Understanding the Costs of Many-Task Computing Workloads on Intel Xeon Phi Coprocessors

Jeff Johnson, Scott Krieder, Benjamin Grimmer, Ioan Raicu
Illinois Institute of Technology
{jjohns14, skrieder, bgrimmer}@hawk.iit.edu, iraicu@cs.iit.edu

Justin Wozniak, Michael Wilde
Argonne National Laboratory
{wozniak, wilde}@mcs.anl.gov,

ABSTRACT
Many-Task Computing (MTC) aims to bridge the gap between HPC and HTC. MTC emphasizes running many computational tasks over a short period of time, where tasks can be either dependent or independent of one another. MTC has been well supported on Clouds, Grids, and Supercomputers on traditional computing architectures, but the abundance of hybrid large-scale systems using accelerators has motivated us to explore the support of MTC on the new Intel Xeon Phi accelerators. The Xeon Phi is a PCI-Express based expansion card comprised of 60 cores supporting 240 hardware threads to produce up to 1 teraflop of double-precision performance in a single accelerator. These cards are already being integrated into super-computing clusters such as Stampede which hosts over 6,400 Xeon Phi Accelerators totaling in over 7 petaflops of double-precision performance. This work provides an in depth understanding of MTC on the Intel Xeon Phi and presents our preliminary results of running several different workloads on pre-production Intel Xeon Phi hardware. By utilizing Intel's provided SCIF protocol for communicating across the PCI-Express bus we have achieved over 90% efficiency near or outperforming OpenMP offloading tasks over 300 uS with our batch framework. This performance opens the opportunity for the development of a framework for executing heterogeneous tasks on the Xeon Phi alongside other potential accelerators including graphics cards for MTC applications. Our framework will provide fine granularity for executing MTC applications across large scale compute clusters. It will be integrated with our existing graphics card framework, GemTC, to provide transparent access to GPU's, Xeon Phi's, and future generations of accelerators to help bridge the gap into exascale computing

Keywords: *MIMD,MTC, Accelerator, Intel Xeon Phi, Coprocessor*

1. INTRODUCTION
In this work we provide preliminary results evaluating MTC workloads running on Intel Xeon Phi Coprocessors. The Intel Xeon Phi Coprocessor is physically similar to other hardware accelerators such as GPGPUs but contains many significant underlying differences.

The High Performance Computing (HPC) community is seeing a large adaptation of general-purpose accelerator cards. In the past years [X], graphics cards have become a significant part of newly built computing clusters providing unprecedented parallelism with a low power footprint. Accelerators often require the use of reworking a program in order to use or fully utilize the device and as a result increase development time.

The pre-production Xeon Phi is a 61-core accelerator featuring 8 GB of GDDR5 connected to the host via a PCI Express bus. The Phi runs an instance of the Linux operating system, which occupies a single core of the accelerator to provide the developer with a familiar programming interface. As of the pre-production Xeon Phi, it is possible to use OpenMP, POSIX threads, OpenCL, Intel Math Kernel Library, MPI, or other popular libraries to develop and offload applications to the accelerator. Intel also provides a socket protocol, SCIF, to aid in data transfer between the host and the Phi to transfer data across the PCI Express bus. Intel also provides the application 'micnativeloadex' to launch an application compiled for the Phi on the accelerator. This application ensures the correct libraries and program code are copied and executed on the accelerator.

2. ARCHITECTURE
In order to provide a seamless and easy to use interface, our framework sets out to provides identical functionality to GeMTC. GeMTC has three types off operations: Push/Pool for sending and receiving jobs, Malloc/Free for preparing device memory, and a memory copy operation to copy data to or from the accelerator. By providing this interface we can easily tie in with Swift/T to open up our solution to multi-node configurations.

2.1 SCIF Implementation
Our SCIF framework employs a client server architecture, which communicates via the Symmetric Communications Interface (SCIF). This interface abstracts communication across the PCI-Express bus to a UNIX socket semantics. The server application uses a single process launched on the Intel Xeon Phi, which then launches a single processing thread per hardware core to handle incoming work from the host. The client starts from a single process and launches no more threads than the server. Each server thread listens via SCIF on a discrete port, which is used to accept and return work to the client.

Each client thread produces a batch of work to the accelerator, which can contain a heterogeneous list of tasks with varying lengths and data payload sizes. The client blocks until the entire batch is completed.

This architecture allows the overhead of launching threads as well as offloading the code to be executed to be ignored and performance becomes only dependent on data transfer rate and processing rate of the processor.

This framework does not provide the feature set of GeMTC due to the complexity of data transfer between the device and host over SCIF. Each party in the communication is forced to

manually parse out received streams of data which is a highly error prone operation when dealing with complex lists of tasks.

2.2 Proof of Architecture

To prove the functionality of the architecture on the Intel Xeon Phi, a OpenMP based framework is constructed in addition to the basic SCIF framework. This sets out to provide the identical functionality as GeMTC mentioned previously in the paper. The framework uses an incoming and outgoing queue stored on the host which contains job descriptions pushed by a consumer of the API. The framework spawns 60 threads on the host which run a super kernel. This super kernel pools the incoming queue waiting for a task to execute. Once a task is placed in the queue a free worker performs the execute_job function which performs the actual offloading of the code.

2.2.1 Challenges in Implementation

Using OpenMP to perform offloading of our abstract interface into the task execution does not trivially work with the design of OpenMP. The largest challenge with this is performing memory copies to and from the device while providing a device pointer to the host. In order to do this, malloc is called in an offloaded OpenMP section which returns a pointer on the device. The memory address of this pointer is cast to a long data type and returned to the API. The API places this inside of a special datatype which keeps track of the pointer and is passed whenever referencing the devices memory.

In OpenMP offloading situations it is customary to define what memory is required for the operation in the pragma of the offloading call. In our configuration where we aim to provide a device pointer this is not possible. As a result, we must use memcpy on the device in an offload section similar to the work done in malloc. The following is a novel method for performing a memory copy from the device to the host:

```
#pragma offload target(mic:MIC_DEV) in(device: alloc_if(1)
free_if(0)) out(host: alloc_if(1) free_if(0))
{
            memcpy(host, device->mic_payload, size);
}
```

The pragma tags prevent the system from prematurely freeing the memory being copied and ensure they are allocated on entrance of the offloaded code. This technique provides a way to provide the same semantics as GeMTC by providing memory access with a pointer that seems to be directly to the device.

3. PRELIMINARY RESULTS

In this section we present preliminary results for running several synthetic benchmarks on pre-production Intel Xeon Phi hardware.

3.1 Sleep Efficiencies

To compare the overhead of our architecture with OpenMP three sleep programs are analyzed. The first uses our SCIF framework to launch a batch of sleep jobs to the Xeon Phi and block until completion then separately tests launching individual jobs to the device and waiting for the result before pushing the next job. The second uses OpenMP to offload a single block, which calls a sleep job 128 times by repetitively placing the function call in the block. This method was chosen to more accurately mimic the loop-unrolling style of our SCIF framework. The final program uses OpenMP to offload a single sleep task and is repeatedly called. Figure 1 demonstrates the results of comparing all four different configurations in varying sleep lengths. By testing varying lengths of sleeps, we find that jobs over 320 uS benefit from the SCIF framework when sent in this length of a batch. At this point the performance is slightly above OpenMP but can enjoy the potential benefits of the framework.

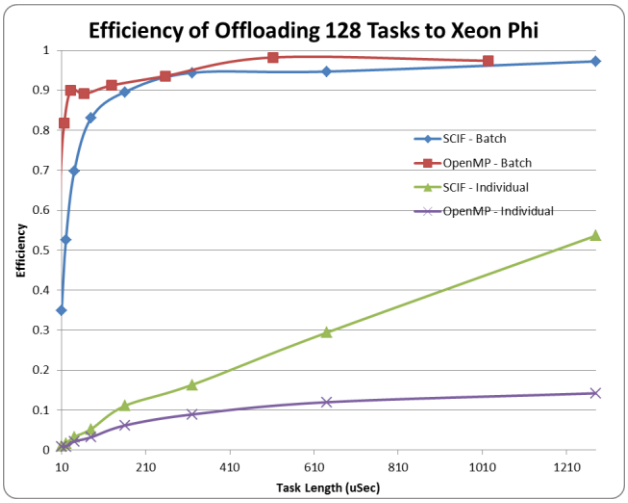

Figure 1: Efficiency of Offloading 128 Tasks to Xeon Phi Comparison between OpenMP and SCIF with individual offloads and batch offloads

3.2 Matrix Multiplication

The following test is performed using the SCIF framework with varying concurrent executing threads and data sizes on the performance of the Matrix Square operation containing floating-point numbers.

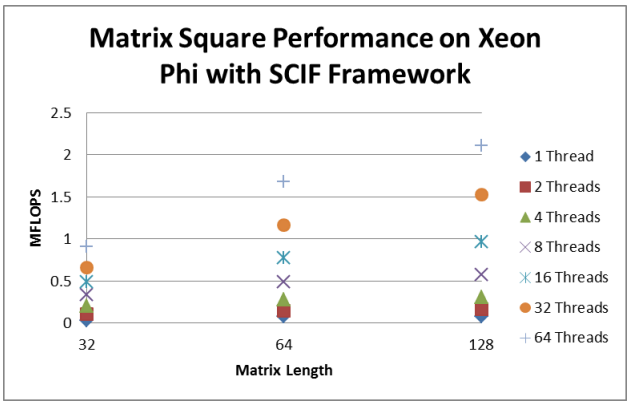

Figure 2: MFLOP Performance of Matrix Square Operation Running on Intel Xeon Phi with SCIF Framework.

The results of our Matrix Square tests show that our performace scales well with increased cores and increased workloads although offering overall low performance.

For comparison, the Matrix Square tests are executed on the host machine in Figure 3.

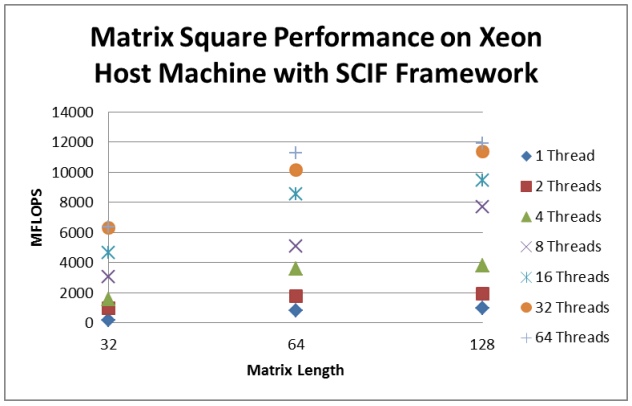

Figure 3: Matrix Square Performance on 24-Core Xeon Machine with SCIF Framework

An extremely significant increase in performance. In this case the Matrix Square operation performs a significant amount of computation on the data. This result shows that the current implementation is likely not taking advantage of the Xeon Phi's vector operations to handle large sets of data at once. The device is only properly utalized when it is able to perform large operations which is not seen in a naieve implementation of matrix multiply.

3.3 Memory Transfers

To analyze the memory transfer rate of the Xeon Phi a client/server application uses SCIF to write a payload to the MIC then read it back to the host. Figure 4 shows the results of this benchmark.

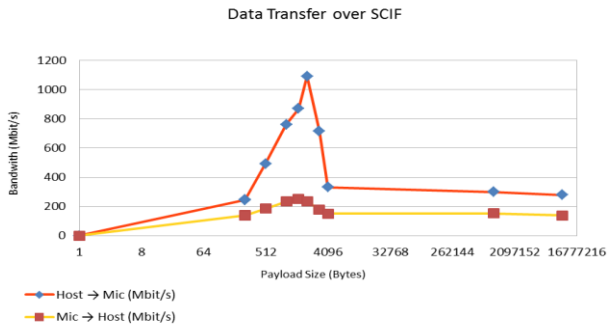

Figure 4: Memory Transfer Benchmark across SCIF Interface

The peak seen in the data transfer benchmark demonstrates a potential sweet spot for performance that our framework can optimize for. This peak can be atributed to the data size able to be transferred across the PCI-Express bus or may be a result of an alignment in cache size with the host or accelerator.

3.4 Vector Add

The following demonstrates a vector add operation of a height of 10 with varying lengths and varying threads.

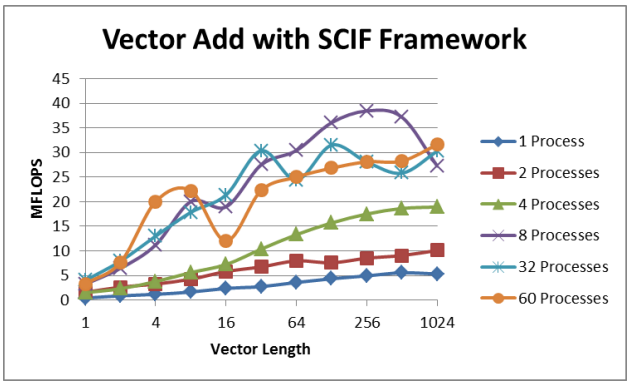

Figure 5: Vector Add with SCIF Framework on Intel Xeon Phi

Again we see inrease in performance with concurrent threads and increased data size.

Running the same benchmark with the vector add being processed on the host across the SCIF framework we see a significant performance increase on the host machine as seen in Figure 6.

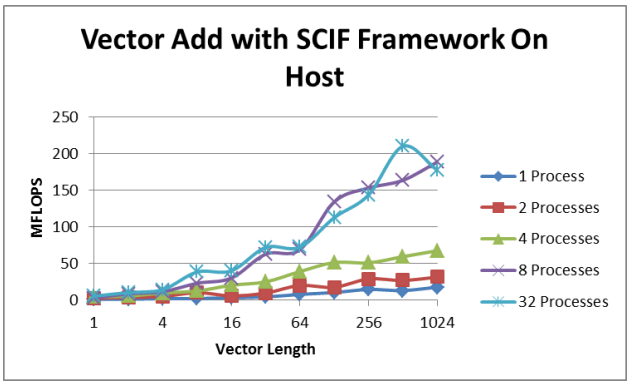

Figure 6: Vector Add with SCIF Framework on 24-Core Xeon Machine

The significant increase in performance likely caused by this type of workload which includes little processing time with high data trasnfer. The SCIF framework requires singificantly more time to transfer data to the MIC then it does to keep the information on the host. This result shows that data bound tasks are not appropriate for this type of offloading.

3.5 OpenMP: MDProxy

The proposed architecture is implemented in OpenMP and three microkernels are developed to mimic the functionality of MDProxy. MDProxy simulates a set of particles in space with varying simulation steps and dimensions. The result of this simulation is caused by the collisions of the particles with determine each steps values for the particles position, velocity, and energy. The simulation was run on the host machine instead of the Xeon Phi because of issues with the memory transferring between the device and host. This demonstrates a problem with our method of performing data transfers and providing a virtual memory pointer from the API. For comparison a Intel Xeon 24-Core machine is compared with an NVIDIA GPU in Figure 7.

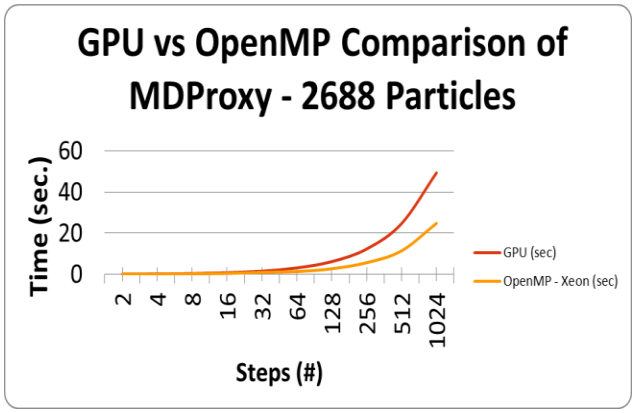

Figure 7: GPU (NVIDIA) vs OpenMP (24-Core Xeon) Comparison of MDProxy with 2688 Particles

As the workload increases and the number of steps increase, the OpenMP implementation running on the host shows its ability to outperform the graphics card implementation of the same algorithm. This result demonstrates the functionality of our architecture and demonstrates the possibility for a SCIF based implementation in the future.

4. RELATED WORK

The Xeon Phi is a very new technology and research is just beginning to show the usefulness of the technology. IRWTH Aachen University and Intel have collaborated on a paper outlining the initial performance of the preproduction Xeon Phi demonstrating the power of its dense processing using OpenMP related to a traditional 128-core system [7]. It was shown that a single Phi had a lower overall performance but the power per core and power efficiency significantly outweighs the latter solution towards many-core. Intel also advocates the offloading capabilities of its compiler and OpenMP with use of the Phi showing that many OpenMP programs can be launched on the Phi without code change [8].

5. Proposed SCIF Framework

The following framework is suggested to build on the architecture of the previously mentioned OpenMP/GeMTC framework. This framework will use SCIF for communication between the host and an Intel Xeon Phi.. OpenMP will be used to offload the server section of the code so that the API can be easily consumed without extra configurations and the API can be ran on a machine lacking a Xeon Phi for development. This framework is to handle multiple Xeon Phi cards on a single machine and handle the scheduling between them. Figure 8 demonstrates the client server architecture and the proposed usage with Swift/T as a driver to the framework.

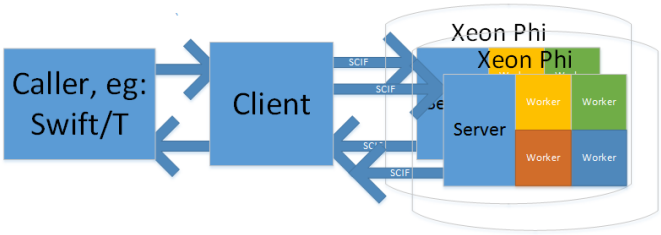

Figure 8: Proposed Architecture of a SCIF Based Offloading Framework for Swift/T Integration

The Xeon Phi runs a single server instance which hosts a thread per core for each worker on the device that executes meaningful work. Each thread is capable of processing unique work from other devices as well as sharing memory between multiple cores if desired. This flexibility should allow for any GeMTC program written for a graphics accelerator to be executed on the Intel Xeon Phi by implementing any proprietary microkernels and without modification to the driver program.

6. CONCLUSIONS & FUTURE WORK

It has been demonstrated that is it possible to achieve minimum overhead with the Xeon Phi by directly communicating between the host and accelerator via SCIF across the PCI Express bus. Using our proposed framework, it will be possible to share the resources of a Xeon Phi across multiple processes and users in a large scale computing environment while maintaining high performance through the use of specialized microkernels. Enabling the Xeon Phi to run heterogeneous workloads can enable the device to be used in a Many-Task Computing environment where resources are shared between users and latency is important. The findings of this paper shows no significant performance tradeoff to the framework opening the way for a promising use of the Xeon Phi in new environments.

7. REFERENCES

[1] S. Krieder and I. Raicu, "Towards the Support for Many-Task Computing on Many-Core Computing Platforms," Doctoral Showcase, SC 2012

[2] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Comput.*, vol. 37, pp. 633-652, 2011

[3] J. Lange, *et al.*, "Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing," in IEEE *Parallel & Distributed Processing (IPDPS), 2010*, pp. 1-12

[4] I. Raicu, I. T. Foster, and Z. Yong, "Many-task computing for grids and supercomputers," in *Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Workshop on*, 2008, pp. 1-11

[5] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster, "Swift/T: Large-scale application composition via. distributed-memory data flow processing.," presented at the CCGrid, 2013

[6] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, pp. 187-198, 2011

[7] T. Cramer, D. Schmidl, M. Klemm, and D. Mey, "OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison," (2012).

[8] C. J. Newburn, R. Deodhar, S. Dimitriev, R. Murty, R. Narayanaswamy, J. Wiegert, and R. McGuire, "Offload Compiler Runtime for Intel Xeon Phi Coprocessor" (2012)