory bandwidth
- can execute 2 instructs per cycle, one on the U-pipe and one on the V-pipe (not all instruction types can be executed by the V-pipe, e.g. vector instructions can only be executed on the U-pipe)
- contains the L1 Icache and Dcache
- each core is connected to a ring interconnect via the Core Ring Interface (CRI)

Vector Processing Unit (VPU)

- the VPU includes the EMU (Extended Math Unit) and executes 16 single-precision floating point, 16 32bit integer operations or 8 double-precision floating point operations per cycle. Each operation can be a fused multiply-add, giving 32 single-precision or 16 double-precision floating-point operations per cycle.
- contains the vector register file: 32 512-bit wide registers per thread context, each register can hold 16 singles or 8 doubles
- most vector instructions have a 4-clock latency with a 1 clock throughput

Core Ring Interface (CRI)

- hosts the L2 cache and the tag directory (TD)
- connects each core to an Intel Xeon Phi coprocessor Ring Stop (RS), which connects to the interprocessor core network

Ring

- includes component interfaces, ring stops, ring turns, addressing and flow control
- a Xeon Phi coprocessor has 2 of theses rings, one travelling each direction

SBOX

- Gen2 PCI Express client logic
- system interface to the host CPU or PCI Express switch
- DMA engine

GBOX

- coprocessor memory controller

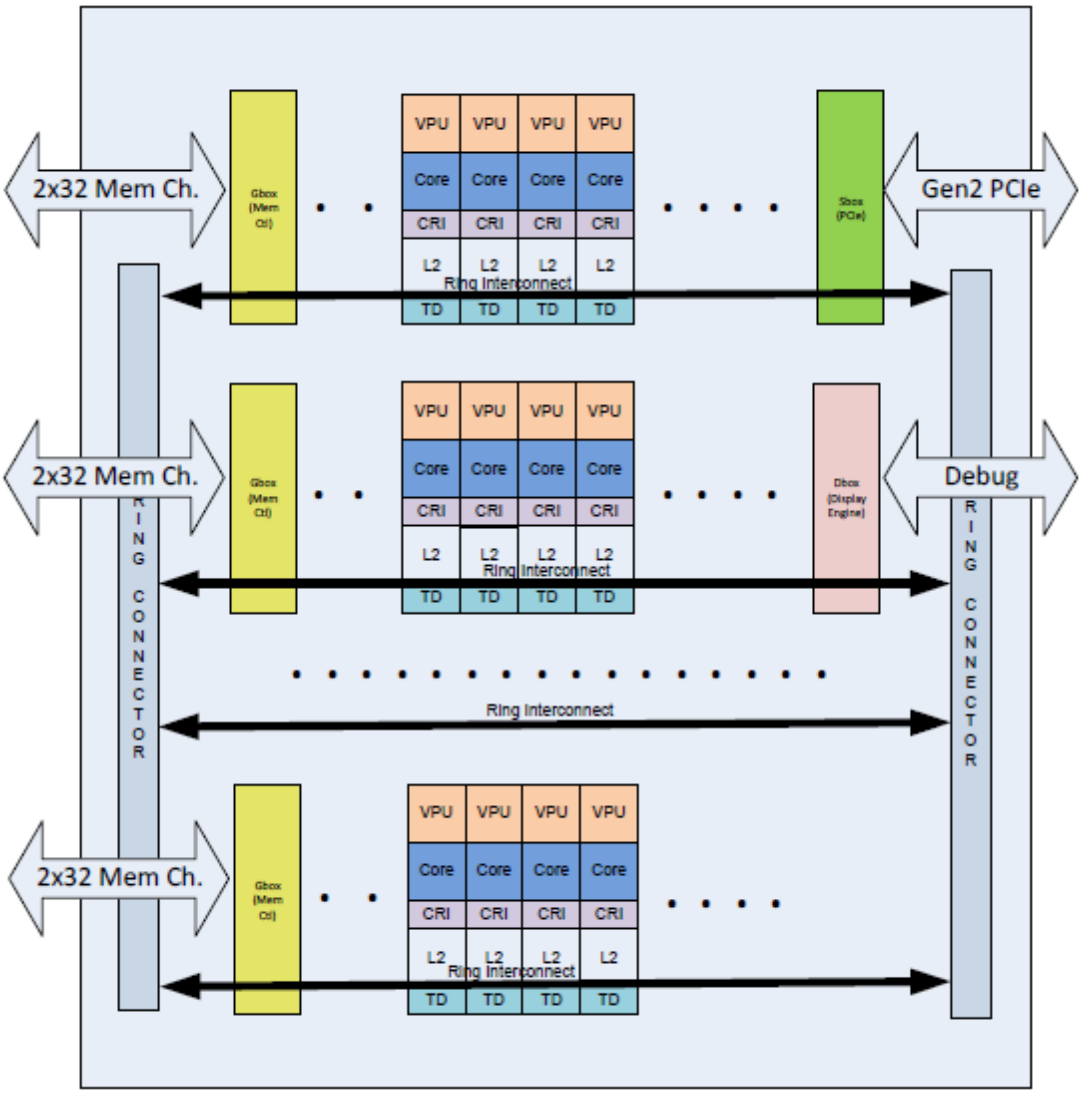
- consists of the FBOX (interface to the ring interconnect), the MBOX (request scheduler) and the PBOX (physical layer that interfaces with the GDDR devices
- There are 8 memory controllers supporting up to 16 GDDR5 channels. With a transfer speed of up to 5.5 GT/s a theoretical aggregated bandwidth of 352 GB/s is provided.

Performance Monitoring Unit (PMU)

- allows data to be collected from all units in the architecture
- does not implement some advanced features found in mainline IA cores (e.g. precise event-based sampling, etc.)

The following picture (from [18]) illustrates the building blocks of the architecture.

Figure 2. Intel MIC architecture overview



2.1.2. The cache hierarchy

Details about the L1 and L2 cache can be found in the System Software Developers Guide [18]. We only cite the most important features here.

The L1 cache has a 32 KB L1 instruction cache and 32 KB L1 data cache. Associativity is 8-way, with a cache line-size of 64 byte. It has a load-to-use latency of 1 cycle, which means that an integer value loaded from the L1 cache can be used in the next clock by an integer instruction. (Vector instructions have different latencies than integer instructions.)

The L2 cache is a unified cache which is inclusive of the L1 data and instruction caches. Each core contributes 512 KB of L2 to the total global shared L2 cache storage. If no cores share any data or code, then the effective total L2 size of the chip is up to 31 MB. On the other hand, if every core shares exactly the same code and data in perfect synchronization, then the effective total L2 size of the chip is only 512 KB. The actual size of the workload-perceived L2 storage is a function of the degree of code and data sharing among cores and thread.

Like for the L1 cache, associativity is 8-way, with a cache line-size of 64 byte. The raw latency is 11 clock cycles. It has a streaming hardware prefetcher and supports ECC correction.

The main properties of the L1 and L2 caches are summarized in the following table (from [18]):

Parameter	**L1**	**L2**
Coherence	MESI	MESI
Size	32 KB + 32 KB	512 KB
Associativity	8-way	8-way
Line Size	64 bytes	64 bytes
Banks	8	8
Access Time	1 cycle	11 cycles
Policy	pseudo LRU	pseudo LRU
Duty Cycle	1 per clock	1 per clock
Ports	Read or Write	Read or Write

2.2. Network configuration & system access

Details about the system startup and the network configuration can be found in [19] and in the documentation coming with the Intel Manycore Platform Software Stack (Intel MPSS) [10].

To start the Intel MPSS stack and initialize the Xeon Phi coprocessor the following command has to be executed as root or during host system start-up:

```
weinberg@knf1:~> sudo service mpss start
```

During start-up details are logged to `/var/log/messages`.

If MPSS with OFED support is needed, further the following commands have to be executed as root:

```
weinberg@knf1:~> sudo service openibd start
weinberg@knf1:~> sudo service opensmd start
weinberg@knf1:~> sudo service ofed-mic start
```

Per default IP addresses 172.31.1.254 , 172.31.2.254 , 172.31.3.254 etc. are then assigned to the attached Intel Xeon Phi coprocessors. The IP addresses of the attached coprocessors can be listed via the traditional `ifconfig` Linux program.

```
weinberg@knf1:~> /sbin/ifconfig
...
```

```
    mic0      Link encap:Ethernet  HWaddr 86:32:20:F3:1A:4A
              inet addr:172.31.1.254  Bcast:172.31.1.255  Mask:255.255.255.0
              inet6 addr: fe80::8432:20ff:fef3:1a4a/64 Scope:Link
              UP BROADCAST RUNNING  MTU:64512  Metric:1
              RX packets:46407 errors:0 dropped:0 overruns:0 frame:0
              TX packets:42100 errors:0 dropped:0 overruns:0 carrier:0
              collisions:0 txqueuelen:1000
              RX bytes:8186629 (7.8 Mb)  TX bytes:139125152 (132.6 Mb)

    mic1      Link encap:Ethernet  HWaddr FA:7A:0D:1E:65:B0
              inet addr:172.31.2.254  Bcast:172.31.2.255  Mask:255.255.255.0
              inet6 addr: fe80::f87a:dff:fe1e:65b0/64 Scope:Link
              UP BROADCAST RUNNING  MTU:64512  Metric:1
              RX packets:5097 errors:0 dropped:0 overruns:0 frame:0
              TX packets:4240 errors:0 dropped:0 overruns:0 carrier:0
              collisions:0 txqueuelen:1000
              RX bytes:522169 (509.9 Kb)  TX bytes:124671342 (118.8 Mb)
```

Further information can be obtained by running the `micinfo` program on the host. To get also PCIe related details the command has to be run with root privileges. Here is an example output for a C0 stepping Intel Xeon Phi prototype:

```
weinberg@knf1:~> sudo /opt/intel/mic/bin/micinfo
MicInfo Utility Log

Created Thu Dec  5 14:43:39 2013


        System Info
                HOST OS                 : Linux
                OS Version              : 3.0.13-0.27-default
                Driver Version          : 5889-16
                MPSS Version            : 2.1.5889-16
                Host Physical Memory    : 66056 MB

Device No: 0, Device Name: mic0

        Version
                Flash Version            : 2.1.02.0381
                SMC Boot Loader Version  : 1.8.4326
                uOS Version              : 2.6.38.8-g9b2c036
                Device Serial Number     : ADKC31202722

        Board
                Vendor ID                : 8086
                Device ID                : 225d
                Subsystem ID             : 3608
                Coprocessor Stepping ID  : 2
                PCIe Width               : x16
                PCIe Speed               : 5 GT/s
                PCIe Max payload size    : 256 bytes
                PCIe Max read req size   : 4096 bytes
                Coprocessor Model        : 0x01
```

```
                Coprocessor Model Ext    : 0x00
                Coprocessor Type         : 0x00
                Coprocessor Family       : 0x0b
                Coprocessor Family Ext   : 0x00
                Coprocessor Stepping     : C0
                Board SKU                : C0-3120P/3120A
                ECC Mode                 : Enabled
                SMC HW Revision          : Product 300W Active CS

        Cores
                Total No of Active Cores : 57
                Voltage                  : 1081000 uV
                Frequency                : 1100000 KHz

        Thermal
                Fan Speed Control        : On
                SMC Firmware Version     : 1.11.4404
                FSC Strap                : 14 MHz
                Fan RPM                  : 2700
                Fan PWM                  : 50
                Die Temp                 : 54 C

        GDDR
                GDDR Vendor              : Elpida
                GDDR Version             : 0x1
                GDDR Density             : 2048 Mb
                GDDR Size                : 5952 MB
                GDDR Technology          : GDDR5
                GDDR Speed               : 5.000000 GT/s
                GDDR Frequency           : 2500000 KHz
                GDDR Voltage             : 1000000 uV

Device No: 1, Device Name: mic1
...
```

Users can log in directly onto the Xeon Phi coprocessor via ssh.

```
weinberg@knf1:~> ssh mic0
[weinberg@knf1-mic0 weinberg]$ hostname
knf1-mic0
[weinberg@knf1-mic0 weinberg]$ cat /etc/issue
Intel MIC Platform Software Stack release 2.1
Kernel 2.6.38.8-g9b2c036 on an k1om
```

Per default the home directory on the coprocessor is `/home/username` .

Since the access to the coprocessor is ssh-key based users have to generate a private/public key pair via `ssh-keygen` before accessing the coprocessor for the first time.

After the keys have been generated, the following commands have to be executed as root to populate the filesystem image for the coprocessor on the host (`/opt/intel/mic/filesystem/mic0/home`) with the new keys. Since the coprocessor has to be restarted to copy the new image to the coprocessor, the following commands have to be used (preferrably only by the system administrator) with care.

```
  weinberg@knf1:~> sudo service mpss stop
```

```
weinberg@knf1:~> sudo micctrl --resetconfig
weinberg@knf1:~> sudo service mpss start
```

On production systems access to reserved cards might be realized by the job scheduler.

Informations how to set up and configure a cluster with hosts containing Intel Xeon Phi coprocessors, based on how Intel configured its own Endeavor cluster can be found in [20].

Since a Linux kernel is running on the coprocessor, further information about the cores, memory etc. can be obtained from the virtual Linux /proc or /sys filesystems:

```
[weinberg@knf1-mic0 weinberg]$ tail -n 25 /proc/cpuinfo
processor       : 227
vendor_id       : GenuineIntel
cpu family      : 11
model           : 1
model name      : 0b/01
stepping        : 2
cpu MHz         : 1100.000
cache size      : 512 KB
physical id     : 0
siblings        : 228
core id         : 56
cpu cores       : 57
apicid          : 227
initial apicid  : 227
fpu             : yes
fpu_exception   : yes
cpuid level     : 4
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic mtrr mca pat fxsr ht syscall
bogomips        : 2208.11
clflush size    : 64
cache_alignment : 64
address sizes   : 40 bits physical, 48 bits virtual
power management:
```

```
[weinberg@knf1-mic0 weinberg]$ head -5 /proc/meminfo
MemTotal:        5878664 kB
MemFree:         5638568 kB
Buffers:               0 kB
Cached:            76464 kB
SwapCached:            0 kB
```

To run MKL, OpenMP or MPI based programs on the coprocessor, some libraries (exact path and filenames may differ depending on the version) need to be copied to the coprocessor. On production systems the libraries might be installed or mounted on the coprocessor already. Root privileges are necessary for the destination directories given in the following example:

```
scp /opt/intel/composerxe/mkl/lib/mic/libmkl_intel_lp64.so root@mic0:/lib64/
```

```
scp /opt/intel/composerxe/mkl/lib/mic/libmkl_intel_thread.so root@mic0:/lib64/
scp /opt/intel/composerxe/mkl/lib/mic/libmkl_core.so root@mic0:/lib64/

scp  /opt/intel/composerxe/lib/mic/libiomp5.so root@mic0:/lib64/
scp  /opt/intel/composerxe/lib/mic/libimf.so root@mic0:/lib64/
scp  /opt/intel/composerxe/lib/mic/libsvml.so root@mic0:/lib64/
scp  /opt/intel/composerxe/lib/mic/libirng.so root@mic0:/lib64/
scp  /opt/intel/composerxe/lib/mic/libintlc.so.5 root@mic0:/lib64/

scp /opt/intel/mpi-rt/4.1.0/mic/lib/libmpi_mt.so.4 root@mic0:/lib64/
scp /opt/intel/mpi-rt/4.1.0/mic/lib/libmpigf.so.4 root@mic0:/lib64/
scp /opt/intel/impi/4.1.0.024/mic/lib/libmpi.so.4 root@mic0:/lib64/

scp /opt/intel/impi/4.1.0.024/mic/bin/mpiexec.hydra root@mic0:/bin
scp /opt/intel/impi/4.1.0.024/mic/bin/pmi_proxy root@mic0:/bin
```

3. Native compilation

The simplest model of running applications on the Intel Xeon Phi coprocessor is native mode. Detailed information about building a native application for Intel Xeon Phi coprocessors can be found in [26].

In native mode an application is compiled on the host using the compiler switch `-mmic` to generate code for the MIC architecture. The binary can then be copied to the coprocessor and has to be started there.

```
weinberg@knf1:~/c> . /opt/intel/composerxe/bin/compilervars.sh intel64
weinberg@knf1:~/c> icc -O3 -mmic  program.c -o program
weinberg@knf1:~/c> scp program mic0:
program                                       100%   10KB  10.2KB/s   00:00
weinberg@knf1:~/c> ssh mic0 ~/program
hello, world
```

To achieve good performance one should mind the following items.

- Data should be **aligned to 64 Bytes (512 Bits)** for the MIC architecture, in contrast to 32 Bytes (256 Bits) for AVX and 16 Bytes (128 Bits) for SSE.

- Due to the large SIMD width of 64 Bytes **vectorization is even more important for the MIC architecture than for Intel Xeon!** The MIC architecture offers new instructions like gather/scatter, fused multiply-add, masked vector instructions etc. which allow more loops to be parallelized on the coprocessor than on an Intel Xeon based host.

- Use pragmas like `#pragma ivdep`, `#pragma vector always`, `#pragma vector aligned`, `#pragma simd` etc. to achieve autovectorization. Autovectorization is enabled at default optimization level `-O2` . Requirements for vectorizable loops can be found in [43].

- Let the compiler generate vectorization reports using the compiler option `-vecreport2` to see if loops were vectorized for MIC (Message "*MIC* Loop was vectorized" etc). The options `-opt-report-phase hlo` (High Level Optimizer Report) or `-opt-report-phase ipo_inl` (Inlining report) may also be useful.

- Explicit vector programming is also possible via Intel Cilk Plus language extensions (C/C++ array notation, vector elemental functions, ...) or the new SIMD constructs from OpenMP 4.0 RC1.

- Vector elemental functions can be declared by using `__attributes__((vector)).` The compiler then generates a vectorized version of a scalar function which can be called from a vectorized loop.

- One can use intrinsics to have full control over the vector registers and the instruction set. Include `<immintrin.h>` for using intrinsics.

- Hardware prefetching from the L2 cache is enabled per default. In addition, software prefetching is on by default at compiler optimization level `-O2` and above. Since Intel Xeon Phi is an inorder architecture, care about prefetching is more important than on out-of-order architectures. The compiler prefetching can be influenced by setting the compiler switch `-opt-prefetch=n`. Manual prefetching can be done by using intrinsics (`_mm_prefetch()`) or pragmas (`#pragma prefetch var`).

4. Intel compiler's offload pragmas

One can simply add OpenMP-like pragmas to C/C++ or Fortran code to mark regions of code that should be offloaded to the Intel Xeon Phi Coprocessor and be run there. This approach is quite similar to the accelerator pragmas introduced by the PGI compiler, CAPS HMPP or OpenACC to offload code to GPGPUs. When the Intel compiler encounters an offload pragma, it generates code for both the coprocessor and the host. Code to transfer the data to the coprocessor is automatically created by the compiler, however the programmer can influence the data transfer by adding data clauses to the offload pragma. Details can be found under "Offload Using a Pragma" in the Intel compiler documentation [30].

4.1. Simple example

In the following we show a simple example how to offload a matrix-matrix computation to the coprocessor.

```
main(){

  double *a, *b, *c;
  int i,j,k, ok, n=100;

  // allocated memory on the heap aligned to 64 byte boundary
  ok = posix_memalign((void**)&a, 64, n*n*sizeof(double));
  ok = posix_memalign((void**)&b, 64, n*n*sizeof(double));
  ok = posix_memalign((void**)&c, 64, n*n*sizeof(double));

  // initialize matrices
  ...
  //offload code
#pragma offload target(mic) in(a,b:length(n*n)) inout(c:length(n*n))
  {
  //parallelize via OpenMP on MIC
#pragma omp parallel for
    for( i = 0; i < n; i++ ) {
      for( k = 0; k < n; k++ ) {
#pragma vector aligned
#pragma ivdep
 for( j = 0; j < n; j++ ) {
   //c[i][j] = c[i][j] + a[i][k]*b[k][j];
   c[i*n+j] = c[i*n+j] + a[i*n+k]*b[k*n+j];

        }
      }
    }
  }
}
```

This example (with quite bad performance) shows how to offload the matrix computation to the coprocessor using the `#pragma offload target(mic)`. One could also specify the specific coprocessor `num` in a system with multiple coprocessors by using `#pragma offload target(mic:num)`.

Since the matrices have been dynamically allocated using `posix_memalign()`, their sizes must be specified via the `length()` clause. Using `in`, `out` and `inout` one can specify which data has to be copied in which direction. It is recommended that for Intel Xeon Phi data is 64-byte aligned. `#pragma vector aligned` tells the compiler that all array data accessed in the loop is properly aligned. `#pragma ivdep` discards any data dependencies assumed by the compiler.

Offloading is enabled per default for the Intel compiler. Use `-no-offload` to disable the generation of offload code.

4.2. Obtaining informations about the offloading

Using the compiler option `-vec-report2` one can see which loops have been vectorized on the host and the MIC coprocessor:

```
weinberg@knf1:~/c> icc  -vec-report2   -openmp offload.c
offload.c(57): (col. 2) remark: loop was not vectorized: vectorization
                                     possible but seems inefficient.
...
offload.c(57): (col. 2) remark: *MIC* LOOP WAS VECTORIZED.
offload.c(54): (col. 7) remark: *MIC* loop was not vectorized: not inner loop.
offload.c(53): (col. 5) remark: *MIC* loop was not vectorized: not inner loop.
```

By setting the environment variable `OFFLOAD_REPORT` one can obtain information about performance and data transfers at runtime:

```
weinberg@knf1:~/c> export OFFLOAD_REPORT=2
weinberg@knf1:~/c> ./a.out
[Offload] [MIC 0] [File]            offload2.c
[Offload] [MIC 0] [Line]            50
[Offload] [MIC 0] [CPU Time]        12.853562 (seconds)
[Offload] [MIC 0] [CPU->MIC Data]   9830416 (bytes)
[Offload] [MIC 0] [MIC Time]        12.208636 (seconds)
[Offload] [MIC 0] [MIC->CPU Data]   3276816 (bytes)
```

If a function is called within the offloaded code block, this function has to be declared with `__attribute__((target(mic)))` .

For example one could put the matrix-matrix multiplication of the previous example into a subroutine and call that routine within an offloaded block region:

```
__attribute__((target(mic))) void mxm( int n,  double * restrict a,
                          double * restrict b, double *restrict c ){
  int i,j,k;
  for( i = 0; i < n; i++ ) {
  ...
}

main(){
```

```
...

#pragma offload target(mic) in(a,b:length(n*n)) inout(c:length(n*n))
  {
    mxm(n,a,b,c);
  }

}
```

Mind the C99 restrict keyword that specifies that the vectors do not overlap. (Compile with -std=c99)

4.3. Syntax of pragmas

The following offload pragmas are available (from [11]):

Pragma	Syntax	Semantic
	C/C++	
Offload pragma	`#pragma offload <clauses> <statement>`	Allow next statement to execute on coprocessor or host CPU
Variable/function offload properties	`_attribute__ ((target(mic)))`	Compile function for, or allocate variable on, both host CPU and coprocessor

Entire blocks of data/code defs	`#pragma offload_attribute(push, target(mic))`
	`...`
	`#pragma offload_attribute(pop)`
		Mark entire files or large blocks of code to compile for both host CPU and co-processor

	Fortran	
Offload directive	`!dir$ omp offload <clauses> <statement>`	Execute OpenMP parallel block on coprocessor
Variable/function offload properties	`!dir$ attributes offload:<mic> :: <ret-name> OR <var1,var2,…>`	Compile function or variable for CPU and coprocessor

Entire code blocks	`!dir$ offload begin <clauses>`
	`...`
	`!dir$ end offload`
		Mark entire files or large blocks of code to compile for both host CPU and co-processor

The following clauses can be used to control data transfers:

Clause	Syntax	Semantic
Multiple coprocessors	`target(mic[:unit])`	Select specific coprocessors
Inputs	`in(var-list modifiers)`	Copy from host to co-processor
Outputs	`out(var-list modifiers)`	Copy from coprocessor to host
Inputs & outputs	`inout(var-list modifiers)`	Copy host to coprocessor and back when offload completes
Non-copied data	`nocopy(var-list modifiers)`	Data is local to target

The following (optional) modifiers are specified:

Modifier	Syntax	Semantic
Specify copy length	`length(N)`	Copy N elements of pointer's type
Coprocessor memory allocation	`alloc_if ( bool )`	Allocate coprocessor space on this offload (default: TRUE)
Coprocessor memory release	`free_if ( bool )`	Free coprocessor space at the end of this offload (default: TRUE)
Control target data alignment	`align ( N bytes )`	Specify minimum memory alignment on coprocessor
Array partial allocation & variable relocation	`alloc ( array-slice ) into ( var-expr )`	Enables partial array allocation and data copy into other vars & ranges

4.4. Recommendations

4.4.1. Explicit worksharing

To explicitly share work between the coprocessor and the host one can use OpenMP sections to manually distribute the work. In the following example both the host and the coporcessor will run a matrix-matrix multiplication in parallel.

```
#pragma omp parallel
  {
#pragma omp sections
    {
#pragma omp section
      {
//section running on the coprocessor
#pragma offload target(mic) in(a,b:length(n*n)) inout(c:length(n*n))
 {
   mxm(n,a,b,c);
 }
      }
#pragma omp section
      {
//section running on the host
 mxm(n,d,e,f);
      }
    }

  }
```

4.4.2. Persistent data on the coprocessor

The main bottleneck of accelerator based programming are data transfers over the slow PCIe bus from the host to the accelerator and vice versa. To increase the performance one should minimize data transfers as much as possible and keep the data on the coprocessor between computations using the same data.

Defining the following macros

```
#define ALLOC   alloc_if(1)
#define FREE    free_if(1)
#define RETAIN  free_if(0)
#define REUSE   alloc_if(0)
```

one can simply use the following notation:

- to allocate data and keep it for the next offload

```
#pragma offload target(mic) in (p:length(l) ALLOC RETAIN)
```

- to reuse the data and still keep it on the coprocessor

```
#pragma offload target(mic) in (p:length(l) REUSE RETAIN)
```

- to reuse the data again and free the memory. (FREE is the default, and does not need to be explicitly specified)

```
#pragma offload target(mic) in (p:length(l) REUSE FREE)
```

More information can be found in the section "Managing Memory Allocation for Pointer Variables" under "Offload Using a Pragma" in the compiler documentation [30] .

4.4.3. Optimizing offloaded code

The implementation of the matrix-matrix multiplication given in Section 4.1 can be optimized by defining appropriate ROWCHUNK and COLCHUNK chunk sizes, rewriting the code with 6 nested loops (using OpenMP collapse for the 2 outermost loops) and some manual loop unrolling (thanks to A. Heinecke for input for this section).

```
#define ROWCHUNK 96
#define COLCHUNK 96

#pragma omp parallel for collapse(2) private(i,j,k)
    for(i = 0; i < n; i+=ROWCHUNK ) {
      for(j = 0; j < n; j+=ROWCHUNK ) {
 for(k = 0; k < n; k+=COLCHUNK ) {
   for (ii = i; ii < i+ROWCHUNK; ii+=6)  {
     for (kk = k; kk < k+COLCHUNK; kk++ ) {
#pragma ivdep
#pragma vector aligned
       for ( jj = j; jj < j+ROWCHUNK; jj++){
  c[(ii*n)+jj] += a[(ii*n)+kk]*b[kk*n+jj];
  c[((ii+1)*n)+jj] += a[((ii+1)*n)+kk]*b[kk*n+jj];
  c[((ii+2)*n)+jj] += a[((ii+2)*n)+kk]*b[kk*n+jj];
  c[((ii+3)*n)+jj] += a[((ii+3)*n)+kk]*b[kk*n+jj];
  c[((ii+4)*n)+jj] += a[((ii+4)*n)+kk]*b[kk*n+jj];
  c[((ii+5)*n)+jj] += a[((ii+5)*n)+kk]*b[kk*n+jj];
             }
           }
         }
       }
     }
    }
}
```

Using intrinsics with manual data prefetching and register blocking can still considerably increase the performance. Generally speaking, the programmer should try to get a suitable vectorization and write cache and register efficient

code, i.e. values stored in registers should be reused as often as possible in order to avoid cache and memory access. The tuning techniques for native implementations discussed in Section 3 also apply for offloaded code, of course.

4.5. Intel Cilk Plus parallel extensions

More complex data structures can be handled by Virtual Shared Memory. In this case the same virtual address space is used on both the host and the coprocessor, enabling a seamless sharing of data. Virtual shared data is specified using the `_Cilk_shared` allocation specifier. This model is integrated in Intel Cilk Plus parallel extensions and is only available in C/C++. There are also Cilk functions to specify offloading of functions and `_Cilk_for` loops. More information on Intel Cilk Plus can be found online under [12].

5. OpenMP and hybrid

5.1. OpenMP

5.1.1. OpenMP basics

OpenMP parallelization on an Intel Xeon + Xeon Phi coprocessor machine can be applied both on the host and on the device with the `-openmp` compiler option. It can run on the Xeon host, natively on the Xeon Phi coprocessor and also in different offload schemes. It is introduced with regular pragma statements in the code for either case.

In applications that run both on the host and on the Xeon Phi coprocessor OpenMP threads do not interfere with each other and offload statements are executed on the device based on the available resources. If there are no free threads on the Xeon Phi coprocessor or less than requested, the parallel region will be executed on the host. Offload pragma statements and OpenMP pragma statements are independent from each other and must both be present in the code. Within the latter construct apply usual semantics of shared and private data.

There are 16 threads available on every Xeon host CPU and 4 times the number of cores threads on every Xeon Phi coprocessor. For offload schemes the maximal amount of threads that can be used on the device is 4 times the number of cores minus one, because one core is reserved for the OS and its services. Offload to the Xeon Phi coprocessor can be done at any time by multiple host CPUs until filling the resources available. If there are no free threads, the task meant to be offloaded may be done on the host.

5.1.2. Threading and affinity

The Xeon Phi coprocessor supports 4 threads per core. Unlike some CPU-intensive HPC applications that are run on Xeon architecture, which do not benefit from hyperthreading, applications run on Xeon Phi coprocessors do and using more than one thread per core is recommended.

The most important considerations for OpenMP threading and affinity are the total number of threads that should be utilized and the scheme for binding threads to processor cores. These are controlled with the environmental variables `OMP_NUM_THREADS` and `KMP_AFFINITY`.

The default settings are as follows:

	`OMP_NUM_THREADS`
OpenMP on host without HT	1 x ncore-host
OpenMP on host with HT	2 x ncore-host
OpenMP on Xeon Phi in native mode	4 x ncore-phi
OpenMP on Xeon Phi in offload mode	4 x (ncore-phi - 1)

For host-only and native-only execution the number of threads and all other environmental variables are set as usual:

```
% export OMP_NUM_THREADS=16
```

```
% export OMP_NUM_THREADS=240
% export KMP_AFFINITY=compact/scatter/balanced
```

Setting affinity to “compact” will place OpenMP threads by filling cores one by one, while “scatter” will place one thread in every core until reaching the total number of cores and then continue with the first core. When using “balanced”, it is similar to “scatter”, but threads with adjacent numbers will be placed on the same core. “Balanced” is only available for the Xeon Phi coprocessor. Another useful option is the verbosity modifier:

```
% export KMP_AFFINITY=verbose,balanced
```

With compiler version 13.1.0 and newer one can use the `KMP_PLACE_THREADS` variable to point out the topology of the system to the OpenMP runtime, for example:

```
% export OMP_NUM_THREADS=180
% export KMP_PLACE_THREADS=60c,3t
```

meaning that 60 cores and 3 threads per core should be used. Still one should use the `KMP_AFFINITY` variable to bind the threads to the cores.

If OpenMP regions exist on the host and on the part of the code offloaded to the Phi, two separate OpenMP runtimes exist. Environment variables for controlling OpenMP behavior are to be set for both runtimes, for example the `KMP_AFFINITY` variable which can be used to assign a particular thread to a particular physical node. For Phi it can be done like this:

```
$ export MIC_ENV_PREFIX=MIC
# specify affinity for all cards
$ export MIC_KMP_AFFINITY=...
# specify number of threads for all cards
$ export MIC_OMP_NUM_THREADS=120
# specify the number of threads for card #2
$ export MIC_2_OMP_NUM_THREADS=200
# specify the number of threads and affinity for card #3
$ export MIC_3_ENV=”OMP_NUM_THREADS=60 | KMP_AFFINITY=balanced”
```

If `MIC_ENV_PREFIX` is not set and `KMP_AFFINITY` is set to “balanced” it will be ignored by the runtime.

One can also use special API calls to set the environment for the coprocessor only, e.g.

```
omp_set_num_threads_target()
omp_set_nested_target()
```

5.1.3. Loop scheduling

OpenMP accepts four different kinds of loop scheduling - static, dynamic, guided and auto. In this way the amount of iterations done by different threads can be controlled. The `schedule` clause can be used to set the loop scheduling at compile time. Another way to control this feature is to specify `schedule(runtime)` in your code and select the loop scheduling at runtime through setting the `OMP_SCHEDULE` environment variable.

5.1.4. Scalability improvement

If the amount of work that should be done by each thread is non-trivial and consists of nested for-loops, one might use the `collapse()` directive to specify how many for-loops are associated with the OpenMP loop construct. This often improves scalability of OpenMP applications (see Section 4.4.3).

Another way to improve scalability is to reduce barrier synchronization overheads by using the `nowait` directive. The effect of it is that the threads will not synchronize after they have completed their individual pieces of work.

This approach is applicable combined with static loop scheduling because all threads will execute the same amount of iterations in each loop but is also a potential threat on the correctness of the code.

5.2. Hybrid OpenMP/MPI

5.2.1. Programming models

For hybrid OpenMP/MPI programming there are two major approaches: an MPI offload approach, where MPI ranks reside on the host CPU and work is offloaded to the Xeon Phi coprocessor and a symmetric approach in which MPI ranks reside both on the CPU and on the Xeon Phi. An MPI program can be structured using either model.

When assigning MPI ranks, one should take into account that there is a data transfer overhead over the PCIe, so minimizing the communication from and to the Xeon Phi is a good idea. Another consideration is that there is limited amount of memory on the coprocessor which favors the shared memory parallelism ideology so placing 120 MPI ranks on the coprocessor each of which starts 2 threads might be less effective than placing less ranks but allowing them to start more threads. Note that MPI directives cannot be called within a pragma offload statement.

5.2.2. Threading of the MPI ranks

For hybrid OpenMP/MPI applications use the thread safe version of the Intel MPI Library by using the `-mt_mpi` compiler driver option. A desired process pinning scheme can be set with the `I_MPI_PIN_DOMAIN` environment variable. It is recommended to use the following setting:

```
$ export I_MPI_PIN_DOMAIN=omp
```

By setting this to omp, one sets the process pinning domain size to be to `OMP_NUM_THREADS`. In this way, every MPI process is able to create `OMP_NUM_THREADS` number of threads that will run within the corresponding domain. If this variable is not set, each process will create a number of threads per MPI process equal to the number of cores, because it will be treated as a separate domain.

Further, to pin OpenMP threads within a particular domain, one could use the `KMP_AFFINITY` environment variable.

6. MPI

Details about using the Intel MPI library on Xeon Phi coprocessor systems can be found in [21].

6.1. Setting up the MPI environment

The following commands have to be executed to set up the MPI environment:

```
# copy MPI libraries and binaries to the card (as root)
# only copying really necessary files saves memory

scp /opt/intel/impi/4.1.0.024/mic/lib/* mic0:/lib
scp /opt/intel/impi/4.1.0.024/mic/bin/* mic0:/bin

# setup Intel compiler variables
. /opt/intel/composerxe/bin/compilervars.sh intel64

# setup Intel MPI variables
. /opt/intel/impi/4.1.0.024/bin64/mpivars.sh
```

The following network fabrics are available for the Intel Xeon Phi coprocessor:

Fabric Name	Description
shm	Shared-memory
tcp	TCP/IP-capable network fabrics, such as Ethernet and InfiniBand (through IPoIB)
ofa	OFA-capable network fabric including InfiniBand (through OFED verbs)
dapl	DAPL–capable network fabrics, such as InfiniBand, iWarp, Dolphin, and XPMEM (through DAPL)

The Intel MPI library tries to automatically use the best available network fabric detected (usually shm for intra-node communication and InfiniBand (dapl, ofa) for inter-node communication).

The default can be changed by setting the `I_MPI_FABRICS` environment variable to `I_MPI_FABRICS=<fabric>` or `I_MPI_FABRICS=<intra-node fabric>:<inter-nodes fabric>`. The availability is checked in the following order: shm:dapl, shm:ofa, shm:tcp.

6.2. MPI programming models

Intel MPI for the Xeon Phi coprocessors offers various MPI programming models:

Symmetric model The MPI ranks reside on both the host and the coprocessor. Most general MPI case.

Coprocessor-only model All MPI ranks reside only on the coprocessors.

Host-only model All MPI ranks reside on the host. The coprocessors can be used by using offload pragmas. (Using MPI calls inside offloaded code is not supported.)

6.2.1. Coprocessor-only model

To build and run an application in coprocessor-only mode, the following commands have to be executed:

```
# compile the program for the coprocessor (-mmic)

mpiicc -mmic -o test.MIC test.c

#copy the executable to the coprocessor
scp test.MIC mic0:/tmp

#set the I_MPI_MIC variable
export I_MPI_MIC=1

#launch MPI jobs on the coprocessor mic0 from the host
#(alternatively one can login to the coprocessor and run mpirun there)
mpirun -host mic0  -n 2 /tmp/test.MIC
```

6.2.2. Symmetric model

To build and run an application in symmetric mode, the following commands have to be executed:

```
# compile the program for the coprocessor (-mmic)
mpiicc -mmic -o test.MIC test.c
```

```
# compile the program for the host
mpiicc -mmic -o test test.c


#copy the executable to the coprocessor
scp test.MIC mic0:/tmp/test.MIC

#set the I_MPI_MIC variable
export I_MPI_MIC=1


#launch MPI jobs on the host knf1 and on the coprocessor mic0
mpirun -host knf1 -n 1 ./test : -n 1  -host mic0 /tmp/test.MIC
```

6.2.3. Host-only model

To build and run an application in host-only mode, the following commands have to be executed:

```
# compile the program for the host,
# mind that offloading is enabled per default
mpiicc -o test test.c

# launch MPI jobs on the host knf1, the MPI process will offload code
# for acceleration
mpirun -host knf1 -n 1 ./test
```

6.3. Simplifying launching of MPI jobs

Instead of specifying the hosts and coprocessors via `-n hostname` one can also put the names into a hostfile and launch the jobs via

```
mpirun -f hostfile -n 4 ./test
```

Mind that the executable must have the same name on the hosts and the coprocessors in this case.

If one sets

```
export I_MPI_POSTFIX=.mic
```

the `.mix` postfix is automatically added to the executable name by mpirun, so in the case of the example above `test` is launched on the host and `test.mic` on the coprocessors. It is also possible to specify a prefix using

```
export I_MPI_PREFIX=./MIC/
```

In this case `./MIC/test` will be launched on the coprocessor. This is specially useful if the host and the coprocessors share the same NFS filesystem.

7. Intel MKL (Math Kernel Library)

The Intel Xeon Phi coprocessor is supported since MKL 11.0. Details on using MKL with Intel Xeon Phi co-processors can be found in [27], [28] and [29]. Also the MKL developer zone [8] contains useful information. All functions can be used on the Xeon Phi, however the optimization level for wider 512-bit SIMD instructions differs.

As of Intel MKL 11.0 Update 2 the following functions are highly optimized for the Intel Xeon Phi coprocessor:

- BLAS Level 3, and much of Level 1 & 2

- Sparse BLAS: ?CSRMV, ?CSRMM
- Some important LAPACK routines (LU, QR, Cholesky)
- Fast Fourier Transformations
- Vector Math Library
- Random number generators in the Vector Statistical Library

Intel plans to optimize a wider range of functions in future MKL releases.

7.1. MKL usage modes

The following 3 usage models of MKL are available for the Xeon Phi:

1. Automatic Offload
2. Compiler Assisted Offload
3. Native Execution

7.1.1. Automatic Offload (AO)

In the case of automatic offload the user does not have to change the code at all. For automatic offload enabled functions the runtime may automatically download data to the Xeon Phi coprocessor and execute (all or part of) the computations there. The data transfer and the execution management is completely automatic and transparent for the user.

As of Intel MKL 11.0.2 only the following functions are enabled for automatic offload:

- Level-3 BLAS functions
  - ?GEMM (for m,n > 2048, k > 256)
  - ?TRSM (for M,N > 3072)
  - ?TRMM (for M,N > 3072)
  - ?SYMM (for M,N > 2048)
- LAPACK functions
  - LU (M,N > 8192)
  - QR
  - Cholesky

In the above list also the matrix sizes for which MKL decides to offload the computation are given in brackets.

To enable automatic offload either the function `mkl_mic_enable()` has to be called within the source code or the environment variable `MKL_MIC_ENABLE=1` has to be set. If no Xeon Phi coprocessor is detected the application runs on the host without penalty.

To build a program for automatic offload, the same way of building code as on the Xeon host is used:

```
icc -O3 -mkl file.c -o file
```

By default, the MKL library decides when to offload and also tries to determine the optimal work division between the host and the targets (MKL can take advantage of multiple coprocessors). In case of the BLAS routines the user can specify the work division between the host and the coprocessor by calling the routine

```
mkl_mic_set_Workdivision(MKL_TARGET_MIC,0,0.5)
```

or by setting the environment variable

```
MKL_MIC_0_WORKDIVISION=0.5
```

Both examples specify to offload 50% of computation only to the 1st card (card #0).

7.1.2. Compiler Assisted Offload (CAO)

In this mode of MKL the offloading is explicitly controlled by compiler pragmas or directives. In contrast to the automatic offload mode, all MKL function can be offloaded in CAO-mode.

A big advantage of this mode is that it allows for data persistence on the device.

For Intel compilers it is possible to use AO and CAO in the same program, however the work division must be explicitly set for AO in this case. Otherwise, all MKL AO calls are executed on the host.

MKL functions are offloaded in the same way as any other offloaded function (see section Section 4). An example for offloading MKL's sgemm routine looks as follows:

```
#pragma offload target(mic) \
  in(transa, transb, N, alpha, beta) \
  in(A:length(N*N)) \
  in(B:length(N*N)) \
  in(C:length(N*N)) \
  out(C:length(N*N) alloc_if(0)) {

     sgemm(&transa, &transb, &N, &N, &N, &alpha, A, &N, B, &N, &beta, C, &N);

  }
```

To build a program for compiler assisted offload, the following command is recommended by Intel:

```
icc –O3 -openmp -mkl \
–offload-option,mic,ld, “-L$MKLROOT/lib/mic -Wl,\
--start-group -lmkl_intel_lp64 -lmkl_intel_thread \
-lmkl_core -Wl,--end-group” file.c –o file
```

To avoid using the OS core, it is recommended to use the following environment setting (in case of a 61-core coprocessor):

```
MIC_KMP_AFFINITY=explicit,granularity=fine,proclist=[1-240:1]
```

Setting larger pages by the environment setting `MIC_USE_2MB_BUFFERS=16K` usually increases performance. It is also recommended to exploit data persistence with CAO.

7.1.3. Native Execution

In this mode of MKL the Intel Xeon Phi coprocessor is used as an independent compute node.

To build a program for native mode, the following compiler settings should be used:

```
icc -O3 -mkl -mmic file.c -o file
```

The binary must then be manually copied to the coprocessor via ssh and directly started on the coprocessor.

7.2. Example code

Example code can be found under `$MKLROOT/examples/mic_ao` and `$MKLROOT/examples/mic_offload` .

7.3. Intel Math Kernel Library Link Line Advisor

To determine the appropriate link line for MKL the Intel Math Kernel Library Link Line Advisor available under [9] has been extended to include support for the Intel Xeon Phi specific options.

8. TBB: Intel Threading Building Blocks

The Intel TBB library is a template based runtime library for C++ code using threads that allows us to fully utilize the scaling capabilities within our code by increasing the number of threads and supporting task oriented load balancing.

Intel TBB is open source and available on many different platforms with most operating systems and processors. It is already popular in the C++ community. You should be able to use it with any compiler supporting ISO C++. So this is one of the advantages of Intel TBB when you intend to keep your code as easily portable as possible.

Typically as a rule of thumb an application must scale well past one hundred threads on Intel Xeon processors to profit from the possible higher parallel performance offered with e.g. the Intel Xeon Phi coprocessor. To check if the scaling would profit from utilising the highly parallel capabilities of the MIC architecture, you should start to create a simple performance graph with a varying number of threads (from one up to the number of cores).

From a programming standpoint we treat the coprocessor as a 64-bit x86 SMP-on-a-chip with an high-speed bi-directional ring interconnect, (up to) four hardware threads per core and 512-bit SIMD instructions. With the available number of cores we have easily 200 hardware threads at hand on a single coprocessor. The multi-threading on each core is primarily used to hide latencies that come implicitly with an in-order microarchitecture. Unlike hyper-threading these hardware threads cannot be switched off and should never be ignored. Generally it should be impossible for a single thread per core to approach the memory or floating point capability limit. Highly tuned codesnippets may reach saturation already at two threads, but in general a minimum of three or four active threads per cores will be needed. This is one of the reasons why the number of threads per core should be parameterized as well as the number of cores. The other reason is of course to be future compatible.

TBB offers programming methods that support creating this many threads in a program. In the easiest way the one main production loop is transformed by adding a single directive or pragma enabling the code for many threads. The chunk size used is chosen automatically.

The new Intel Cilk Plus which offers support for a simpler set of tasking capabilities fully interoperates with Intel TBB. Apart from that Intel Cilk Plus also supports vectorization. So shared memory programmers have Intel TBB and Intel Cilk Plus to assist them with built-in tasking models. Intel Cilk Plus extends Intel TBB to offer C programmers a solution as well as help with vectorization in C and C++ programs.

Intel TBB itself does not offer any explicit vectorization support. However it does not interfere with any vectorization solution either.

In relevance to the Intel Xeon Phi coprocessor TBB is just one available runtime-based parallel programming model alongside OpenMP, Intel Cilk Plus and pthreads that are also already available on the host system. Any code running natively on the coprocessor can put them to use just like it would on the host with the only difference being the larger number of threads.

8.1. Advantages of TBB

There exists a variety of approaches to parallel programming, but there are several advantages to using Intel TBB when writing scalable applications:

- TBB relies on generic programming: Writing the best possible algorithms with the fewest possible constraints enables to deliver high performance algorithms which can be applied in a broader context. Other more traditional libraries specify interfaces in terms of particular types or base classes. Intel TBB specifies the requirements on the types instead and in this way keeps the algorithms themselves generic and easily adaptable to different data representations.

- It is easy to start: You don't have to be a threading expert to leverage multi-core performance with the help of TBB. Normally you can successfully thread some programs just by adding a single directive or pragma to the main production loop.

- It obeys to logical parallelism: Since with TBB you specify tasks instead of threads, you automatically produce more portable code which emphasizes scalable, data parallel programming. You are not bound to platform-dependent threading primitives; most threading packages require you to directly code on low-level constructs close to the hardware. Direct programming on raw thread level is error-prone, tedious and typically hard work since it forces you to efficiently map logical tasks into threads and it is not always leading to the desired results. With the higher level of data-parallel programming on the other hand, where you have multiple threads working on different parts of a collection, performance continues to increase as you add more cores since for a larger number of processors the collections are just divided into smaller chunks. This is a great feature when it comes to portability.

- TBB is compatible with other programming models: Since the library is not designed to address all kinds of threading problems, it can coexist seamlessly with other threading packages.

- The template-based approach allows Intel TBB to make no excuses for performance. Other general-purpose threading packages tend to be low-level tools that are still far from the actual solution, while at the same time supporting many different kinds of threading. In TBB every template solves a computationally intensive problem in a generic, simple way instead.

All of these advantages make TBB popular and easily portable while at the same time facilitating data parallel programming.

Further advanced concepts in TBB that are not MIC specific can be found in the Intel TBB User Guide or in the Reference Manual, both available under [14].

8.2. Using TBB natively

Code that runs natively on the Intel Xeon Phi coprocessor can apply the TBB parallel programming model just as they would on the host, with no unusual complications beyond the larger number of threads.

In order to initialize your compiler environment variables needed to set up TBB correctly, typically the `/opt/intel/composerxe/tbb/bin/tbbvars.csh` or `tbbvars.sh` script with `intel64` as the argument is called by the `/opt/intel/composerxe/bin/compilervars.csh` or `compilervars.sh` script with `intel64` as argument. (e.g. `source /opt/intel/composerxe/bin/compilervars.sh intel64`)

Normally there is no need to call the `tbbvars` script directly and it is not advisable either since the `compilervars` script also calls other subscripts taking i.e. care of the debugger or Intel MKL and running the subscripts out of order might result in unpredictable behavior.

A minimal C++ TBB example looks as follows:

```
#include “tbb/task_scheduler_init.h”
#include “tbb/parallel_for.h”
#include "tbb/blocked_range.h"

using namespace tbb;

int main() {

task_scheduler_init init;
```

```
return 0;
}
```

The `using` directive imports the namespace `tbb` where all of the library's classes and functions are found. The namespace is explicit in the first mention of a component, but implicit afterwards. So with the `using namespace` statement present you can use the library component identifiers without having to write out the namespace prefix `tbb` before each of them.

The task scheduler is initialized by instantiating a `task_scheduler_init` object in the main function. The definition for the `task_scheduler_init` class is included from the corresponding header file. Actually any thread using one of the provided TBB template algorithms must have such an initialized `task_scheduler_init` object. The default constructor for the `task_scheduler_init` object informs the task scheduler that the thread is participating in task execution, and the destructor informs the scheduler that the thread no longer needs the scheduler. With the newer versions of Intel TBB as used in a MIC environment the task scheduler is automatically initialized, so there is no need to explicitely initialize it if you don't need to have control over when the task scheduler is constructed or destroyed. When initializing it you also have the further possibility to tell the task scheduler explicitly how many worker threads there are to be used and what their stack size would be.

In the simplest form scalable parallelism can be achieved by parallelizing a loop of iterations that can each run independently from each other.

The `parallel_for` template function replaces a serial loop where it is safe to process each element concurrently.

A typical example would be to apply a function `Foo` on all elements of an array over the iterations space of type `size_t` going from 0 to n-1:

```
void SerialApplyFoo( float a[], size_t n ) {
for( size_t i=0; i!=n; ++i )
Foo(a[i]);
}
```

becomes

```
void ParallelApplyFoo( float a[], size_t n) {
parallel_for(size_t(0), n, [=](size_t i) {Foo(a[i]);});
}
```

This is the TBB short form of a `parallel_for` over a loop based on a one-dimensional iteration space consisting of a consecutive range of integers (which is one of the most common cases). The expression `parallel_for(first,last,step,f)` is synonymous to `for(auto i=first; i!=last; i+=step) f(i)` except that each `f(i)` can be evaluated in parallel if resources permit. The omitted step parameter is optional. The short form implicitly uses automatic chunking.

The long form would be:

```
void ParallelApplyFoo( float* a, size_t n ) {
parallel_for( blocked_range<size_t>(0,n),
[=](const blocked_range<size_t>& r) {
for(size_t i=r.begin(); i!=r.end(); ++i)
Foo(a[i]);
});
}
```

Here the key feature of the TBB library is more clearly revealed. The template function `tbb::parallel_for` breaks the iteration space into chunks, and runs each chunk on a separate thread. The first parameter of tem-

plate function call `parallel_for` is a `blocked_range` object that describes the entire iteration space from 0 to n-1. The `parallel_for` divides the iteration space into subspaces for each of the over 200 hardware threads. `blocked_range<T>`is a template class provided by the TBB library describing a one-dimensional iteration space over type `T`. The `parallel_for` class works just as well with other kinds of iteration spaces. The library provides `blocked_range2d` for two-dimensional spaces. There exists also the possibility to define own spaces. The general constructor of the `blocked_range` template class is `blocked_range<T>(begin,end,grainsize)`. The `T` specifies the value type. `begin` represents the lower bound of the half-open range interval `[begin,end)` representing the iteration space. `end` represents the excluded upper bound of this range. The `grainsize` is the approximate number of elements per sub-range. The default `grainsize` is 1.

A parallel loop construct introduces overhead cost for every chunk of work that it schedules. The MIC adapted Intel TBB library chooses chunk sizes automatically, depending upon load balancing needs. The heuristic normally works well with the default `grainsize`. It attempts to limit overhead cost while still providing ample opportunities for load balancing. For most use cases automatic chunking is the recommended choice. There might be situations though where controlling the chunk size more precisely might yield better performance.

When compiling programs that employ TBB constructs, be sure to link in the Intel TBB shared library with `–ltbb`. If you don't undefined references will occur.

```
icc -mmic –ltbb foo.cpp
```

Afterwards you can use `scp` to upload the binary and any shared libraries required by your application to the coprocessor. On the coprocessor you can then export the library path and run the application.

8.3. Offloading TBB

The Intel TBB header files are not available on the Intel MIC target environment by default (the same is also true for Intel Cilk Plus). To make them available on the coprocessor the header files have to be wrapped with `#pragma offload` directives as demonstrated in the example below:

```
#pragma offload_attribute (push,target(mic))
#include “tbb/task_scheduler_init.h”
#include “tbb/parallel_for.h”
#include "tbb/blocked_range.h"
#pragma offload_attribute (pop)
```

Functions called from within the offloaded construct and global data required on the Intel Xeon Phi coprocessor should be appended by the special function attribute `__attribute__((target(mic)))`.

Codes using Intel TBB with an offload should be compiled with `-tbb` flag instead of `-ltbb`.

8.4. Examples

The TBB library provides support for parallel algorithms, concurrent containers, tasks, synchronization, memory allocation and thread control. Amongst it, the templates cover well-known parallel-programming patterns like parallel loops or reduction operations, task-based constructs or pipelines. An easy and flexible formulation of parallel computations is possible. The programmer's task is mainly to expose a high degree of parallelism to the processor, to allow or support the vectorization of calculations, and to maximize the cache-locality of algorithms. Principles of vectorization and striving for implementations with high cache-locality are general topics of performance optimization.

8.4.1. Exposing parallelism to the processor

The concrete parallelization of a certain algorithm is of course dependent on the specific floating-point operations used, its memory access pattern, the possible decoupling into parallel operations resp. vice-versa the dependencies

between the calculations. However, some estimation can be made what degree the needed parallelism should have in order to achieve efficient execution on the coprocessor.

Xeon Phi processors can execute 4 hyperthreads on each core. That results in 240 threads active at the same time on the 60 cores of the coprocessor. A rule of thumb is that one should have about 10 computational tasks per thread in order to compensate delays from load imbalances. That results in 2400 parallel tasks. Efficient calculations have to apply the vector units that can process 16 single precision floating-point numbers in one instruction. Therefore, we would need several ten thousand operations that can be executed independently from each other in order to keep the processor busy.

On the other hand, good load balancing or the placement of several MPI tasks on the Xeon Phi can lower the needed degree of parallelism. Considerations also have to be made if it would really increase the performance to use all 4 hyperthreads of a core. Those hyperthreads share certain hardware resources, and depending on the algorithm, competition for the resources can occur causing an increase of the overall execution speed.

8.4.1.1. Parallelisation with the task construct

This example shows how one can implement a multi-threaded, task-oriented algorithm. Nevertheless, there is no need to deal with the subtleties of thread creation and their control for the implementer of the algorithm.

The class `tbb:task` represents a low-level task with low overhead. It will be used by higher-level constructs like `parallel_for` or `task_group`.

Our example shows the use of task groups in a walkthrough through a binary tree. The sum of values stored in each node will be calculated during this walkthrough.

The tree nodes are defined as follows:

```
struct tree {
    struct tree *l, *r;  // left and right subtree
    long depth;          // depth of the tree starting from here
    float v;             // an important value
};
```

The serial implementation could be written as:

```
float tree_sum( struct tree *r )
{
    float mysum = r->v;

    if ( root->l != NULL ) mysum += tree_sum( r->l );
    if ( root->r != NULL ) mysum += tree_sum( r->r );

    return mysum;
}
```

A TTB task that performs the same computation is given with the following class `ParallelSumTask`. It must contain a method `exec()`. This method does either a serial computation of the value sum if its subtree has a depth lower than 10 levels or creates two subtasks that calculate the value sums of the left resp. right child tree. These tasks will be added to the task list of the active scheduler. After their termination will the current task add the value sums of both child trees and their own value and terminate itself.

```
class ParallelSumTask : public tbb::task {
    float       *mysum;
    struct tree *myr;

public:
```

```
    ParallelSumTask( TreeNode *r, float *sum ) : myr(r), mysum(sum) {}

    task *execute() {
        if( root->depth < 10 ) {
            *mysum = tree_sum(myr);
        }
        else {
            float sum_l, sum_r;
            tbb::task_list tlist;

            // Create subtasks for processing of left and right subtree.
            int count = 1;
            if( myr->l ) {
                ++count;
                tlist.push_back( *new(allocate_child())
                                  ParallelSumTask(myr->l, &sum_l));
            }
            if( myr->r ) {
                ++count;
                list.push_back( *new(allocate_child())
                                    ParallelSumTask(myr->r, &sum_r));
            }
            // Start task processing
            set_ref_count(count);
            spawn_and_wait_for_all(list);

            // Add own value and sums of left and right subtree.
            *mysum = myr->v;
            if( myr->l ) *mysum += sum_l;
            if( myr->r ) *mysum += sum_r;
        }
        return NULL;
    }
};
```

The calculation of the value sum could be started in the following way:

```
float value_sum;
struct tree *tree_root;

// ... Initialize tree ...

// Initialize a task scheduler with the maximum number of threads.
tbb::task_scheduler_init init( 240 );
// Perform the calculation.
ParallelSumTask *a = new(tbb::task::allocate_root())
                           ParallelSumTask(tree_root, &value_sum);
tbb::task::spawn_root_and_wait*(a);
// The sum of all values stored in the tree nodes
// has been calculated in variable "value_sum"
```

8.4.1.2. Loop parallelisation

TBB provides several algorithms that can be used for the parallelisation of loops, again without forcing the implementer do deal with the details of low-level threads. The following example provides the details how a loop can be parallelized with the `parallel_for` template. The implemented algorithm shall calculate the sum of the vector elements.

The serial implementation could be written as follows:

```
float serial_vecsum( float *vec, int len) {
   float sum = 0;

   for (int i = 0; i < len; ++i)
      sum += vec[i];
   return sum;
}
```

The parallel algorithm shall compute the partial sums of partial vectors. The overall sum of all partial vectors will be computed as sum of these partial sums.

The TBB template `parallel_reduce` provides the basic implementation of subdividing a range to iterate over into several subranges. The subranges will be assigned to different threads that perform the iterations on them. The program has to provide a "loop body" that that will be applied to each array element as well as a function that processes the results of the iterations over the subranges of the vector.

These functionality for the vector summation will be provided in the class `adder`. The transformation function working on the vector elements has to be provided as `operator()` method. The function that reduces the results of the parallel work on different subvectors has to be implemented as method `join()`, which takes a reference to another instance of the adder class as argument. Finally, we have to equip the adder class with a copy constructor because each thread will get one copy of the originally provided adder instance.

```
class adder {
    float *myvec // pointer o array with values to sum.

public:
    float sum;

    adder ( float *a ): myvec(a), sum(0) { }

    adder ( adder &a, split ): myvec(a.myvec, su (0) { }

    void operator()( const tbb::blocked_range<float> &r ) {
        for( float i = r.begin(); i != r.end( ); ++i )
            sum += myvec[i];
    }

    void join( const adder &other ) {
        sum += other.sum;
    }
};
```

The listing shows how the partial sum will be calculated by iterating over the elements of the subvector in `operator()`. The reduction of two partial sums to one value is implemented in `join()`.

The summation of all array elements can be performed with the following code piece:

```
float *vec; // Vector data, initialise them.
int veclen; // ...

adder vec_adder(vec); // Create root object for decomposition.
parallel_reduce(tbb::blocked_range<int>(0, size, 5), vec_adder);
   // Define range and grainsize for iteration.
// vec_adder.sum contains the sum of all vector elements.
```

8.4.2. Vectorization and Cache-Locality

Vectorization and cache-locality should be used in order to increase the program efficiency. Many details and examples are given in the programming and compilation guide of the MIC architecture [25].

Vectorization can be achieved by auto vectorization [http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-win/index.htm#GUID-7D541D6D-4929-4F35-A58D-B67F9A897AA0.htm] and checked with the vec-report [http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-win/index.htm#GUID-3D61D83A-857D-49C3-A6C9-A1037BFA63CD.htm] option. There is the possibility to apply user mandated vectorization [http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-win/index.htm#GUID-42986DEF-8710-453A-9DAC-2086EE55F1F5.htm] for constructs that are not covered by automatic vectorization. Users of Cilk Plus also have an option to use extensions for array notation [http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-win/index.htm#GUID-B4E06ED4-184F-40E6-A8B4-117947D8C7AD.htm].

8.4.3. Work-stealing versus Work-sharing

Using work-stealing for task scheduling means that idle threads will take over tasks from other busy threads. This approach has been mentioned already above when it was said that there should be a certain amount of tasks per thread available in order to compensate load imbalances. The scheduler could keep idle processors busy if sufficient many tasks are available.

Work-sharing as method of the task scheduling is worthwhile for well-balanced workloads. Work-sharing is typically implemented as task pool and achieves near optimal occupancy for balanced workload.

9. IPP: The Intel Integrated Performance Primitives

Remark. This section gives a general overview about IPP based on version 8.0, the latest version at the moment of writing. IPP cannot be used on Xeon Phi devices at the moment. There is not yet a decision whether or when IPP will be supported in the coprocessors. The latest release, published in September 2013, contains a technology preview of asynchronous implementations as well as algorithms that have been implemented in the OpenCL programming language in order to support processing on GPU.

9.1. Overview of IPP

Intel Integrated Performance Primitives (IPP) is a software library, which provides a large collection of functions for signal processing and multimedia processing. It is useable on MS-Windows, Linux, and Mac OS X platforms.

The covered topics are

- Signal processing, including
  - FFT, FIR, Transformations, Convolutions etc
- Image & frame processing, including
  - Transformations, filters, color manipulation
- General functionality, including
  - Statistical functions
  - Vector, matrix and linear algebra for small matrices.

The functions in this library are optimised to use advanced features of the processors like SSE and AVX instruction sets. Many functions are internally parallelised using OpenMP.

Practical aspects. IPP requires support of Streaming SIMD extensions (SSE) or Advanced Vector Extensions (AVX). The validated operating systems include newer versions of MS Windows (Windows 2003, Windows 7, Windows 2008), Redhat Enterpreise Linux (RHEL) 5 and 6, and Mac OS X 10.6 or higher. More detailed information can be found in the product documentation.

The library is compatible to Intel, Microsoft, GNU and cctools compilers. It contains freely distributable runtime libraries in order to allow the execution of programs for users without having the need to install the development tools.

9.2. Using IPP

9.2.1. Getting Started

The following example has been taken from the IPP User's Guide [31].

```
#include "ipp.h"
#include <stdio.h>

int main(int argc, char* argv[])
{

    const IppLibraryVersion *lib;
    Ipp64u fm;

    ippInit();                  //Automatic best static dispatch
    //ippInitCpu(ippCpuSSSE3); //Select a specific static library level
                               //Can be useful for debugging/profiling
    // Get version info
    lib = ippiGetLibVersion();
    printf(%s %s\n, lib->Name, lib->Version);
    //Get CPU features enabled with selected library level
    fm = ippGetEnabledCpuFeatures();
    printf(SSE2  %c\n,(fm>>2)&1? 'Y' : 'N');
    printf(SSE3  %c\n,(fm>>3)&1? 'Y' : 'N');
    printf(SSSE3 %c\n,(fm>>4)&1? 'Y' : 'N');
    printf(SSE41 %c\n,(fm>>6)&1? 'Y' : 'N');
    printf(SSE42 %c\n,(fm>>7)&1? 'Y' : 'N');
    printf(AVX   %c OS Enabled %c\n,
           (fm>>8)&1 ? 'Y' : 'N', (fm>>9)&1 ? 'Y' : 'N');
    printf(AES   %c CLMUL      %c\n,
           (fm>>10)&1 ? 'Y' : 'N', (fm>>11)&1 ? 'Y': 'N');

    return 0;
}
```

This program contains three steps. The initialisation with `ippInit()` makes sure that the best possible implementation of IPP functions will be used during the program execution. The next step provides nformation about the used library version, and finally will be the enabled hardware features listed.

Building of the program. The first step to build the program is to provide the correct environment settings for the compiler. Intel's default solution is a shellscript compilervars.sh in the bin directory of the installation that should be executed. Some other software like the modules environment is available on some HPC computer systems too. Please refer to the locally available documentation.

The program, saved in the file ipptest.cpp can be compiled and linked with the following command line:

```
icc ipptest.cpp -o ipptest \
     -I$IPPROOT/include -L$IPPROOT/lib/intel4 \
     -lippi -lipps -lippcore
```

The executable can be started with the following command:

```
./ipptest
```

9.2.2. Linking of Applications

IPP contains different implementations of each function that provide the best performance on different processor architectures. They will be selected by means of dispatching while the programmer uses always the same API.

Dynamic Linking. Dynamic linking of IPP can be achieved by using Intel's compiler switch `-ipp` or by linking with the default libraries that have names that do not end in `_l` resp. `_t`. The dynamic libraries use internally OpenMP in order to achive multi-threaded execution. The multithreading within the IPP routines can be turned off by calling the function `ippSetNumThreads(1)`. Other options are static single-threaded linking or to build a single-threaded shared library from the static single-threaded libraries.

Static Linking. The libraries with names ending in `_l` must be used for static single- threaded linking, while the libraries with names ending in `_t` provide will provide multi-threaded implementations based again on OpenMP.

More information. The exact choice of the linking model depends on several factors like intended processor architectures for the execution, useable memory size during the execution, installation aspects and more. A white paper available online focuses on such aspects and provides an in-depth discussion of the topic.

9.3. Multithreading

The use of treads within the IPP should also be taken into consideration by the application developer.

- IPP uses OpenMP in order to achieve internally a multi-threaded execution as mentioned in the section about linking. IPP uses processors up to the minimum of `$OMP_NUM_THREADS` and the number of available processors. Another possibility to get and set the number of used processors are the functions `ippSetNumThreads(int n)` and `ippGetNumThreads()`.
- Some functions (for example FFT) are designed to use two threads that should be mapped onto the same die in order to use a shared L2 cache if available. The user should set the following environment variable when using processors with more than two cores per die in order to ensure best performance: `KMP_AFFINITY=compact`.
- There is a risk of thread oversubscription and performance degradation in the case that an application uses OpenMP additionally. The use of threads within IPP can be turned off by calling `ippSetNumThreads(1)`. However, some OpenMP-related functionality could be active regardless of that. Therefore, single-threaded execution can be achieved best by using the single-threaded libraries.

9.4. Links and References

Intel Software Documentation Library [32].

Technical information: Selecting the Intel Integrated Performance Primitives (Intel IPP) libraries needed by your application [33].

10. Further programming models

The programming models OpenCL and OpenACC have become popular to program GPGPUs and have also been enabled for the Intel MIC architecture.

10.1. OpenCL

OpenCL (Open Computing Language) [34] is the first open, royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers and handheld/embedded devices. OpenCL is maintained by the non-profit technology consortium Khronos Group. It has been adopted by Apple, Intel, Qualcomm, Advanced Micro Devices (AMD), Nvidia, Altera, Samsung, Vivante and ARM Holdings. OpenCL 2.0 is the latest significant evolution of the OpenCL standard, designed to further simplify cross-platform programming, while enabling a rich range of algorithms and programming patterns to be easily accelerated.

A coding guide for developing OpenCL applications for the Intel Xeon Phi coprocessor can be found in [35]. More details are provided in the Intel SDK for OpenCL Applications XE – Optimization Guide [36].

10.2. OpenACC

The OpenACC Application Program Interface [37] describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators. OpenACC is designed for portability across a wide range of accelerators and coprocessors, including APUs, GPUs, many-core and multi-core implementations. The standard is developed by Cray, CAPS, Nvidia and PGI.

Intel Xeon Phi support is provided by the French company CAPS [38]. Based on the directive-based OpenACC and OpenHMPP standards, CAPS compilers enable developers to incrementally build portable applications for various many-core systems such as NVIDIA and AMD GPUs, and Intel Xeon Phi.

11. Debugging

Information about debugging on Intel Xeon Phi coprocessors can be found in [22].

The GNU debugger (gdb) has been enabled by Intel to support the Intel Xeon Phi coprocessor. The debugger is now part of the recent MPSS release and does not have to be downloaded separately any more.

There are 2 different modes of debugging supported: native debugging on the coprocessor or remote cross-debugging on the host.

11.1. Native debugging with gdb

- Run gdb on the coprocessor

```
ssh -t mic0 /usr/bin/gdb
```

- One can then attach to a running application with process ID pid via

```
(gdb) attach pid
```

- or alternatively start an application from within gdb via

```
(gdb) file /path/to/application
(gdb) start
```

11.2. Remote debugging with gdb

- Run the special gdb version with Xeon Phi support on the host

```
/usr/linux-k1om-4.7/bin/x86_64-k1om-linux-gdb
```

- Start the gdbserver on the coprocessor by typing on the host gdb

```
(gdb) target extended-remote| ssh -T mic0 gdbserver -multi -
```

- Attach to a remotely running application with the remote process ID pid

```
(gdb) file /local/path/to/application
(gdb) attach pid
```

- It is also possible to run an application directly from the host gdb

```
(gdb) file /local/path/to/application
(gdb) set remote exec-file /remote/path/to/application
```

12. Tuning

Information and material on performance tuning from Intel can be found in [40] and [41].

A single Xeon Phi core is slower than a Xeon core due to lower clock frequency, smaller caches and lack of sophisticated features such as out-of-order execution and branch prediction. To fully exploit the processing power of a Xeon Phi, parallelism on both intruction level (SIMD) and thread level (OpenMP) is needed.

The following sections describe a few basic methodologies for improving the performance of a code on a Xeon Phi. As a first step, we consider methods for improving the performance on a single core. We then continue by thread-level parallelization in shared memory.

12.1. Single core optimization

12.1.1. Memory alignment

Xeon Phi can only perform memory reads/writes on 64-byte aligned data. Any unaligned data will fetched and stored by performing a masked unpack or pack operation on the first and last unaligned bytes. This may cause performance degredation, especially if the data to be operated on is small in size and mostly unaligned. In the following, we list a few of the most common ways to let the compiler to align the data or to assume that the data has been aligned.

Compiler flags for alignment

C/C++ `n/a`

Fortran Align arrays: `-align array64byte`

Align fields of derived types: `-align rec64byte`

Compiler directives for alignment

C/C++ Align variable `var`:

```
float var[100] __attribute__((aligned(64)));
```

Inform the compiler of the alignment of variable `var`:

```
__assume_aligned(var, 64)
```

Declare a loop to be aligned:

```
#pragma vector aligned
```

Fortran Align variable `var`:

```
real var(100)
```

```
!dir$ attributes align:64::var
```

Inform the compiler of the alignment of variable `var`:

```
!dir$ assume_aligned var:64
```

Declare a loop to operate on aligned data:

```
!dir$ vector aligned
```

Allocation of aligned dynamic memory

C/C++ Use `_mm_malloc()` and `_mm_free()` to allocate and free memory. These take the desired byte-alignment as a second input argument. When using a aligned variable `var`, use `__assume_aligned(var, 64)` to inform the compiler about the alignment

Fortran Use `-align array64byte` compiler flag to enforce aligned heap memory allocation.

For a more detailed description of memory alignment on Xeon Phi, see [45].

12.1.2. SIMD optimization

Each Xeon Phi core has a 512-bit VPU unit which is capable of performing 16SP flops or 8 DP flops per clock cycle. VPU units are also capable of Fused Multiply-Add (FMA) or Fused Multiply-Subtract (FMS) operations which effectively double the theoretical floating point performance.

Intel compilers have several directives to aid vectorization of loops. These are listed in the following in short. For details, refer to the compiler manuals.

Let the compiler know there are no loop carried dependencies, but only affect compiler heuristics.

C/C++ `#pragma IVDEP`

Fortran `!DIR$ IVDEP`

Let the compiler know the loop should be vectorized, but only affect compiler heuristics.

C/C++ `#pragma VECTOR`

Fortran `!DIR$ VECTOR`

Force the compiler vectorize a loop, independent of heuristics.

C/C++ `#pragma SIMD` or `#pragma vector always`

Fortran `!DIR$ SIMD` or `!DIR$ VECTOR ALWAYS`

In order to aid vectorization, Intel compilers can report details on whether vectorization is successful or not. The reports can be generated with `-vec-reportN` compiler flag, where `N` denotes the report level. The vector report levels are:

Level	Description
N=0	No diagnostic information.
N=1	Report vectorized loops.
N=2	Report vectorized and non-vectorized loops.
N=3	Report vectorized and non-vectorized loops with any proven or assumed data dependencies.

N=4	Report non-vectorized loops.
N=5	Report non-vectorized loops with a reason why they were not vectorized.
N=6	Use greater detail on reporting vectorized and non-vectorized loops with any proven or assumed data dependencies.
N=7	Very detailed report, not intended to be human readable. Python script for gathering information and annotating the vector report with source code available from Intel [46].

For a more detailed description of SIMD vectorization with Intel Xeon Phi, see [47] and references therein.

12.2. OpenMP optimization

Threading parallelism with Intel Xeon Phi can be readily exploited with OpenMP. All threading constructs are equivalent for both offload and native models. We expect the basic concepts and syntax of OpenMP to be known. For OpenMP, see for instance "Using OpenMP" By Chapman et al [5], the OpenMP forum [13], and references therein.

The high level of parallelism available on a Xeon Phi available is very likely to reveal any performance problems related to threading previously been unnoticed in the code. In the following, we introduce a few of the most common OpenMP performance problems and suggest some ways to correct them. We begin by considering thread to core affinity and thread placement among the cores. For further details, we refer to [48] .

12.2.1. OpenMP thread affinity

Each Xeon Phi card contains a shared-memory environment with approximately 60 physical cores which, in turn, are divided into 4 logical cores each. We refer to this as node topology.

Each memory bank resides closer to some of the cores in the topology and therefore access to data laying in a memory bank attached to another socket is generally more expensive. Such a non-uniform memory access (NUMA) can create performance issues if threads are allowed to migrate from one logical core to another during their execution.

In order to extract maximum performance, consider binding OpenMP threads to logical and physical cores across different sockets on a single Xeon Phi card. The layout of this binding in respect to the node topology has performance implications depending on the computational task and is referred as thread affinity.

We now briefly show how to set thread affinity using Intel compilers and OpenMP-library. For a complete description, see "Thread affinity interface" in the Intel compiler manual.

The thread affinity interface of the Intel runtime library can be controlled by using the `KMP_AFFINITY` environment variable or by using a proprietary Intel API. We now focus on the former. A standardized method for setting affinity is available with OpenMP 4.0.

`KMP_AFFINITY`=[modifier,...]<type>[,<permute>][,<offset>]]

modifier	**default**=noverbose, respect, granularity=core
	granularity=<{fine, thread, core}>, norespect, noverbose, nowarnings, proclist={<proc-list>}, respect, verbose, warnings.
type	**default**=none
	balanced, compact, disabled, explicit, none, scatter
permute	**default**=0.
	$\geq$ 0, integer. Not valid with type=explicit,none,disabled.
offset	**default**=0.

$\geq$ 0, integer. Not valid with type=explicit,none,disabled

In most cases it is sufficient only to specify the affinity and granularity. The most important affinity types supported by Intel Xeon Phi are

balanced	Thread affinity balanced is a mixture of scatter and compact affinities. Threads from <1> to <n_p> will be spread across the topology as evenly as evenly as possible in the granularity context, where <n_p> denotes the number of physical cores. For thread <k> from threads <n_p+1> to <n> will be assigned as close as possible to thread <k+1>.
compact	Thread <k+1> will be assigned as close as possible to thread <k> in the granularity context according to which the threads are placed.
none	Threads are not bound to any contexts. Use of affinity none is not recommended in general.
scatter	Threads from <1> to <n> will be spread across the topology as evenly as possibly in the granularity context according to which the threads are placed.

The most important granularity types supported by Intel Xeon Phi are

core	Threads are bound to a single core, but allowed to float within the context of a physical core.
fine/thread	Threads are bound to a single context, i.e., a logical core.

12.2.2. Example: Thread affinity

We now consider the effect of thread affinity to matrix-matrix multiply. Let A, B and C=AB be real matrices of size 4000-by-4000. We implement the data initialization and multiplication operation in Fortran90 (without blocking) by using jki loop-ordering and OpenMP as follows

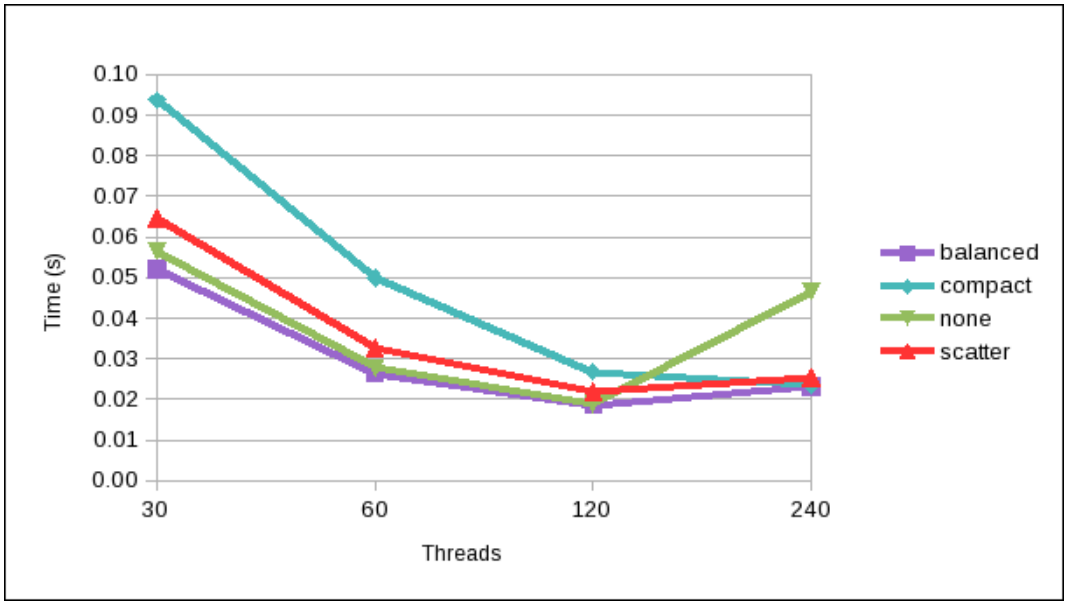
```
! Initialize data per thread
!$OMP PARALLEL DO DEFAULT(NONE) &
!$OMP SHARED(A,B,C) &
!$OMP PRIVATE(j)
DO j=1,n
   ! Initialize A
   CALL RANDOM_NUMBER(A(1:n,j))
   ! Initialise B
   CALL RANDOM_NUMBER(B(1:n,j))
   ! Initialise C
   C(1:n,j)=0D0
END DO
!$OMP END PARALLEL DO

! C=A*B
!$OMP PARALLEL DO DEFAULT(NONE) &
!$OMP SHARED(A,B,C) &
!$OMP PRIVATE(i,j,k)
DO j=1,n
  DO k=1,n
    DO i=1,n
      C(i,j)=C(i,j)+A(i,k)*B(k,j)
    END DO
  END DO
```

```
END DO
!$OMP END PARALLEL DO
```

Running and timing the matrix-matrix multiplication part on a single Intel Xeon Phi 7110 card, we have the following results.

Threads	balanced, t(s)	compact, t(s)	none, t(s)	scatter, t(s)
1	1.43E+00	1.45E+00	1.42E+00	1.58E+00
30	5.21E-02	9.37E-02	5.65E-02	6.45E-02
60	2.64E-02	5.00E-02	2.77E-02	3.26E-02
120	1.86E-02	2.68E-02	1.89E-02	2.18E-02
240	2.31E-02	2.35E-02	4.66E-02	2.54E-02

Figure 3. Results of Example: Thread affinity



As seen from the results, changing the thread affinity has implications to the performance of even a very simple test case. Using affinity `none` is generally not recommended and is the slowest when all available threads are being used. Using affinity `balanced` is generally a good compromise, especially if one wishes to use less than the number of threads available at maximum.

12.2.3. OpenMP thread placement

Xeon Phi runtime for OpenMP includes an extension for placing the threads over the cores on a single card. For a given number of cores and threads, the user has control where (relative to the first physical core, i.e., core 0) the OpenMP threads are placed. In conjunction with offloading from several MPI processes to a single Xeon Phi card, such control can be very useful to avoid oversubscription of cores.

The placement of the threads can be controlled with the environment variable `KMP_PLACE_THREADS`. It specifies the number of cores to allocate with an optional offset value and number of threads per core to use. Effectively `KMP_PLACE_THREADS` defines the node topology for `KMP_AFFINITY`.

`KMP_PLACE_THREADS`= (*int* ["C" | "T"] [*delim*] | *delim*) [*int* ["T"] [*delim*]] [*int* ["O"]],

where *int* is a simple integer constant and *delim* is either "," or "x".

C **default** if "C" or "T" not specified

Indicates number of cores

T Indicates number of threads

O **default**=0O.

Indicates number of cores to offset, starting from core 0. Offset ignores granularity.

As an example, we consider the case with `OMP_NUM_THREADS`=20.

`KMP_PLACE_THREADS` value	**Thread placement**
5C,4T,0O or 5,4,0 or 5C or 5	Threads are placed on cores from 0 to 4 with 4 threads per core.
5C,4T,10O or 5,4,10 or 5C,10O	Threads are placed on cores from 10 to 14 with 4 threads per core.
20C,1T,20O or 20,1,20	Threads are placed on cores from 20 to 40 with 1 thread per core.

12.2.4. Multiple parallel regions and barriers

Whenever an OpenMP parallel region is encountered, a team of threads is formed and launched to execute the computations. Whenever the parallel region is ended, threads are joined and the computation proceeds with a single thread. Between different parallel regions it is up to the OpenMP implementation to decide whether the threads are shut down or left in an idle state.

Intel OpenMP -library leaves the threads in a running state for a predefined amount of time before setting them to sleep. The time is defined by `KMP_BLOCKTIME` and `KMP_LIBRARY` environment variables. The default is 200ms. For more details, see Sections "Intel Environment Variables Extensions" and "Execution modes" in the Intel compiler manual.

Repeatedly forming and disbanding thread-teams and setting idle threads to sleep has some overhead associated with it. Another common source of threading overhead in OpenMP computations are implicit or explicit barriers. Recall that many OpenMP constructs have an implicit barrier attached to the end of the construct. Then, especially if the amount of work done inside an OpenMP construct is relatively small, thread synchronization with several threads may be a source of significant overhead. If the computations are independent, the implicit barrier at the end of OpenMP constructs can be removed with the optional NOWAIT parameter.

12.2.5. Example: Multiple parallel regions and barriers

We now consider the effect of multiple parallel regions and barriers to performance. Let `v` be a vector with real entries with size `n`=1000000. Let `f(x)` denote a function, defined as `f(x)=x+1`.

We implement an OpenMP loop to apply `f(x)` successively `repeats`=10000 times to a given vector `v`. We consider three different implementations. In the first one, OpenMP parallel region is re-initialized for each successive application of `f(x)`. The second one initializes the parallel region once, but contains two implicit barriers from OpenMP constructs. In the third implementation the parallel region is initialized once and one barrier is used to synchronize the repetitions.

Implementation 1: parallel region re-initialized repeatedly.

```
DO rep=1,repeats
  !$OMP PARALLEL DO DEFAULT(NONE) NUM_THREADS(threads) &
  !$OMP SHARED(vec1, rep, n) &
  !$OMP PRIVATE(i)
  DO i=1,n
    vec1(i)=vec1(i)+1D0
  END DO
  !$OMP END PARALLEL DO
```

```
  ops = ops + 1
END DO
```

Implementation 2: parallel region initialized once, two implicit barriers from OpenMP constructs.

```
!$OMP PARALLEL DEFAULT(NONE) NUM_THREADS(threads) &
!$OMP SHARED(vec1, repeats, ops, n) &
!$OMP PRIVATE(i, rep)
DO rep=1,repeats
  !$OMP DO
  DO i=1,n
    vec1(i)=vec1(i)+1D0
  END DO
  !$OMP END DO

  !$OMP SINGLE
  ops = ops + 1
  !$OMP END SINGLE
END DO
!$OMP END PARALLEL
```

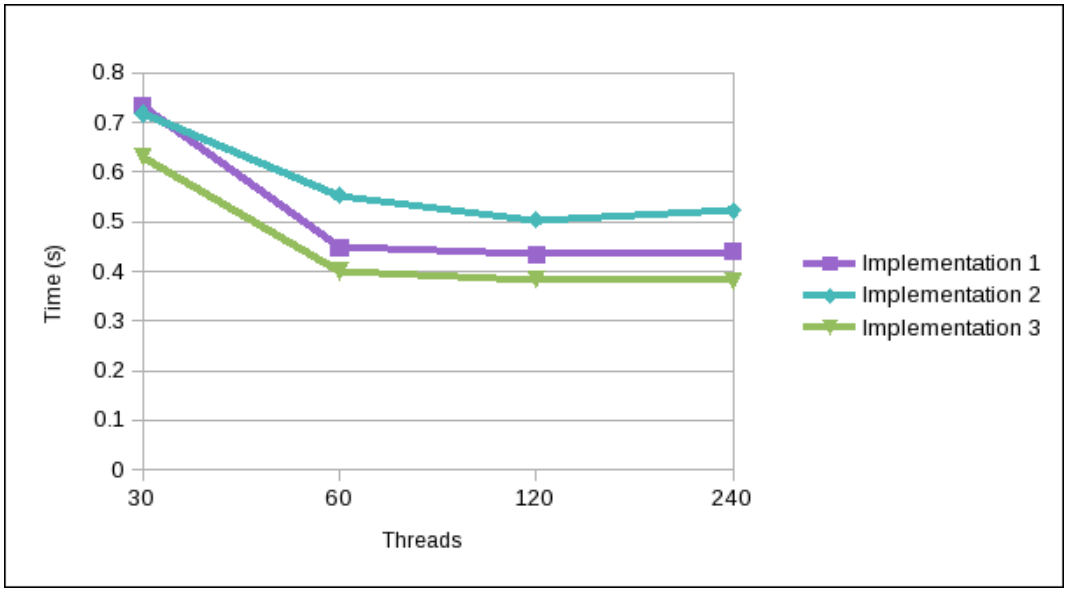
Implementation 3: parallel region initialized once, one explicit barrier from OpenMP construct.

```
!$OMP PARALLEL DEFAULT(NONE) NUM_THREADS(threads) &
!$OMP SHARED(vec1, repeats, ops, n) &
!$OMP PRIVATE(i, rep)
DO rep=1,repeats
  !$OMP DO
  DO i=1,n
    vec1(i)=vec1(i)+1D0
  END DO
  !$OMP END DO NOWAIT

  !$OMP SINGLE
  ops = ops + 1
  !$OMP END SINGLE NOWAIT

  ! Synchronize threads once per round
  !$OMP BARRIER
  END DO
!$OMP END PARALLEL
```

The results on a single Intel Xeon Phi 7110 card with `KMP_AFFINITY`=granularity=fine,balanced and `KMP_BLOCKTIME`=200 are presented in the following table.

Threads	**Implementation 1, t(s)**	**Implementation 2, t(s)**	**Implementation 3, t(s)**
1	2.40E+01	2.37E+01	2.37E+01
30	7.33E-01	7.18E-01	6.31E-01
60	4.48E-01	5.53E-01	4.00E-01
120	4.34E-01	5.04E-01	3.83E-01
240	4.40E-01	5.23E-01	3.81E-01

Figure 4. Results of Example: Multiple parallel regions and barriers



As the number of threads used increases, parallel threading overhead becomes more apparent. The implementation with only one barrier is the fastest by a fair margin. Implementation 1 comes out as the second fastest. This is due to Implementation 1 having only one barrier (at the end of the parallel region) versus two in Implementation 2 (at the end of both of the OpenMP constructs). With Implementation 1, the parallel region is re-initialized immediately after it has ended and thus waiting time for threads is less than `KMP_BLOCKTIME`, i.e., the threads are not being put to sleep before the next parallel iteration begins.

12.2.6. False sharing

On a multiprocessor shared-memory system, each core has some local cache, which must be kept coherent the among the cores in the system. Processor cache is organized into several cache lines, each of which map to some part of the main memory. On an Intel Xeon Phi, cache line size is 64 bytes. For reference and details, see [50] .

If more than one core accesses the same data in the main memory, a cache line is shared. Whenever a shared cache line is updated, to maintain coherency an update is forced to the caches of all the cores accessing the cache line.

False sharing occurs when several cores access and update different variables which reside on a single shared cache line. The resulting updates to maintain cache coherency may cause a significant performance degradation. The processors may not be actually sharing any data, it is sufficient that the data resides on a same cache line, hence the name false sharing. Due to the ring-bus architecture of the Xeon Phi, false sharing among the cores can cause severe performance degredation.

False sharing can be avoided by carefully considering write access to shared variables. If a variable is updated often, it may be worthwhile to use a private variable in stead of a shared one and use reduction at the end of the work sharing loop.

Given a code with performance problems, false sharing may be hard to localize. Intel VTune Performance Analyzer can be used to locate false sharing. For details, we refer to [49].

12.2.7. Example: False sharing

We now consider a simple example where false sharing occurs. Let `v` be a vector with real entries and size `n`=1E +08. Let `f(x)` denote a function which counts the number of entries in `v` which are smaller than zero.

We implement `f(x)` with OpenMP in two different ways. In the first implementation, each thread counts the number of negative entries it has found in `v` to a globally shared array. To avoid race conditions, each thread uses its own entry in the shared array, uniquely determined by thread id. When a thread has finished its portion

of vector, a global counter is atomically incremented. The second implementation is practically equivalent to the first one, except that each thread has its own private array for counting the data.

Implementation 1: False sharing with an array counter.

```
!$OMP PARALLEL DEFAULT(NONE) NUM_THREADS(threads) &
!$OMP SHARED(vec, count, counter, n) &
!$OMP PRIVATE(i,TID)

TID=1
!$ TID=omp_get_thread_num()+1

!$OMP DO
DO i=1,n
  IF (vec(i)<0) counter(TID)=counter(TID)+1
END DO
!$OMP END DO

!OMP ATOMIC
count = counter(TID)+count

!$OMP END PARALLEL
```

Implementation 2: Private array used to avoid false sharing.

```
!$OMP PARALLEL DEFAULT(NONE) NUM_THREADS(threads) &
!$OMP SHARED(vec, count, n) &
!$OMP PRIVATE(i, counter, TID)

TID=1
!$ TID=omp_get_thread_num()+1

!$OMP DO
DO i=1,n
  IF (vec(i)<0) counter(TID)=counter(TID)+1
END DO
!$OMP END DO

!OMP ATOMIC
count = counter(TID)+count

!$OMP END PARALLEL
```

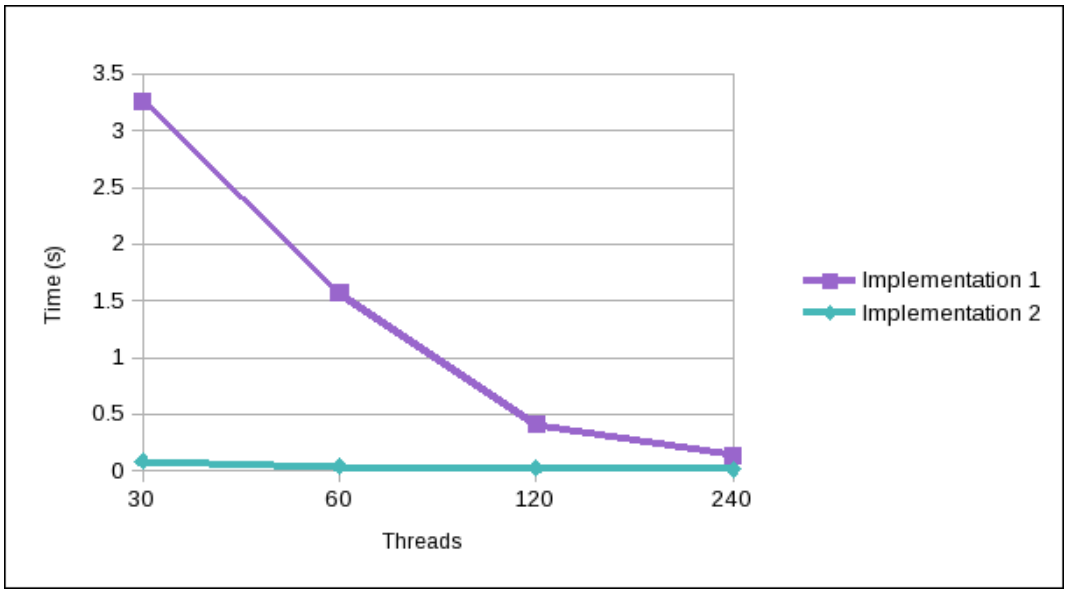
We note that a better implementation for this particular problem will be given in the next section.

The results on a single Intel Xeon Phi 7110 card with `KMP_AFFINITY`=granularity=fine,balanced are presented in the following table. We note that to obtain the results, we compiled the test code with `-O1`. On optimization level `-O2` and higher, at least in this case, the possibility of false sharing with multiple threads was recognized and corrected by the Intel Fortran compiler.

Threads	**Implementation 1, t(s)**	**Implementation 2, t(s)**
1	1.94E+00	2.05E+00
30	3.26E+00	8.30E-02
60	1.57E+00	4.18E-02

Threads	Implementation 1, t(s)	Implementation 2, t(s)
120	4.13E-01	2.53E-02
240	1.40E-01	1.34E-02

Figure 5. Results of Example: False sharing



As expected, Implementation 2 is faster than Implementation 1, with a difference of an order of magnitude. Although in this case we had to lower the optimization level to prevent the compiler from correcting the situation, we cannot completely rely on the compiler to detect false sharing, especially if the code to be compiled is relatively complex.

12.2.8. Memory limitations

Available memory per core on Xeon Phi is very limited. When an application is run using all the available threads, approximately 30Mb of memory is available per thread assuming none of the data is shared. Excessive memory allocation per thread is therefore highly discouraged. Care should be also taken when assigning private variables in order to avoid unnecessary data duplication among threads.

12.2.9. Example: Memory limitations

We now return to the example given in the previous section. In the example, we prevented threads from doing false sharing by modifying the definition of the vector containing the counters. What is important to note is that in doing so, each thread now implicitly allocates a vector of length nthreads, i.e., the memory consumption is quadratic in terms of the number of threads. A better alternative is to let each thread to store the local result in a temporary variable and use a reduction to count the number of elements smaller than zero.

Implementation 3: Temporary variable with reduction used to store local results.

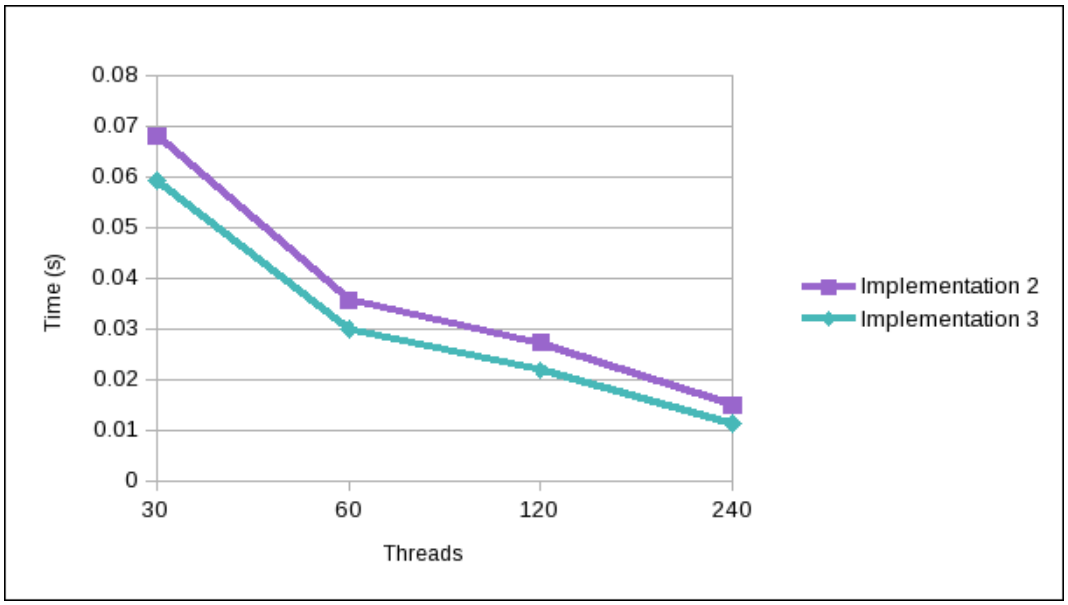
```
count = 0
!$OMP PARALLEL DEFAULT(NONE) NUM_THREADS(threads) &
!$OMP SHARED(vec, n) &
!$OMP PRIVATE(i, Lcount, TID) &
!$OMP REDUCTION(+:count)

TID=1
!$ TID=omp_get_thread_num()+1
```

```
Lcount = 0
!$OMP DO
DO i=1,n
  IF (vec(i)<0) Lcount=Lcount+1
END DO
!$OMP END DO

count = Lcount+count
!$OMP END PARALLEL
```

We have the following results, where Implementation 2 refers to second implementation given in the previous section. As previously, results have been computed on a single Intel Xeon Phi 7110 card by using `KMP_AFFINITY`=granularity=fine,balanced and optimization level `-O1`.

Threads	Implementation 2, t(s)	Implementation 3, t(s)
1	2.04E+00	1.78E+00
30	6.81E-02	5.93E-02
60	3.57E-02	2.99E-02
120	2.73E-02	2.19E-02
240	1.50E-02	1.13E-02

Figure 6. Results of Example: Memory limitations



The main difference between Implementation 2 and 3 is memory use. This is because Implementation 2 uses a private array for storing the result (memory usage grows quadratically with the number of threads), whereas in Implementation 3 the result is stored a single scalar per thread. In total 57600 elements have to be stored in Implementation 2 with 240 threads, whereas just 240 elements suffice in Implementation 3 for the same amount of threads. We note that if an array of results is to be computed, one should prefer using an implementation where the size of the work arrays does grow with the number of threads used.

12.2.10. Nested parallelism

Due to the limited amount of memory available, sometimes using all available threads on an Intel Xeon Phi to parallelize the outer loop of some computation is not possible. In some cases this may not be due to inefficient structure of the code, but because the data needed for computations per thread is too large. In this case, to take advantage of all the processing power of the coprocessor, an option is to use nested OpenMP parallelism.

When nested parallelism is enabled, any inner OpenMP parallel region which is enclosed within an outer parallel region will be executed with multiple threads. The performance impact of using nested parallelism is similar to performance impact of using multiple parallel regions and barriers.

Enabling OpenMP nested parallelism is done by setting environment variable `OMP_NESTED=TRUE` or with an API call to `omp_set_nested` -function. The number of nested threads within each OpenMP parallel region is done by setting the environment variable `OMP_NUM_THREADS=n_1, n_2,n_3,...`, where `n_j` refers to the number of threads on the `j`th level. The number or threads within each nesting level can be also set with an API call to `omp_set_num_threads` -function.

12.2.11. Example: Nested parallelism

Consider a case where several independent matrix-matrix multiplications have to be computed. Let `A` be an `n`-by-`n` matrix and let matrix B_k be defined as $B_k = A^T A$.

We let `m`=1000, `n`=1000 and `k`=240 and study the effect of parallelizing the computation of B_k's in three different ways. The first case is to use parallelize the loop over `k` with all available threads. A second case is to parallelize the computation over different `k`'s to physical Intel Xeon Phi cores and use nested parallelism with varying levels of hardware threads in the computation of the matrix-matrix multiplications. The third case uses parallelization only over physical cores. For this, we have the following implementation.

Implementation: possibly nested parallel loop for computing $A^T A$.

```
!$OMP PARALLEL DO DEFAULT(NONE) NUM_THREADS(nthreads) &
!$OMP SCHEDULE(STATIC) &
!$OMP SHARED(A, B, m, n, k)
DO i=1,k
  CALL DGEMM('T','N', n, n, m, 1D0, A, m, A, m, 0D0, B(1,1,i), n)
END DO
!$OMP END PARALLEL DO
```

The results on a single Intel Xeon Phi 7110 card with `KMP_AFFINITY`=granularity=fine,balanced are presented in the following table.

Threads	**240/1, t(s)**	**60/4, t(s)**	**120/1, t(s)**	**60/2, t(s)**	**60/1, t(s)**
time	1.06E+01	4.98E+00	5.64E+00	4.80E+00	4.81E+00

Intuitively one would expect using 240 threads to be the most efficient in this case. The results show otherwise, however. Since the threads are competing for the same cache, the performance is lowered. Nested parallelism does not offer significant improvements from just using 60 threads in a flat fashion. One reason for this might be that with the affinity policies used, it is difficult to control the thread placement on the second level of thread parallelism.

12.2.12. OpenMP load balancing

With any parallel processing, some processes or threads may require more resources and be more time consuming than others. In a regular distributed memory program, load balancing requires programming effort to redistribute parts of the computation among the processors. In a shared memory program with a runtime, such as OpenMP, load balancing can be in some cases automatically handled by the runtime itself with little overhead.

OpenMP loop constructs support an additional SCHEDULE-clause. The syntax for this is

SCHEDULE(<kind>[,chunk_size]),

kind **default**=STATIC.

<STATIC,DYNAMIC,GUIDED,RUNTIME>

chunk_size	**default**=value depends on the schedule kind. >0, integer.

Different schedule kinds supported by OpenMP runtime on a Xeon Phi are

STATIC	With the static scheduling policy, iteration indices are divided into chucks of chunk_size and distributed to threads in a round-robin fashion. If chuck_size is not defined, iteration indices are divided into chucks that are roughly equal in size.
DYNAMIC	With the dynamic scheduling policy, iteration indices are assigned to threads in chunks of chunk_size. Threads request and process new chucks with assigned iteration indices until the whole index range has been processed.
GUIDED	With the guided scheduling policy, iteration indices are assigned to threads in chunks of size chunk_size at minimum. In the beginning of the iteration, the chunk_size actually assigned to be processed is proportional to the number of unassigned iteration indices versus the number of available threads. The assigned chunk_size and can be larger than the minimum.
RUNTIME	The scheduling policy and chunk size will be decided at runtime based on the `OMP_SCHEDULE` environment variable.

13. Performance analysis tools

13.1. Intel performance analysis tools

The following Intel performance analysis tools have been enabled for the Intel Xeon Phi coprocessor:

- Intel trace analyzer and collector (ITAC)
- Intel VTune Amplifier XE

More information on performance analysis can be found in [39] [42]. Details will be included in a future version of this guide.

13.2. Scalasca

Scalasca [44] is a scalable automatic performance analysis toolset designed to profile large scale parallel Fortran, C and C++ applications that use MPI, OpenMP and hybrid MPI+OpenMP parallelization models. It is portable on Intel Xeon Phi architecture in native, symmetric and offload models. Version 2.x uses the Score-P instrumenter and measurment libraries. The following examples are taken from [51].

13.2.1. Compilation of Scalasca

On the host Scalasca can be normally compiled, while on the device one must perform a cross-compilation and add -mmic compiler option.

13.2.2. Usage

Instrumentation of the code to be profiled is done with the command skin.

```
$ skin mpiifort -O -openmp *.f
```

This produces the instrumented executable that will be executed on the host, while

```
$ skin mpiifort -O -openmp -mmic *.f
```

produces an executable for the coprocessor. Further, measurement is than performed with the scan command:

```
$ scan mpiexec -n 2 a.out.cpu       // on the host
% scan mpiexec -n 61 a.out.mic    // on the device
```

It can also be launched on more than one node on the host:

```
$ scan mpiexec.hydra -host node0 -n 1 a.out.cpu : -host node1 -n 1 a.out.cpu
```

and on more than one coprocessor, if available:

```
$ scan mpiexec.hydra -host mic0 -n 30 a.out.mic : -host mic1 -n 31 a.out.mic
```

For symmetric execution one can use:

```
$ scan mpiexec.hydra -host node0 -n 2 a.out.cpu : -host mic0 -n 61 a.out.mic
```

Finally, the collected data can be analyzed with the square command. The scan output would look something like `epik_a_2x16_sum` for the runs performed on the host and `epik_a_mic61x4_sum` for those performed on the coprocessor. For data collected on the host and on the device we have:

```
$ square epik_a_2x16_sum
$ square epik_a_61x4_sum
```

respectively. To analyze the data collected in a run from a symmetric execution type:

```
$ square epik_a_2x16+mic61x4_sum.
```

Further documentation

Books

[1] *James Reinders, James Jeffers, Intel Xeon Phi Coprocessor High Performance Programming, Morgan Kaufman Publ Inc, 2013 http://lotsofcores.com [http://lotsofcores.com] .*

[2] *Rezaur Rahman: Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers, Apress 2013 .*

[3] *Parallel Programming and Optimization with Intel Xeon Phi Coprocessors, Colfax 2013 http://www.colfax-intl.com/nd/xeonphi/book.aspx [http://www.colfax-intl.com/nd/xeonphi/book.aspx] .*

[4] *Michael McCool, James Reinders, Arch Robison, Structured Parallel Programming: Patterns for Efficient Computation , Morgan Kaufman Publ Inc, 2013 http://parallelbook.com [http://parallelbook.com] .*

[5] *Barbara Chapman, Gabriele Jost and Ruud van der Pas, Using OpenMP, MIT Press Cambridge, 2007, http://mitpress.mit.edu/books/using-openmp [http://mitpress.mit.edu/books/using-openmp].*

Forums, Download Sites, Webinars

[6] *Intel Developer Zone: Intel Xeon Phi Coprocessor, http://software.intel.com/en-us/mic-developer [http://software.intel.com/en-us/mic-developer].*

[7] *Intel Many Integrated Core Architecture User Forum, http://software.intel.com/en-us/forums/intel-many-integrated-core [http://software.intel.com/en-us/forums/intel-many-integrated-core]*.

[8] *Intel Developer Zone: Intel Math Kernel Library, http://software.intel.com/en-us/forums/intel-math-kernel-library [http://software.intel.com/en-us/forums/intel-math-kernel-library]*.

[9] *Intel Math Kernel Library Link Line Advisor, http://software.intel.com/sites/products/mkl/ [http://software.intel.com/sites/products/mkl/]*.

[10] *Intel Manycore Platform Software Stack (MPSS), http://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss [http://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss]*.

[11] *Intel Xeon Processors & Intel Xeon Phi Coprocessors – Introduction to High Performance Applications Development for Multicore and Manycore – Live Webinar, 26.-27.2.2013, recorded http://software.intel.com/en-us/articles/intel-xeon-phi-training-m-core [http://software.intel.com/en-us/articles/intel-xeon-phi-training-m-core]*.

[12] *Intel Cilk Plus Home Page, http://cilkplus.org/ [http://cilkplus.org/]*.

[13] *OpenMP forum, http://openmp.org/wp/ [http://openmp.org/wp/]*.

[14] *Intel Threading Building Blocks Documentation Site, http://threadingbuildingblocks.org/documentation [http://threadingbuildingblocks.org/documentation]*.

Manuals, Papers

[15] *Intel Xeon Phi Coprocessor Developer's Quick Start Guide, http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-developers-quick-start-guide [http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-developers-quick-start-guide]*.

[16] *PRACE-1IP Whitepapers, Evaluations on Intel MIC, http://www.prace-ri.eu/Evaluation-Intel-MIC*.

[17] *Intel Xeon Phi Coprocessor (codename Knights Corner), http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner [http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner]*.

[18] *Intel Xeon Phi Coprocessor System Software Developers Guide, https://secure-software.intel.com/sites/default/files/article/334766/intel-xeon-phi-systemsoftwaredevelopersguide.pdf [https://secure-software.intel.com/sites/default/files/article/334766/intel-xeon-phi-systemsoftwaredevelopersguide.pdf]*.

[19] *System Administration for the Intel Xeon Phi Coprocessor, http://software.intel.com/sites/default/files/article/373934/system-administration-for-the-intel-xeon-phi-coprocessor.pdf [http://software.intel.com/sites/default/files/article/373934/system-administration-for-the-intel-xeon-phi-coprocessor.pdf]*.

[20] *Configuring Intel Xeon Phi coprocessors inside a cluster, http://software.intel.com/en-us/articles/configuring-intel-xeon-phi-coprocessors-inside-a-cluster*.

[21] *Using the Intel MPI Library on Intel Xeon Phi Coprocessor Systems, http://software.intel.com/en-us/articles/using-the-intel-mpi-library-on-intel-xeon-phi-coprocessor-systems [http://software.intel.com/en-us/articles/using-the-intel-mpi-library-on-intel-xeon-phi-coprocessor-systems]*.

[22] *Debugging on Intel Xeon Phi Coprocessor Use Case Overview, http://software.intel.com/en-us/articles/debugging-on-intel-xeon-phi-coprocessor-use-case-overview [http://software.intel.com/en-us/articles/debugging-on-intel-xeon-phi-coprocessor-use-case-overview]*.

[23] *Intel Xeon Phi Coprocessor Instruction Set Reference Manual, http://software.intel.com/sites/default/files/forum/278102/327364001en.pdf [http://software.intel.com/sites/default/files/forum/278102/327364001en.pdf]*.

[24] *An Overview of Programming for Intel Xeon processors and Intel Xeon Phi coprocessors, http://software.intel.com/sites/default/files/article/330164/an-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors_1.pdf [http://software.intel.com/sites/default/files/article/330164/an-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors_1.pdf]*.

[25] *Programming and Compiling for Intel Many Integrated Core Architecture, http://software.intel.com/en-us/articles/programming-and-compiling-for-intel-many-integrated-core-architecture [http://software.intel.com/en-us/articles/programming-and-compiling-for-intel-many-integrated-core-architecture]*.

[26] *Building a Native Application for Intel Xeon Phi Coprocessors, http://software.intel.com/en-us/articles/building-a-native-application-for-intel-xeon-phi-coprocessors [http://software.intel.com/en-us/articles/building-a-native-application-for-intel-xeon-phi-coprocessors]*.

[27] *"Using Intel Math Kernel Library on Intel Xeon Phi Coprocessors" section in the MKL User's Guide, http://software.intel.com/sites/products/documentation/doclib/mkl_sa/11/mkl_userguide_lnx/index.htm [http://software.intel.com/sites/products/documentation/doclib/mkl_sa/11/mkl_userguide_lnx/index.htm]*.

[28] *"Support Functions for Intel Many Integrated Core Architecture" section in the MKL Reference Manual, http://software.intel.com/sites/products/documentation/doclib/mkl_sa/11/mklman/index.htm [http://software.intel.com/sites/products/documentation/doclib/mkl_sa/11/mklman/index.htm]*.

[29] *Intel Math Kernel Library on the Intel Xeon Phi Coprocessor, http://software.intel.com/en-us/articles/intel-mkl-on-the-intel-xeon-phi-coprocessors [http://software.intel.com/en-us/articles/intel-mkl-on-the-intel-xeon-phi-coprocessors]*.

[30] *Intel Compiler 13.0 User Guide and Reference Manual, http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-lin/index.htm [http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-lin/index.htm]*.

[31] *IPP User's Guide, http://software.intel.com/sites/products/documentation/doclib/ipp_sa/71/ipp_userguide_lnx/index.htm*.

[32] *Intel Software Documentation Library, http://software.intel.com/en-us/intel-software-technical-documentation*.

[33] *Selecting the Intel Integrated Performance Primitives (Intel IPP) libraries needed by your application, http://software.intel.com/en-us/articles/selecting-the-intelr-ipp-libraries-needed-by-your-application/*.

[34] *OpenCL Webpage, https://www.khronos.org/opencl/*.

[35] *OpenCL Design and Programming Guide for the Intel® Xeon Phi Coprocessor, http://software.intel.com/en-us/articles/opencl-design-and-programming-guide-for-the-intel-xeon-phi-coprocessor*.

[36] *Intel SDK for OpenCL Applications XE 2013, http://software.intel.com/sites/products/documentation/ioclsdk/2013XE/OG/index.htm*.

[37] *OpenACC Webpage, http://www.openacc-standard.org/*.

[38] *CAPS Compilers , http://www.caps-entreprise.com/products/caps-compilers/*.

[39] *"Using Intel Trace Analyzer and Collector for Intel Many Integrated Core Architecture" in Intel Cluster Studio 2013 Tutorial, http://software.intel.com/sites/products/documentation/hpc/ics/ics2013/ics_tutorial/index.htm#Linux_itac.htm [http://software.intel.com/sites/products/documentation/hpc/ics/ics2013/ics_tutorial/index.htm#Linux_itac.htm]*.

[40] *Advanced Optimizations for Intel MIC Architecture, http://software.intel.com/en-us/articles/advanced-optimizations-for-intel-mic-architecture [http://software.intel.com/en-us/articles/advanced-optimizations-for-intel-mic-architecture]*.

[41] *Optimization and Performance Tuning for Intel Xeon Phi Coprocessors - Part 1: Optimization Essentials, http://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-1-optimization [http://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-1-optimization].*

[42] *Optimization and Performance Tuning for Intel Xeon Phi Coprocessors, Part 2: Understanding and Using Hardware Events, http://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-2-understanding [http://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-2-understanding].*

[43] *Requirements for Vectorizable Loops, http://software.intel.com/en-us/articles/requirements-for-vectorizable-loops/ [http://software.intel.com/en-us/articles/requirements-for-vectorizable-loops/].*

[44] *Scalasca , http://www.scalasca.org/ [http://www.scalasca.org/].*

[45] *Data Alignment to Assist Vectorization, http://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization.*

[46] *VecAnalysis Python Script for Annotating Intel C++ and Fortran Compilers Vectorization Reports , http://software.intel.com/en-us/articles/vecanalysis-python-script-for-annotating-intelr-compiler-vectorization-report.*

[47] *Vectorization essentials, http://software.intel.com/en-us/articles/vectorization-essential.*

[48] *Open MP Thread Affinity Control, http://software.intel.com/en-us/articles/openmp-thread-affinity-control.*

[49] *Avoiding and Identifying False Sharing Among Threads, http://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads.*

[50] *Intel Xeon Phi Core Micro-architecture, http://software.intel.com/en-us/articles/intel-xeon-phi-core-micro-architecture.*

[51] *Brian Wylie, Wolfgang Frings: Scalasca support for Intel Xeon Phi, XSEDE13 22-25 July 2013, San Diego, https://www.xsede.org/documents/384387/561679/XSEDE13-Wylie.pdf [https://www.xsede.org/documents/384387/561679/XSEDE13-Wylie.pdf].*