Preparing for Future Heterogeneous Systems Using Migrating Threads

Peter Kogge
Jayden Vap
Derek Pepple
kogge@nd.edu
jvap2@nd.edu
dpepple@nd.edu
University of Notre Dame
Notre Dame, Indiana, USA

Abstract

Heterogeneity in computing systems is clearly increasing, especially as "accelerators" burrow deeper and deeper into different parts of an architecture. What is new, however, is a rapid change in not only the number of such heterogeneous processors, but in their connectivity to other structures, such as cores with different ISAs or smart memory interfaces. Technologies such as chiplets are accelerating this trend. This paper is focused on the problem of how to architect efficient systems that combine multiple heterogeneous concurrent threads, especially when the underlying heterogeneous cores are separated by networks or have no shared-memory access paths. The goal is to eliminate today's need to invoke significant software stacks to cross any of these boundaries. A suggestion is made of using migrating threads as the glue. Two experiments are described: using a heterogeneous platform where all threads share the same memory to solve a rich ML problem, and a fast PageRank approximation that mirrors the kind of computation for which thread migration may be useful. Architectural "lessons learned" are developed that should help guide future development of such systems.

CCS Concepts: • **Computer systems organization → Architectures**; **Parallel architectures**; **Multicore architectures**; • **Networks → Network protocols**;

Keywords: parallel, heterogeneous, multi-threading, migrating threads

ExHET '24, March 02–06, 2024, Edinburgh, United Kingdom

ACM ISBN 979-8-4007-0537-3/24/03
https://doi.org/10.1145/3642961.3643801

ACM Reference Format:
Peter Kogge, Jayden Vap, and Derek Pepple. 2024. Preparing for Future Heterogeneous Systems Using Migrating Threads. In *The 3rd International Workshop on Extreme Heterogeneity Solutions (ExHET '24), March 02–06, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3642961.3643801

1 Introduction

Heterogeneous systems involve multiple architecturally different processing units, each optimized for different functionality, that are ganged together to solve compute-intensive problems. Fig. 1 diagrams an architecture typical of many of today's commodity servers. Multiple conventional multicore sockets are in the same node with multiple GPU modules.

Software support for such system architectures is fairly mature, with languages such as CUDA [2, 20] and OPENCL [26] providing interfaces between programs in the conventional cores and accelerator routines in the other accelerator cores. Such interfaces are, however, rather one-directional. It is pedestrian for conventional cores to launch accelerator cores, but the opposite is still a challenging novel task [11, 27].

An even more disjointed situation occurs in systems with potentially thousands of such nodes. Now not only can "accelerator" threads not spawn conventional threads, but a conventional thread in one node cannot even spawn an accelerator thread in a separate node without traversing multiple software stacks. Such calls then use an entirely separate protocol to communicate with a thread in the other node that in turn should do the first thread's bidding. Completion acknowledgements must then again be relayed indirectly back through a similar tortuous path. All of this extra software adds significant latency and represents wasted computational potential on cores on both sides of the interface.

Looking forward, there are two trends: "accelerators" are coupling even closer to conventional cores, moving on-module or on-die. Second, they are also dispersing even further away and deeper into the memory systems or networks of large parallel systems.

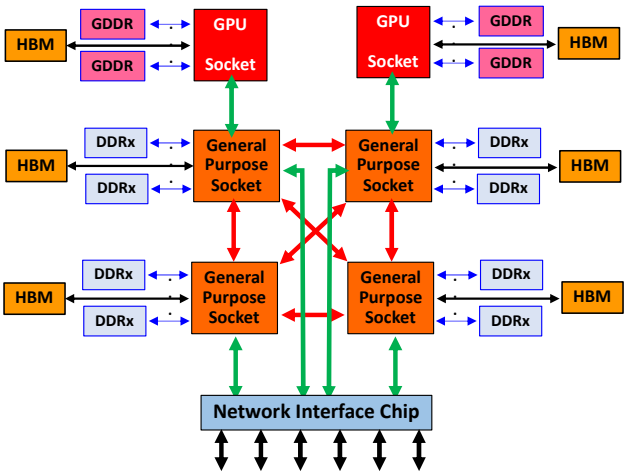

Figure 1. Heterogeneity Today.

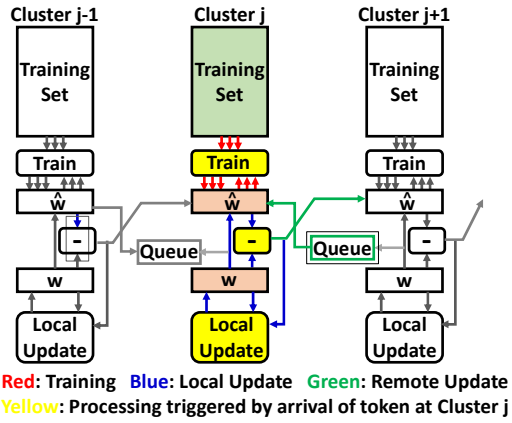

Figure 2. Hogwild++ Data flow Associated with Cluster j.

All of these cases are the focus of this paper, where potentially huge numbers of accelerators may be found in a system, and where conventional processors capable of running sophisticated runtimes may not be "nearby" the accelerators. It is clear that in such cases, unless there is a change in architectures, the cost of spawning and managing activities in heterogeneous cores will become more and more onerous. In such cases, the software costs of deciding where data resides, what is the appropriate accelerator to perform the desired function, and where is the processor that is close enough to manage the computation request will become enormous.

This paper explores the beginnings of an architecture where communication intended to start remote operations "elsewhere" in a system does not require any, let alone deep, software stacks, nor even any particular program-level knowledge as to on which node particular data objects live. Threads are not constrained to live out their existence on specific nodes, but can "migrate" freely as needed without software intervention throughout the system.

To explore what kind of, and how much, functionality might have to be carried by such migrations, and what might be the nature of the "controllers' for such accelerators, this paper draws observations from two experiments. The first explores a parallel machine learning application where localized training happens on accelerators (GPUs) and the exchange of model vector updates must be handled by a functionality with a bigger system view. The second explores a fast PageRank application that has significant remote function invocation.

Section 2 discusses the two problems used in the experiments discussed here. Section 3 describes the setup for the experiments. Section 4 reviews the lessons observed from the two experiments. Section 5 discusses a proposed architecture that remediates the major issues found. Section 6 discusses the semantics of cross-border function invocations. Section 7 concludes.

2 Example Heterogeneous Problems

2.1 Hogwild! Overview

Machine learning (ML) is an essential part of modern computing. The inferencing part of ML is often straightforward to perform efficiently, especially by purpose-built hardware (cf. Google's TPU [12]). However, learning is far more complex. One well known technique is Stochastic Gradient Descent (SGD) that repeatedly uses training data against an evolving *model vector*. When a projection (inference) is incorrect, the "direction" of the error is used to modify the model vector slightly. A series of studies have explored a wide range of different parallel SGD algorithms [8, 24, 29, 30], largely on multi-core chips, with at best limited scalability. Memory issues such as inter-socket coherency traffic and false sharing have been primary. Perhaps the most influential of these algorithms was Hogwild! [17, 18] - a multi-threaded implementation where independent trainers could update the same model vector at the same time with a feature-at-a-time atomic update. This algorithm demonstrated scaling of up to 4.5X on 10 core systems for sparse problems. A follow-on, Hogwild++ [30], provided better speedups (up to 9.5X on 40 cores for problems for which the conventional algorithms do poorly) by creating "clusters" of cores where each cluster had an affinity to a distinct memory channel (cf. Fig. 2). Each cluster ran a separate training session using Hogwild! on just local data, but with a rotating "token" whose arrival on a cluster caused it to exchange changes in its model vector with the next cluster.

2.2 Frogwild! Overview

An important metric in graph processing is "vertex importance" as computed by techniques such as PageRank [6].

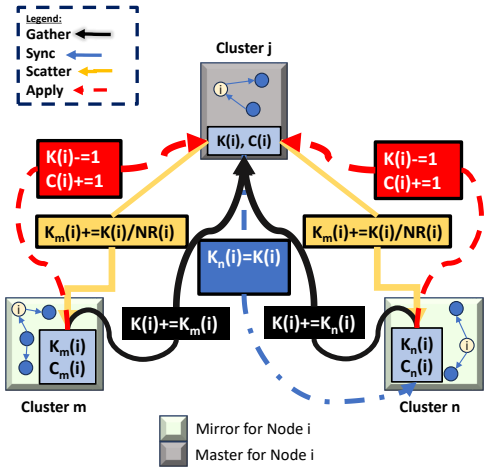

Figure 3. Frogwild! Data flow Associated with Cluster J.

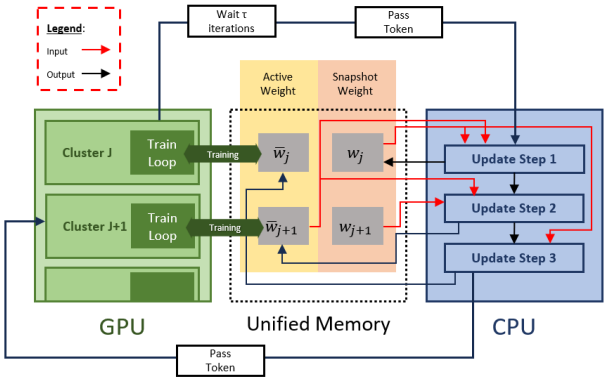

Figure 4. The heterogeneous Hogwild++ configuration.

Evaluation via power method techniques puts a large demand on memory as the order of the graph increases. Further, precisely ranking all vertices by importance is computationally intensive, and unnecessary for certain applications. Approximation techniques bypass complexity to quickly and efficiently evaluate the PageRank vector [3, 5, 7, 15]. An example is Frogwild! [16] (Fig. 3), a distributed PageRank approximation to capture the top-k vertices with a slightly altered GraphLab engine, by initializing multiple random walks across the graph, and tracking their traversals across each vertex.

2.3 Heterogeneity

Hogwild++ (Fig. 2) has a natural implementation as a multi-node heterogeneous system, where each node executes a single cluster. The main training loop is an excellent match for a GPU-like engine, whereas the inter-cluster token-passing is more appropriate for a conventional core, with the local updates being supported by either aperiodic GPU or conventional operations.

Frogwild (Fig. 3) is also a candidate for heterogeneity, but with much more irregularity than Hogwild++. GPU-like cores with wide vector capability are a good match for processing vertices when visited, and following all outgoing edges when visited. More conventional cores are then needed when the graphs are bigger than one node, and traversals between nodes become necessary.

The Appendix summarizes a list of similarities and differences.

3 Experimental Setup

A good image to justify the following experiments is that a future system to support them may very well be constructed from a "sea" of 3D memory stacks where a logic die on each module supports networking with other modules, and there is one of more "accelerators" included on each stack, but in nowhere the numbers found on today's 300W Goliaths. Also any conventional cores present on the stacks may themselves be quite limited, and not what one would use to run conventional system software or robust runtimes.

The following sections describe experiments constructed with such future architectures in mind. First in each is a description of an experimental platform, and what components of the platform relate to different aspects of heterogeneity.

In neither case could the experimental code be considered optimally designed for today's GPU execution. Rather, these codes were engineered as if specifically for accelerator cores on a future platform, with various forms of processors providing assistance. By designing these algorithms in such fashion, it permits an examination of a variety of questions including:

- What kinds of remote interactions are needed,
- how much processing is triggered by a remote "spawn,"
- how long is the lifetime of such computations,
- how many operands accompany such remote spawns,
- how much other data must be transferred?

3.1 Hogwild Experiment

To implement the Hogwild++ experiment, we used an NVIDIA Jetson [19] that was composed of a Tegra X1 (TX1) processor with a small GPU and 4 ARM cores, all of which share the same physical memory. Each of the Jetson's GPU cores was teamed with an ARM and a slice of the available memory as if they were independent "clusters" as discussed in Section 2.1, but implemented on the bottom of its own 3D memory stack.

Fig. 4 diagrams the partitioning of the components of the algorithm for the experiment onto the Jetson platform. A CUDA program running in each ARM core sets up independent training sets, with the paired GPU core tasked to perform the training process on one of those sets as an independent cluster as in Fig. 4.

After initiating these training loops asynchronously, the ARM acts as the token thread handler by interacting with

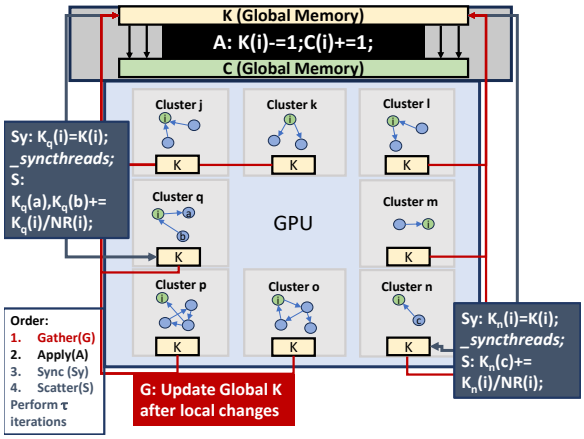

Figure 5. GPU Frogwild! For Vertex i

each model vector, and uses the access to shared memory to perform its processing. Actual migration was not needed to be simulated as the ARM thread had access to the GPU memory, but the equivalent of the shifting of attention happened naturally by the change in which chunks of memory were accessed.

In Hogwild++, each cluster has two copies of the model vector. The first, $\overline{w}_j$ ("active weights" in Fig. 4), is directly updated by the cluster while performing the SGD training. The second, w_j ("snapshot weights" in Fig. 4), stores the result of the most recent synchronization step. These vectors were stored contiguously in the Jetson's global memory (as shown in Fig. 4) and each thread was provided a pointer to the start of its block of memory. Additionally, each cluster had its own copy of a bias term, which was updated and synchronized in a mostly similar way to the model vectors.

Two variables were used to synchronize the data between CPU and GPU. First, the "token" described in the Hogwild++ algorithm is stored as an integer value of the index of the cluster currently performing synchronization. Second, a "sync" variable is used to allow the CPU and GPU to communicate their present state to each other. This is used to signal when a token has arrived and updates needed, and when it is time to "pass" the token.

3.2 Frogwild! Experiment

To reproduce Frogwild! in a heterogeneous environment, a discrete system made up of an Intel Core i9-13900KF processor and NVIDIA RTX 4080 was used. This system was employed for two main reasons. The first is the large memory on the GPU, able to better support the memory requirements for storing large graph structures. Second, the large number of CUDA cores enabled quick vertex-parallel programming. As with the Hogwild! experiment, each of the SM blocks in the GPU was programmed to act as if it was alone with its chunk of memory, and needed to interact with the other blocks as if they were "distant."

To properly partition graphs, degree-based vertex cuts were performed on the GPU. Once edges are assigned clusters, the main data structures formed by the host to represent the partitioned graph were created in a CSR (Compressed Sparse Row) format. These data structures were intentionally designed for better cache utilization on the GPU, and for mirroring near-memory computing in a future architectures.

To emulate the relation between master and mirror vertices described in Section 2.2, K and C acted as the master for all mirrors, while all vertices distributed among clusters were treated as mirrors. For initialization, a sublinear number M (compared to the degree of the graph) walkers were initialized on the M vertices with the largest out degrees to promote propagation of walkers. These initial walkers were then stored in the array K as integer values prior to execution of Frogwild!. To adhere to the distributed nature of this algorithm, the specified clusters are modeled with CUDA blocks via intentional selection of execution configuration parameters.

This CUDA variant involved four different kernels to mirror the original experiment as outlined in Section 2.2 and summarized in Table 1 which were performed iteratively.

Table 1. Frogwild Kernel Description

Kernel	Utility	Operations
Gather	Threads of each cluster send its vertex's local updates to the master	Mirror threads synchronized in the last iteration send updates to the master data set. All remaining threads lay dormant.
Apply	Grid of threads is launched s.t. each thread is responsible with updating a specific entry of K and C	Master threads are spawned, while mirror threads are asleep. Atomic operations of addition and subtraction are applied to entries of C and K, respectively
Sync	$K_m(i)$ of Mirror vertices selected for synchronization are updated to match global K(i)	Mirror threads are spawned to evaluate if they will be synchronized, and if so go to the master data to fetch K(i). Non-synchronized mirror threads will become dormant.
Scatter	Intra-cluster Successors of synchronized mirrors are sent a number of frogs equal to K(i)/NR(i), where NR(i) is the number of synchronized peer mirror threads corresponding to vertex i	Non-synchronized mirror threads remain dormant. Synchronized mirror threads go to the master data to fetch NR(i).

4 Results and Issues

Table 2 outlines the primary observations and issues identified from implementing Hogwild++ and Frogwild!.

The key issues involve memory and thread spawning. The state of the art CUDA for GPUs ([1]) and GASNet have remediated the said memory issues for systems by providing an

Table 2. Lessons and Observations

Experiment Components	Observations
Malloc (HF)	Due to non-homogenous memory models between host and device, Malloc consumes a large amount of time. Memory mapping should be more controlled.
Memory Caching (H)	Managed Memory allows caching, but the CPU and GPU cannot simulatenously access the data during runtime. Pinned Memory allows simulatenous memory accesses for the CPU and GPU, but this memory cannot be cached.
One-Directional Spawning (H)	CPU must perform a spin lock while waiting on the GPU to give any signals. Leads to complications with Inter-Kernel Interaction and Synchronization.
One-Directional Spawning (F)	During the execution of *Apply*, iterating through all the walkers on a vertex is a major bottleneck. Could be remediated by offloading computation to the CPU or another processor.
CUDA Shared Memory (H)	Improved performance due to less memory latency as opposed to fetching global memory. Required careful timing of data transfers to the global memory for synchronization
CUDA Shared Memory (F)	Due to the networks demands' inverse relation to the number of clusters, shared memory could only be applied in the presence of a larger number of clusters, limiting the opportunity for optimization.
Process Grouping (F)	There is no efficient way to define process groups with NVIDIA GPUs for this type of problem without introducing block divergence (even with block synchronization with compute capability 9.0). This necessitates additional memory by creating global versions of K and C, in addition to K arrays for each cluster.
"H" stands for Hogwild++ and "F" for Frogwild!	

interface with memory that can extend across multiple physical nodes. Intel's CAPI interface [28], and HSA architectures [23] have both provided a comparable memory interfaces to what the Jetson provided, with the added benefit of enhanced address translation. All these systems' capabilities would abate the issues of token passing and inter-kernel synchronization in Hogwild!. For Frogwild!, these paradigms would allow the use of pointer-based structures, e.g. linked lists, to better traverse partitioned graphs. GASNet's active messaging or RMA capabilities would subside the memory latency incurred by the large volume of atomic updates seen within Frogwild!. The issue of additional thread spawns could have been managed via CUDA's dynamic parallelism, but this would only consume more hardware resources on the GPU. Neither would this address the asymmetry of crossing architecture boundaries to either "conventional" or alternative accelerator cores. Rather, both the experiments studied here would see performance benefits from spawning threads on an alternative processor. Currently, the concept of multi-directional thread spawns only exist as a thought experiment.

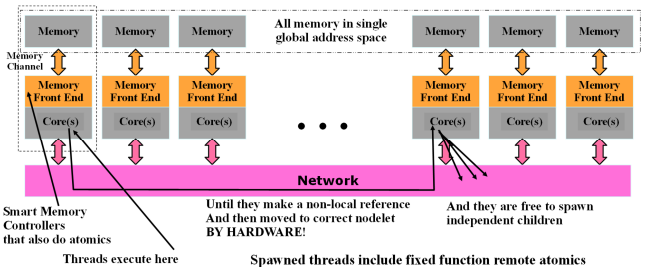

Figure 6. A generic migrating thread architecture.

5 A Proposed Architecture

The prior section suggests a variety of mechanisms that address some of the issues raised by the two experiments. However, none address the longer range issues of remote and reverse spawning and the cost of deep software stacks to handle them.

Fig. 6 diagrams one such possible architecture that avoids much of the software overhead by placing all memory in all nodes in a single logical Partitioned Global Address Space (PGAS), and supporting hardware-based spawning and migration of threads to alternative physical partitions if the memory they wish to access is not local [13]. To support the varying number of threads that may be resident in the same node at the same time, the cores in such a system (termed *MT cores* here) must become heavily multi-threaded. Thus a program thread may work with memory anywhere in the system without needing program-level awareness of where the data resides and where the thread is currently located.

Prototypes of such an architecture exist[1] [9]. The architecture also uses cheap thread spawning coupled with the migration mechanism to support very fast light weight remote memory operations for very inexpensive "action at a distance". The processing logic for such remote operation resides in the memory controllers (Memory Front End (MFE) in Fig. 6). A tool chain based on the Cilk language[2] [4], plus a rich library of intrinsic functions to perform both local and remote atomics, is an extraordinarily efficient match to the architecture. The combination has proven highly effective (cf. [21]) in scaling codes that normally involve large amounts of inter-node communication to handle irregular or dynamic execution profiles such as the updates due to walkers crossing nodes in Frogwild!.

While an architecture such as Fig. 6 has many desirable properties, it doesn't currently support integration with heterogeneous accelerators larger than simple remote operations performed in an MFE, especially when a "node" is greatly simplified, such as on the bottom of a 3D DRAM stack. In contrast, Fig. 7 gives a notional picture of a possible basic node to serve as the glue for a more heterogeneous

[1] See the Pathfinder system of 32 nodes, each with 16 MT cores, from the Lucata Corp. and located in Georgia Tech's CRNCH Center

[2] C plus the ability to specify a function call to be performed asynchronously.

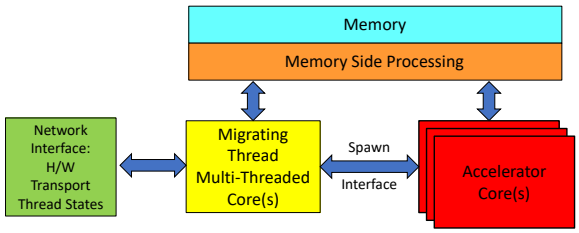

Figure 7. A migrating thread-based heterogeneous node.

system. As with Fig. 6 all memory is "logically" part of a common address space. A physical "node" as pictured in Fig. 7 has perhaps just a few cores, of potentially different types, that are optimized for specific processing. One of more MT cores sits at the interface between the network and the other cores. Both the accelerators and the MT cores have access to the local memory, although only the MT core need have knowledge of the memory in the rest of the system (it needs this to be able to determine where to migrate when necessary). The combination of the MT core(s), the accelerator core(s), the MFE, and the network interface could make a viable chiplet, with different chiplet designs differing only in what accelerators are present.

The ability to spawn new threads can also be more symmetric here. The MT core can spawn new MT threads (as in Cilk and the Lucata systems, and of both heavy and light weight), and accelerator threads (as in CUDA). Also, the accelerator cores may also spawn MT threads in the MT core[3]. As with both CUDA and Cilk, at a spawn in the MT core, the parent thread's registers form the basis for the registers for either a new MT thread or an accelerator thread.

6 Spawning Options

Prior work [14] has discussed some of the semantics of such border crossings, but have not suggested a matching hardware architecture. Going from the MT to the accelerator, there are at least two possibilities to consider. First is where the accelerator can perform multiple concurrent independent computations. This may be because it is multi-threaded or it has multiple cores, as in SM cores in modern GPUs. In either case, the spawn may look like that of an MT core spawning another MT thread. A register set on the accelerator side receives all necessary arguments, and then is released for execution.

An extension that allows multiple such register sets to be created, each with some different arguments, would also be useful. This would mirror either a CUDA call in a GPU program today, or a *cilk_for* loop in the Cilk programming language.

In either case, it may be that when an MT thread arrives and tries to spawn an accelerator thread, there is no space on the accelerator side for the new accelerator thread. An approach implemented in the Pathfinder prototype returns an indication that a spawn was not possible. The parent thread can try again, enqueue itself, or spawn a a child migrating thread that waits for the accelerator to be available while the parent continues.

A second model of MT to accelerator spawning is where the accelerator functions must be done atomically, one at a time, as in some sort of producer/consumer process. Here an Actor-like model is appropriate [10, 22]. If the accelerator is not busy, the MT thread can spawn an accelerator thread. If busy, the MT thread can enqueue either itself or the state for the desired accelerator call on a memory or hardware queue, which would be tested by the accelerator code when it completes an operation. Either the MT thread would be restarted (to try initiating the accelerator thread again), or the enqueued accelerator state would become active.

Understanding the reverse direction is also important. An accelerator code must also be capable of spawning an MT thread, which could travel anywhere in the system and start new computation. In addition, however, it should also be possible for an accelerator thread to be given a key that can be used to restart a waiting MT thread that spawned it.

Also note that the same mechanism that allows a node's hardware to know where to route an MT thread when it tries to access a non-local address is also useful for supporting RDMA operations that perform memory-to-memory transfers.

What is important about this model is the significant reduction in software complexity and runtime cost. There is no need for an inter-node messaging protocol, and only one set of drivers need to be written for each new accelerator type - namely between an MT core and each accelerator type. Further, such drivers can take advantage of hardware spawning mechanisms, and worry only about assembling the right parameters in the right "registers" for the accelerator thread. Finally, the asynchronous options within MT threads allow the code written for accelerators to be blind to how their functionality will play with that of other computations. Currently, there exists a commutable multi-threaded software, OpenCilk [25], mirroring aspects of the proposed model, with support for optimized compilers and work schedulers to ensure proper work-load balance and procedural scheduling. This allows developers to avoid bromidic tasks such as memory transfers, message passing, etc. seen in current interfaces, via underlying mechanisms, such as work-stealing, for optimal execution. A future extension of this which supports heterogeneous systems consisting of varying cores, will help a proposed model like Fig. 7 become attainable.

7 Conclusions

With the growth of chiplet and 3D technology, the expansion of systems with heterogeneous core designs can only go into

[3] In the Lucata design such a reverse spawn happens between the conventional hosts and the MT cores to initiate computation.

overdrive, with the concurrent need to devise architectures that simplify the software complexity of communicating between cores of different types. The intent of this paper was not to fully design a new architecture for such systems but to run some experiments that had many of the aspects we would expect from such systems, and use that to articulate what observations (both positive and negative) we saw that ought be considered in such an architectural definition.

Future explorations would involve constructing simulators for Fig; 7 where better implementations of apps such as Hogwild! and Frogwild! can be built to emulate larger systems such as seas of 3D stacks. In addition, given that threads are not bound to particular cores, it is worth investigating extensions to allow significant *multi-tenancy* (multiple separate applications sharing the same data sets and the same accelerators). Finally, such architectures resonate well with alternative programming models such as Actors.

Acknowledgments

This material was supported in part by the National Science Foundation Grant No. 1822939 and in part by the Univ. of Notre Dame, with collaboration with Vivek Sarkar of Georgia Tech.

References

[1] [n. d.]. https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper

[2] 2023. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[3] K. Avrachenkov and et al. 2010. Monte carlo methods for top-k personalized pagerank lists and name disambiguation. *arXiv preprint arXiv:1008.3775* (2010).

[4] Robert D. Blumofe and et al. 1995. Cilk: An Efficient Multithreaded Runtime System. In *Proc. of the Fifth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming* (Santa Barbara, California, USA) *(PPOPP '95)*. ACM, New York, NY, USA, 207–216. https://doi.org/10.1145/209936.209958

[5] Christian Borgs, Michael Brautbar, Jennifer Chayes, and Shang-Hua Teng. 2012. A sublinear time algorithm for pagerank computations. In *Int. Workshop on Algorithms and Models for the Web-Graph*. Springer, 41–53.

[6] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems* 30, 1-7 (1998), 107–117.

[7] Atish Das Sarma, Anisur Rahaman Molla, Gopal Pandurangan, and Eli Upfal. 2013. Fast distributed pagerank computation. In *Int. Conf. on Distributed Computing and Networking*. Springer, 11–26.

[8] Jeffrey Dean, Greg Corrado, and et al. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1223–1231. http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf

[9] Timothy Dysart, Peter Kogge, Martin Deneroff, and et al. 2016. Highly Scalable Near Memory Processing with Migrating Threads on the Emu System Architecture. In *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*. 2–9. https://doi.org/10.1109/IA3.2016.007

[10] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proc. of the 3rd Int. Joint Conf. on Artificial Intelligence* (Stanford, USA) *(IJCAI'73)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 235–245.

[11] Joseph Huber and Jon Chesterfield. 2023. OpenMP Reverse Offloading Using Shared Memory Remote Procedure Calls. In *Int. Workshop on OpenMP*. Springer, 226–238.

[12] Norman P. Jouppi and et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Comput. Archit. News* 45, 2 (jun 2017), 1–12. https://doi.org/10.1145/3140659.3080246

[13] P.M. Kogge. 2004. Of Piglets and Threadlets: Architectures for Self-Contained, Mobile, Memory Programming. In *Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'04)*. 130–138. https://doi.org/10.1109/IWIA.2004.10005

[14] Peter M. Kogge. 2019. Multi-threading Semantics for Highly Heterogeneous Systems Using Mobile Threads (nominated for best paper). In *2019 Int. Conf.on High Performance Computing Simulation (HPCS)* (Dublin, Ireland). 281–289. https://doi.org/10.1109/HPCS48598.2019.9188165

[15] Wenting Liu, Guangxia Li, and James Cheng. 2015. Fast PageRank approximation by adaptive sampling. *Knowledge and Information Systems* 42 (2015), 127–146.

[16] Ioannis Mitliagkas, Michael Borokhovich, Alexandros G Dimakis, and Constantine Caramanis. 2015. FrogWild!–fast PageRank approximations on graph engines. *arXiv preprint arXiv:1502.04281* (2015).

[17] Lam M. Nguyen, Phuong Ha Nguyen, and et al. 2018. SGD and Hogwild! Convergence Without the Bounded Gradients Assumption. (2018). https://doi.org/10.48550/ARXIV.1802.03801

[18] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. 2011. HOGWILD!: A Lock-free Approach to Parallelizing Stochastic Gradient Descent. In *Proc. of the 24th Int. Conf. on Neural Information Processing Systems* (Granada, Spain) *(NIPS'11)*. Curran Associates Inc., USA, 693–701. http://dl.acm.org/citation.cfm?id=2986459.2986537

[19] NVIDIA. [n. d.]. Jetson TX1 module. https://developer.nvidia.com/embedded/jetson-tx1

[20] NVIDIA. 2023. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#

[21] Brian A. Page and Peter Kogge. 2022. The Evolution of a New Model of Computation. In *2022 IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. 9–18. https://doi.org/10.1109/IA356718.2022.00008

[22] Sri Raj Paul, Akihiro Hayashi, Kun Chen, and Vivek Sarkar. 2022. A Productive and Scalable Actor-Based Programming System for PGAS Applications. In *Computational Science – ICCS 2022*, Derek Groen, Clélia de Mulatier, Maciej Paszynski, Valeria V. Krzhizhanovskaya, Jack J. Dongarra, and Peter M. A. Sloot (Eds.). Springer Int. Publishing, Cham, 233–247.

[23] P Rogers. 2016. HSA overview. In *Hetero. System Architecture*. Elsevier, 7–18.

[24] Christopher De Sa, Ce Zhang, Kunle Olukotun, and Christopher Ré. 2015. Taming the Wild: A Unified Analysis of HOG WILD! -Style Algorithms. In *Proc. of the 28th Int. Conf. on Neural Information Processing Systems - Volume 2* (Montreal, Canada) *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 2674–2682.

[25] Tao B. Schardl and I-Ting Angelina Lee. 2023. OpenCilk: A Modular and Extensible Software Infrastructure for Fast Task-Parallel Code. In *Proc. of the 28th ACM SIGPLAN Annual Symp. on Principles and Practice of Parallel Programming* (Montreal, QC, Canada) *(PPoPP '23)*. ACM, New York, NY, USA, 189–203. https://doi.org/10.1145/3572848.3577509

[26] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering* 12, 3 (2010), 66–73. https://doi.org/10.1109/MCSE.2010.69

[27] Ján Veselỳ, Arkaprava Basu, Abhishek Bhattacharjee, Gabriel H Loh, Mark Oskin, and Steven K Reinhardt. 2018. Generic system calls for GPUs. In *2018 ACM/IEEE 45th Annual Int. Symp. on Computer*

Table 4. Differences between Hogwild! and Frogwild!

Attribute	Hogwild!	Frogwild!
Parallelization Style	Data Parallel	Object/Vertex Parallel
Non-local memory updates	Infrequent, expensive	Frequent, cheap
Order of Updates	Deterministic Ordering	Stochastic Ordering
Perf. Bottleneck	Computationally Bound	Memory Bound
Network Constraints	Bandwidth	Large Volume of Short Messages
Payload Size/Transaction	O(n)	O(1)
Network Demands	Independent of node count	Inversely related to node count

Architecture (ISCA). IEEE, 843–856.

[28] Bruce Wile. 2014. *Coherent Accelerator Processor Interface (CAPI) for POWER8 systems*. Technical Report.

[29] Ce Zhang and Christopher Ré. 2014. DimmWitted: A Study of Main-Memory Statistical Analytics. *Proc. VLDB Endow.* 7, 12 (Aug. 2014), 1283–1294. https://doi.org/10.14778/2732977.2733001

[30] H. Zhang, C. J. Hsieh, and V. Akella. 2016. HogWild++: A New Mechanism for Decentralized Asynchronous Stochastic Gradient Descent. In *2016 IEEE 16th Int. Conf. on Data Mining (ICDM)*. 629–638. https://doi.org/10.1109/ICDM.2016.0074

A Appendix

A.1 Similarities and Differences

Tables 3 and 4 outline the similarities and differences between these two algorithms as implemented on heterogeneous platforms. Parallel computations in Hogwild! primarily involve vector-vector multiplication and vector addition, whereas Frogwild! involves atomic additions and subtractions done simultaneously by different threads. The updates to the model vector in Hogwild! are predictable w.r.t to the order of clusters. These involve a large payload of size n, which capitalizes on contiguous memory accesses, but places a demand on the network's bandwidth. The updates in Frogwild! are frequent and sporadic, done by threads randomly accessing singular memory locations in different clusters.

Table 3. Similarities between Hogwild! and Frogwild!

Attribute	Similarities
Implementation Environment	Distributed
Heterogeneity	Core computations accelerator-friendly, but with need for "conventional" glue
Network Consequences	Reduced Traffic w.r.t Original Implementation
Function spawn	Main accelerator code spawned once at start
Performance Consequences	Lower Execution Time
Accuracy Consequences	Comparable Accuracy w.r.t Original Implementation