QUARQ: A Novel General Purpose Multicore Architecture for Cognitive Computing

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

***Abstract*—The recent success of deep neural networks on challenging problems have provoked both academia and industry to design efficient specialized hardware to improve the performance and energy requirements for training and evaluating state-of-the-art neural networks. In this context, many specialized accelerator designs have been proposed. For this work, instead of designing a specialized piece of hardware, we propose a general purpose multi-core architecture called QUARQ that provides state-of-the-art performance for neural network workloads. QUARQ is a tiled multi-core architecture which contains both shared memory and explicit messaging capabilities for inter-core communication. QUARQ enables scalable computation, memory and communication for neural networks. For computation, QUARQ enables a short-SIMD (64-bit) pipeline per core. Each short-SIMD can perform four 16-bit precision MAC operations. QUARQ enables an intelligent 2-level cache hierarchy that exploits data reuse at the L1 levels, and caches per-frame dataset on-chip in the L2 cache, avoiding expensive off-chip memory accesses leading to near optimal memory bandwidth usage. QUARQ enables extremely scalable communication by combining the hardware based cache coherence protocol with in-hardware explicit messaging. This allows fine-grain concurrency to be exploited within a single neuron since communication overheads are much smaller than the benefits achieved from data reuse enabled by fine-grain concurrency. The simulation based evaluations results show that 512 cores with a short-SIMD pipeline per core provides 165 frames per second performance for ALEXNET, which is a classical neural network workload.**

I. INTRODUCTION

The recent success of 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]. Specially, GPUs are shown to be effective in processing of DNNs due to their high FLOP rate, memory bandwidth, and large concurrency capabilities.

In this work, we propose to use a general purpose hybrid multicore architecture described in [8] for inference phase of deep neural networks instead of specialized hardware. The overview of a tile of the system can be seen in Figure 1. The proposed system is a tiled multicore that combines the shared memory with in–hardware explicit messaging. It utilizes RISCV ISA with extensions for explicit messaging instructions. We also extended the core described in [8] with 4–way short–SIMD to increase FLOP rate of the system. In addition, since it is shown in the literature that 16–bit floating point is enough for neural networks [9], support for 16–bit floating point is added to reduce the pressure on caches. Each SIMD instruction performs four 16–bit floating point operations. To accelerate communication, four basic explicit messaging instructions are added to the ISA and implemented at the hardware level.

1) Non-blocking send instruction (*send*) requires a destination address along with the data to be sent. 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 inserting it into a send queue for transmission on the on-chip

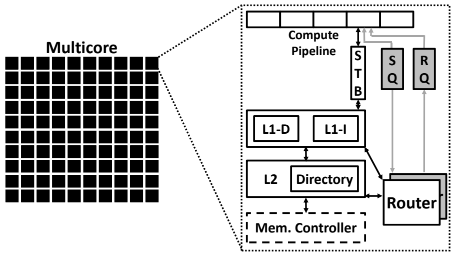

Figure 1: Overview of a tile in the proposed system with explicit messaging support.

network. 2) Blocking receive instruction (*recv*) is used to receive the sent data. When a new message is received, 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, 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 [8].

As a representative neural network application, a classical DNN AlexNet [3] is employed to show the applicability of the proposed architecture on neural networks. Two parallelization strategies are presented. The first one is a naive coarse–grained parallelization in which neurons are divided among available threads and synchronized with a barrier after each layer. The barriers are implemented using explicit messaging instructions *sendr/resumer* as explained in [8]. The second one is a fine–grained parallelization strategy in which threads are clustered into groups and the neurons are divided among the groups, so that each thread group works on the same set of neurons. This approach deploys explicit messaging instructions to enable fine–grained concurrency. Since multiple threads work on the same set of neurons, the threads in the group generates partial sums and the partial sums are accumulated using explicit messaging instruction. Similarly, the threads are synchronized with a barrier after each layer work is done. We also implemented a version of fine–grained parallelization by using SIMD instructions of QUARQ to improve the FLOP rate of the system. The results shows that the proposed system provides 92 frames per second for AlexNet with a single image when using fine–grained parallelization strategy with SIMD support. This outperforms the GPU with equivalent FLOP rate (NVDIA Tegra X1) which offers 67 frames per second [9]. In addition, a core count scaling and memory bandwidth sensitivity studies are conducted to show the bottlenecks of the system

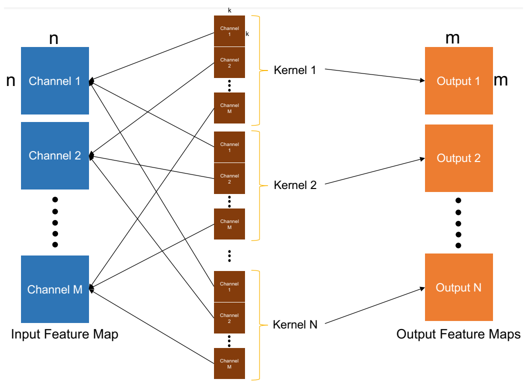

Figure 2: Overview of convolutional layer.

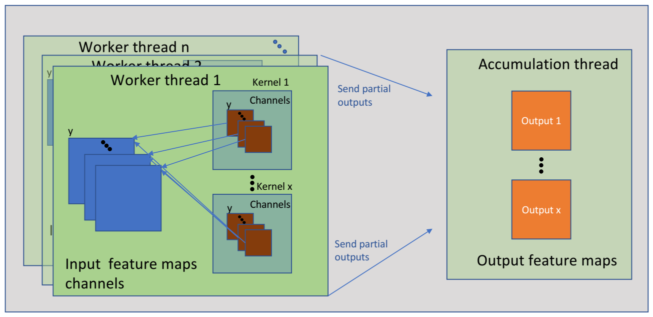

Figure 3: Overview of convolutional layer.

at different architectural configurations for the AlexNet benchamrk.

II. Machine Learning

We have evaluated AlexNet as representative machine learning workload. AlexNet consists of five convolutional layers and three fully–connected layers. In addition, some of the convolutional layers contain pooling and normalization layers. Since most of the computation is in convolutional layers, here we only discuss the parallelization of the convolutional layer in detail. Briefly, other layers are parallelized by dividing the neurons among available threads.

The general overview of a convolutional layer can be seen in Figure 3. There are multiple kernels to produce multiple output feature maps. Each kernel has multiple channels corresponding to input channels. Each neuron output is calculated with the accumulation of 2D convolution operations using the channels of the corresponding kernel and input. All three data structures are reused for various purposes. Each kernel is reused for each neuron in an output feature map, and the input is also reused with each kernel to generate different output feature maps. Similarly, each output is also reused when accumulating the 2D convolutions. To minimize the performance overheads, reuse of these data in L1 cache should be maximized.

A naive coarse–grained parallelization strategy is that all the neurons are tiled and tiles are divided among available threads. Each thread performs all the computation for the neurons in its tiles. This approach enables reuse of the kernel data. Each kernel channel is brought to L1 data cache one by one, and reused for all the neurons in each tile. This approach also allows data reuse for the output data structure if the neurons in the tile, 2D kernel and the corresponding input fits in L1. That is to say, when bringing the next kernel channel for the same tile, the outputs are still in the L1 and

Algorithm 1 Fine–grained parallelization of convolution

```
 1: << Worker Threads >>
 2: Divide the channels among group of threads
 3: start = tid * nChannels/nThreads
 4: stop = (tid+1) * nChannels/nThreads
 5: for each ch in range(start, stop) do
 6:     for each y in range(0,outH) do
 7:         for each x in range(0,outW) do
 8:             Perform convolution for one channel
 9:             psum = convolution(filter, input, ch, y, x)
10:             Send psum to accumulation core
11:             sendmsg(AccumCore, psum,y,x)
12: << Accumulation Threads:>>
13: num_msg = 0;
14: while $num\_msg < nChannels$ do
15:     recvmsg(&addr, &psum, &y, &x)
16:     output[y][x] += psum
17: end while
```

loaded without expensive L1 miss. However, this approach creates imbalance when using larger tile size in some of the layers. For example, layer 3 of AlexNet contains 384 13×13 neurons. If we use 13×13 tiles to have good reuse of filter and output data in L1, when using 256–core system, some of the cores get more work than others. This causes underutilization of the system. If we reduce the tile size to have more concurrency, then it hurts the data reuse.

More optimized implementation makes use of fine–grained parallelization strategy. This is achieved by dispatching multiple threads to work on one neuron. This requires updating the same neuron output by multiple threads. Therefore, this must be implemented using critical code sections. The critical section work can be realized utilizing shared memory spin locks. However, it does not scale well due to the large overheads of shared memory locks. Therefore, we have implemented it using the explicit messaging capabilities of the QUARQ architecture, as depicted in Algorithm 1. In this approach, as it is seen in the algorithm, the cores are clustered into small thread groups and each group works on a tile of neurons. To calculate a neuron output, the kernel channels are divided among the threads in the group, and each thread calculates partial sums using its kernel channels. One of the threads in each group is used to accumulate the partial sums. The partial sum of each neuron for each kernel channel is calculated (line 9), and sent to the accumulation thread using *send* instruction to be accumulated (line 11). The accumulation thread (lines 14–17) receives the partial sums from the other threads, and accumulates it on the output to be used in the following layers. In this approach, the neuron outputs reside in separate cores that perform the accumulation work, and they are reused in L1. Similarly, the kernel channels are also reused for all the neurons in the tile. Because this approach deploys a fine–grain parallelization strategy, it enables higher concurrency without loosing the data reuse benefits. To get the best performance out of this approach, one needs to adjust the number of threads per group. For this work, we empirically decided the optimal number of threads per group as 8.

Both approaches discussed here are implemented as scalar. In addition to two scalar implementations, the fine–grained parallelization approach is also realized using the SIMD capabilities of the system. This approach implements the previously mentioned 2D convolutions by using 4-way SIMD instructions and significantly reduces the instruction footprint. Since SIMD version also utilizes 16 bit floating point, it also reduces the pressure on the memory subsystem. Similar to convolutional layers, other layers are also implemented by utilizing SIMD capabilities.

III. Evaluation Methodology

A. Compiler Support

GCC 7.0.1 for RISV is used as the compiler. The compiler itself does not inherently understands explicit messaging. Instead, it simply wraps the explicit messaging 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.

Architectural Parameter	**Value**
Number of Cores	256 @ 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 KB, 4–way Assoc., 1 cycle
L1–D Cache per core	8 KB, 4–way Assoc., 1 cycle
L2 Inclusive Cache per core	16 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	8
DRAM Bandwidth/Latency	10 GBps per Controller/ 100ns
DRAM Latency	100 ns
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	
Send queue per core	4 entry (1 entry = 4 words)
Receive queue per core	48 entries

Table I: Architectural parameters for evaluation.

B. Simulator Setup

The proposed architecture is implemented using an in–house industry–class simulator and the associated GCC compiler. A futuristic many–core tiled multicore processor, a two–level private L1, shared L2 cache hierarchy per core is evaluated. The default architectural parameters used for evaluation are shown in Table I.

C. Performance Models

All experiments are performed using the core, cache hierarchy, coherence protocol, memory system, and on–chip interconnection network models implemented within the multicore simulator. The electrical mesh interconnection network uses XY routing. We model a 2–cycle per hop delay; we also account for the appropriate pipeline latencies associated with loading and unloading a packet onto the network [10]. In addition to the fixed per–hop latency,

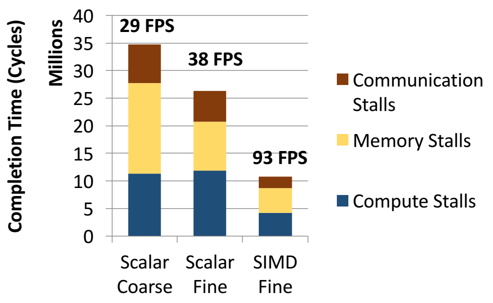

Figure 4: Completion time results for coarse–grained, fine–grained and fine–grained SIMD implementations of AlexNet at 256–core system.

network contention delays are also modeled. These models are derived from Graphite multicore simulator [11]. The performance models are extended to accurately account for explicit communication instructions.

D. Evaluation Metrics

The workload is run to completion, and the completion time of *parallel* region is measured. The parallel completion time is 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. The communication latency includes stalls caused by *send* (due to flow control restrictions), *recv* (waiting for a message to arrive), and *sendr* (round–trip latency to the destination, wait time in the destination's receive queue, and destination's execution latency) instructions.

IV. Evaluation

Figure 4 shows the completion time breakdowns and the respective frames per second for three AlexNet implementations. As seen from the figure that memory stalls are the main bottleneck for the coarse–grained implementation due to limited data reuse at level 1 cache. This is improved with fine–grained implementation by enhancing the data reuse for input, kernel and output data. Even though fine–grained implementation helps to improve performance, it still has noticeable memory and compute stalls. To improve the compute stalls, the fine–grained version is implemented using SIMD capabilities of QUARQ. Similarly, the memory stalls are improved by using 16 bit floating point instead of 32 bit. As the figure illustrates, while the compute stalls go down by $3\times$, the memory stalls decrease by $2\times$, and SIMD implementation of a 256–core QUARQ architecture provides 93 frames per second. The achieved performance under QUARQ is better than NVDIA's Tegra X1, which offers 67 frames per second with 256 CUDA cores [9]. At the respective architecture configurations, both NVidia's Tegra X1 and the proposed 256–core QUARQ are rated at 1 teraflops. Ideally 4-way SIMD should decrease the instruction count by $4\times$, however due to some scalar region in the implementation, we observe the observed deviation from the ideal case.

To observe the performance scaling trends of AlexNet and the bottlenecks at different core counts, a scaling experiment is conducted for the SIMD fine–grained implementation. Figure 6 shows that AlexNet scales to 512 cores and provides 165 fps, which is $1.8\times$ better over 256 cores. We also

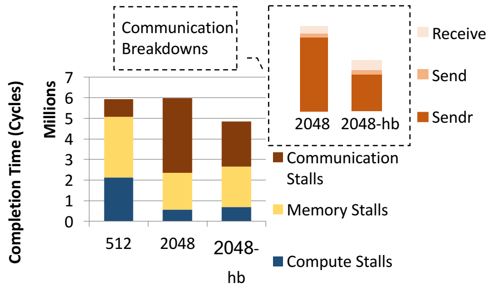

Figure 5: Completion time breakdowns for SIMD implementation at 512 and 2048 cores, and 2048 cores with hierarchical barriers.

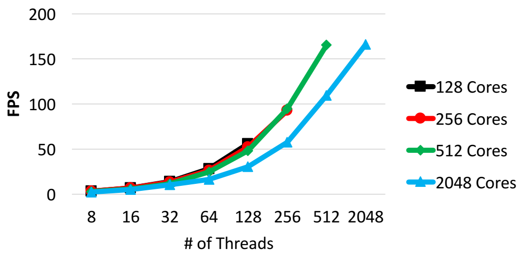

Figure 6: Scaling results for SIMD implementation of AlexNet at 128, 256, 512 and 2048 cores.

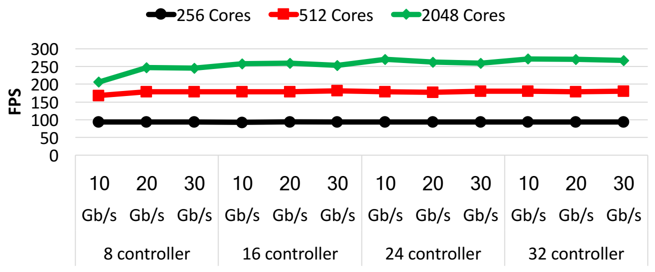

Figure 7: Bandwidth sensitivity study at 256, 512 and 2048 cores.

observe that the performance does not improve at 2048 cores compared to 512 cores. As seen in Figure 5, the memory and compute stalls at 2048 cores are improved compared to 512 cores, however the blowup in communication stalls prevents further increase in performance scaling of AlexNet. The figure also shows the breakdown of the communication stalls for the 2048–cores system. *Sendr* stalls constitute almost 90% of the communication stalls, which is used to implement barrier synchronization. Therefore, to be able to scale to 2048 cores, thread synchronization should be improved. One of the ways of improving barrier synchronization is to deploy hierarchical barriers. As seen in the figure, replacing the barriers with hierarchical ones at 2048–cores system (2048-hb) reduces the *sendr* stalls (barrier overhead) by 40% and provides 206 fps. However, the communication stalls still remain as the biggest bottleneck for 2048–cores system. As a future work, we plan to explore multiple pipelines in a tile to reduce barrier overheads. Using four pipelines in each core at 512–cores system provides the same compute power as 2048–cores system but keeps the network same which is expected to have similar barrier overhead as 512 cores with single pipeline per core.

As a last experiment, we have also conducted a memory bandwidth sensitivity study to show the impact of the bandwidth on the performance. Since AlexNet has 3 fully–connected layers, it has large amount of data for weights to be read from the main memory. In this experiment, the number of memory controllers and bandwidth per memory controller is varied for 256, 512 and 2048–cores system as shown in Figure 7. The figure shows that for 256–cores system, the performance is not impacted at all by both the number of memory controllers and the bandwidth per controller. However, as the number of cores increases, more concurrent requests from the main memory happens, hence either number of memory controllers needs to be increased or the bandwidth per controller should go up. For 512 cores, the performance does not get better beyond the total bandwidth of 160 Gb per second. On the other hand, as seen from the figure, at 2048 cores, increasing the number of memory controllers helps more than improved bandwidth per controller. With hierarchical barriers and enhanced memory bandwidth, 2048 cores provide 271 frames per second.

V. Conclusion

This paper presents a general–purpose hybrid multicore architecture for deep neural network applications. Our analysis on a classical neural network AlexNet shows that the proposed architecture with a short–SIMD and 16–bit floating point support provides a state-of-the-art performance by enabling low overhead fine–grained communication of threads via in–hardware explicit messaging. The proposed system scales to 512 cores and offers 165 fps for AlexNet.

VI. Acknowledgments

This work was supported in part by Semiconductor Research Corporation (SRC). We would like to thank José A. Joao of ARM and Christopher Hughes of Intel for their reviews and feedback.

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] 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.

[9] 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.

[10] S. Park, T. Krishna, C. Chen, B. Daya, A. Chandrakasan, and L.-S. Peh, "Approaching the theoretical limits of a mesh noc with a 16-node chip prototype in 45nm soi," in *DAC*, 2012.

[11] 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.