The Intel® Programmable and Integrated Unified Memory Architecture (PIUMA) Graph Analytics Processor

Sriram Aananthakrishnan, Shamsul Abedin, Vincent Cavé, Fabio Checconi, Kristof Du Bois, Stijn Eyerman, Joshua B. Fryman, Wim Heirman,[*] Jason Howard, Ibrahim Hur, Samkit Jain, Marek M. Landowski, Kevin Ma, Jarrod Nelson, Robert Pawlowski, Fabrizio Petrini, Sebastian Szkoda, Sanjaya Tayal, Jesmin Jahan Tithi, Yves Vandriessche
Intel Corporation

***Abstract*—High performance large scale graph analytics are essential to timely analyze relationships in big data sets. Conventional processor architectures suffer from inefficient resource usage and bad scaling on those workloads. To enable efficient and scalable graph analysis, Intel® developed the Programmable Integrated Unified Memory Architecture (PIUMA) as a part of the DARPA Hierarchical Identify Verify Exploit (HIVE) program. PIUMA consists of many multi-threaded cores, fine-grained memory and network accesses, a globally shared address space, powerful offload engines and a tightly integrated optical interconnection network. This paper presents the PIUMA architecture, and documents our experience in designing and building a prototype chip and its bring-up process. PIUMA silicon has successfully powered on demonstrating key aspects of the architecture, some of which will be incorporated into future Intel products.**

■ **CURRENT PRACTICES** in data analytics and artificial intelligence (AI) 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. However, 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*. 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*. The resulting access stream (Figure 1b) is irregular and has no locality, making conventional prefetching and caching useless.

Traditionally, algorithmic analysis is based on the principle that compute is precious and

[*] Corresponding author: `wim.heirman@intel.com`

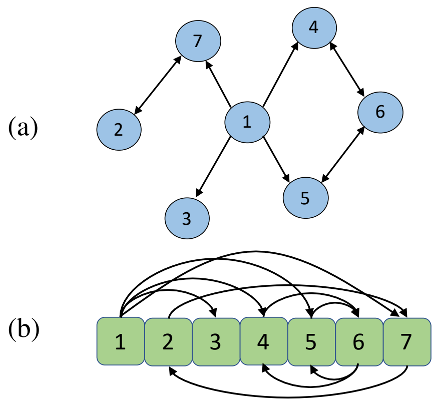


Figure 1. (a) A sparse graph with directed edges, (b) Memory access patterns observed when moving data along the edges of (a)

communications are free. Yet, today's physics of implementation shows mostly the opposite. This is especially relevant for at-scale problems in AI and high-performance computing (HPC), which are showing this trend clearly, exhibiting very low utilization on "classical" dense architectures. The combination of low performance and very large graph sizes limits the practical use of graph analytics. Recognizing both the increasing importance of this field, and the need for vastly improved sparse computation performance compared to traditional approaches, DARPA launched their Hierarchical Identify Verify Exploit (HIVE) program to achieve at least 1000× Performance/Watt breakthrough on such large problems before the end of 2022.

This paper introduces Intel's response to this challenge with its design called Intel® Programmable Integrated Unified Memory Architecture (PIUMA). The PIUMA machine is designed for graph analytics at massive scales. PIUMA enables high-performance graph processing by addressing limitations across the network, memory, and compute architectures that typically limit performance on graph workloads.

1. Challenges

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

1.1. Challenge 1: Cache and Bandwidth Utilization

Graph analysis applications, when executed on a conventional cache based processor with prefetcher, typically waste a large fraction of main memory bandwidth [7]. For every 64-byte cache line fetched from memory, often just eight bytes or less are used because many data loads are sparse word-sized accesses with no spatial locality. A typical pattern in graph applications is a chain of indirect loads [10], similar to a pointer chasing pattern: a vertex's neighbors are stored in a list, which are used to index the data array. Since neighbor lists do not show regularity or locality, accesses to the data array are intrinsically sparse. Other memory access behaviors exhibit increased locality (*e.g.,* fetching the neighbor list itself), leading to spatial (but no temporal) cache line reuse. These lists are limited in size, causing a high rate of useless prefetches that extend 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. Over-provisioning 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.

1.2. 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.

Analysis reveals that graph applications are heavy on branches and memory operations [4].

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: data dependencies between cache misses limit the amount of instruction-level parallelism, while hard-to-predict data-dependent branches restrict the amount of useful speculation [5]. In PIUMA, this observation was the incentive to use single issue in-order pipelines with many threads to hide memory latency and avoid speculation.

1.3. 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.

1.4. 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 and would require a large-scale multi-node platform to 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, for small graphs, GPUs usually perform better on graph algorithms than CPUs [9], because they have more threads, which hides memory latency, and much higher memory bandwidth, brute-forcing the inefficient bandwidth utilization. However, GPUs also have limited memory capacity and 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 densifying the adjacency matrix for an 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.,* compressed sparse row (CSR) format) which avoids the need for densification.

2. Introducing PIUMA

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 PIUMA architecture is designed to cope with the challenges imposed by graph workloads.

2.1. 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. Figure 2 shows a block diagram of the PIUMA archirecture. PIUMA multi-threaded cores (MTC) are round-robin multi-threaded in-order pipelines. At any moment, each thread can only have one in-flight instruction (implicitly making them stall-on-miss at the thread level), which considerably simplifies the core de-

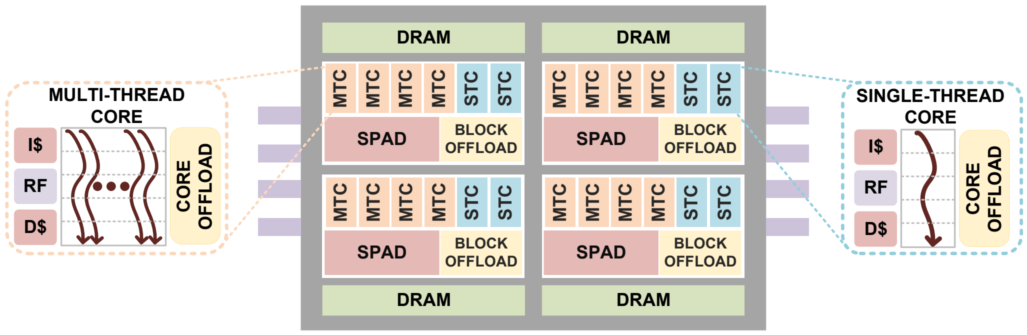


Figure 2. High-level diagram of the PIUMA architecture

sign for better energy efficiency. Single-threaded cores (STC) are used for single-thread performance sensitive tasks, such as memory and thread management (*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 which enables easy thread migration.

Each core has a small data and instruction cache (D$ and I$), and a large register file (RF) with 32 registers per thread. Because of the low locality in graph workloads, no higher cache levels are included, avoiding useless chip area and power consumption of large caches. Special cache instructions are built into the instruction set to support I$ and D$ software prefetches, invalidations, and write-backs. A MOESI-F protocol [11] maintains coherency across all data-caches on the die, and allows for migration of dirty cache lines between data caches. For scalability, caches are not coherent across the whole system. It is the responsibility of the programmer to avoid modifying shared data that are 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 can manipulate address bits to point to specific memory map regions (scratchpad, main memory, configuration registers, etc.). A cache bit determines whether the memory access is cached. The runtime language can choose to *e.g.,* cache the execution stack and not cache static data by default. There are no hardware 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.

2.2. Offload Engines

Although the MTCs hide much 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, multiple memory offload engines are 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 and scatter/gather. This engine has the capability to interpret various compressed sparse representations commonly used in neighbor lists (*e.g.,* CSR) and perform other data transformations (*e.g.,* transpose, basic arithmetic operations, etc.).

Indirect operations accelerate pointer chasing (*e.g.,* `A[B[i]]`). Conventionally, this is done in three steps: an offset (`B[i]`) is loaded from memory into a register, the load/store address is computed by the core by shifting the loaded offset and adding it to the base address (`A`), and finally the load or store operation is performed on the created address. ISA support for indirect loads and stores [10] eliminates the need to bring the offset to the core by performing these three steps on the fly. This can save a round-trip to remote memory when both the offset and data values reside at the same remote memory controller.

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 or scratchpad 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 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.

2.3. 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 with one address range. 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.

2.4. 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, a hierarchical network with all-to-all connections on each level. To ensure power-efficient, high-bandwidth communication, optical links are used that are tightly integrated into the PIUMA chip package. The hierarchical topology and optical links enable PIUMA to efficiently scale out to many nodes, maintaining easy and fast remote access.

2.5. 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 consisted of multiple many-threaded cores with no large caches and a memory-coherent network. It did not support fine-grained 8-byte accesses, wasting bandwidth on loading full cache lines. Furthermore, it had no offload memory engines, such as the DMA, queue and remote atomics in PIUMA, leading to more memory stalls in the pipelines.

The Emu architecture [3] is a recently proposed architecture for big data analysis, including graph analysis workloads. It 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. Therefore, optimizing data locality is crucial for obtaining good performance on Emu, 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.

Song *et al.* [12] 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 [6] 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 not to be 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.

3. Software stack

The PIUMA software stack includes all the tools necessary for developers to write, compile, execute, and debug codes. Unlike most accelerators, the PIUMA hardware and its programming stack do not impose upon the programmer a restricted parallel programming model, derived from hardware limitations. Instead, it offers flexibility in leveraging host and accelerator resources, and takes a layered approach. Each abstraction layer offers a trade-off between programmer control and productivity, catering to a range of developer audiences.

Standalone x86 C/C++ workloads are usually straightforward to port to the PIUMA software stack, often simplifying the code in the process by removing the need for complex locality-improving datastructures.

3.1. PIUMA Software Development Kit

The PIUMA Software Development Kit (SDK) provides developers with a fully featured software stack composed of a set of familiar C and C++ programming interfaces, toolchain, runtime, driver, supporting libraries, and debugger.

The SDK tools and libraries make extensive use of PIUMA hardware-backed features such as atomics, queues, collectives, and DMAs. Additionally, some of the hardware features are automatically leveraged by the toolchain. For example, code can be compiled to automatically make use of PIUMA's indirect-load [10] and bitwise instructions or could directly use PIUMA-specific compiler builtins to access custom RISC ISA instructions.

Since the PIUMA accelerator is deployed alongside a host, developers decide whether to program PIUMA in standalone or hybrid mode.

In standalone mode, the full program is compiled to run natively on PIUMA. The LLVM-based PIUMA toolchain contains the expected suite of tools: a compiler, binary utilities (assembler, linker, ELF tools, etc.), and support libraries (libc, libcxx, libunwind, etc.) tailored for PIUMA.

PIUMA-native binaries can be launched as POSIX-style processes through a convenience launcher tool on the x86 Host. This launcher relies on more general user-space libraries (Host API) providing management, communication, and debugging support on top of the PIUMA device driver. On the accelerator's side, the program is handed off to the PIUMA runtime, which manages hardware resources and program executions, fulfilling the role of a light-weight operating system.

In hybrid mode, the main program runs on x86 and coordinates the execution of tasks or kernels with the PIUMA accelerator using the Host-API. The Host-API covers basic functionalities such as dynamically allocating memory, executing data transfers in and out, as well as scheduling parallel functions for execution.

3.2. Programming Interfaces

Most programmers will want to use one of the supported parallel programming interfaces: a single program, multiple data (SPMD) programming model, OpenMP, or graph algorithms plugins for Anaconda's Metagraph.

The PIUMA SPMD programming model bridges the gap between system-level programming and application-level programming, implementing enough of a runtime and OS-like functionality to offer the user a familiar (shared-memory) system view and parallel programming approach (bulk-synchronous style). The PIUMA SPMD layer is a good target for developers that require low overheads and tight control, as it allows the underlying runtime to scale well to a large number of threads. A PIUMA SPMD program is a standard C program: the `main()` function is the entry point and is executed sequentially. There, developers can use the provided SPMD API to execute user-defined functions across the available hardware threads. The SPMD programming layer API also provides a thin library for: thread identification and system geometry information to orchestrate parallel computations, scratchpad and global memory allocations, point-to-point synchronization through the use of atomics and hardware queues, and global synchronization with hardware collectives.

The OpenMP layer is built on top of the SPMD layer and provides developers with increased productivity and a familiar programming environment. Some of PIUMA's hardware features find a natural fit in standard OpenMP pragmas (atomics, reductions) while others require the creation of new pragmas. For example, a pragma can be applied to a specific section of code to instruct the PIUMA compiler to accelerate code with indirect access instructions.

For domain scientists mostly interested in using off-the-shelf graph algorithms, we have developed PIUMA plugins for Anaconda's Metagraph Python library. This approach enables users to leverage PIUMA hardware from a familiar Python programming environment such as Jupyter notebooks. When invoking graph algorithms in the Metagraph framework, developers can use annotations to request the use of a PIUMA implementation. The Metagraph dispatcher then finds and executes the corresponding PIUMA plugin, which takes care of all the necessary steps to transparently offload data and compute from the host to the accelerator.

4. 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, performance simulation and analysis teams as well as FPGA emulation teams. In parallel with the hardware design and compiler development, 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.

The functional part of the simulator was created by extending FSim [2] to handle the custom RISC ISA as well as the functional emulation of the offload engines and network collectives. For performance modeling, Sniper [1] was used to model both the stall-on-use single-threaded (STC) cores as well as the stall-on-miss multi-threaded (MTC) cores. We also extended the memory model to handle scratchpads, selective caching, and the HyperX topology of the network. In addition to performance estimations of running a workload on PIUMA, it provides an extensive set of performance analysis reports, such as Cycle per instruction (CPI) stacks and detailed performance information on each memory structure and each instruction. This enabled 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 responded with an updated design, feeding a continuous cycle of gradual improvements to hardware and software. We validated the resulting simulator against the PIUMA FPGA emulation platform as well as against A0 silicon once that became available.

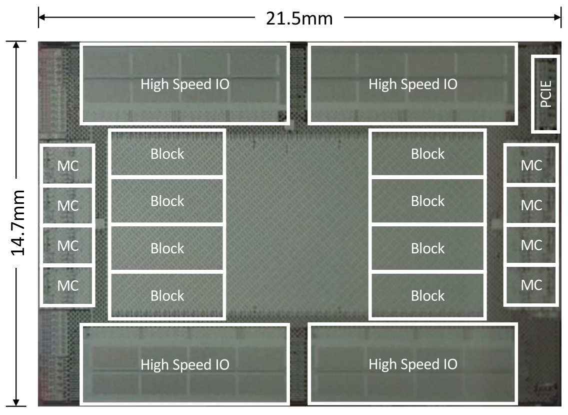
5. PIUMA A0 silicon implementation

We designed and fabricated an A0 test chip that implements the PIUMA architecture. This chip was powered on and characterized in the lab, and was healthy enough (with some software mitigations in place) to run actual workloads. A 16-chip PIUMA *node* is also being constructed to allow validation of the inter-chip network and conduct scaling experiments.

5.1. A0 hardware design

We designed and built a 27.6B-transistor, 316mm^2 prototype chip in 7nm FinFET CMOS. The chip integrates eight PIUMA blocks (running a total of 528 threads), 32MB of on-die scratchpad memory, and all off-chip interfaces (memory, network, and a PCIe link to the host system).

Memory transactions are distributed over eight narrow-channel on-die DDR5 controllers that have been optimized for 8-byte native accesses for efficient bandwidth utilization on sparse graph data sets. Main memory is built from standard



Technology	TSMC 7nm FinFET
Interconnect	15 metal layers
Die transistors	27.6B
Die area	316 mm^2
Block transistors	1.2B
Block area	9.3 mm^2
Signals	705
Package	3275-pin BGA

Figure 3. Full-chip micrograph and characteristics

DDR5 memory chips in a custom SODIMM form factor. Each of the eight on-die memory controllers can be accessed in parallel at an aggregate peak memory bandwidth of 35.2GB/s.

Four x8 high speed I/O links are divided into 32 channels for 1TB/s per direction of off-die signalling. The 32 channels are configured into a HyperX topology using the high-radix of the die to provide a low-diameter network, reducing network hops and resulting in both low latency and low energy communication. The HyperX topology seamlessly extends the on-die network by using the same packet format and network protocols, allowing for efficient system scaling. In total, the processor's HyperX network can scale out to 131,072 dies using three network levels, where nodes at each level are fully connected.

Figure 3 shows the layout of the chip. The most critical resource was the edges of the chip (shoreline) required to fit the memory controllers and off-chip network interfaces. Much of the internal area is taken up by routing of the on- and off-die networks while the cores and memories take up just 24%.

We have a healthy PIUMA A0 chip compared to other chips of this scale on a brand new architecture and a grounds up new design. Few hardware bugs have been uncovered, although some required software workarounds with significant performance impact.

5.2. Software bring-up

The software stack was fully enabled in the last phase of the A0 power-on, running complete workloads on the PIUMA bring-up boards. Because of the close co-design process, the software stack was able to be hit the ground running. The period between tape-out and power-on was used to focus on preparing the software stack for execution on the A0 platform. This involved removing simulation tricks, such as magic instructions, working around known errata and completing the host-side development framework. This quick software bring-up proved to be extremely valuable, as it enabled finding hardware and software faults that arose because of rare and complex interactions.

5.3. Learnings

A certain class of hardware errors only occur under load, for instance, states triggered by buffer back-pressure. These would typically only occur during complex workload execution scenarios on the actual hardware platform. We found that FPGA emulation or functional simulation platforms would initially not produce these execution states, mainly because of the relative speed difference between what is inside and outside the simulation. For example, hardware DDR memory or a Xeon host PCIe interface run at native speeds, so an FPGA-emulated PIUMA core running at a few tens MHz is unable to saturate these. To catch such classes of errors earlier requires validation methodology improvements.

The early software bring-up on the A0 platform did present additional challenges. Enabling a bespoke software stack on an A0 stepping of a novel hardware architecture creates a combination of unknowns; neither the hardware nor any software layers can be fully trusted. Errors arising during full-stack workload runs can be difficult and time consuming to reproduce and root cause. In such an environment, the software stack benefits from having several intermediate layers that can be sounded out incrementally. During the A0 software bring-up, several new test suites were created with increasing level of abstraction and complexity. This enabled us to test incrementally, build trust in each layer and catch complex errors in a simpler environment.

6. Conclusions

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 multi-node computers, which perform poorly on distributed graph applications.

We built an A0 test chip, powered it on in the lab and were able to run workloads on it. The effective hardware/software co-design process of PIUMA guaranteed highly optimized hardware, and ensured that system and development tools were available by the time we had silicon in the lab. Despite finding hardware bugs that required software workarounds, we were able to assess potential performance and energy efficiency targets that make PIUMA a highly capable graph analytics platform.

Acknowledgements

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Distribution Statement A—Approved for Public Release, Distribution Unlimited

REFERENCES

1. T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.
2. N. P. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I. Ganev, R. A. Golliver, R. Knauerhase *et al.*, "Runnemede: An architecture for ubiquitous high-performance computing," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2013.
3. 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. Ruttenberg, M. Ruttenberg, and S. Stein, "Highly scalable near memory processing with migrating threads on the Emu system architecture," in *Workshop on Irregular Applications: Architectures and Algorithms*, 2016.
4. S. Eyerman, W. Heirman, K. Du Bois, J. B. Fryman, and I. Hur, "Many-core graph workload analysis," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2018.
5. S. Eyerman, W. Heirman, S. Van den Steen, and I. Hur, "Enabling branch-mispredict level parallelism by selectively flushing instructions," in *International Symposium on Microarchitecture (MICRO)*, 2021.
6. T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *International Symposium on Microarchitecture (MICRO)*, 2016.
7. W. Heirman, S. Eyerman, K. Du Bois, and I. Hur, "Automatic sublining for efficient sparse memory accesses," *ACM Trans. Archit. Code Optim.*, vol. 18, Apr. 2021.
8. A. Kopser and D. Vollrath, "Overview of the next generation Cray XMT," in *Cray User Group Proceedings*, 2011, pp. 1–10.
9. H. Liu and H. H. Huang, "SIMD-X: Programming and processing of graph algorithms on GPUs," in *USENIX Annual Technical Conference*, 2019.
10. P. Ossowski, S. Szkoda, A. Perdeusz, L. Odzioba, P. Karpinski, M. Koss, M. M. Landowski, "Automatic indirect memory access instructions generation for pointer chasing patterns," 2022. [Online]. Available: https://llvm.org/devmtg/2022-11/slides/QuickTalk10-AutomaticIndirectMemoryAccessInstructions.ppsx
11. R. Pawlowski, B. Krishnamurthy, V. Cavé, J. M. Howard, A. More, and J. B. Fryman, "Multi-processor system with configurable cache sub-domains and cross-die memory coherency," U.S. Patent 10,795,819, issued October 6, 2020.
12. W. S. Song, V. Gleyzer, A. Lomakin, and J. Kepner, "Novel graph processor architecture, prototype system, and results," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2016.

Sriram Aananthakrishnan is a Software Research Engineer in Intel's Office of the CTO.

Shamsul Abedin is a Principal Engineer in Intel's Office of the CTO.

Vincent Cavé is a research scientist at Intel Corporation. His background and interests are in parallel programming models and their implementations, with a focus on whole software stack optimizations and hardware/software co-design.

Fabio Checconi is a Research Engineer in Intel Labs.

Kristof Du Bois is a research scientist at Intel Corporation, Belgium. His research interests include architectural simulation, performance evaluation, and performance analysis of future CPU architectures. Kristof received a Ph.D. degree in computer science engineering from Ghent University, Belgium in 2014.

Stijn Eyerman obtained his PhD in Computer Engineering at Ghent University in 2008 and works as a Research Scientist for Intel since 2016. His interests are in processor performance analysis, modeling and simulation.

Joshua B. Fryman received the Ph.D. degree in computer architecture from the Georgia Institute of Technology. He is currently an Intel Fellow as part of Intel's Office of the CTO with Datacenter Technologies. His research interests include novel microprocessor architecture, co-design of workloads and systems, disaggregated system architecture, high-performance computing, embedded systems, novel memory architectures, photonic chip networks, and at-scale system fabrics.

Wim Heirman obtained his PhD in Computer Engineering at Ghent University in 2008 and works as a Principal Engineer for Intel in the Extreme Scale Computing (ESC) team. He is a co-developer of the Sniper multicore simulator.

Jason Howard is a Principal Engineer for the Extreme Scale Computing (ESC) team within Intel's Office of the CTO (OCTO). His research interests include parallel architectures, exascale processing, non-traditional computer mathematics.

Ibrahim Hur is a Senior Principal Engineer in Intel's Office of the CTO. He obtained his PhD in Electrical and Computer Engineering at The University of Texas Austin. His research interests include computer architecture and performance modeling/analysis.

Samkit Jain received his MS in Electrical Engineering from University of Minnesota, Twin Cities, USA in 2014 and works as a Software Research Scientist for Intel Corporation. His research interests include parallel computer architectures, performance modeling, parallel programming and AI.

Marek M. Landowski is a software research manager at Intel Poland. His research interests focus on software for novel architectures, QoS networks and neuromorphic computing.

Kevin Ma is a Systems Engineer for the Extreme Scale Computing (ESC) team within Intel's Office of the CTO (OCTO). His research interests includes exascale system architectures, and cloud system architectures.

Jarrod Nelson is an Engineering Manager in Intel's Office of the CTO.

Robert Pawlowski obtained his PhD in Computer Engineering at Oregon State University in 2014 and works as a Research Engineer for Intel in the Extreme Scale Computing (ESC) team. His interests include computer architecture and system resiliency.

Fabrizio Petrini is a Senior Principal Engineer of the Intel Parallel Computing Labs in Santa Clara, CA. His research interests include data-intensive algorithms for graph analytics and sparse linear algebra, Exascale computing, high performance interconnection networks and novel architectures.

Sebastian Szkoda is a Software Research Scientist for the Extreme Scale Computing (ESC) team at Intel Corporation. His research interests include high performance computing and software optimization.

Sanjaya Tayal is the Director of Engineering at Intel and leads the Extreme Scale Computing Group. He is engaged in building advanced research prototypes with novel architectures and technologies.

Jesmin Jahan Tithi is an AI Research Scientist at Intel, specializing in high-performance computing and software-hardware codesign for next-generation processors for large-scale machine learning and graph applications. She completed her Ph.D. from Stony Brook University and is also a founding member of Z-inspection, a process for trustworthy and ethical AI assessment.

Yves Vandriessche works on performance analysis and high-performance software development. He has a Science PhD from the Vrije Universiteit Brussel (VUB) and works as a research scientist at Intel Corporation.