TARUC: A Topology-Aware Resource Usability and Contention Benchmark

Gavin Baker
Sandia National Laboratory
Livermore, California 94550
gmbaker@sandia.gov

Chris Lupo
California Polytechnic State University
San Luis Obispo, California 93407
clupo@calpoly.edu

ABSTRACT

Computer architects have increased hardware parallelism and power efficiency by integrating massively parallel hardware accelerators (coprocessors) into compute systems. Many modern HPC clusters now consist of multi-CPU nodes along with additional hardware accelerators in the form of graphics processing units (GPUs). Each CPU and GPU is integrated with system memory via communication links (QPI and PCIe) and multi-channel memory controllers. The increasing density of these heterogeneous computing systems has resulted in complex performance phenomena including non-uniform memory access (NUMA) and resource contention that make application performance hard to predict and tune. This paper presents the Topology Aware Resource Usability and Contention (TARUC) benchmark. TARUC is a modular, open-source, and highly configurable benchmark useful for profiling dense heterogeneous systems to provide insight for developers who wish to tune application codes for specific systems. Analysis of TARUC performance profiles from a multi-CPU, multi-GPU system is also presented.

1. INTRODUCTION

Non-uniform memory access (NUMA) as well as communication link bandwidth contention can reduce performance for applications used in heterogeneous computing system topologies. This paper presents *TARUC*, a new Topology Aware Resource Usability and Contention benchmark that utilizes memory and thread pinning techniques to provide detailed performance profiles of how system topology and usage patterns affect memory performance. The benchmark is tested on an x86 computing system and the results and experiences gained may be used to inform application developers about best practices for manual control of computing resources.

While parallel computational performance has improved following the trend predicted by Moore's Law, the bandwidth of memory systems and the communication links that connect peripheral coprocessor cards to main memory or network fabrics have both failed to match increasing demand for bandwidth [5, 7]. The existence of variable bandwidth and latency communication paths between CPUs, GPUs, main memory, and the network fabric has resulted in a hierarchical communication topology along with NUMA. NUMA effects can result in longer latencies for critical path operations which in turn can reduce the overall performance of the application [8]. TARUC is designed to help HPC application developers understand system topology and potentially mitigate the effects that variable topologies have on application performance.

The remainder of this paper is organized into the following sections: 2) discussion of modern system level architecture, 3) discussion of the design and implementation of the TARUC benchmark, 4) review of results acquired on a HPC test system, 5) related work on the topic of resource management of heterogeneous computing systems, 6) a summary of the most significant conclusions and topology trends and a review of continuing effort in this research area.

ICPE'17, April 22 - 26, 2017, L'Aquila, Italy

ACM ISBN 978-1-4503-4404-3/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3030207.3030230

2. BACKGROUND

For this paper, we focus on x86-based microprocessor computing systems that utilize coprocessor hardware to explore the architecture of densely packed computing systems. These systems comprise the majority of supercomputers [10].

2.1 NUMA

One solution for scaling beyond a single CPU is to add multiple CPU sockets to a single computing system. Separate DIMMs and corresponding memory management units (MMUs) are often provided for each CPU socket. To ensure cache and memory coherence, each CPU is connected together via a high-speed low-latency cache coherence link. One side effect is that some memory accesses require extra time/latency due to the cache coherence link. Cache coherency links are high bandwidth and low latency compared to other peripheral connections, but still fail to match either the bandwidth or latency of having a single memory bus connection between main memory and CPUs.

The effect of having memory spaces connected to cores with differing latency and bandwidth connections is called non-uniform memory access (NUMA).

2.2 Modern x86 System Architecture

Modern x86 computers typically consist of one or more CPUs, main memory, persistent storage, and a number of I/O interfaces. Each CPU integrates both memory control and high speed peripheral communication links onto a sin-

gle package. Processors can contain multiple cores per package and multiple processing units per core in simultaneous multithreaded (SMT) architectures. The sharing of memory resources can be of concern in NUMA systems because of the potential for unbalanced memory usage where many cores access the memory or cache of a specific NUMA node. Figure 1 shows a basic x86 system with main memory, I/O, and multiple cores.

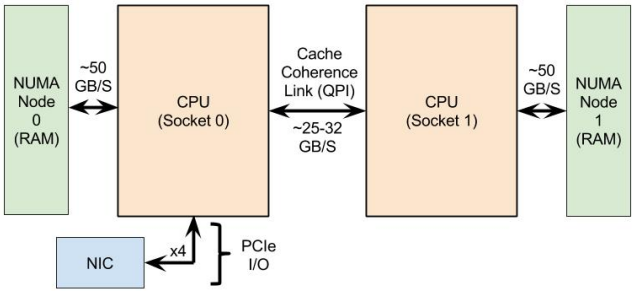

Figure 1: An x86 Cache Coherent NUMA Machine

For multiple CPU systems, cache coherency links are used to maintain the shared memory programming model guaranteed by today's symmetric multiprocessing systems. QPI is utilized on Intel processors to connect the memory subsystems of two or more CPUs within a single machine. The bidirectional bandwidth of QPI ranges from 25-32GB/s [4]. Memory access on NUMA systems may require the request to travel from one CPU over the QPI bus to the MMUs of an adjacent processor. This transaction is transparent to the application, other than the performance reduction for repeated remote access.

Peripheral devices such as NICs and GPU or Xeon Phi coprocessors may be attached via a PCIe connection. The current generation PCIe 3.0 is a point-to-point communication link that can dynamically scale between 2, 4, 8, or 16 lanes between a single device and host. Each lane can carry out 8.0GT/s (giga transfers per second) for a total bidirectional bandwidth of 32GB/s [6]. PCIe combined with integrated memory control hardware on the CPU allow for direct address (DMA) of pinned memory by PCIe devices contained on node-local PCIe slots.

2.3 Memory, Thread, and Process Management

The Linux operating system allows for the setting of memory and thread management policy to control the hardware location of allocated resources. Manual control of the location of thread or process execution can be carried out by modifying the thread's individual hardware affinity. The allocation policy of memory can be set to *firsttouch*, *bind*, *interleave*, and *nexttouch* with each policy defining where a block of memory will be placed upon allocation or use. In the TARUC benchmark, the *bind* policy is used to ensure correct memory NUMA locations during tests.

2.4 Programming Parallel Architectures

A number of software libraries allow for the use of parallel hardware through different software abstractions. Most significantly OpenMP, OpenAcc, OpenCL and CUDA provide some type of node-local software parallelism. TARUC uses OpenMP and CUDA to test heterogeneous systems for NUMA effects. This choice was made because of the flexibility and vendor support for these libraries provided on our test systems. OpenMP enjoys widespread compiler support and relatively low overhead. Compiling with OpenMP is a matter of adding relevant threading directives, the header library declaration, and linking flags. For GPU functionality, the CUDA runtime API version 7.5 is utilized. CUDA was chosen because it provides thorough functionality including a variety of host-device memory management operations and has been optimized for the NVIDIA coprocessors present on our test systems.

3. THE TARUC BENCHMARK

The Topology Aware Resource Usability and Contention (TARUC) Benchmark consists of a number of micro-benchmarks used to measure communication and memory access bandwidth within complex heterogeneous architectures.

The TARUC benchmark examines NUMA and contention effects in the context of memory transfer and operation bandwidth during single threaded and multithreaded memory migrations. Threads and memory resources are pinned to simulate specific resource utilization situations and measure performance influences. TARUC tests the effectiveness of memory types and copy methods in specific hardware allocation situations. Memory types include device memory, host pageable, host pinned, and host write-combined memories while migration methods include manual unified memory and asynchronous memory copies. In addition to built-in memory transfers, a variety of manual memory access patterns including copy assignments and triads (copy-/scale/add) are tested for throughput and latency.

TARUC examines resource contention within the memory access pipeline by profiling simultaneous memory transfers and access operations between memory resources on different computing resources. In contrast, NUMA effects and other overhead costs of utilizing different system memory types are examined in isolated, single-threaded allocations, transfers, and operations.

TARUC supports analysis of NVIDIA GPUs using the CUDA runtime along with system topology detection and control through the HWLOC library. Host multi-threading is provided by calls to OpenMP parallel tasks. HWLOC also provides memory and thread resource pinning functions through integration with pthreads and libnuma, allowing isolation of individual computing resources within benchmark test cases. Each recorded micro-benchmark data set is automatically fed into graphing scripts that iteratively generate a variety of comparative plots for performance analysis. The entire benchmark may be run with a single command and tuning is provided through a simple input parameter file. The TARUC benchmark may be downloaded from the TARUC_Bench GitHub repository located at the following URL:

https://github.com/gabaker/TARUC_Bench.git

3.1 Specification

The TARUC benchmark is defined by its feature set, software and hardware requirements, and the individual micro-benchmarks that implement the actual benchmark test cases.

3.1.1 Features

- GPU device management and control via CUDA and the NVIDIA Management Library (NVML),

- topology awareness of coprocessors, CPUs and memory systems via the portable Hardware Localitly (HWLOC) library,
- test automation of all micro-benchmark tests,
- verbose output including topology and GPU device information,
- automated graphing of micro-benchmark test cases, and
- cleanup and topology scanning scripts.

3.1.2 Non-Features

TARUC does not support the following:

- NUMA effects from distributed shared-memory programming,
- GPUs or other coprocessors that do not contain NVIDIA chipsets,
- hybrid CPU + GPU or embedded system on a chip (SoC) architectures that do not contain an NVIDIA chipset,
- systems with non-Linux operating systems, and
- NVIDIA coprocessors that do not support CUDA runtime version 7.5 or newer with compute ability 3.5+.

3.1.3 Requirements

A complete set of hardware and software requirements for building and using TARUC can be found in [1].

3.2 Micro-Benchmarks

In order to isolate resource contention and NUMA effects, TARUC is broken into four micro-benchmarks. These micro-benchmarks may be run in any combination or order. Each micro-benchmark dynamically creates a number of test cases at runtime based on the detected topology.

3.2.1 Memory Overhead

The memory overhead micro-benchmark measures allocation and deallocation cost for each memory type using various memory binding and thread pinning combinations to isolate NUMA effects caused by heterogeneous topology. A range of memory block sizes are timed if ranged test runs are enabled. All CPU socket and NUMA node combinations are tested for each of the three host memory types. Allocation and deallocations are timed consecutively for each memory type and block size step. For device memory allocations the `cudaMalloc()` function is timed for all GPU devices present within the test system after pinning the host thread to each CPU socket.

3.2.2 Memory Bandwidth

The memory transfer bandwidth micro-benchmark consists of three individual sub-tests that focus on different types of automated memory transfers (copies) and communication pipelines. Host-to-host, host-device, and device-to-device transfer bandwidth are all measured separately. For transfers involving host memory, pinned, pageable, and write-combined memory types are tested with varied memory access patterns. TARUC uses repeated, ascending and descending memory address access patterns to reduce caching effects.

Each single-threaded memory migration task is executed while pinning the executing CPU thread to a specific CPU socket/core. For transfers involving host memory, each memory type and access pattern is measured for bandwidth and transfer time. Each host memory block is bound to a specific NUMA region to test for NUMA effects while measuring pipeline bandwidth. For device-only memory transfers, additional memory copy types are considered including unified virtual addressing and direct peer-peer transfers.

3.2.3 Non-Uniform Random Memory Access (NURMA)

The NURMA micro-benchmark stresses streamed random memory access by using a large, dynamically defined stride gap to separate consecutive memory accesses. This allows the simulation of small, fixed-access strides like those present in certain stencil operations as well as larger pseudo-random memory access patterns. The main focus of this micro-benchmark is understanding how NUMA situations affect random memory access patterns.

3.2.4 Resource Contention

The resource contention micro-benchmark has one test for each of the QPI, PCIe, and integrated multi-channel memory controller communication paths. Simultaneous memory transfers to different combinations of host and device memory systems are used to isolate each communication pipeline. Threads are pinned to various combinations of CPU cores while different devices are used as sources or destinations. In this micro-benchmark host-only, device-only, and host-device transfers are measured.

3.3 Design

The TARUC benchmark suite consists of topology aware NUMA and resource contention micro-benchmarks as well as automated graphing scripts. A number of libraries are utilized for topology detection and control including HWLOC, NVML, CUDA, and OpenMP. These libraries allow for host thread and memory pinning, GPU coprocessor management and computation offload, as well as hardware affinity detection for both host and device processing units.

Figure 2 shows a block diagram of the TARUC benchmark at the software level. During benchmark initialization, the parameter values are read in from the input file and the system topology is detected via the `SystemTopo` class. A number of safety checks also occur during this phase to ensure benchmark parameters can be satisfied by system resources. These checks include a verification of the availability of NVIDIA GPUs as well as memory resources for large block allocations.

After initialization, TARUC proceeds to the benchmarking phase where each micro-benchmark is called in the order indicated by the parameter file. Each micro-benchmark reads relevant parameters from the `BenchParams` class and then generates test cases depending on the provided parameters. Ranged and multithreaded memory access tests save data for various block sizes into a multi-dimensional C++ `Vector`. This data is processed to produce either a calculated bandwidth or transfer latency value for each vector entry. These processed values are then saved to CSV files within the results folder.

Post-processing of ranged test data is carried out via Python

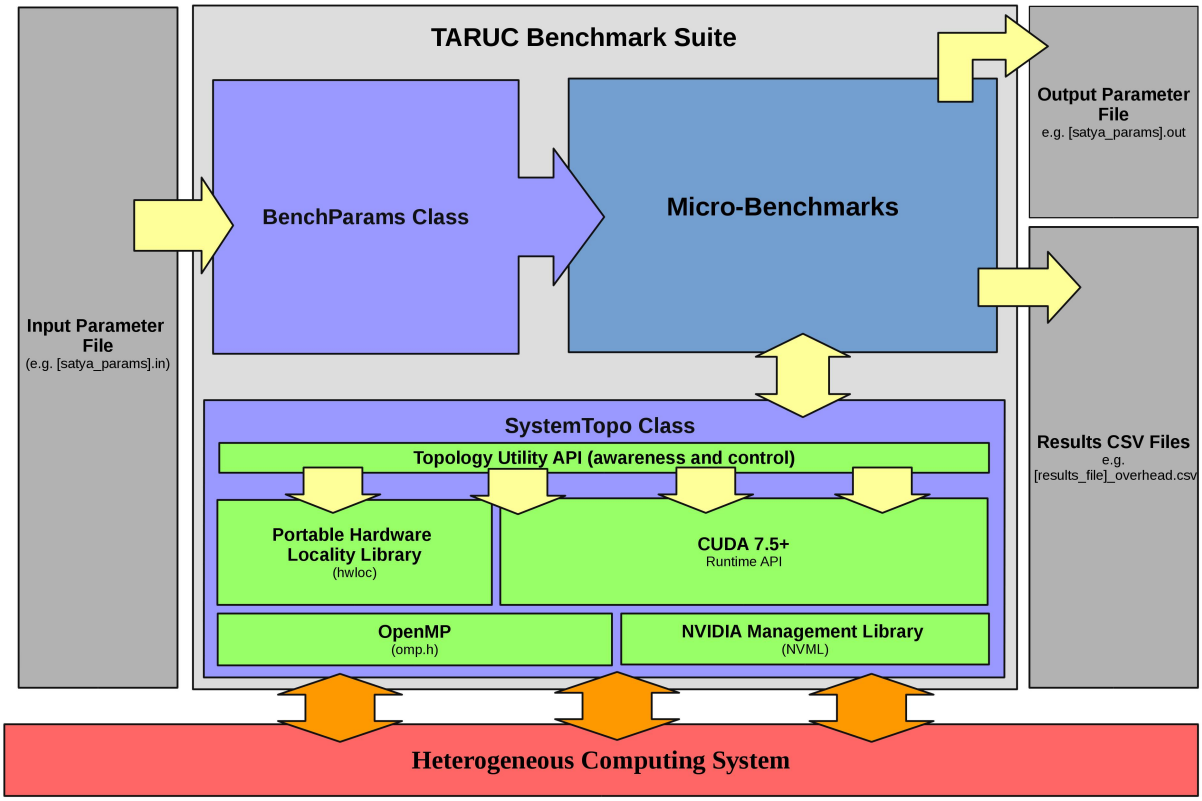

Figure 2: TARUC Benchmark Block Diagram

graphing scripts, each of which utilizes *NumPy* for data scanning/storage and *matplotlib* for plot creation. Each micro-benchmark has a separate plotting script used to graph output data. TARUC can produce a large number of graphs that the user can evaluate.

The sections below discuss the class interfaces for the `SystemTopo`, `BenchParams`, and `Timer` classes.

3.3.1 *SystemTopo Class*

Hardware topology detection, thread scheduling and memory policy management functionality is integrated into the `SystemTopo` C++ class. This class combines NVIDIA Management Library (NVML), CUDA runtime API, and HWLOC functionality into a single interface. At instantiation, the `SystemTopo` class parses a generalized topology object tree provided by the HWLOC API as well as device information from the NVML and CUDA libraries. All system topology information is automatically detected after instantiation of the `SystemTopo` class object. HWLOC provides detailed information about the number of CPU sockets, NUMA nodes, cores, and processing units (PUs) as well as the sizes of cache and main memory. PCIe device location information is also available. The topology information is parsed into device statistics and relational information that is later used to manage thread affinity and memory policy. The CUDA runtime API is then queried to provide basic device information including the size of memory resources and the functionality of peer-to-peer support for GPU pairs. Finally, NVML is utilized to inform the `SystemTopo` class about the proximity of GPU devices to processing units.

After the initialization phase, the class provides a number of memory, thread, and device management functions. These functions provide the ability to pin threads to execution units including CPU sockets, cores or processing units. The HWLOC library provides this functionality on Linux systems through the modification of scheduler affinity. Memory can be directly allocated on any specified NUMA node (called memory binding), or the memory policy can be modified such that memory is placed on NUMA regions after being used (such as interleaving on first touch).

3.3.2 *BenchParams Class*

The `BenchParams` class is a storage and utility class for keeping track of all benchmark parameters. Because of the number of class variables and intended use within the TARUC benchmark, each parameter is made publicly accessible rather than abstracted via member access functions. Each value within the parameter file corresponds to a variable within the parameter class.

Some of the available parameters include: minimum and maximum block sizes for the range tests, whether to test all GPUs in the system, whether to test all memory types, and if each socket should be included in the tests. There are many parameters available that determine the behavior of each test. Full discussion of all of TARUC's parameters is omitted from this paper due to space considerations.

3.3.3 *Timer Class*

The `Timer` class is a C++ class that integrates host-based timing together with a CUDA event-based device timer. The timing object is instantiated with a single `boolean` value in-

dicating the type of timing the user wishes to use. When the value is true, the `chrono::high_resolution_clock` standard library class is utilized to provide host-based timing functionality. The `chrono` class was chosen because of its nanosecond accuracy and simple programmer interface. If the value is set to false, then the timer relies on the CUDA event interface to provide timing results. While event timings come with a lower (millisecond) accuracy than the host timer, they provide the best mechanism that is capable of timing asynchronous CUDA API calls. To prevent synchronization issues between different timers, a non-default CUDA stream is initialized and passed to event timing calls.

4. BENCHMARK RESULTS

Some of the more interesting results and insights from running TARUC on a high-performance test system are provided. These results quantify the importance of NUMA awareness and demonstrate potential uses for TARUC. Some insight based on experiences building the TARUC benchmark is also provided. The primary output from TARUC is performance profile graphs from each of the micro-benchmarks. This section presents a small subset of the graphs available. These graphs are useful for measuring the significance of NUMA effects on GPUs and host memory subsystems as well as doing comparative analysis of different GPU hardware and topologies.

4.1 Test System

The test system consists of two Intel Xeon E5-2650 Sandy Bridge CPUs that have 8 discrete 2.0 GHZ cores, and 16 processing units with hyperthreading enabled. Each CPU socket has 32 GB of attached RAM for a total of 64 GB of main memory. Cores have 32KB instruction and data L1 caches and a 256 KB L2 cache. All cores on a single socket share 20 MB of L3 cache. Attached to the PCIe slots are four discrete GPUs: two Tesla K20Xm compute-class cards, one Tesla K40c, and one GeForce GTX Titan Black graphics card. Each K20Xm and the Titan have 6GB of memory while the K40c GPU has 12 GB of memory. Each of the GPU memories have different memory clock frequencies resulting in varying memory access latency. The K20Xm, K40c and Titan GPUs have 2600 MHz, 3004 MHz, and 3500 MHz memory clock rates, respectively. The test system provides a unique testing environment because different generation GPUs allow comparison of NUMA effects on different GPU hardware. Figure 3 shows a hardware overview of the test system.

4.2 Profiling Results

The following sections contain the results of running the TARUC benchmark on the test system. These graphs are a subset of the graphs that are generated with TARUC.

The parameters have been kept consistent over all the benchmarking systems to minimize variance in graph structure. The following parameters were used for the benchmarking runs on all systems. Figure 4 below shows a simplified list of test parameters.

4.2.1 Memory Management Overhead

The graphs of memory overhead tests are log-log scale graphs showing the elapsed time in microseconds of an allocation of varied memory block sizes. Log scale graphs are useful for showing large variations in memory block sizes. However, log scale graphs can obscure overhead costs that are actually much larger than they appear.

Three types of host memory are tested within the TARUC benchmark. Pageable, pinned, and write-combined memories are measured along with device memory located on each of the on-node GPUs. Memory allocations and deallocations are measured between 100B and 1.5GB.

Figure 5 shows that allocations of non-local host memory result in a fixed overhead of 20-50%. This NUMA overhead decreases from its maximum as the block size increases past the inflection point of 1MB for non-pageable memories. This graph also shows that non-pageable memories have increased allocation overhead compared to pageable memory.

Allocations of GPU memory have no significant effect when NUMA bindings are changed. The fixed overhead associated with the allocation of GPU memory over PCIe depends more on the hardware specifications of the specific GPU device. In particular, the memory clock frequency of the GPU seems to determine the relative performance of a specific GPU for memory allocation and deallocation operations. Figure 6 shows that the GTX Titan GPU has the lowest cost for block allocations likely as a result of it having the highest memory clock frequency.

Figure 7 shows NUMA related effects in the deallocation of memory for the K20Xm GPU devices when the executing thread is pinned to the non-local CPU socket. This performance artifact can be clearly linked to NUMA as the K20Xm devices do not suffer from the performance degradation when the thread is pinned to CPU 1 as shown in Figure 8. The two K20Xm GPUs are located on PCIe connections attached to controllers on CPU 1 while the executing thread is run on CPU 0. This trend is significant, as it appears to be machine specific and only occurs for device memory deallocation events.

It is interesting that there is a very clear point where at certain memory block sizes the noise dissipates as indicated near 10MB. We do not speculate why this performance artifact occurs due to the opaqueness of the CUDA runtime system.

These graphs provide evidence that system performance can be influenced from hardware specific phenomena outside the control of the developer. It is essential that the performance profile of the memory system be benchmarked to avoid wasting time on application debugging if performance results are other than what is expected.

4.2.2 Memory Transfer Bandwidth

Host-Host Block Memory Transfers.

Host-Host memory bandwidth is measured using a single thread of execution pinned to various CPUs. This thread transfers memory from one memory block to another, each of which has an individual memory policy binding. The comparative effect of block size, memory and thread policy, and access pattern are presented. Each of the mentioned variables may change the effectiveness of caching at reducing NUMA effects. In particular, it is expected that small block sizes along with those that use repeated memory address ranges will experience less NUMA overhead due to caching.

Figure 9 shows the change in execution time associated with exceeding the available space of each level of cache. The test uses cacheable paged memory with a repeated memory

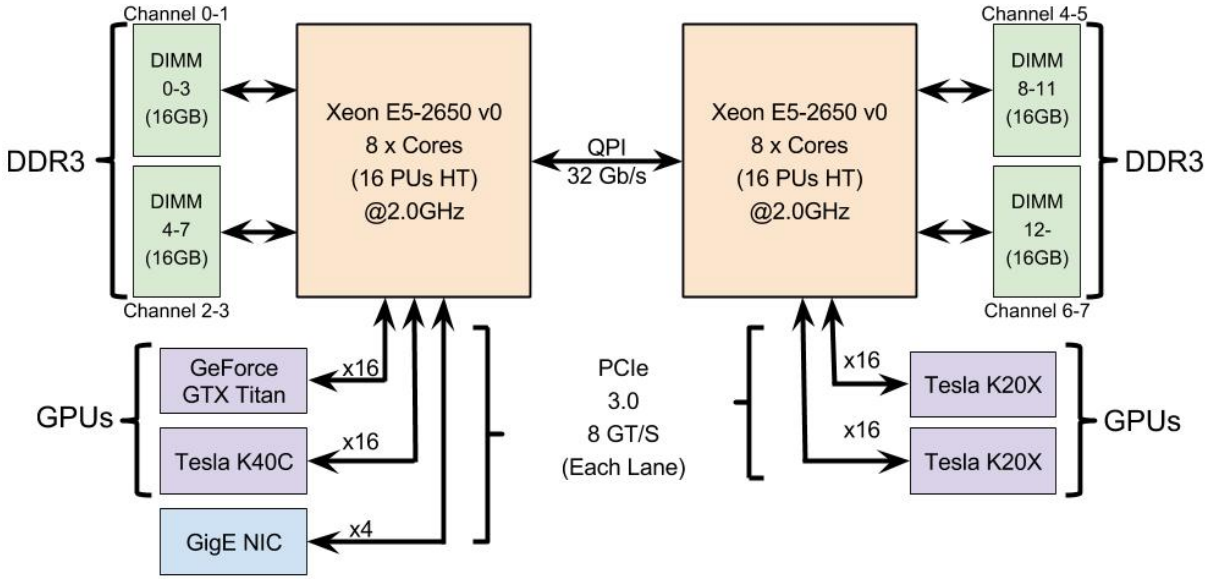

Figure 3: Test System Architecture Overview

```
1  Test All Sockets:                true
2  Test Host Mem Types:             true
3  Test all Devices:                true
4  Test Access Patterns:            true
5  Sustained Tests:                 true
6  # Repeated Steps:                20
7  Number Steps Per Magnitude:      10
8  Overhead/Bandwidth Range:        10 - 1500000000 (bytes)
9  NURMA Access Gap/Stride:         657 (doubles)
10 NURMA Host Memory Block Size:    200000000 (doubles)
11 NURMA Range (doubles):           10, 10000000 (min,max)
12 Contention Memory Block Size:    100000000 (bytes)
```

Figure 4: TARUC Test Parameters

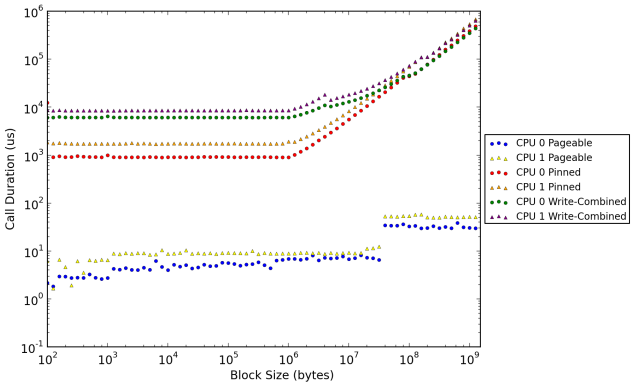

Figure 5: Memory Allocation - Memory Bound To Node Zero

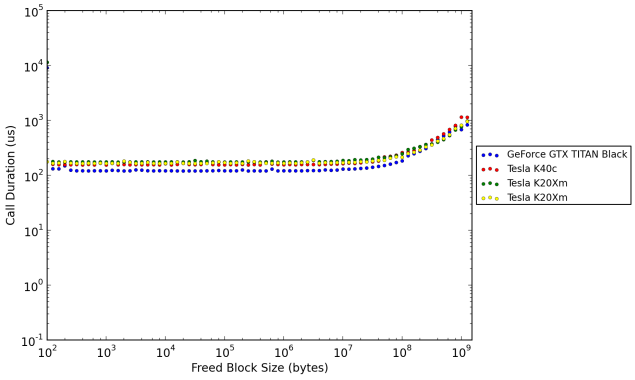

Figure 6: Memory Allocation - Thread Pinned To CPU Zero With All Devices

address. Interestingly, the local memory transfer when the executing thread is local to the memory node provides the best performance only after the size of the memory block exceeds L3 cache size.

Figure 10 shows the NUMA costs of different node bindings when the executing thread is pinned to CPU 1. The worst case for NUMA binding overhead is a 15% reduction in bandwidth when a thread executes completely non-local memory transfers. The best performance is achieved when both source and destination memory blocks remain local to the executing thread. Results also show a significant reduction in overhead if the destination memory block is on the same NUMA region as the executing thread's CPU. This indicates that cache write-back policy is slowed by having to write cache lines to non-local memory. The source block is never written which prevents it being marked as modified, and allows the cache line to be invalidated without being written back to main memory. Another result is that page-

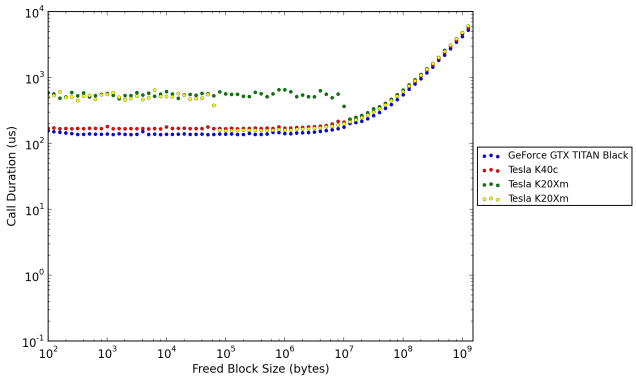

Figure 7: Memory Deallocation - Thread Pinned To CPU Zero With All Devices

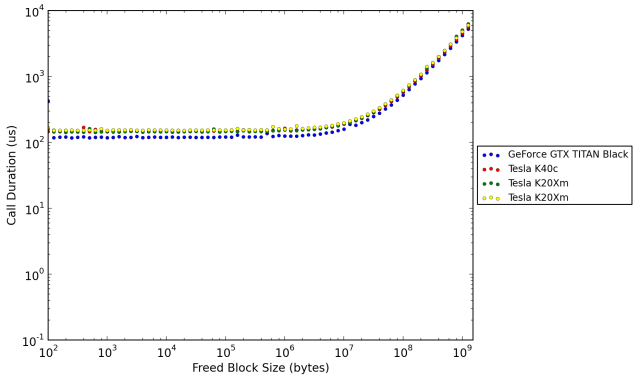

Figure 8: Memory Deallocation - Thread Pinned To CPU One With All Devices

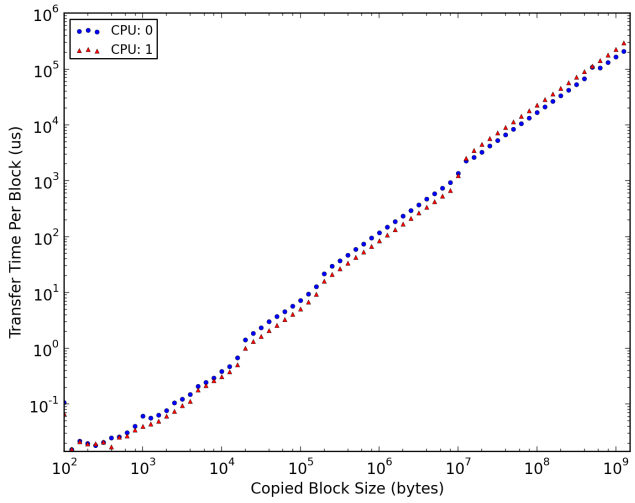

Figure 9: Host Memory Copy - Pageable Memory Bound To Node Zero With A Repeated Memory Address

able and pinned memory benefit from cache despite the use of varied access patterns.

To demonstrate that different memory address access patterns do not significantly influence the presence of NUMA

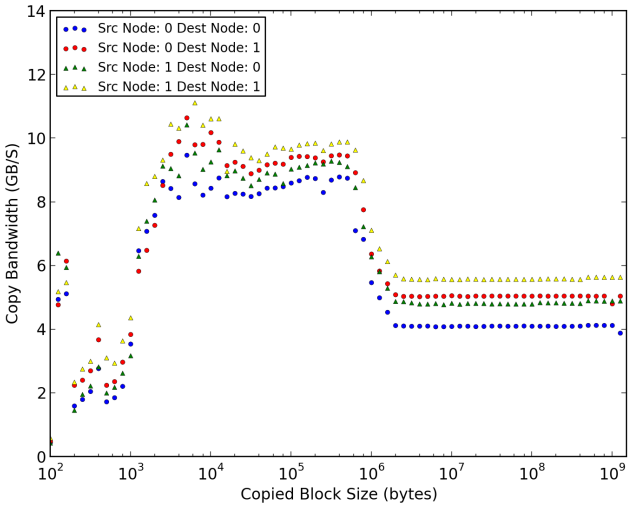

Figure 10: Host Memory Copy - Thread Pinned To CPU One And Pageable Host Memory With Linear Decreasing Memory Address

within test runs, the total transfer time of each pattern is plotted for different thread pinning combinations. Figure 11 shows that while there is a minor change in the overall execution time of a memory copy for certain patterns, this change does not outweigh NUMA performance influences.

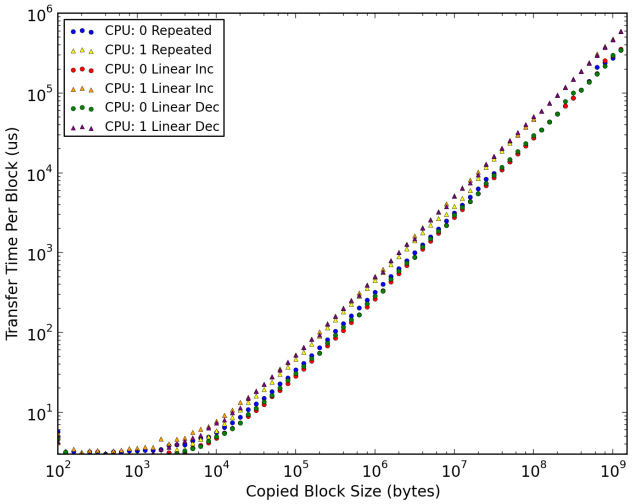

Figure 11: Host Memory Copy - Memory Bound To Node Zero, Repeated Address, And Pinned Dest Memory

Pageable memory seems to provide better performance than pinned memory on the test system. Figure 12 shows pinned memory performance and demonstrates a 50% reduction in bandwidth for non-local NUMA bindings.

Host-Device Block Memory Transfers.

Benchmarking memory copy bandwidth between GPU accelerators and host memory involves timing CUDA runtime calls for various memory types. Asynchronous and syn-

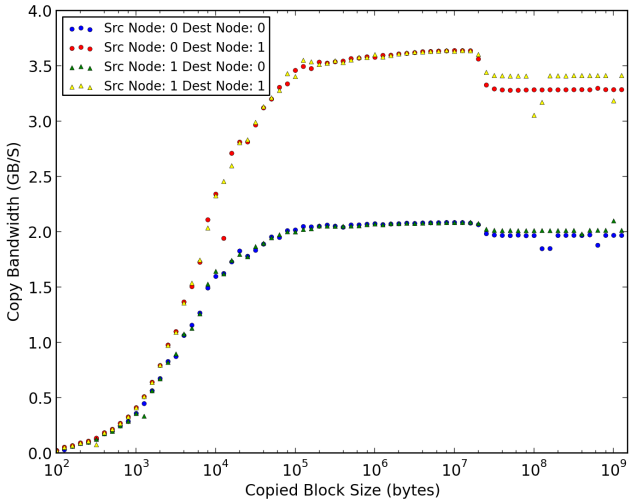

Figure 12: Host Memory Copy - Thread Pinned CPU One Using Repeated Memory Address And Pinned Memory

chronous copy types are tested depending on the host memory type. Pinned and write-combined memories allow the runtime to use asynchronous copies between host and device memory spaces. Such copies remove the need to temporarily transfer memory to a pinned buffer before copying the block to the device. The test system demonstrated a 28% increase in bandwidth when using asynchronous runtime copies over the baseline pageable memory transfer (for the largest block sizes tested).

NUMA effects on host-device memory transfers were of particular interest when the TARUC benchmark was developed. Different thread pinning and NUMA binding combinations were tested for both asynchronous and synchronous copies. Removing direct thread involvement in the copy ends the need for the memory to be copied to a buffer that is local to that executing thread. We see a significant bandwidth reduction when using pageable host memory for transfers involving non-local memory. Figure 13 illustrates this effect. Transfers involving pinned and write-combined host memory are unaffected by NUMA locality of the host memory.

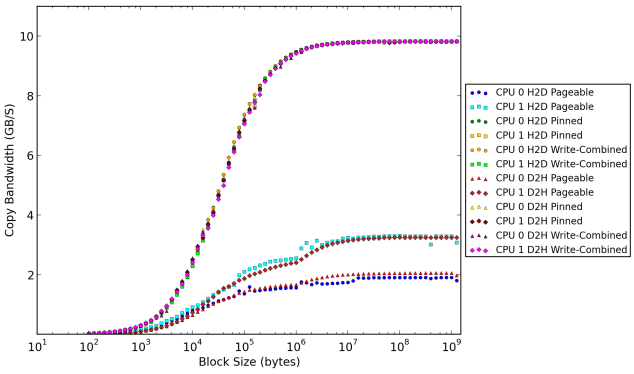

Figure 13: Host-Device Memory Copy - Host NUMA Node One And K40c GPU Using Linear Increasing Memory Address

Similar NUMA effects are demonstrated during transfers for all GPUs on the test system. Figure 14 illustrates that the most significant NUMA effect is the locality of memory to the executing host thread. When memory is non-local, the pageable host memory must be transferred before a host-device transfer can be initiated. Differing GPUs also report varying host-device bandwidth. This shows that the difference in bandwidth is a result of different device hardware rather than NUMA effects. Bandwidth rather than latency is the limiting factor when non-pageable memory transfers are concerned.

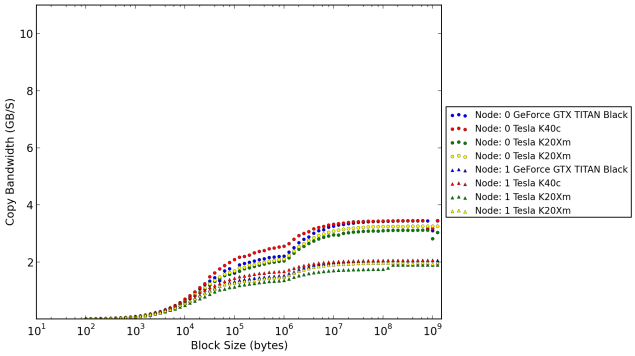

Figure 14: Device-To-Host Memory Copy - Thread Pinned To CPU Zero Using Pageable Host Memory

Device-Device Block Memory Transfers.

Direct device-device memory transfers allow the accelerators to share data during synchronization events scheduled by a host thread. Direct transfers are desirable in situations where the host does not need intermediate application data in between time steps (or other incremental steps) of a run. If the host does not need intermediate step data then GPU devices can do direct transfers. Device-device transfers reduce contention on the host memory system as well as the QPI bus. In TARUC, peer-peer and device-host-device transfers (shortened device-device) are tested. Both types of asynchronous transfers remove host thread involvement in the memory transfer. Device-device transfers involve an intermediate copy by the CUDA runtime to host memory before the memory block is copied to the destination GPU.

The first test of device performance is internal device memory transfer bandwidth. In the case of Figure 15, intra-device transfers are profiled. The actual memory system bandwidth is doubled because the bandwidth is measured from the perspective of the block being transferred. Intra-device transfers involve a read and a write to different memory blocks within the same GPU. Measuring intra-node GPU memory bandwidth shows the difference in accelerator generation and type (consumer or compute class).

Figures 16 and 17 show that peer transfers provide higher bandwidth for small to medium memory block transfer sizes, but underperform for some transfers of large device memory blocks. This result is contrary to expectation since peer transfers should allow for lower latency and higher bandwidth when enabled. Instead, we see that For large block sizes there can be an advantage to using device-device memory transfers. The direction of the transfer can also influence the bandwidth. Transferring memory between the con-

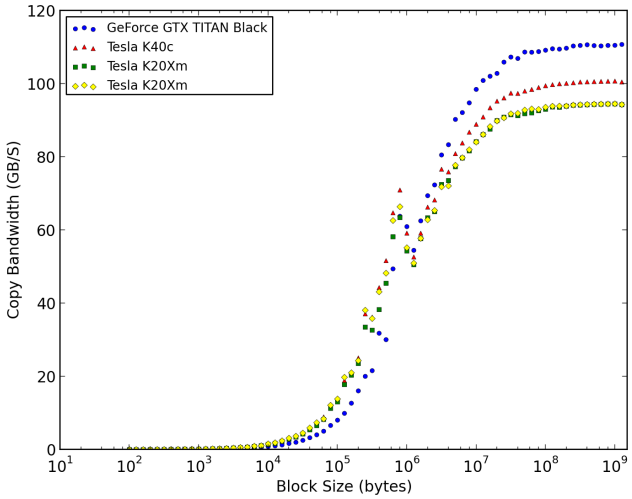

Figure 15: Intra-Device Memory Transfer - Thread Pinned To CPU Zero

sumer class GTX Titan GPU and the compute class K40c is a clear demonstration of this effect. Something relating to hardware generation or optimization levels between the two cards reduces the performance when transferring from the K40c to the Titan GPU. This is consistent no matter which CPU core is pinned for the host thread. Bandwidth decreases are more significant during peer transfers (than device-device) for certain transfer directions when copying memory between different GPU models.

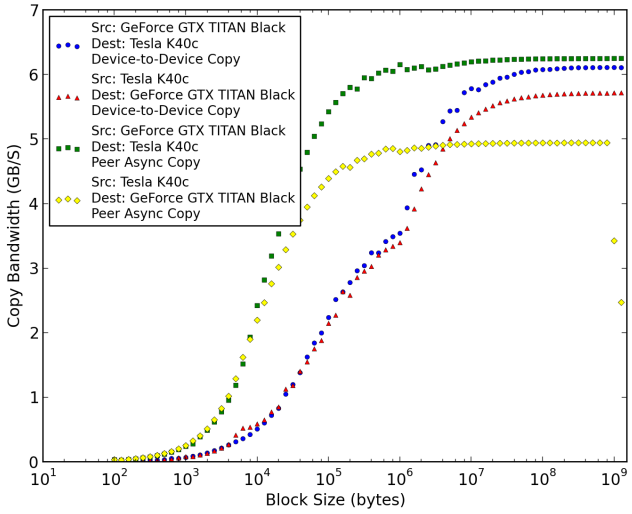

Figure 16: Device-Device Memory Copy - Titan And K40c GPUs With Thread Pinned To CPU Zero

The K20Xm GPUs consistently show almost identical bandwidth no matter the transfer direction. They also show the premature maxing of peer transfer bandwidth when compared to device-device transfers of large enough block sizes.

4.2.3 *NURMA Latency*

The NURMA micro-benchmark is useful for understanding how different memory access patterns are affected by

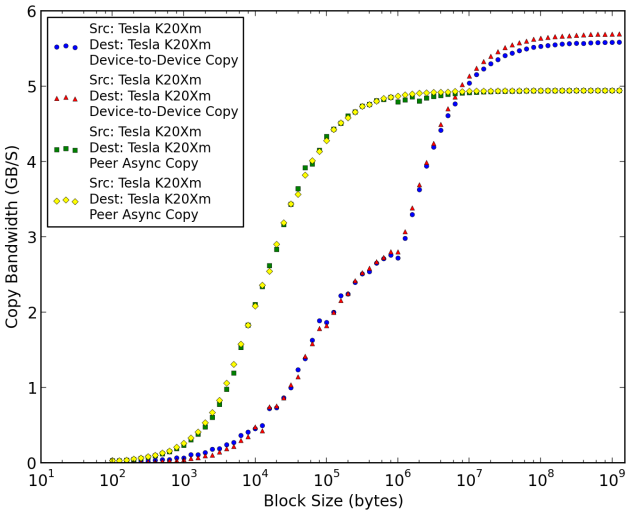

Figure 17: Device-Device Memory Copy - Both K20Xm GPUs With Thread Pinned To CPU Zero

NUMA. Small memory gap sizes mimic strided memory access while larger gap sizes that exceed cache line width and page size mimic random memory access. If the gap size is reduced to zero, streamed memory transfer is simulated.

Figure 18 illustrates that there is no significant difference in performance or NUMA impact on pinned versus pageable memory for streamed random access. Both types of memory demonstrate differences in performance for local and non-local memory access.

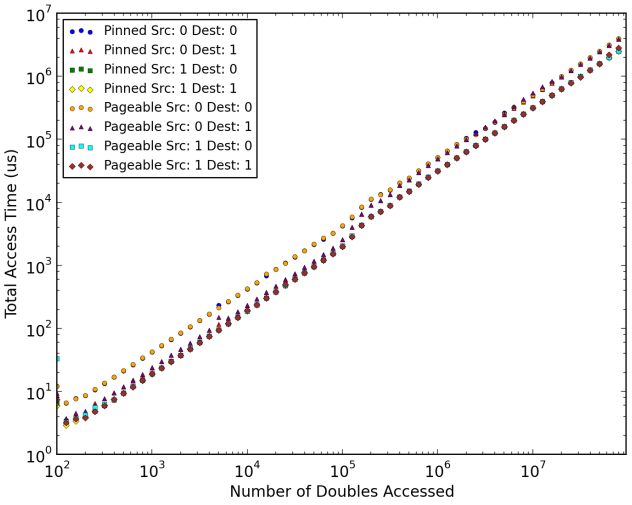

Figure 18: NURMA Test - Thread Pinned to CPU One With All Memory Types And NUMA Bindings

4.2.4 *Resource Contention*

Local Memory Aggregate Bandwidth.

In general, ideal real-world memory bandwidth will be achieved when the locality of memory access is maximized. To achieve ideal memory locality, threads should be pinned

to execution spaces where NUMA policy allocates memory. In the case of the local memory bandwidth test, memory is accessed from the local NUMA node for all executing threads. Figure 19 shows bandwidth scaling for single-node or multi-node memory situations. Bandwidth peaks when the number of simultaneous threads matches the number of memory channels connecting each CPU to its local memory space. This graph shows a bump in performance when oversubscribing threads to cores by 1.5x. This could be a result of the OS scheduling threads that are bottlenecked by memory transfers more efficiently. Once thread-to-core oversubscription exceeds 1.5x, there is a decrease in performance to slightly below the bandwidth of the one thread to one core ratio.

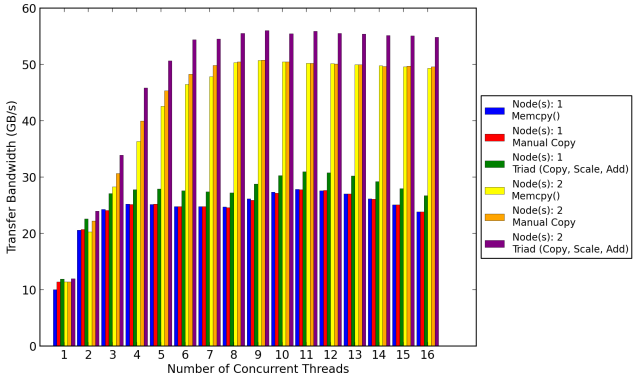

Figure 19: Local Memory Bandwidth - All NUMA Bindings And Memory Operation Types

Inter-Socket QPI Contention.

The worst case NUMA related contention occurs when simultaneously executing threads access host memory from adjacent NUMA regions. QPI links quickly saturate on the test system for inter-socket transfers. since they have a maximum bidirectional bandwidth roughly the same as a single 16x PCIe 3.0 connection. Figure 20 shows a roughly 50% drop in bandwidth compared to maximum host memory bandwidth when QPI is saturated.

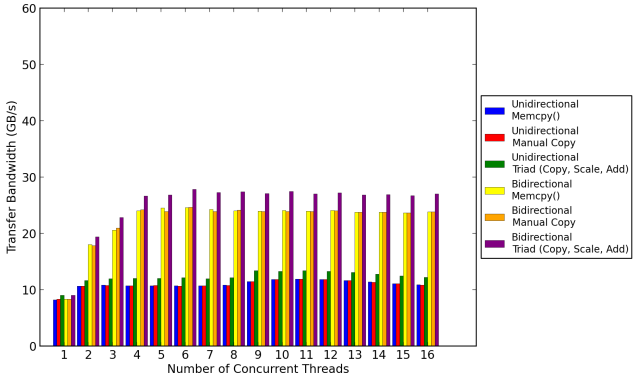

Figure 20: Inter-Socket Host Memory Access Bandwidth - All Directions And Memory Operation Types

Host-Device PCIe Contention.

To provide accurate assessment of PCIe contention, a system-wide baseline of aggregate performance provides a reference for realistic bandwidth scaling of different GPU assignments. Figure 21 shows the aggregate PCIe bandwidth when all GPUs are copying memory simultaneously to and from local CPU memory. Bidirectional memory copies are measured to ensure pipeline saturation. Each GPU has two execution and memory copy engines, so this test is an oversubscription of 4x since each thread does a bidirectional transfer.

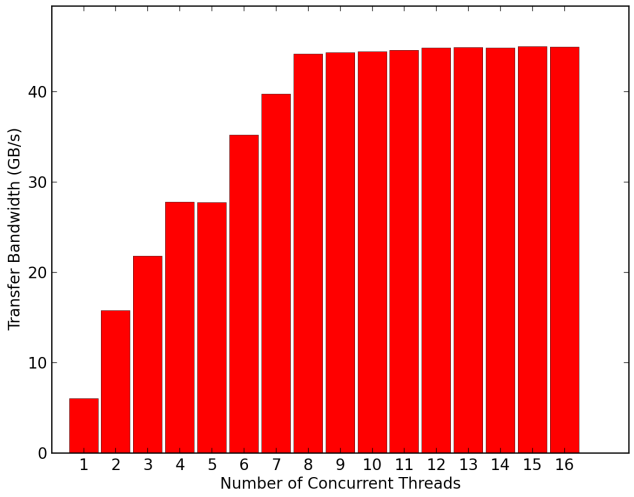

Figure 21: Baseline PCIe Bandwidth - Four Threads Per GPU With Bidirectional Transfers, All Threads On Local CPUs

Local assignment of GPUs to CPU memory is critical to maximize bandwidth because QPI bandwidth is significantly lower than host memory or the aggregate bandwidth of multiple PCIe connections. Moving half of the threads to a non-local CPU reduces the aggregate bandwidth by roughly 6 GB/S. This amounts to a 13% drop in overall bandwidth. For a system with less contention, aggregate bandwidth might be less affected making this test somewhat less valid. However, pipeline contention certainly does occur in balanced applications that utilize GPUs for computation when kernel operations synchronize with host memory. In previous tests it was determined that for asynchronous pinned memory host-device transfers, little in the way of NUMA effects exist when contention is not present. Pageable memory transfers do not benefit from a lack of NUMA effects due to the need to transfer the pageable memory block to a local pinned buffer. Minimizing QPI contention is essential for maximizing bandwidth during synchronization events.

Figure 22 displays the per node host-device bandwidth of the test system when up to four threads per GPU are executing simultaneous transfers. Results show that node 1 has slightly lower bandwidth than node 0, which verifies that that the GTX Titan and K40c have higher aggregate bandwidth than the two K20Xm GPUs. Because the K40c and the Titan are local to CPU 0, they have a direct higher bandwidth connection that does not travel over QPI. Allocating GPUs based on the locality of CPU resources is a viable option assuming memory is local to the executing

thread. Otherwise, allocating GPU resources based on the locality of memory the thread is using should be preferred. It is important to note that the per node transfer bandwidth graph shows roughly the same performance reduction as partially non-local baseline bandwidth test. The performance of both local memory bandwidth and aggregate PCIe bandwidth appears to be limited by the host memory system for the test system rather than PCIe bandwidth. This is true for contended systems when all GPUs simultaneously execute bidirectional memory transfers.

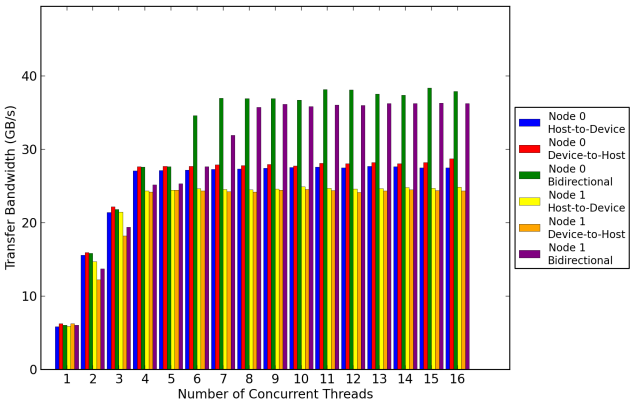

Figure 22: Per NUMA Node Aggregate PCIe Bandwidth

These results show that the TARUC benchmark can be used to profile a target hardware system to find the inflection points in bandwidth and execution time or possible hardware specific performance artifacts.

5. RELATED WORK

The existence of NUMA effects and the necessity of designing NUMA aware schedulers and algorithms has been well documented by a number of researchers. Research related to the material covered in this paper has mainly benchmarked host memory contention via either synthetic memory kernels [2] or real applications [8].

The synthetic benchmarks addressed specific instances of NUMA effects in much the same way the host-based TARUC tests do by carrying out memory operations over a number of NUMA pinning situations. Application performance benchmark tests on NUMA systems generally utilize application codes along with command line memory policy tools (such as *hwloc-bind*) to test poorly scheduled thread and memory binding combinations on overall application performance. Some new architectural enhancements like large virtual page sizes have been demonstrated to negatively impact the performance of certain applications on NUMA machines [3]. Additionally, some work has been carried out to develop a NUMA aware thread scheduler with knowledge of critical path operations. This type of scheduler has been shown to allow the deployment of near ideal thread scheduling algorithms for NUMA machines [9].

Most of the previous research has not dealt with the effect NUMA has on coprocessor enabled systems. Previous NVIDIA conference presentations allude to NUMA effects on GPUs with the solution being to statically define GPU locality based on initial problem partitioning. Based on TARUC benchmark results on x86 systems, we observe that NUMA effects result in more complicated performance phenomenon than simple bandwidth or latency reductions. Both load balancing and NUMA awareness are required along with a system specific understanding of NUMA and contention effects. In addition, few benchmarking tools have been developed to test the significance of on-node NUMA effects on differing systems.

To the best of our knowledge, no publicly available tools that benchmark NUMA effects also take into account the significance of PCIe based coprocessors in addition to measuring memory contention.

6. CONCLUSIONS AND FUTURE WORK

This paper presents the Topology Aware Resource Usability and Contention (TARUC) benchmark that allows performance profiling of heterogeneous computing systems. TARUC consists of micro-benchmarks that demonstrate how NUMA and contention influence performance. These benchmark subtests address memory allocation and deallocation performance for a number of host and device memory types. Tests of host-host, host-device, and device-device pipeline bandwidth can be measured over various thread, memory type and memory binding combinations. In addition to streamed consecutive memory access, the non-uniform random memory access (NURMA) micro-benchmark addresses other possible memory access patterns on memory performance on NUMA machines. Contention micro-benchmark tests seek to stress system communication pipelines including main memory, I/O links like PCIe and cache-coherence links like QPI.

TARUC performance profiles were acquired from a test system containing four NVIDIA GPU devices. Our results show NUMA effects can have a significant impact on achieved memory performance, and that TARUC can be used as a guide for application developers who wish to tune the performance of their application on a specific system. It is our hope that the TARUC benchmark will be useful for HPC application developers for improving performance on complex heterogeneous NUMA systems.

For future work we plan to measure communication and resource contention on Intel Xeon Phi Knights Landing systems, as they will present a new NUMA region in the form of High-Bandwidth Memory.

7. REFERENCES

[1] G. Baker. An emperical study of contention and NUMA effects on heterogeneous computing systems. Master's thesis, California Polytechnic State University, June 2016.

[2] L. Bergstrom. Measuring NUMA Effects With the STREAM Benchmark. *arXiv preprint arXiv:1103.3225*, 2011.

[3] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quéma. Large Pages May be Harmful on NUMA Systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 231–242, 2014.

[4] Intel. White paper: An Introduction to the IntelÂő QuickPath Interconnect. Technical report, Intel Corporation, January 2009.

[5] P. Jacob, A. Zia, O. Erdogan, P. M. Belemjian, J.-W. Kim, M. Chu, R. P. Kraft, J. F. McDonald, and K. Bernstein. Mitigating Memory Wall Effects in High-Clock-Rate and Multicore CMOS 3-D Processor Memory Stacks. *Proceedings of the IEEE*, 97(1):108–122, 2009.

[6] J. Lawley. White paper: Understanding Performance of PCI Express Systems. Technical report, XILINX, October 2014.

[7] S. A. McKee. Reflections on the Memory Wall. In *Proceedings of the 1st conference on Computing frontiers*, page 162. ACM, 2004.

[8] K. Spafford, J. S. Meredith, and J. S. Vetter. Quantifying NUMA and Contention Effects in Multi-GPU Systems. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, page 11. ACM, 2011.

[9] C. Su, D. Li, D. S. Nikolopoulos, M. Grove, K. Cameron, and B. R. De Supinski. Critical Path-Based Thread Placement for NUMA Systems. *ACM SIGMETRICS Performance Evaluation Review*, 40(2):106–112, 2012.

[10] The Top500 List of Supercomputers. http://www.top500.org. Accessed: 2016-4-14.