Utilizing Moving Compute to Data Model to Improve Scaling of Graph and Machine Learning Algorithms in QUARQ Multicore Architecture at 1000-cores Scale

Halit Dogan*, Brian Kahne†, Omer Khan*
†NXP Semiconductors, Austin, TX, USA
*University of Connecticut, Storrs, CT, USA

***Abstract*—The recent success of graph analytics and machine learning on challenging problems have provoked both academia and industry to design efficient specialized hardware. In this context, many accelerator designs have been proposed. Instead of designing a specialized hardware, we propose a general purpose multicore architecture called QUARQ that provides state-of-the-art performance for graph and neural network workloads. QUARQ is a tiled multicore architecture that supports both hardware cache coherence and explicit messaging capabilities. QUARQ enables scalable computation, data access, and synchronization. For computation, QUARQ enables multiple short-SIMD (64-bit) pipelines per tile. Each short-SIMD can perform four 16-bit precision MAC operations. QUARQ enables an intelligent coherent cache hierarchy that exploits data reuse at the private cache levels, and caches the dataset on-chip in last-level cache, avoiding expensive off-chip memory accesses. QUARQ also enables an extremely scalable thread synchronization paradigm that utilizing in-hardware explicit messaging on top of shared memory. A moving computation to data model (MC) enables fine-grain execution of critical code sections at dedicated cores. This paper evaluates a prototype implementation of the QUARQ architecture's MC model on the Tilera TILE-Gx72 multicore platform, as well as a multicore simulator to evaluate graph and machine learning workloads at the 1000–cores scale. The QUARQ architecture is empirically compared against several competitive single-chip platforms, such as NVidia GPUs and Intel multicores.**

I. Introduction

The recent success of graph analytics on real world graphs, and deep neural networks (DNNs) on computer vision [1] [2] [3] and natural language processing [4] have attracted the attention of both academia and industry. In this context, many accelerators are proposed for both high performance [5] and low energy applications [6] [7] [8]. Specially, GPUs are shown to be effective in processing of DNNs due to their high floating point operations (FLOP) rate, memory bandwidth, and large concurrency capabilities.

We have proposed a general-purpose multicore architecture called QUARQ in [9] and [10]. The overview of a tile of the system can be seen in Figure 1. The proposed system is a tiled multicore that combines hardware cache coherence with in–hardware explicit messaging for low-latency, non-blocking core–to–core communication. It utilizes the RISCV ISA with extensions for explicit messaging instructions. We also extended the core with 4–way short–SIMD execution unit to increase FLOP rating of the system. In addition, since it is shown in the literature that 16–bit floating point is enough for neural networks [11], support for 16–bit floating point is added to reduce the pressure on caches. Each SIMD instruction performs up to four 16–bit floating point operations. To accelerate communication between cores,

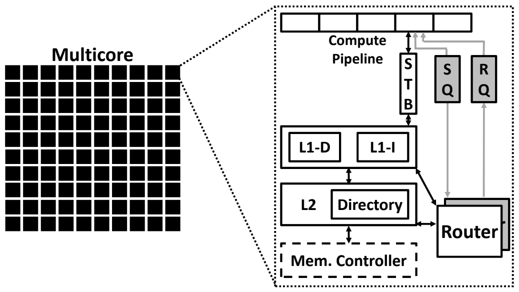
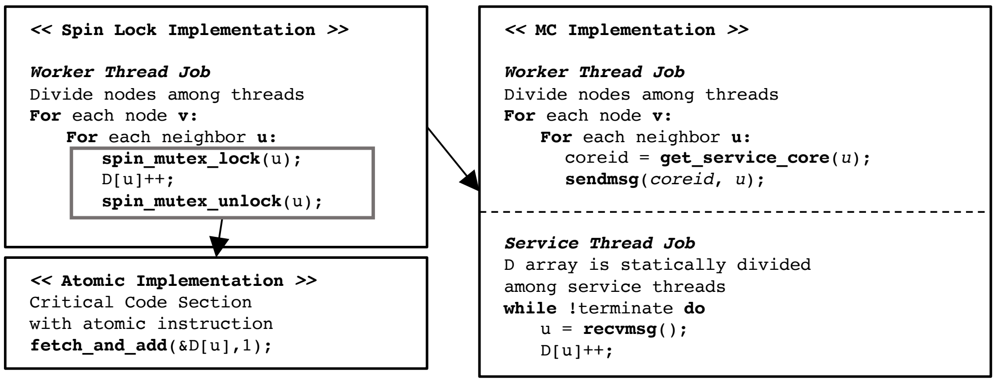


Figure 1: Overview of the QUARQ multicore architecture.

four basic explicit messaging instructions are added to the ISA and implemented at the hardware level.

1) The *send* is a non-blocking instruction that requires a destination address along with the data to be sent from a sender core to a receiver core. Both destination address and data are setup in the register file explicitly using load/store instructions that precede the send instruction. A message is composed by reading the register file, and inserted into a send queue for transmission on the on-chip network. 2) Each send instruction is paired with a blocking receive instruction (*recv*) at the receiver core. When a new message is received at a core, it is buffered in a receive queue until handled via a receive instruction. A core's pipeline is stalled if it gets to a *recv* instruction but hasn't received the message yet. The receive instruction loads the message contents into already setup registers by the programmer, and an ACK message is generated and sent back to the original sender core to enforce flow-control. 3) Blocking send with rendezvous instruction (*sendr*) is similar to a send instruction with the exception that it always blocks the compute pipeline until an explicit reply is received from the destination thread. 4) Non-blocking resume rendezvous instruction (*resumer*) is used to respond to a *sendr* instruction from a sender. More details about the explicit messaging support on top of shared memory architecture can be found in [9].

The explicit messaging protocol is used to deploy a novel thread synchronization model that moves compute to data (MC) to improve performance scalability at 1000–cores scale. The MC model moves a critical code section to a dedicated thread called *service thread*. The shared data to be updated in the critical code section is pinned to the core executing the *service thread*. The *worker threads* request execution of the critical code section from the *service thread* via non-blocking low–latency messages without any involvement of the cache coherence protocol. Using a single thread may lead to higher serialization overhead of critical code section execution, hence multiple threads are assigned to perform the *service*



```
<< Spin Lock Implementation >>

Worker Thread Job
Divide nodes among threads
For each node v:
   For each neighbor u:
      spin_mutex_lock(u);
      D[u]++;
      spin_mutex_unlock(u);
```

```
<< Atomic Implementation >>
Critical Code Section
with atomic instruction
fetch_and_add(&D[u],1);
```

```
<< MC Implementation >>

Worker Thread Job
Divide nodes among threads
For each node v:
   For each neighbor u:
      coreid = get_service_core(u);
      sendmsg(coreid, u);
-----------------------------------------
Service Thread Job
D array is statically divided
among service threads
while !terminate do
   u = recvmsg();
   D[u]++;
```

Figure 2: Pseudo code of triangle counting (TC) using **Spin**, **Atomic** and **MC** models.

thread work to alleviate the impact of serialization. Since the MC model pins shared data to *service threads*, it eliminates the cache line ping–pong in traditional thread synchronization primitives, and enhances the data locality for shared data. Moreover, the MC model overlaps communication latency with other stalls and useful works by allowing *workers* to send non–blocking critical section requests. Consequently, it is anticipated to scale to higher core counts as compared to the traditional spin-locks (**Spin**), and atomic instructions (**Atomic**) based synchronization models.

As representative applications, Single Source Shortest Path (SSSP) and Triangle Counting (TC) from graph processing domain, and a classical DNN AlexNet [3] and SqeueezeNet [12] are employed to show the performance advantage of the QUARQ architecture. The MC model is first realized on the Tilera TILE-Gx72 multicore platform that incorporates in–hardware explicit messaging on top of hardware cache coherent shared memory paradigm. Next, an industry-class multicore simulator is configured to characterize QUARQ at 1000–cores scale. The QUARQ architecture is empirically compared against competitive single-chip machines, such as NVidia GPUs and Intel multicores.

II. Application Illustrations

A. Triangle Counting (TC)

Triangle counting (TC) algorithm is ported from the CRONO benchmark suite [13]. Figure 2 shows the implementation of TC using the various synchronization models in the QUARQ architecture. As seen in the upper left box, the nodes are divided among threads, and the threads calculate the triangles in their chunk of the graph. The algorithm performs critical section for each neighbor, which results in acquiring a lock multiple times for each node. Therefore, synchronization on shared data is expected to be high for this algorithm. As a result, the lock acquisition overhead is elevated due to retries and cache line ping–pong for both shared data and the lock variables. Implementing the algorithm using lock–free data structures by employing the atomic fetch–and–add (FAA) instruction (lower left box in the figure) removes the overhead of acquiring locks as the atomic FAA instruction does not fail. However, the shared data itself still ping-pongs between cores.

The MC model implementation is presented in the right side of the figure. The code section that needs atomic

```
<< SSSP Quarq Implementation >>

Divide nodes among threads
While !done:
   // Traverse the graph
   For each node v:
      Int min = D[v];
      For each neighbor u:
         If (min > D[u] + W [v, u])
         {
            min = D[u] + W [v, u]
            Flag = True;
         }
      D[v] = min;
   Barrier;

   CheckForDoneSignal(Flag);
   Barrier;
```

Figure 3: Pseudo code of SSSP using **Spin**, **Atomic** and **MC** models.

updates is migrated to *service threads*. Only the neighbor node id is needed for critical section invocation, hence each *worker thread* sends one word of data as a message to the corresponding *service thread*. Similar to the Atomic model, the MC model also eliminates locks. In addition, it also distributes and pins shared data to *service threads*, and prevents cache line bouncing between cores. Moreover, by utilizing non–blocking messaging, it overlaps communication overheads with other useful work in the *worker threads*.

B. Single Source Shortest Path (SSSP)

The SSSP benchmark is ported from the Pannotia benchmark suite [14]. The algorithm utilizes static node distribution, similar to TC. As seen in Figure 3, the node distances are updated without protection of locks. Even though the threads may end up reading the stale distance values of the neighboring nodes, the iterative nature of the algorithm compensates it in the next iterations. The original GPU implementation utilizes a temporary array to write the new distance values locally, and push to global array at barrier synchronization points at the end of each iteration. However, the temporary array is not needed in a cache coherent multicore system since the updated distance values are coherently observed by all threads at the hardware level. Therefore, the temporary distance array is removed, and the updates are performed on the global array directly using load/store instructions. SSSP requires only barrier synchronization at the end of each iteration. Hence, the barriers are implemented using the shared memory spin-locks, atomic instructions, and the explicit messaging based MC model.

C. AlexNet and SqueezeNet

Two machine learning workloads are investigated. While AlexNet is a large network with over $250MB$ of on-chip model size, SqueezeNet has a smaller network size of $\sim 5MB$. For both workloads, most of the computation is in convolutional layers, therefore only parallelization of the convolutional layers is discussed. Both benchmarks are realized using the 4-way SIMD with 16-bit floating point capability per core.

The coarse–grain parallelization strategy is that all neurons are tiled, and tiles are divided among all available threads. Each thread performs the computation for the neurons in its tiles. The tiling is done in a way that the data reuse is maximized in the private caches. The only synchronization required is a barrier at the end of each layer. This approach is implemented using the shared memory spin-locks, atomic instructions, and the explicit messaging based MC barriers. Both AlexNet and SqueezeNet are realized using this approach.

An optimized implementation for AlexNet from [10] makes use of a fine–grain parallelization strategy. This is achieved by dispatching multiple threads to work on a single neuron. The accumulation on a neuron by multiple threads can be realized utilizing shared memory spin locks. However, it does not scale as well as the coarse-grain approach due to additional synchronization overheads. This approach can also be implemented using an atomic floating point fused-multiply-and-add instruction. However, this type of operation is not available as a single atomic instruction in the RISCV ISA or the Tilera machine used for evaluating the Atomic model. The **MC** model is generalizable for any critical code section implementation, hence it is realized for the fine-grain synchronization implementation of AlexNet.

III. Evaluation Methodology

A. Tilera Machine

TILE-Gx72 multicore platform incorporates hardware-based core-to-core messaging as an auxiliary capability to hardware cache coherence and atomic instructions based synchronization. Hence, this platform is deployed for prototyping the QUARQ architecture. The Tilera platform is a 72 tiles processor. Each tile contains a VLIW core, $32KB$ private level-1 instruction and data caches, and $256KB$ shared level-2 cache. It executes at 1 GHz and is equipped with 16 GB of DDR3 main memory. It runs a linux version that is modified for Tilera architecture. A modified version of GCC 4.4.7 that supports Tilera specific features is utilized for the compilation of the benchmarks.

8 to 64 cores in the system are utilized for performance evaluation. Completion time is measured by running all the workloads to completion, and only the parallel region is measured in each application. Every run is repeated ten times and the average number is reported to obtain more accurate benchmarking time.

Architectural Parameter	**Value**
Number of Cores	upto 1024 @ 1 GHz
Compute Pipeline per Core	In–Order, Single–Issue
Word Size	64 bits
Physical Address Length	48 bits
Memory Subsystem	
L1–I Cache per core	8-32 KB, 4–way Assoc., 1 cycle
L1–D Cache per core	8-32 KB, 4–way Assoc., 1 cycle
L2 Inclusive Cache per core	16-256 KB, 8–way Assoc. 2 cycle tag, 4 cycle data
Cache Line Size	64 bytes
Directory Protocol	Invalidation–based MESI ACKwise$_4$
Num. of Memory Controllers	4-16
DRAM Bandwidth/Latency	10 GBps per Controller/ 100ns
Electrical 2–D Mesh with XY Routing	
Hop Latency	2 cycles (1–router, 1–link)
Contention Model	Only link contention (Infinite input buffers)
Flit Width	64 bits
Explicit Communication	
Receive queue per core	2.4 KB

Table I: Architectural parameters for evaluation.

B. QUARQ Simulator

QUARQ architecture is implemented using an in–house industry–class simulator of a tiled multicore processor. Each tile implements an in-order RISCV core, and a two–level private L1, shared L2 cache hierarchy. The cache sizes are configured for each core count to match the total on-chip cache capacity of the TILE-Gx72 machine. This is done to keep the area overhead of the multicore in check as the number of cores on chip are scaled from 8 to 1024. The default architectural parameters are shown in Table I.

Performance models of the core, cache hierarchy, coherence protocol, memory system, and on–chip network are derived from the Graphite multicore simulator [15]. The performance models are extended to accurately account for explicit communication instructions, as well as the RISCV ISA. GCC is used to compile the benchmarks. The compiler itself does not inherently understands explicit messaging and SIMD instructions. Instead, it simply wraps the instructions within assembly blocks, using the GCC extended asm block syntax to instruct the compiler as to what registers are inputs or outputs. This allows the compiler to allocate registers properly and schedule the code.

Both graph benchmarks use the California Road Network [16] as the input graph. Moreover, the machine learning benchmarks use an image from the ImageNet dataset [17] for inference. Each benchmark is run to completion, and the completion time of the *parallel* region is measured and broken down into the following categories: *Compute Stalls* is the time spent retiring instructions, waiting for functional unit (ALU, FPU, Multiplier, etc.), and the stall time due to mis-predicted branch instructions. *Memory Stalls* is the stall time due to load/store queue capacity limits, fences, and waiting for load completion and L1 instruction cache misses. *Communication Stalls* is the stall time due to explicit messaging instructions.

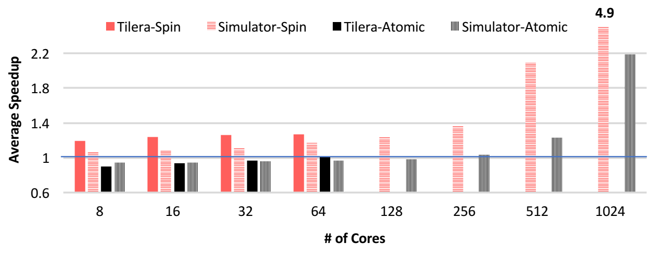


Figure 4: Average speedup of **MC** over **Spin** and **Atomic** as the core count increases.

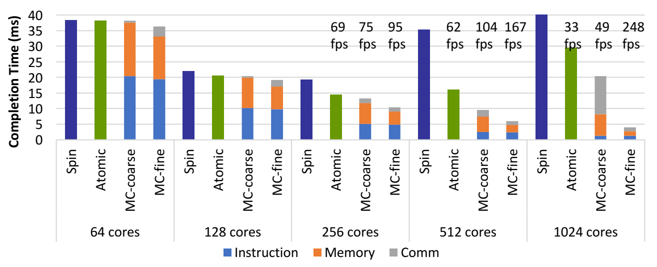


Figure 5: Completion time results for **Spin**, **Atomic**, **MC-coarse** and **MC-fine** implementations of AlexNet at different core counts.

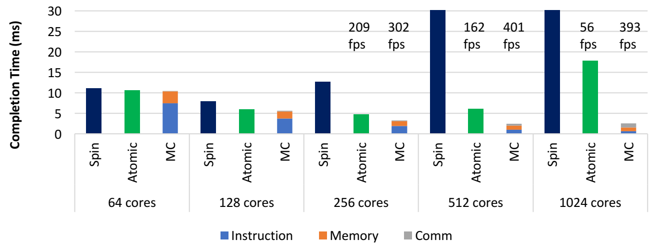


Figure 6: Completion time results for **Spin**, **Atomic** and **MC** implementations of SqueezeNet at different core counts.

IV. Evaluation

A. Core Count Scaling of Synchronization Models

Figure 4 shows the average speedup of **MC** over **Spin** and **Atomic** synchronization models for the TILE-Gx72 machine, as well as the QUARQ simulator. While the core count is varied from 8 to 64 in TILE-Gx72, it is ranged from 8 to 1024 in the QUARQ simulator. The overall performance trends are very similar between the Tilera machine and the simulator. The relative performance of **MC** with respect to **Spin** and **Atomic** improves as the core count goes up in both TILE-Gx72 and in the simulator. However, at core counts up to 64, the **Atomic** model outperforms **MC** by more than 10%. The **MC** model closes the gap and provides comparable performance at 64 cores in both TILE-Gx72 and the simulator. It utilizes the non-blocking aspects of critical code section requests to overcome the on-chip network latencies at higher core counts. As the core counts approach 256 and higher, the **MC** model handily outperforms the **Atomic** model. A detailed analysis of these performance benefits relative to other competitive machines, such as GPUs and multicores is discussed next.

B. Machine Learning Benchmarks

Figure 5 shows AlexNet's completion time breakdowns and the performance in frames per second (at 1 GHz frequency) for various core counts. The **Spin** and **Atomic** models stop scaling beyond 256 cores due to expensive barrier overheads caused by cache line ping–pongs. The **MC-coarse** version continues to scale with the explicit messaging based barrier implementation until 512 cores. On the other hand, the **MC-fine** version continues scaling to 1000 cores due to reasons discussed in [10]. QUARQ achieves 95 frames per second at 256 cores (1 TFLOPs), while machines with similar compute capabilities achieve lower performance for their open source AlexNet implementations. For example, NVDIA's Tegra X1 with 256 CUDA cores [11] gives 67, and Skylake I7 6700K 81 frames per second with Intel proprietary machine learning libraries. Moreover, as the core count increase, QUARQ improves performance by 1.75× at 512 cores, and 2.61× at 1024 cores over the 256 cores setup.

Figure 6 illustrates the completion time breakdowns and the respective frames per second for the SqueezeNet implementations at various core counts. As mentioned in Section II-C, SqueezeNet does not have fine–grained parallelization as opposed to AlexNet. As seen, the version that utilizes the **Spin** model does not scale to even 256 cores because SqueezeNet contains less work between barriers making them prohibitively expensive. This is improved using the **Atomic** model, which also does not scale beyond 256 cores due to expensive cache line ping–pongs in the barrier. On the other hand, the **MC** model scales up to 512 cores and performance remains flat as core counts are further increase to 1024. At 1024 cores, the work between barriers gets very small compared to the synchronization overhead, hence performance does not scale. The performance of QUARQ at 256 cores delivers 302 frames per second, which is better than the performance of NVDIA GTX 750-Ti (239 frames per second) and Skylake I7 6700K at 4 GHz (285 frames per second).

C. Graph Benchmarks

Figure 7 shows the completion time results for the Triangle Counting (TC) benchmark at various core counts. Due to extra locking overheads, the **Spin** version does not perform as good as **Atomic** and **MC** even though it scales up to 512 cores. The performance gap also increases as the core counts increase. Due to expensive cache line ping–pongs, the **Atomic** model also starts degrading performance at 512 cores, and **MC** provides 1.63× better performance as it pins shared data to dedicated cores to exploit data locality. Moreover, it utilizes non–blocking send messages that help worker cores to hide communication latency with other useful work. As

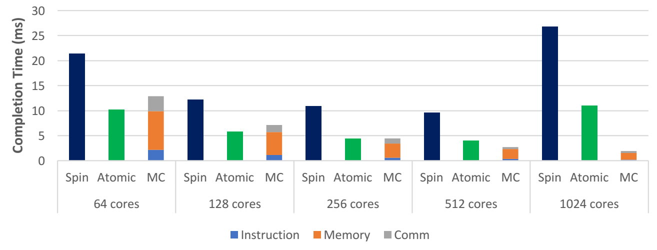


Figure 7: Completion time results for **Spin**, **Atomic** and **MC** implementations of TC at different core counts.

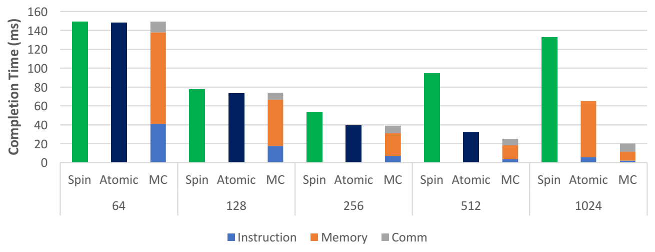


Figure 8: Completion time results for **Spin**, **Atomic** and **MC** implementations of SSSP at different core counts.

a result, **MC** continues to scale to 1024 cores. At 256 and 512 cores, both **Atomic** and **MC** models provide less than **5 ms** of completion time. This is an order of magnitude better than NVDIA GTX 750-Ti (122 ms), GTX 970 (67 ms), and Intel Xeon Phi multicore (133 ms) processors. The GPU and Xeon Phi utilize state of the art atomic instructions in their optimized implementations.

Figure 8 shows he completion time results for the SSSP benchmark at various core counts. All three synchronization models scale to 256 cores, and **Spin** starts degrading beyond that. Similarly, the **Atomic** model does not scale beyond 512 cores. On the other hand, the **MC** model shows enhanced performance up to 1024 cores. SSSP is an iterative algorithm and each iteration is separated by barrier synchronization. Hence, as core counts increase, synchronization through cache coherence becomes expensive and gets dominant in the completion time. The **MC** implementation gives **39 ms** at 256 cores, and **25 ms** at 512 cores, whereas at similar scale GPUs, GTX 750-Ti and GTX 970 provide 169 ms and 47 ms, respectively. As mentioned in Section II-B, the shared memory implementation does not use the temporary D-array, which is implemented in the GPU version. Using temporary array delays observing the updated distances to the next iteration. This is necessary in GPUs as it does not have cache coherence. However, in QUARQ, updates on a global array are efficiently done using hardware cache coherence. Therefore, making the distance updates on the global distance array helps threads to see the changes made by other other threads in the same iteration. This reduces the iteration count from 48 to 28 iterations, and helps QUARQ achieve better performance than the GPU implementations.

V. Conclusion

This paper presents performance analysis for deep neural network and graph processing applications on the QUARQ multicore architecture. QUARQ features short–SIMD and 16–bit floating point support per core, and accelerates thread synchronization using a novel moving compute to data model that ships critical code sections to dedicated cores using auxiliary low-latency, non–blocking explicit messaging instructions. A real TILE-Gx72 machine is prototyped to demonstrate the practicality and applicability of the proposed synchronization models. QUARQ scales to 1024 cores, and offers best performance of 248 frames per second for AlexNet, 401 frames per second for SqueezeNet, and $5ms$ for single source shortest path and $30ms$ for triangle counting benchmarks executing on the California Road Network graph. These performance results are shown to be competitive with various state-of-the-art NVidia GPUs and Intel multicores executing optimized implementations of these benchmarks.

VI. Acknowledgments

This work was supported in part by Semiconductor Research Corporation (SRC). We would like to thank José Joao of ARM, and Christopher Hughes of Intel for their discussions and feedback towards the evolution of the QUARQ architecture.

References

[1] P. Sermanet and Y. LeCun, "Traffic sign recognition with multi-scale convolutional networks," in *Neural Networks (IJCNN), The 2011 International Joint Conference on*. IEEE.

[2] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "Show and tell: A neural image caption generator," *CoRR*, vol. abs/1411.4555, 2014. [Online]. Available: http://arxiv.org/abs/1411.4555

[3] A. Krizhevsky, I. Sutskever, and G. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012.

[4] Y. Goldberg, "A primer on neural network models for natural language processing." 2016.

[5] S. Kaz, Y. Cliff, and P. David, "An in-depth look at Google's first Tensor Processing Unit (TPU)," https://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu, May 2017.

[6] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[7] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM Sigplan Notices*, vol. 49, no. 4. ACM, 2014, pp. 269–284.

[8] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3, pp. 105–117, 2016.

[9] H. Dogan, F. Hijaz, M. Ahmad, B. Kahne, P. Wilson, and O. Khan, "Accelerating graph and machine learning workloads using a shared memory multicore architecture with auxiliary support for in-hardware explicit messaging," in *IPDPS, 2017*, 2017.

[10] H. Dogan, B. Kahne, and O. Khan, "Quarq: A novel general purpose multicore architecture for cognitive computing," in *TECHCON, 2017*, 2017.

[11] NVIDIA, "GPU-Based Deep Learning Inference: A Performance and Power Analysis," https://www.nvidia.com/content/tegra/embedded-systems/pdf/jetson_tx1_whitepaper.pdf, Nov 2015.

[12] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size," *arXiv:1602.07360 preprint*, 2016.

[13] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores," in *Workload Characterization (IISWC), 2015 IEEE Int. Symp. on*, Oct 2015, pp. 44–55.

[14] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 185–195.

[15] J. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *HPCA*, 2010.

[16] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, 2009.

[17] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 2009, pp. 248–255.