

PIUMA: Programmable Integrated Unified Memory Architecture

Sriram Ananthakrishnan*, Nesreen K. Ahmed*, Vincent Cavé*, Marcelo Cintra*, Yigit Demir*, Kristof Du Bois*, Stijn Eyerman*, Joshua B. Fryman*, Ivan Ganev*, Wim Heirman*, Hans-Christian Hoppe*, Jason Howard*, Ibrahim Hur*, MidhunChandra Kodiyath*, Samkit Jain*, Daniel S. Klowden*, Marek M. Landowski*, Laurent Montigny*, Ankit More^{†§}, Przemyslaw Ossowski*, Robert Pawlowski*, Nick Pepperling*, Fabrizio Petrini*, Mariusz Sikora*, Balasubramanian Seshasayee*, Shaden Smith^{†§}, Sebastian Szkoda*, Sanjaya Tayal*, Jesmin Jahan Tithi*, Yves Vandriessche*, Izajasz P. Wrosz*

*Intel Corporation

†Microsoft Corporation

Abstract—High performance large scale graph analytics is essential to timely analyze relationships in big data sets. Conventional processor architectures suffer from inefficient resource usage and bad scaling on graph workloads. To enable efficient and scalable graph analysis, Intel developed the Programmable Integrated Unified Memory Architecture (PIUMA). PIUMA consists of many multi-threaded cores, fine-grained memory and network accesses, a globally shared address space and powerful offload engines. This paper presents the PIUMA architecture, and provides initial performance estimations, projecting that a PIUMA node will outperform a conventional compute node by one to two orders of magnitude. Furthermore, PIUMA continues to scale across multiple nodes, which is a challenge in conventional multinode setups.

I. INTRODUCTION

Current practices in data analytics and artificial intelligence perform tasks such as object classification on unending streams of data. Computing infrastructure for classification is predominantly oriented toward “dense” compute, such as matrix computations. The continuing exponential growth in generated data [1] has shifted compute to offload to GPUs and other focused accelerators across multiple domains that are dense-compute dominated.

The next step in both AI and data analytics is reasoning about the *relationships* between these classified objects, typically represented as a graph. Determining the relationships between entities in a graph is the basis of *graph analytics* [2]. Graph analytics poses important challenges on existing processor architectures due to its sparse structure. This sparseness leads to scattered and irregular memory accesses and communication, challenging the optimizations implemented for decades that have gone into traditional dense compute solutions. Consider the common case of *pushing* data along the graph edges, see the example graph in Figure 1. All vertices initially store a value locally and then proceed to add their value to all neighbors along outgoing edges. This basic computation is ubiquitous in graph algorithms such as *PageRank* [3]. The resulting access stream (Figure 1b) is

[§]Ankit More and Shaden Smith were with Intel when working on this project.

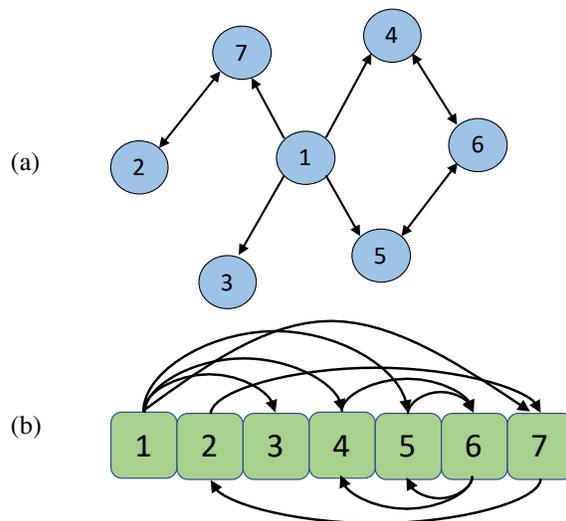


Fig. 1. (a) A sparse graph with directed edges. (b) Memory access patterns observed when moving data along the edges of (a).

irregular and has no locality, making conventional prefetching and caching useless.

The combination of low performance and very large graph sizes limits the usability of graph analytics. Recognizing both the increasing importance of graph analytics and the need for vastly improved sparse computation performance compared to traditional approaches, DARPA launched their HIVE program to achieve at least 1000× Performance/Watt breakthrough on such large problems before the end of 2022 [4].

This paper introduces the Programmable Integrated Unified Memory Architecture (PIUMA) developed for the DARPA HIVE program. The PIUMA machine is designed for graph analytics at massive scales. PIUMA enables high-performance graph processing by addressing constraints across the network, memory, and compute architectures that typically limit performance on graph workloads.

II. GRAPH WORKLOAD CHALLENGES

Graph algorithms face several major scalability challenges on existing architectures, because of their irregularity and

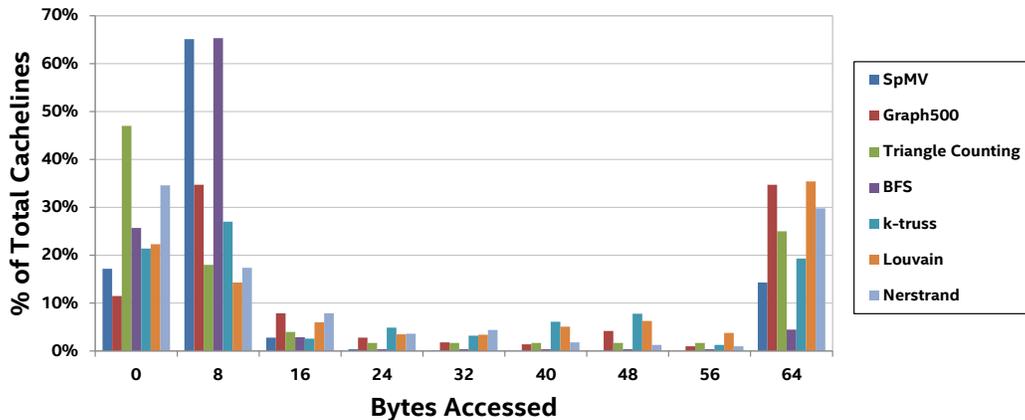


Fig. 2. Cache line utilization for graph workloads.

sparsity.

A. Challenge 1: Cache and Bandwidth Utilization

Figure 2 shows the cache line utilization for a variety of graph analysis applications when executed on a conventional cache based processor with prefetcher. For every 64 byte cache line fetched from memory, it shows how many bytes are actually used by the CPU. For most cache lines, either 0, 8, or the full 64 bytes are used.

The zero usage fraction stems from cache lines that were prefetched but never used. Cache lines with 8 or fewer bytes used are caused by sparse accesses with no spatial locality. A typical pattern in graph application is a chain of indirect loads, similar to a pointer chasing pattern: a vertex’s neighbors are stored in a list, which are used to index the data array. Because neighbor lists do not show regularity or locality, accesses to the data array are intrinsically sparse. Other memory accesses have limited locality (*e.g.*, fetching the neighbor list itself), as shown by the fraction where all 64 bytes are used, but they are limited in size, explaining the useless prefetches that fetch past the end of the list.

As a result, the execution of graph applications suffers from inefficient cache and bandwidth utilization: caches are thrashed with single-use sparse accesses and useless prefetches, and most of the 64 byte memory fetches contain only one 8-byte useful data element. Overprovisioning memory bandwidth and/or cache space to cope with sparsity is inefficient in terms of power consumption, chip area and I/O pin count. Instead, PIUMA uses limited caching and small granularity memory accesses to efficiently deal with the memory behavior of graph applications.

B. Challenge 2: Irregular Computation and Memory Intensity

Further analysis of graph algorithms shows additional problems in optimizing performance. The computations are *irregular*: they exhibit skewed compute time distributions, encounter frequent control flow instructions, and perform many memory accesses. The compute time for a vertex in the PageRank example is proportional to the number of outgoing edges (degree) of that vertex. Graphs such as the one illustrated

in Figure 1 have skewed degree distributions, and thus the work per vertex has a high variance, leading to significant load imbalance.

Our analysis reveals that graph applications are heavy on branches and memory operations. Furthermore, conditional branches are often data dependent, *e.g.*, checking the degree or certain properties of vertices, leading to irregular and therefore hard to predict branch outcomes. Together with the high cache miss rates caused by the sparse accesses, conventional performance oriented out-of-order processors are largely underutilized: most of the time they are stalled on cache misses, while a large part of the speculative resources is wasted due to branch mispredictions. In PIUMA, this observation was the incentive to use single issue in-order pipelines with many threads to hide memory latency and avoid speculation.

C. Challenge 3: Fine- and Coarse-Grained synchronization

Graph algorithms require frequent fine- and coarse-grained synchronization. For example, PageRank requires fine-grained synchronizations (*e.g.*, atomics) to prevent race conditions when pushing values along edges. Synchronization instructions that resolve in the cache hierarchy place a large stress on the cache coherency mechanisms for multi-socket systems, and all synchronizations incur long round-trip latencies on multi-node systems. Additionally, the sparse memory accesses result in even more memory traffic for synchronizations due to false sharing in the cache coherency system.

Coarse-grained synchronizations (*e.g.*, system-wide barriers and prefix scans) fence the already-challenging computations in graph algorithms. These synchronizations have diverse uses including resource coordination, dynamic load balancing, and the aggregation of partial results. These synchronizations can dominate execution time on large-scale systems due to high network latencies and imbalanced computation.

D. Challenge 4: Massive Datasets

Current commercial graph databases exceed 20 TB as an in-memory representation. Such large problems exceed the capabilities of even a rack of computational nodes of any type, which requires a large-scale multi-node platform to

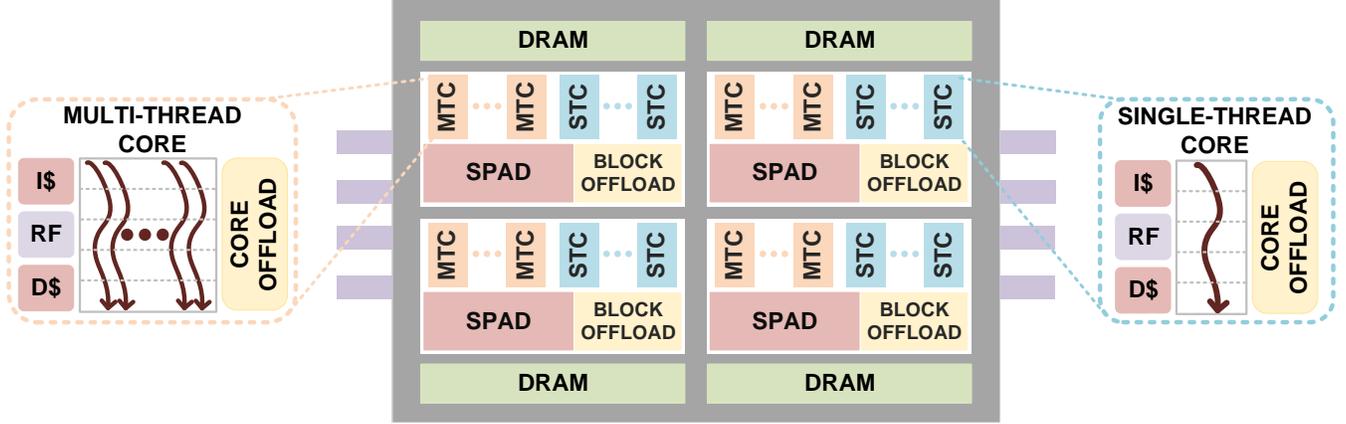


Fig. 3. High-level diagram of PIUMA architecture.

even house the graph’s working set. When combined with the prior observations – poor memory hierarchy utilization, high control flow changes, frequent memory references, and abundant synchronizations – any architecture that targets graph workloads must focus on reducing latency to access remote data, combined with latency hiding techniques in the processing elements.

Although the analysis in this section focuses on CPUs, the same challenges apply for GPUs: sparse accesses prevent memory coalescing, branches cause thread divergence and synchronization limits thread progress. Nevertheless, GPUs usually perform better on graph algorithms than CPUs for small graphs [5], because they have more threads, which hides memory latency, and much higher memory bandwidth, brute-forcing the inefficient bandwidth utilization. However, GPUs have limited memory capacity and limited scale-out capabilities, which means that they are unable to process large, multi-TB graphs. Furthermore, graphs are extremely sparse ($\ll 1\%$ non-zeros), so the typical GPU trick to densify the connectivity matrix for a more efficient GPU execution leads to another few orders of magnitude increase in memory usage, restricting it to small graphs only. PIUMA directly operates on sparse data (e.g., CSR).

III. PIUMA ARCHITECTURE

The observations on graph analysis workloads guided the PIUMA design, targeting breakthrough performance per Watt for graph analytics. We discuss how each component of the UMA architecture is designed to cope with the challenges imposed by graph workloads.

A. PIUMA Cores

The design of PIUMA cores builds on the observations that most graph workloads have abundant parallelism, are memory bound and are not compute intensive. These observations call for many simple pipelines, with multi-threading to hide memory latency, see Figure 3. PIUMA multi-threaded cores (MTC)

are round-robin multi-threaded in-order pipelines [6]. At any moment, each thread can only have one in-flight instruction, which considerably simplifies the core design for better energy efficiency. Single-threaded cores (STC) are used for single-thread performance sensitive tasks, such as memory and thread management threads (e.g., from the operating system). These are in-order stall-on-use cores that are able to exploit some instruction and memory-level parallelism, while avoiding the high power consumption of aggressive out-of-order pipelines. Both core types implement the same custom RISC instruction set.

Each core has a small data and instruction cache (D\$ and I\$), and a register file (RF) to support its thread count. Because of the low locality in graph workloads, no higher cache levels are included, avoiding useless chip area and power consumption of large caches. For scalability, caches are not coherent across the whole system. It is the responsibility of the programmer to avoid modifying shared data that is cached, or to flush caches if required for correctness. MTCs and STCs are grouped into *blocks*, each of which has a large local scratchpad (SPAD) for low latency storage. Programmers are responsible for selecting which memory accesses to cache (e.g., local stack), which to put on SPAD (e.g., often reused data structures or the result of a DMA gather operation) and which not to store locally. There are no prefetchers to avoid useless data fetches and to limit power consumption. Instead, the offload engines described below can be used to efficiently fetch large chunks of useful data.

B. Offload Engines

Although the MTCs hide some of the memory latency by supporting multiple concurrent threads, their in-order design limits the number of outstanding memory accesses to one per thread. To increase memory-level parallelism and to free more compute cycles to the cores, a memory offload engine is added to each block. The offload engine performs memory operations typically found in many graph applications in the background, while the cores continue with their computations. The direct memory access (DMA) engine performs operations

such as (strided) copy, scatter and gather. Queue engines are responsible for maintaining queues allocated in shared memory, alleviating the core from atomic inserts and removals. They can be used for work stealing algorithms and dynamically partitioning the workload. Collective engines implement efficient system-wide reductions and barriers. Remote atomics perform atomic operations at the memory controller where the data is located, instead of burdening the pipeline with first locking the data, moving the data to the core, updating it, writing back and unlocking. They enable efficient and scalable synchronization, which is indispensable for the high thread count in PIUMA.

The engines are directed by the PIUMA cores using specific PIUMA instructions. These instructions are non-blocking, enabling the cores to perform other work while the operation is done in the background. Custom polling and waiting instructions are used to synchronize the threads and offloaded computations.

C. Memory Organization

Sparse and irregular accesses to a large data structure are typical for graph analysis applications. Therefore, accesses to remote memory should be done with minimal overhead. PIUMA implements a hardware distributed global address space (DGAS), which enables each core to uniformly access memory across the full system (multiple PIUMA nodes) with one address space. Besides avoiding the overhead of setting up communication for remote accesses, a DGAS also greatly simplifies programming, because there is no implementation difference between accessing local and remote memory. Address translation tables (ATT) contain programmable rules to translate application memory addresses to physical locations, to arrange the address space to the need of the application (e.g., address interleaved, block partitioned, etc.).

The memory controllers (one per block) are redesigned to support native 8-byte accesses, while supporting standard cache line accesses as well. Fetching only the data that is actually needed reduces memory bandwidth pressure and utilizes the available bandwidth more efficiently.

D. Network

The network connecting the blocks is responsible for sending memory requests to remote memory controllers. Similar to the memory controller, it is optimized for small 8 byte messages. Furthermore, due to the high fraction of remote accesses, network bandwidth exceeds local DRAM bandwidth, which is different from conventional architectures that assume higher local traffic than remote traffic.

To obtain high bandwidth and low latency to remote blocks, the network needs to have a high radix and a low diameter. This is achieved with a HyperX topology [7], with all-to-all connections on each level. Links on the highest levels are optical to ensure power-efficient, high-bandwidth communication. The hierarchical topology and optical links enable PIUMA to efficiently scale out to many nodes, maintaining easy and fast remote access.

E. Comparison to other Graph Processors

The Cray Urika-GD graph processor [8] was one of the first commercial graph-oriented big data processors. Similar to PIUMA, it consists of multiple many-threaded cores with no large caches and a memory-coherent network. It does not support fine-grained 8 byte accesses, wasting bandwidth on loading full cache lines. Furthermore, it has no offload memory engines, such as the DMA, QMA and remote atomics as in PIUMA, leading to more memory stalls in the pipelines. As we will show in the results section, 8 byte accesses and offload memory engines are important contributors to PIUMA's performance and energy efficiency.

The Emu architecture [9] is a recently proposed architecture for big data analysis, including graph analysis workloads. Similar to PIUMA and Urika-GD, it consists of many small cores with many hardware threads per core to hide memory latency. It also features 8 byte DRAM accesses and is completely cacheless. Unique is its low-overhead thread migration scheme, which enables moving threads to a core near to the memory controller that owns the required data instead of moving the data to the current core. Moving threads to data is beneficial if the overhead of moving the thread is compensated by the amount of locally consumed data. Young *et al.* [10] report that migrating a thread involves moving 200 bytes, which means that at least 25 local 8 byte accesses are needed to compensate for the thread migration. Therefore, optimizing data locality is crucial for obtaining good performance on Emu [10], which is often hard to obtain for graph analysis applications. In contrast, PIUMA does not rely on any locality. Instead, it uses the offload engines to perform complex system-wide memory operations in parallel, and only moves the data that is eventually needed to the core that requests it. For example, a DMA gather will not move the memory stored indices or addresses of the data elements to gather to the requesting core, only the requested elements from the data array.

Song *et al.* [11] propose a graph processor based on sparse matrix algebra, building on the observation that many graph applications can be represented as operations on sparse matrices. Their architecture has overlaps with PIUMA, such as the absence of caches, and fine-grained communication and memory accesses. Graphicionado [12] is a graph analysis accelerator, implementing a vertex-centric compute paradigm. While these accelerators are likely more energy efficient for analyzing small graphs, PIUMA's goal is to provide a flexible instruction set architecture, optimized for typical graph analysis operations, and is not limited to algorithms that use sparse matrix algebra or vertex-centric operations. Furthermore, none of these proposals scale out to multi-TB graphs with trillions of vertices.

IV. HARDWARE/SOFTWARE CO-DESIGN

Crucial for the pathfinding and development of PIUMA was the hardware/software co-design process. This process requires the involvement of multiple multi-disciplinary teams: architects, system software developers, workload analysis teams, and performance simulation and analysis teams.

We developed a C/C++ compiler based on LLVM that supports the RISC ISA of PIUMA, including basic library functions. PIUMA specific operations, such as the offload engines and remote atomics, are accessible using intrinsics. Additionally, we constructed a runtime environment that implements basic memory and thread management, supporting common programming models, such as gather-apply-scatter, task-based and SPMD-style parallelism.

In parallel, we developed an architectural simulator for PIUMA, simulating the timing of all instructions in the pipelines, engines, memory and network, based on the hardware specifications. Next to performance estimations of running a workload on PIUMA, it provides an extensive set of performance analysis reports, such as CPI stacks and detailed performance information on each memory structure and each instruction. This enables workload owners to quickly detect bottleneck causes, and to use these insights to optimize the workload for PIUMA and report hardware bottlenecks to the hardware design team. The hardware team then responds with an updated design, feeding a continuous cycle of gradual improvements to hardware and software.

V. PERFORMANCE RESULTS

We implemented a variety of graph analysis applications for PIUMA and evaluated their performance using our detailed timing simulator. The applications are selected from a list of high priority workloads suggested by DARPA for the HIVE project, as well as common sparse applications. We compare the estimated PIUMA performance with timing measurements of the same applications on a high-end Intel Xeon server (Intel Xeon Gold 6140), containing 4 sockets of 18 cores each. Initial power estimates show that a PIUMA node, containing 256 PIUMA blocks, consumes about the same amount of power as the Xeon server. Therefore, a performance comparison between 1 PIUMA node and the Xeon server is also an energy efficiency comparison.

Because most applications are basic kernels, the baseline Xeon implementations are hand-crafted and optimized, avoiding potential library overhead. The PIUMA implementations are also written from scratch. For Xeon, graph applications do not scale well beyond a single node, with even worse performance for small node counts, due to the overhead of the fine-grained communication [13]. For PIUMA, the application code does not need to change for multinode execution, thanks to the system-wide shared memory. Our simulator can practically simulate systems up to a few dozen blocks, we use an analytical model to extrapolate performance for larger systems. The analytical model takes into account increased network latencies and network bandwidth restrictions as the system scales out.

PIUMA specifically targets scale-out and tera-to-peta-scale workloads, which is why we do not compare against GPUs and other graph accelerators, because these do not support such large workloads.

A. SpMV analysis

To provide insight into how the PIUMA design choices have an impact on performance, we first perform a detailed

TABLE I
SINGLE NODE PIUMA SPEEDUP FOR SpMV VERSIONS.

	Base	Selective caching	With DMA
Versus Xeon	10.0×	19.8×	29.2×
Versus Base	1×	2.0×	2.9×

analysis for sparse matrix dense vector multiplication (SpMV). The basic operation of SpMV is very similar to that of the PageRank algorithm: a multiply-accumulate of sparse matrix elements (the (weighted) graph edges) and a dense vector (the pagerank values). We implement multiple PIUMA versions for SpMV, gradually adding more PIUMA specific operations, to show the impact of each optimization individually. We apply SpMV on an RMAT-30 synthetic matrix, stored in compressed sparse row (CSR) format, both on Xeon and PIUMA. Table I shows the speedup of each of the implementations versus Xeon.

The first PIUMA version is a straightforward implementation of SpMV, with each thread calculating one or more elements of the result vector. The rows are partitioned across the threads based on the number of non-zeros for a balanced execution. It does not make use of DMA operations, and all accesses are non-cached 8-byte, which is the default for PIUMA (except for thread local stack accesses, these are cached by default). This basic implementation already outperforms Xeon by a factor 10 through using a higher thread count and 8 byte memory accesses, avoiding memory bandwidth saturation.

The next implementation uses selective caching: accesses to the matrix values are cached, while the sparse accesses to the vector bypass caches. In the compressed sparse row (CSR) representation of a sparse matrix, all non-zero elements on a row are stored consecutively and accessed sequentially, resulting in spatial locality. The dense vector, on the other hand, is accessed sparsely, because only a few of its elements are needed for the multiply-accumulate, *i.e.*, the indices of the non-zeros in the row of the matrix. So we cache the accesses to the matrix, while the vector accesses remain uncached 8 byte accesses. This leads to another 2× speedup. We also simulated caching *all* accesses, which led to *lower* performance compared to the base version. Caching the sparse accesses to the vector causes an increase in memory traffic, because 64 bytes are fetched for every access. The extra traffic saturates the bandwidth, leading to lower overall performance. This experiment shows why uncached 8-byte accesses are required to achieve higher performance and better efficiency for sparse graph applications.

Lastly, we use a DMA gather operation to fetch the elements of the dense vector that are needed for the current row from memory, and put them on local scratchpad. The multiply-accumulate reduction is then done by the core, fetching the matrix elements from cache and the vector elements from scratchpad. Not only does this significantly reduce the number of load instructions, it also reduces data movement: the index list does not need to be transferred to the requesting core, only the final gathered vector elements. While data is gathered, the thread is stalled, allowing other threads that have already fetched their data to compute a result vector element. Using the

TABLE II
PIUMA SPEEDUP VERSUS ONE 4-SOCKET XEON NODE.

Application	1 node	16 nodes
Application Classification	6.9×	111×
Random Walks	279×	2,606×
Graph Search	34×	544×
Louvain Community	41×	555×
TIES Sampler	93×	419×
Graph2Vec	42×	178×
Graph Sage	3.1×	46×
Graph Wave	8.0×	125×
Parallel Decoding FST	6.8×	109×
Geolocation	15×	243×
SpMV	29×	467×
SpMSPV	111×	1,387×
Breadth-first Search	7.5×	117×

DMA gather offload improves performance by 47%, leading to a total 29× speedup versus Xeon. This implementation uses more than 95% of the available memory bandwidth, while not wasting bandwidth on useless and sparse accesses.

B. All applications

Table II shows the speedup of a single PIUMA node versus the Xeon server for all evaluated applications. It also shows the speedup of a PIUMA system with 16 nodes, which is the first proof-of-concept system that will be built for the HIVE project. The base is still the single node Xeon performance, we expect that the performance of a 16-node Xeon will not be much higher than that of a single node for graph applications.

Single node PUMA performance exceeds Xeon performance by one to two orders of magnitude. The performance benefit of low-compute low-locality applications, such as Random Walks, is the highest, while more compute-intensive applications, such as Application Classification, benefit less, but still outperform Xeon. The main reasons are the much higher thread count support (144 threads for Xeon, more than 16K threads for PIUMA), enabling threads to progress while others are stalled on memory operations, efficient small size local and remote memory operations, and powerful offload engines that allow for more memory/compute overlap. The speedups for 16 nodes show that PIUMA scales out well. Scaling is not perfectly linear, due to the larger latencies and bandwidth restrictions, but it significantly outperforms conventional multinode Xeon configurations.

VI. CONCLUSION

PIUMA is a graph analysis oriented architecture developed by Intel in response to the DARPA HIVE project. Based on the observation that graph workloads are dominated by irregular sparse accesses, it features many highly-threaded simple cores to hide the latency of remote memory accesses. Combined with small access granularity to memory and network, and powerful offload engines, PIUMA outperforms current high-end processors for typical graph workloads. Furthermore, it is designed to scale out efficiently thanks to the high bandwidth network and shared address space, increasing the performance gap

with current multinode computers, which perform poorly on distributed graph applications. The effective hardware/software co-design process of PIUMA guarantees highly optimized hardware, *and* ensures that system and development tools are available by the time PIUMA will be released.

ACKNOWLEDGEMENT

This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

REFERENCES

- [1] D. Reinsel, J. Gantz, J. Rydning, “The Digitization of the World – From Edge to Core,” ser. An IDC White Paper – #US44413318, Nov. 2018.
- [2] M. U. Nisar, A. Fard, and J. A. Miller, “Techniques for graph analytics on big data,” in *2013 IEEE International Congress on Big Data*, June 2013, pp. 255–262.
- [3] L. Page, S. Brin, R. Motwani, and T. Winograd, “The PageRank citation ranking: Bringing order to the web.” Stanford InfoLab, Tech. Rep., 1999.
- [4] W. Shen, “Hierarchical identify verify exploit (HIVE).” [Online]. Available: <https://www.darpa.mil/program/hierarchical-identify-verify-exploit>
- [5] H. Liu and H. H. Huang, “Simd-x: Programming and processing of graph algorithms on gpus,” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Jul. 2019, pp. 411–428.
- [6] A. Snaveley, L. Carter, J. Boisseau, A. Majumdar, K. S. Gatlin, N. Mitchell, J. Feo, and B. Koblenz, “Multi-processor performance on the Tera MTA,” in *SC’98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. IEEE, 1998.
- [7] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber, “HyperX: topology, routing, and packaging of efficient large-scale networks,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 41.
- [8] A. Kopsper and D. Vollrath, “Overview of the next generation Cray XMT,” in *Cray User Group Proceedings*, 2011, pp. 1–10.
- [9] T. Dysart, P. Kogge, M. Deneroff, E. Bovell, P. Briggs, J. Brockman, K. Jacobsen, Y. Juan, S. Kuntz, R. Lethin, J. McMahon, C. Pawar, M. Perrigo, S. Rucker, J. Rutenber, M. Rutenber, and S. Stein, “Highly scalable near memory processing with migrating threads on the Emu system architecture,” in *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA³ ’16, 2016, pp. 2–9.
- [10] J. S. Young, E. Hein, S. Eswar, P. Lavin, J. Li, J. Riedy, R. Vuduc, and T. Conte, “A microbenchmark characterization of the Emu chick,” *Parallel Computing*, vol. 87, pp. 60 – 69, 2019.
- [11] W. S. Song, V. Gleyzer, A. Lomakin, and J. Kepner, “Novel graph processor architecture, prototype system, and results,” in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, 2016, pp. 1–7.
- [12] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, “Graphicionado: A high-performance and energy-efficient accelerator for graph analytics,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.
- [13] B. A. Page and P. M. Kogge, “Scalability of hybrid sparse matrix dense vector (SpMV) multiplication,” in *2018 International Conference on High Performance Computing Simulation (HPCS)*, July 2018, pp. 406–414.