

Offload Compiler Runtime for the Intel® Xeon Phi™ Coprocessor^{*}

Chris J. Newburn, Rajiv Deodhar, Serguei Dmitriev, Ravi Murty,
Ravi Narayanaswamy, John Wiegert, Francisco Chinchilla, and Russell McGuire

Intel Corporation

Abstract. The Intel® Xeon Phi™ coprocessor platform enables offload of computation from a host processor to a coprocessor that is a fully-functional Intel® Architecture CPU. This paper presents the C/C++ and Fortran compiler offload runtime for that coprocessor. The paper addresses why offload to a coprocessor is useful, how it is specified, and what the conditions for the profitability of offload are. It also serves as a guide to potential third-party developers of offload runtimes, such as a gcc-based offload compiler, ports of existing commercial offloading compilers to Intel® Xeon Phi™ coprocessor such as CAPS®, and third-party offload library vendors that Intel is working with, such as NAG® and MAGMA®. It describes the software architecture and design of the offload compiler runtime. It enumerates the key performance features for this heterogeneous computing stack, related to initialization, data movement and invocation. Finally, it evaluates the performance impact of those features for a set of directed micro-benchmarks and larger workloads.

Keywords: multicore, heterogeneous, coprocessor, offload, compiler, runtime, acceleration.

1 Introducing Offload for a Fully-Capable Compilation Target

This paper describes the runtime infrastructure and performance features of the offload runtime for the Intel® Xeon Phi™ coprocessor. This section covers the architecture, offload execution model, suitability criteria for offload, and contributions.

There has been a long-standing interest in using many power-efficient and more memory- and I/O-bandwidth-capable GPUs and CPUs to efficiently accelerate computation [20,5,26]. Taking advantage of such architectures has often depended on largely rewriting user codes. Intel has taken a different approach: make the coprocessor compilation target very similar to the host, incrementally extend existing programming models for parallelism to take advantage of that coprocessor, and leverage the host when single-thread performance is important, e.g. for I/O. This paper describes one way of doing that: through offload of computation from a host CPU to a fully-capable Intel® Architecture processor that enables higher levels of thread and SIMD parallelism and bandwidth for data-parallel workloads than the Intel® Xeon® processor line. It also offers greater power efficiency: it won Green 500 at SC12 [6]. The first product in that family is

^{*} For more complete information about compiler optimizations, see Intel's Optimization Notice at <http://software.intel.com/en-us/articles/optimization-notice>

codenamed Knights Corner [10]. It is a coprocessor in a PCIe [2] card form factor. One or more cards may connect to host chips that may be part of a larger cluster, as shown in Figure 1.

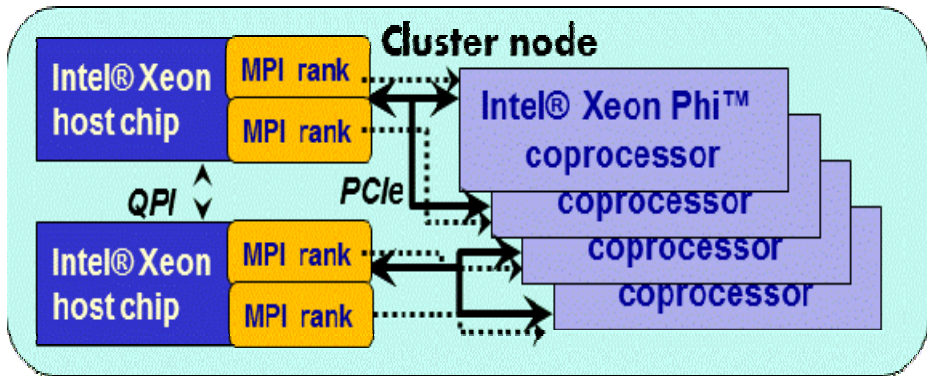


Fig. 1. Intel Xeon Phi™ coprocessor system architecture. MPI ranks may execute on subsets of host cores, and computation may be offloaded to the coprocessors from those ranks, or ranks may be distributed across the host and coprocessor (not shown).

1.1 Execution Models

This platform supports several execution models, as shown in Figure 2. Applications may use the different execution models in different phases, or use a mix of execution models concurrently. And in the symmetric message passing interfaces [7,8, 13] programming model, ranks may run on a heterogeneous set of nodes, concurrently mixing execution on hosts and coprocessors. In hybrid [25] execution, MPI ranks on the host or coprocessor are threaded with OpenMP [22].

- When applications have more than a nominal serial fraction, they are best executed in **host** mode, on a general-purpose CPU that has a higher clock speed and a more aggressive micro-architecture, e.g., an Intel® Xeon® processor (left).
- When applications are highly parallel, with many threads and either highly-SIMD-vectorized code [24] or that demand very high bandwidth to the memory system, **native** execution on an Intel Xeon Phi™ coprocessor may offer higher performance and greater power efficiency (middle or right).
- The highly-parallel phases of the application may be **offloaded** from the Intel® Xeon® host processor to the Intel®Xeon Phi™ coprocessor (middle). In this mode, shown in Figure 1, input data and code are sent to the coprocessor from the host, and output data is sent back to the host when offloaded computation completes. Execution may be **concurrent** on host and coprocessor.

Users may choose to use the offload model instead of native execution because that's higher performing, or for other reasons, such as minimizing complexity or fitting into memory. For example, it may be easier to make an Intel® Math Kernel Library [12] call that transparently results in offload to a coprocessor than to heterogeneously mix MPI ranks on hosts and coprocessors. If the working set [4] fits in the memory on the coprocessor (up to 8GB initially), but the entirety of the memory footprint does

not, it may be necessary to explicitly manage memory locality by partitioning work into computational subtasks that are offloaded to the coprocessor.

1.2 Suitability for Offload

Offloading incurs overhead costs for initialization, marshaling and transferring data, and invocation. These costs vary in their significance. Thus offload may or may not provide a speedup, even if native execution would result in a speedup, as shown in Section 5. There are two important metrics for evaluating the profitability of offload: host-side throughput and response time (latency). Host-side throughput improves if there is a reduction in host-side work that exceeds the host-side overhead of offloading it. Response time improves if the reduction in host-side work exceeds the sum of host-side overhead and the time spent waiting for the coprocessor. The latency formulation accounts for overheads on both the host and coprocessor. Response time and throughput speedups will be identical in the case of single-threaded workloads doing synchronous offload. Offload profitability depends on a mix of the following factors:

- Application characteristics: Applications must have a high ratio of computation to communication (sending code and input data, invocation, and returning output data) for the offloaded portion. Offload communication overheads can be hidden if code is structured to overlap them with computation, or if data is reused across offloads.
- Offload runtime: Offload profitability depends on the efficiency of the host- and coprocessor-side runtimes in marshaling and moving data and invoking code.
- Performance on the *coprocessor and system*: It must benefit from the highly-parallel hardware (SIMD width, threads, and high bandwidth) of the Intel® Xeon Phi™ coprocessor. For example, a serial application that is I/O intensive is unlikely to be a good candidate for that coprocessor.

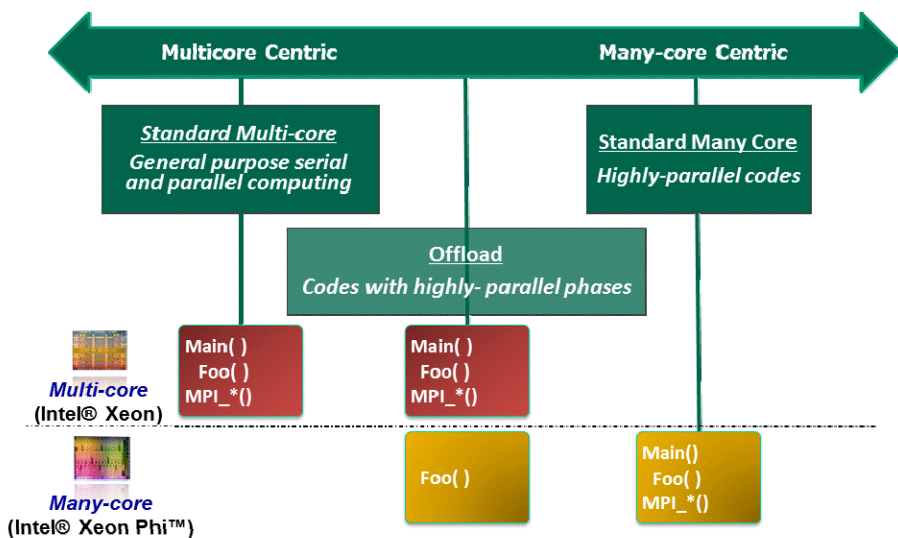


Fig. 2. Execution models: host-only, offload from host to coprocessor and native on coprocessor only

The paper shows examples of workloads with high and with low computation to communication ratios, as well as good and poor native speedups. It focuses on compiler runtime implementation and performance issues related to reducing offload overheads on the host and coprocessor. The performance is evaluated on a few workloads and several directed tests that are in the form of micro- benchmarks.

1.3 Contributions and Scope

The contributions of the paper are as follows. This is the first known description of a production offload compiler and offload runtime software infrastructure for a widely-deployed commercial coprocessor that is as capable as the host CPU. This is the first published description of the Intel® Xeon Phi™ coprocessor offload runtime software architecture. It enumerates issues that matter most to offload compiler runtime performance, describes the solutions to those performance issues, and evaluates their impact on both micro-benchmarks and customer workloads.

There are several other issues of potential interest which are out of scope for this paper. One is a detailed description of the offload programming models described here, and an assessment of their ease of use, performance and productivity relative to other programming models such as CUDA® [20] or OpenACC® [21] or hand-coded assembly. Another is a comparison of our OpenMP 4.0 TR1 [23] implementation with the `#pragma offload` approach. Another is techniques for evaluating and improving the suitability of parallelized code for Intel® Many-Integrated Core, which is documented online [11]. This paper does not delve into the details of the non-compiler runtime components and their performance. The paper’s focus on the offload runtime leaves aside micro-architecture-specific issues related to code generation, like vectorization and prefetching. Finally, issues related to sharing the virtual address space between the host and coprocessor [29] are not covered in this paper.

2 Software Architecture

The Intel® Xeon Phi™ coprocessor software architecture is shown in Figure 3. There are essentially four layers in the software stack: offload tool runtimes, user-level offload libraries, a low-level communication layer that’s split between user-level libraries and kernel drivers, and the operating system. There is a host-side and coprocessor-side component for each. Everything below the offload runtimes is part of the Intel® Manycore Platform Software Stack (MPSS).

This paper focuses on the compiler offload runtime of the Intel® Composer XE 2013 compiler, but there are several other tool offload runtimes listed in Section 3. Intel provides a user-level offload library, called the-Intel® Coprocessor Offload Infrastructure (COI) [15]. This library provides services to create coprocessor-side processes, create FIFO pipelines between the host and coprocessor, move code and data, invoke code (functions) on the coprocessor, manage memory buffers that span the host and coprocessor, enumerate available resources, etc. Offload runtime implementations aren’t strictly required to use COI, but doing so can relieve developers of significant implementation complexity and tuning effort, and it provides portability

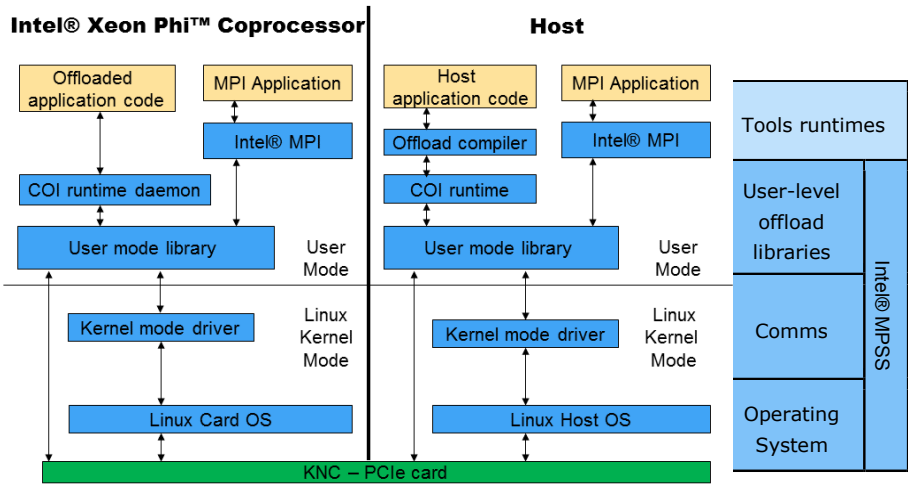


Fig. 3. Intel® Xeon Phi™ coprocessor software architecture

to other host OSes such as Microsoft® Windows™. This paper discusses COI's APIs, but the paper covers implementation and performance optimizations only for the compiler runtime, not other lower-level runtimes like COI and SCIF [15, 28].

Finally, the host and coprocessor have separate operating systems. The snapshot of supported host OSes at the time of evaluation includes RHEL [27] 6.0 kernel 2.6.32-71 and 6.1 kernel 2.6.32-131 and SLES 11 SP1 kernel 2.6.32.12-0.7 [31]. The coprocessor OS is a stock Linux kernel 2.6.34.11 with few architecture-specific modifications. Using a standard OS eases restrictions on what code is offloaded, enables building third-party software and eliminates the need for many proprietary services.

3 Programming Models

A brief background on programming models is provided here, for context. This is not the focus of this paper. Offload is accomplished by runtimes which implement:

- **language pragmas**, e.g. `#pragma omp target data device(1) map()` used by various compilers, e.g. from Intel [9] and CAPS [3], and `#pragma offload`, an Intel-specific extension (see Figure 4)
- **language keywords**, e.g. the `_Cilk_shared` keyword used by the Intel compiler [9,29] or language constructs used by CUDA [20] or OpenCL [16] such as Intel® SDK for OpenCL Applications XE for the coprocessor [14]
- **library calls**, e.g. Intel® Math Kernel Library (MKL) [12], MAGMA [17,1] or NAG [19] calls that divide work across the host and coprocessor

There are also third-party preprocessors that use cost models to evaluate when and how to make use of those tools [26].

This paper focuses on a preliminary version of a proposed but not yet ratified `#pragma omp offload` extension to the OpenMP 4.0 standard [23] implemented by the Intel compiler. An extended version of this paper [18] also covers an Intel-proprietary

set of directives, based on `#pragma offload`. Single-threaded code that uses various offload constructs is illustrated in Figure 4. This code shows how the host synchronously invokes work on the coprocessors, as though DGEMM were invoked on the host with the Intel® Math Kernel Library. See Section 4 for details.

In general, the programmer has to explicitly specify what to do using the offload pragmas. But in some cases, there is a default, for ease of programming. In the `#pragma omp` case, data transfer of variables to and from the coprocessor for offloaded code is explicitly named with `map(to:...)`, `map(from:...)` and `map(tofrom:...)` clauses, but the `map(tofrom:...)` clause is implicitly used for all variables that are not explicitly named but are visible in the scope of a construct during compilation. The coprocessor offload target is specified like `device(1)`.

The offload compiler runtime doesn't manage parallelism. It's the programmer's responsibility to enable and expose vectorization and the compiler's responsibility to extract it, to take advantage of SIMD parallelism. It's the programmer's responsibility to take advantage of thread parallelism. There are two approaches to using hardware threads: using MPI to parallelize across ranks, or using a threading runtime like pthreads, OpenMP [22] or Threading Building Blocks [32] to create and parallelize across a team of threads. These threading runtimes execute natively on the coprocessor. The offload runtime's only interaction with threading runtimes is to set up environment variables they need, e.g. for CPU affinities. By default, the offload runtime sets the CPU affinity mask to allow threading runtimes to execute across all coprocessor threads except those that are used by system software.

4 Offload Compiler Runtime Design

Offload compiler runtime performance matters when the act of offloading is on the critical path. Offload can sometimes be taken off of the critical path with extra coding effort. But in the general case, offload compiler runtime performance can be critical to the profitability of offload, as shown by the data below. This section outlines the key compiler runtime (not COI) performance features that impact performance.

4.1 Overview

The left of Figure 4 shows the host-side code for an offload example, and the right of the figure shows the corresponding sequence of compiler offload runtime actions. When the coprocessor is booted, before execution of the application ever begins, a COI daemon is created (not shown), which handles process creation. Compiler offload runtime initialization discovers the coprocessor and establishes a connection to the COI daemon, which creates a coprocessor-side process that corresponds to the host-side process and that has appropriate environmental settings. At appropriate times, code that is embedded in the host-side binary is extracted, moved and invoked, and memory management is begun.

When data transfers or invocations are specified by the programmer, they trigger specific actions that are executed by the compiler offload runtime and its supporting libraries. Data must be allocated, marshaled, efficiently transferred, unmarshaled and deallocated as necessary. The addresses of coprocessor-side functions must be looked

up and they must have their stack frames set up and be invoked with appropriate parameters, once data transfer dependencies are satisfied.

The COI daemon and the host-side COI infrastructure robustly handle planned and unexpected process termination and error handling on the host and coprocessor.

In the following sections, we focus on each of the performance-critical cases.

4.2 Initialization

The `OFFLOAD_INIT` environment variable controls whether initialization happens at program startup (`on_start`), on first use for each device (`on_offload`, the default), or for all devices at the first offload to any device (`on_offload_all`).

The first step of initialization is to evaluate the conditions for offload. The compiler offload runtime checks for the presence of the COI software stack and the availability of Intel® Xeon Phi™ coprocessors in the system: it gets the count of “engines” (coprocessors) that are currently available. If all offload conditions are satisfied, the compiler offload runtime creates a process on the coprocessor with `COIProcessCreateFromFile` that corresponds to the host process; creates, marshals and moves code and data; and the startup image that is part of the compiler offload runtime that is loaded from disk performs initialization there.

The coprocessor side of the application consists of only those parts of code and data which are marked by the user for offloading. These parts include all routines defined with the `target` attribute, regions of code following offload pragmas, and variables with static storage which are defined with the `target` attribute. All other code and data variables are filtered out. The offload compiler statically generates the coprocessor code and data segments, and embeds it in the host-side binary executable. These get loaded onto the coprocessor at process creation time using `COIProcessLoadLibraryFromMemory`. Additional coprocessor-side code may be available in dependent shared libraries, and these dependencies are detected, resolved, and copied to the coprocessor by COI.

The compiler offload runtime needs the coprocessor-side address for all variables marked with the `target mirror` attribute, in order to know where to transfer data. The address is transferred from the coprocessor to the host during initialization and it is used throughout the execution for transferring data between CPU and coprocessor. This early transfer of the address for heap and static data avoids the need to DMA from the host into a temporary buffer on the coprocessor, and to rely on coprocessor-side code to copy the data to the final destination.

4.3 Invocation

If the offload initialization conditions are not met, there is no invocation of the coprocessor. If the user has mandated the use of offload, e.g. with the `device(1)` clause, then the runtime produces an error. In the absence of such a mandate, a host version of the offloaded region is executed on the host instead of using offload.

Each host thread that has offload pragmas has a corresponding coprocessor-side thread, linked by a data object called a `COIPipeline`. This object facilitates ordered invocation with a FIFO command queue. That thread and pipeline span multiple transactions between the host and coprocessor, enabling offloaded code to preserve thread-local variable values, OpenMP teams, etc. from one offload to another.

Host-side source code	Corresponding offload runtime actions Only highlighted actions are on coprocessor
<pre>double A[SIZE], B[SIZE], C[SIZE]; ... void f() { int transa, transb, N; double alpha, beta; double *A, *B, *C; A = malloc(...); B = malloc(...); C = malloc(...); ... // Define a data region and allocate // memory for A, B and C on the // coprocessor #pragma omp target device(0) map (alloc : A[0:length], \ B[0:length], C[0:length]) { A = ...; B = ... // Xfer the data to the coprocessor #pragma omp target update \ device(0) to (A[0:length], \ B[0:length]) // Perform the computation #pragma omp target device(0) \ map (to : transa, transb, N, alpha, \ beta) {dgemm(&transa, &transb, &N, \ &N, &N, &alpha, A, &N, B, &N, \ &beta, C, &N); } // Transfer the result from the // coprocessor #pragma omp target update \ device(0) from (C[0:length]) // End of data region } }</pre>	<pre>// Coprocessor initialization, on demand by default if ((count = COIEngineGetCount()) <= 0) // fatal error if target unavailable eng = COIEngineGetHandle(0) proc=COIProcessCreateFromFile(eng, startup_image...) COIProcessLoadLibraryFromMemory(proc,codegen_image) thunk = COIProcessGetFunctionHandles(...) pipe = COIPipelineCreate() / // transfer variable mapping info – not shown // Buffer allocation, same for B, C bufAcpu = COIBufferCreateFromMemory(A,bufASize) bufAmic = COIBufferCreate(bufASize) COIBufferAddReference(bufAmic) // Separate data transfer since >30KB; same for B COIBufferCopy(bufAmic, bufAcpu, bufASize 0, 0, NULL) // no input dependencies //Start dgemm on the coprocessor // pack misc_data with name of function calling dgemm memcpy(misc_data, nameOfFunctionCallingDgemm) COIPipelineRunFunction(misc_data, thunk, // offload function in runtime 0, 0, // no buffers 0, 0, // no dependencies &runEvent) // Transfer results from coprocessor to CPU COIBufferCopy(bufCcpu, bufCMic, bufCsize, &runEvent, 1, // input dependencies &outEventC) // completion event dgemm(...), same for B, C COIBufferReleaseReference(bufAmic) // Destroy buffer for A, same for B, C COIBufferDestroy(bufAcpu) COIBufferDestroy(bufAmic)</pre>

Fig. 4. Code example illustrating the use of OpenMP offload extensions in discussion for OpenMP 4.0 RC2

Each offloaded region following an offload pragma is outlined by the compiler into a separate routine having a unique name. The routine's name and address compose a unique entry which is added to a compiler-generated function lookup table.

Function invocation uses indirection, through a thread-safe, coprocessor-side thunk. An invocation helper function is used to invoke coprocessor code that is shared across pipelines. `COIProcessGetFunctionHandles` is used by the host to get a handle for the thunk. The host puts a string with the coprocessor-side function name in a "misc_data" buffer, and uses `COIPipelineRunFunction` for function invocation. Its function arguments include a handle for that thunk, a pointer to misc_data, input buffers, and dependence objects. The resolution of those dependencies, e.g. movement of data down to the co-processor, as checked by `COIPipelineRunFunction`, gates the invocation of the thunk. The thunk extracts the coprocessor function name string from misc_data, looks it up in a function address table, and invokes the outlined function. The compiler-generated outlined function is responsible for demarshaling the data passed during invocation and executing the offloaded region on the coprocessor.

4.4 Memory Management

In the example in Figure 4, a buffer object is created on the host by `COIBufferCreateFromMemory`. This function creates a corresponding buffer on the coprocessor, and the transfer is accomplished with `COIBufferCopy`. Having distinct host and coprocessor buffers, rather than sharing across the PCIe aperture, protects from subsequent modification of the host buffer that could lead to data races.

Coprocessor-side data may be global/static, on the heap, or on the stack. The handling is different for each case. For global/static data, the addresses are already fixed at code generation time, and a mapping table is created upon initialization that the compiler offload runtime uses to set up direct DMAs between host and coprocessor with the correct physical addresses. For heap data, a host-coprocessor address mapping is established and communicated at runtime. In both cases, extra copies through temporary buffers are not necessarily required, and data is persisted as needed.

Each offload invocation creates and destroys its own coprocessor-side stack frame. Coprocessor-side interrupts that occur between invocations use, and hence clobber, the user stack. Because of this, data from the stack frame of one offload function may not persist until the next such invocation. The host-side data that falls within the invoker's scope gets moved and copied onto the coprocessor-side stack upon each invocation and copied back after completion. The programming interface can hide that, thereby emulating persistence, but keeping large variables on the stack isn't recommended for performance reasons. Therefore, use of stack variables for offload is less efficient than using heap or global variables.

Heap data introduces some special issues: allocation and alignment. `#pragma omp` allocates and frees data on coprocessor on the outermost `#pragma omp` region. Data inside nested constructs are implicitly tracked and are ignored if they are also specified in the map clause. To synchronize variables between host and coprocessor inside a `#pragma omp` data construct use the `#pragma omp update` construct. COI uses a reference counting scheme to manage data persistence, which explains the use of `COIBufferAddReference` and `COIBufferRelease-Reference`.

Handling of multiple coprocessors is generally outside the scope of this paper, but it should be noted that the offload runtime supports more than one coprocessor using the target clause. It is entirely the user’s responsibility to synchronize data across different coprocessors, unlike the transparent abstraction offered by MYO [29].

Fortran arrays use dope vectors to describe the number and size of dimensions. This extra 72 or more bytes of metadata must be included in data transfers. At present, with `#pragma omp` directives, only contiguous data transfers are supported, although the base languages permit writing array-slice expression with arbitrary strides.

Table 1. Platform configuration parameters

Host	SNB-EP (2 sockets) 2.6 GHz, Intel® Xeon® E5-2670, Crown Pass Platform
Coprocessor	Pre-production Intel® Xeon Phi™ coprocessor, 61 4-thread cores, 1.09GHz, 5.5GT/s, 8GB (all apps fit)
Host OS	RHEL 6.2, kernel 2.6.32-220.el6.x86_64
Compiler	Composer XE Beta
MPSS	2.1.3653-8, kernel 2.6.34.11

5 Performance Evaluation

When there are multiple architectural families available, like Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors, developers must make a choice as to which is most suitable. The Intel® Xeon Phi™ product family is an extension of the Intel® Xeon® processor family, with different design objectives. A careful consideration of how application characteristics map onto each platform helps direct users to perform a theoretical or empirical analysis of the app, to optimize accordingly.

Performance of offload runtime features is evaluated in two ways: at the directed test level and at the workload level. Directed tests measure the impact of a particular feature in isolation; workloads measure the impact of that feature in the context of a workload. Evaluation at the workload level focuses on overall overheads and profitability. We first evaluate overall profitability, and then deep-dive into the impact of specific features. The evaluations below were based on the `#pragma offload` form of directives, on the platforms described in Table 1. The performance of the `#pragma omp` form is materially identical.

5.1 Offload Profitability

Table 2 provides a summary characterization of a few workloads, and the numerical analysis below is based on that data only, vs. being a general characterization of customer code. The workloads were selected to span a variety of application domains. They came from customers, and are meaningfully representative to them. Some workload names are occluded in deference to customers. The first row of data shows the overall speedup attained using the coprocessor with the offload runtime, relative to host-only execution. Workloads in this paper use OpenMP, but not MPI.

Table 2. Offload performance and overheads for workloads							
Workload	3DFD PDE Stencil	Convolution Resampling	Hogbom-Clean	QR	Iterative closest point, DP	Adaptive Sparse Grid	Black Scholes Compute SP
Domain	Seismic	Astronomy	Astronomy	Physics	Manufac- turing	Physics	Financial
Speedup (x): offload vs. host only	2.03	1.56	2.31	1.40	1.54	1.32	6.92
Compute % of total execution	97.7	44.3	72.9	97.3	94.5	95.3	99.4
Host offload overhead (x), with (top) & without (bottom) init	0.21	1.77	0.44	0.03	0.02	0.06	0.00
	0.18	1.60	0.25	0.01	0.01	0.05	0.00
Computation/communication ratio	5.42	0.62	3.99	68.6	90.1	19.7	1640

The percentage of total execution time that is offloaded computation varies from 44% to 99%. The overhead for moving data and invocation varies from being negligible to being a 1.77x multiple of the computation time. Coprocessor-side overheads were measured but are not shown, since they are negligible. The ratio of computation to communication ranges from 0.62x to 1640x. It was generally over 4x, except for the convolution case, whose scaling across threads and SIMD elements on the coprocessor allows the computation speedup to outweigh the overheads.

Black Scholes, which has the highest computation to communication ratio, the highest offload fraction, and the lowest overheads, is the big performance winner at 6.92x speedup of a dual-socket Sandy Bridge. Recall that the offload speedup depends on several factors, so it is sometimes but not always highest when the computation to communication ratio is high. To illustrate that point, there was temporary compiler regression in which the coprocessor execution time shot up, and the speedup for Adaptive Sparse Grid fell to parity between host only and offload.

The impact of communication and offload overheads can be partially hidden with the use of asynchronous invocation. Using the asynchronous form of `#pragma offload` (see [18]) benefitted SHOC [30] Triad by 1.65x (3.16/1.91), as shown by comparing the two columns at the right of Table 3. That speedup didn't come from reducing the offload overhead, which was only a 1.11x (8.51/7.67) improvement; it was from overlapping computation with communication, only some of which (0.96/0.72=1.33x) can be captured by the available runtime stats.

Table 3. Offload performance and overheads and native performance for SHOC

Workload	Triad Async	Triad Sync	SPMV-SP	SPMV-DP	Sort	SCAN	S3D-SP	S3D-DP	Reduction-SP	Reduction-DP	MD-SP	MD-DP	GEMM-SP	GEMM-DP	FFT-SP	FFT-DP
Data set	4096KB	4096KB	2684354_elements	4007383_elements _62451_rows	4007383_elements _62451_rows	8388608_items	262144_gridPoints	262144_gridPoints	16777216_items	8388608_items	73728_atoms	73728_atoms	4096 x 4096	2048 x 2048	N= 33554432	N= 16777216
Speedup with of-fload (top) and native only	3.16	1.91	0.07	0.07	0.21	0.25	1.03	0.95	0.20	0.20	0.44	0.81	3.25	3.30	0.45	0.30
	NA	NA	0.62	0.80	0.71	4.29	1.37	1.40	4.25	4.14	1.20	1.94	3.88	4.47	5.13	3.33
% execution time in offload	5.8	3.4	6.91	8.22	17.4	68.0	6.92	7.53	19.0	37.3	0.26	19.3	59.1	33.0	3.62	3.58
Host of-fload overhead, with & without init	7.67	8.51	6.99	6.09	3.66	0.19	9.45	9.75	2.42	0.92	17.1	2.42	0.34	1.08	14.8	13.9
	1.04	1.39	1.54	1.60	2.47	0.02	3.09	4.96	0.48	0.41	0.54	0.52	0.07	0.13	1.09	1.20
Computation: communication	0.96	0.72	0.65	0.63	0.40	56.9	0.32	0.20	2.10	2.46	1.85	1.92	14.0	7.74	0.92	0.83

Offload runtime and application performance tends to improve with tuning. For example, allocation and initialization time, which depend on the amount of memory to be initialized, was reduced from several seconds down to between 0.25 and 2.5s for the workloads in this paper. As shown in the SHOC [30] data, allocation and initialization costs are still significant for small kernels. Further efforts to hide those costs is expected to bring them down.

5.2 Runtime Performance Feature Impact

As mentioned above, the performance of the offload runtime in initialization, data transfer, and invocation can determine whether offload is profitable. This section analyzes the most important offload compiler runtime performance features: transfer avoidance, copy avoidance and page size.

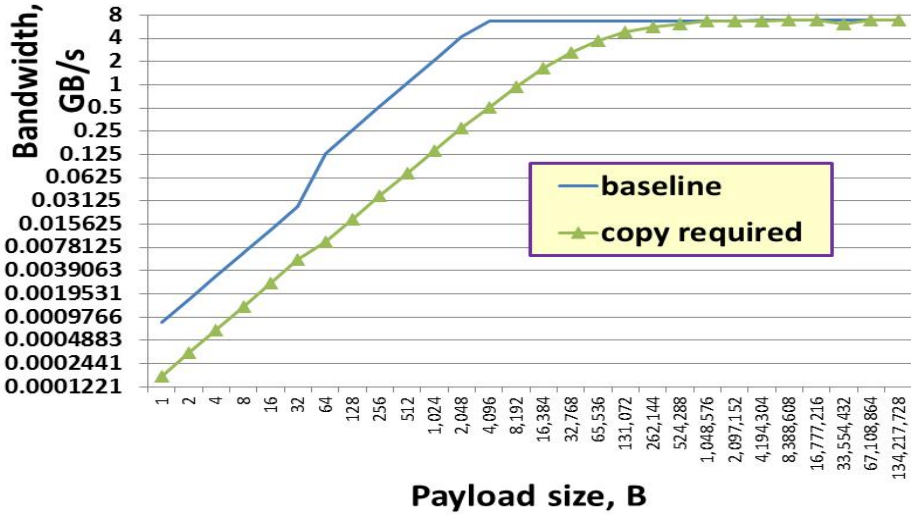


Fig. 5. DMA performance, as a function of payload size, for host to coprocessor transfers. For the baseline, the destination memory location is known. The other curve shows the cost of an extra copy.

When data can be persisted, there may be no need to transfer data at all. Consider the case of a temporary variable that is only written to on the coprocessor side, is never used on the host side, but that is needed by several offloaded functions. If that variable is static, it never needs to be copied back and forth across the PCIe, but if it lives on the stack of an enclosing function, it must be copied in and updates must be copied back to the host in each function. Notice, in Tables 2 and 3, that the ratio of computation to communication can be low enough that eliminating communication can significantly impact performance.

The cost of an extra copy is shown in Figure 5, where the median speedup from copy avoidance is 2.6x and the mean ratio is 3.9x. The baseline curve is for transfers whose destination address is already known to the host, that use 2MB pages and that otherwise meet the optimal conditions.

When large amounts of data must be transferred between the host and coprocessor, the hardware DMA (direct memory access) engine is used. We describe three cases where a direct transfer without extra copies to the final destination may be impossible.

The first case is mutual misalignment between host and coprocessor offsets of addresses within a cache line. For example, there is mutual misalignment when the modulus of differences in global/static addresses isn't 0, or when a host address is an odd multiple of 32, and there is a `#pragma offload` with an `align` clause value of 64. This is important because mutual alignment within a 64B cache line enables a direct DMA into the destination, whereas mutual misalignment within a 64-byte cache line on the initial production member of the Intel® Xeon Phi™ product family isn't supported by the DMA engine. Without mutual alignment, an extra buffer copy is necessary, incurring the costs quantified below. Static data can have mutual misalignment within a cache line. By default, the alignment within a cache line of the coprocessor-side heap data is set to match that of the host-side data.

In the second case, data that is resident on the stack has to be copied from a pinned DMA buffer onto the coprocessor's stack. In the third case, when the destination physical address is not known to the host prior to the DMA, there is a DMA to a temporary buffer, followed by a copy to the address which is already known to the coprocessor code. The compiler offload runtime avoids that copy by sending the coprocessor variable memory locations at initialization.

The overall median improvement of 2MB vs. 4KB pages for several customer work-loads is 1.051x and the average is 3.20x. Almost all of those workloads benefited from large pages, whose use is enabled by the offload runtime environment variable. The average gain for using large pages in DMA transfers is only 1.064x. See [18] for more detail.

6 Summary and Conclusions

We demonstrated the potential for making offload to the Intel® Xeon Phi™ coprocessor profitable. We showed sample workloads with speedups over a dual-socket Intel® Xeon® E5-2670, codenamed Sandy Bridge, in the 1.3x to 6.9x range as customer-relevant examples that span different application domains. But offload is not always profitable, as the SHOC data showed. The paper explores what can enhance or inhibit offload speedup. First, where response time is of concern, there must be a speedup from native execution on only the coprocessor, relative to execution on the host. This is based on the performance of the code on coprocessor and system, such as whether it's highly threaded, and either uses SIMD well or its bandwidth demand exceeds what's available on the host. Second, the ratio of computation to communication must generally be high. In the workloads evaluated, that ratio ranged from 4x to 1640x for all but one case. This depends on characteristics of the application, and how it is structured for offload. For example, a simple restructuring of SHOC Triad to use of asynchronous offload boosted performance. Third, the offload runtime overheads must be small relative to the computation. This depends on the offload runtime implementation.

Runtime offload overheads ranged from negligible to a factor of 1.77x of the actual computation time, on sample workloads for which offload is profitable. These overheads are important to performance; an optimized implementation of the offload compiler runtime is a key aspect of platform performance for offloaded apps. This paper highlights the key facets of the offload compiler runtime implementation which impact performance, describes those performance features, and evaluates their performance. The performance impact of the offload compiler runtime performance features range from a few percent for page size, up to a mean across payload sizes of 3.9x for incurring an extra copy in cases of stack residency, DMA mutual misalignment, or an unknown DMA destination address.

This paper provides the first description of the Intel® Xeon Phi™ coprocessor software architecture, and serves as a guide both to end users wanting to evaluate whether offload is likely to be profitable, and to third-party implementers of heterogeneous runtimes for Intel® Xeon Phi™ coprocessors and potentially other targets seeking guidance in design principles and in what's most important to optimize.

The authors would like to acknowledge many contributors to this paper, who include implementers of the tools and system software described herein, and the many

applications engineers who tuned and measured performance on customer codes. We thank the customers who shared their workload examples, and reviewers who helped improve the presentation of the material.

References

1. Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S.: Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects SciDAC 2009: Scientific Discovery through Advanced Computing, San Diego, California. Journal of Physics: Conference Series, vol. 180, p. 012037. IOP Publishing (2009)
2. Budruk, R., Anderson, D., Shanley, T.: PCI Express System Architecture, 1st edn., 1120 pages (2003) ISBN 978-0-321-15630-3
3. CAPS, <http://www.caps-entreprise.com/technology/hmpps>
4. Denning, P.J., Schwartz, S.C.: Properties of the Working-Set model. Communications of the ACM 15, 191–198 (1972)
5. Donaldson, A.F., Dolinsky, U., Richards, A., Russell, G.: Automatic offloading of C++ for the Cell BE Processor: A case study using offload. In: Proceedings of the 2010 International Conference on Complex, Intelligent and Software Intensive Systems, pp. 901–906 (2010)
6. Green 500: The Green500 List (November 2012), <http://www.green500.org>
7. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message Passing Interface, 2nd edn. MIT Press, Cambridge (1999)
8. Gropp, W., Lusk, E., Thakur, R.: Using MPI-2: Advanced Features of the Message-Passing Interface. MIT Press, Cambridge (1999)
9. Intel® C/C++ compiler, <http://www.intel.com/Software/Products>
10. Intel® Many Integrated Core, <http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html>
11. Intel® Many Integrated Core SW development pages, <http://software.intel.com/mic-developer>
12. Intel® Math Kernel Library, <http://www.intel.com/Software/Products>
13. Intel® Message Passing Interface, <http://software.intel.com/en-us/intel-mpi-library/>
14. Intel® OpenCL for Intel® Xeon Phi™ Coprocessor, <http://software.intel.com/en-us/vcsources/tools/opencl-sdk-xe>
15. Jeffers, J., Reinders, J.: Intel® Xeon Phi™ Coprocessor High Performance Programming. Morgan Kaufmann (2013)
16. Khronos, <http://www.khronos.org/opencl/>
17. MAGMA, <http://icl.cs.utk.edu/magma/>
18. Newburn, C., Deodhar, R., Dmitriev, S., Murty, R., Narayanaswamy, R., Wiegert, J., Chin-chilla, F., McGuire, R.: Offload Runtime for the Intel® Xeon Phi™ Coprocessor, <http://software.intel.com/en-us/articles/offload-runtime-for-the-intelr-xeon-phitm-coprocessor>
19. Numerical Algorithms Group, Ltd., <http://www.nag.com/>

20. NVIDIA CUDA reference manual, version 5.0 (October 2012),
http://docs.nvidia.com/cuda/pdf/CUDA_Toolkit_Reference_Manual.pdf
21. OpenACC, <http://www.openacc-standard.org/>
22. OpenMP (March 2013),
http://www.openmp.org/mp-documents/OpenMP_4.0_RC2.pdf
23. OpenMP (November 2012),
http://www.openmp.org/mp-documents/TR1_167.pdf
24. Patterson, D., Hennessey, J.: Computer Organization and Design: the Hardware/Software Interface, 2nd edn., p. 751. Morgan Kaufmann Publishers, Inc., San Fran (1998)
25. Rabenseifner, R., Hager, G., Jost, G., Keller, R.: Hybrid MPI and openMP parallel programming. In: Mohr, B., Träff, J.L., Worringer, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, p. 11. Springer, Heidelberg (2006)
26. Ravi, N., Yang, Y., Bao, T., Chakradhar, S.: Apricot: An optimizing compiler and productivity tool for x86-compatible many-core coprocessors. In: Proc. of the 26th ACM International Conference on Supercomputing, pp. 47–58. ACM, New York (2012)
27. Redhat, <http://www.redhat.com/products/enterprise-linux/>
28. Reinders, J., <http://parallelbook.com/blogs/james>
29. Saha, B., Zhou, X., Chen, H., Gao, Y., Yan, S., Rajagopalan, M., Fang, J., Zhang, P., Ronen, R., Mendelson, A.: Programming model for a heterogeneous x86 platform. SIGPLAN Not. 44(6), 431–440 (2009)
30. SHOC 1.1.1 manual,
http://ft.ornl.gov/doku/_media/shoc/shoc-manual-1.1.1.pdf
31. SUSE, <https://www.suse.com/promo/sle11.html>
32. Threading Building Blocks, <http://threadingbuildingblocks.org>