Linking Parallel Algorithmic Thinking to Many-Core Memory Systems and Speedups for Boosted Decision Trees

James A. Edwards
University of Maryland
College Park, Maryland
jedward5@umd.edu

Uzi Vishkin
University of Maryland
College Park, Maryland
vishkin@umd.edu

ABSTRACT

The current focus of research on parallel computing takes current commercial hardware for granted. Here, we consider an alternative approach: start with a time-tested algorithmic theory and develop a supporting computer architecture and toolchain. This paper focuses on the hybrid memory architecture of this computer platform, which is designed to efficiently support execution of both serial and parallel code and switching between the two. A key part of this architecture is a flexible all-to-all interconnection network that connects processors to shared memory modules. To understand some recent advances in GPU memory architecture and how they relate to this hybrid memory architecture, we use microbenchmarks including list ranking.

A second part of this work contrasts the scalability of applications with that of routines. In particular, regardless of the scalability needs of full applications, some routines may involve smaller problem sizes, and in particular smaller levels of parallelism, perhaps even serial. To see how a hybrid memory architecture can benefit such applications, we simulate a computer with such an architecture and demonstrate the potential for a speedup of 3.3X over NVIDIA's most powerful GPU to date on boosted decision trees, a timely machine learning application.

CCS CONCEPTS

• **Computing methodologies → Shared memory algorithms**;
• **Computer systems organization → Multicore architectures**;

KEYWORDS

parallel algorithmic thinking, memory systems

ACM Reference Format:
James A. Edwards and Uzi Vishkin. 2018. Linking Parallel Algorithmic Thinking to Many-Core Memory Systems and Speedups for Boosted Decision Trees. In *The International Symposium on Memory Systems (MEMSYS), October 1–4, 2018, Old Town Alexandria, VA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3240302.3240321

1 INTRODUCTION

Since the circa 2004 transition of mainstream computing to parallelism, efforts of the research community have been centered around commercial multi-core or GPU hardware, unwittingly ceding strategic intellectual leadership of the field to vendors. Extensive work on mapping and tuning algorithms and performance programs for a generation of products has dominated contemporary conferences, journals and research dissertations. Vendors have been changing their designs at a rather brisk pace rendering this work Sisyphean: Even for cases where a vendor and a product line remained in business, this work had to often be redone for successive generations, sometimes ab initio. System architecture research has not fared much better. The focus of the "quantitative approach" is on exploring limited updates to commercial systems. As committee peer review is required to rank technically incomparable submissions, it unwittingly conforms with dominant modes of operation. Thus, publication and funding incentives risk upholding futile efforts. The state of educating CS undergraduates to properly benefit from parallelism widely available in the very machines they use suggest further alarming evidence. Reflecting a rather broad computer systems community, the sixth edition [7] justifies a recent shift to heterogeneous platforms by stating: "it seems unlikely that some form of simple multicore scaling will provide a cost-effective path to growing performance". However, we are concerned that such a shift may augment Sisyphean efforts with Babel-Tower-type problems, making a bad situation even worse.

We believe that promoting fundamental understandings and robust knowledge, independent of commercial players, is key to basic academic research. Thus, it would make sense for MEMSYS to ask whether we can do better.

A completely different approach, dubbed "Explicit Multi-Threaded (XMT)", is discussed in [12]. The lead immediate concurrent execution (ICE) abstraction underlying PRAM, the main theory of parallel algorithms, is the concept that each step of a program needs to state all the operations that can done concurrently and assume their lock-step execution in unit time, but nothing else. The horizon envisioned by XMT is that of having the parallel programmer express parallelism using ICE without ever needing to be concerned with threading, race conditions or locality, while achieving competitive performance. A vertically integrated hardware/software on-chip system has been introduced and extensively prototyped, demonstrating speedups by order of magnitude over same-generation commercial platforms for irregular and fine-grained applications. Removing the last obstacles to efficiently implementing textbook PRAM algorithms as-is, this effort culminated in demonstrating [6] that ICE programs can fully match the performance of manually optimized multi-threaded code on XMT, thereby establishing feasibility of the XMT-envisioned horizon. But, would an XMT/PRAM/ICE approach lead to more robust insights? For algorithms the answer is yes. The PRAM algorithms theory has been stable since the 1980s.

MEMSYS, October 1–4, 2018, Old Town Alexandria, VA, USA
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6475-1/18/10.
https://doi.org/10.1145/3240302.3240321

Using PRAM algorithms as-is per [6] is very appealing and holds promise, especially if supported by architecture as for XMT. But, what are the prospects that architecture insights and, in particular, memory architecture ones meet the test of time?

The hybrid memory architecture underlying XMT features: (i) A master CPU with a traditional cache ("serial mode") and a plurality of CPUs using shared memory cache; none of the parallel CPUs has local write caches. And (ii) low-overhead transition between these serial and parallel memories. In conjunction with a high-bandwidth, on-chip, all-to-all interconnection network, these allow competitive performance regardless of how much parallelism a given code presents.

This paper presents: (i) new evidence that both multi-core and GPU design have been getting much closer to this hybrid memory architecture given their original starting principles that guided them in the opposite direction, reasoning that their current quest for more effective support of fine-grained irregular parallelism drew them closer to such memory architecture; and (ii) new speedup results of 3.3X over NVIDIA's most powerful GPU to date for a timely machine learning algorithm.

There are several reasons why we believe that our paper can be stimulating for MEMSYS:

- It provides a unique perspective. In particular, raising the provocative question whether the recent shift to heterogeneous platforms has turned prior Sisyphean efforts into an even more problematic Babel Tower is likely to get some commotion from at least some of the audience. We hope that memory and system researchers will realize the need for operating outside the spell of incremental improvements to commercial systems, and the opportunity for doing that, especially once such incremental approaches are contrasted with the PRAM-based option to do better on both robustness and homogeneity.
- The evidence on multi-core and GPU design getting much closer to the XMT hybrid memory architecture raises the question whether the fundamental nature of parallel algorithms and programs may have a similar effect to gravitation power, drawing us in a certain direction regardless if we are aware of it or not.
- The above mentioned demonstration of ICE programming on XMT, along with success stories such as having XMT programming taught to about 700 students in a single high school (Thomas Jefferson High School for Science and Technology, Alexandria, VA) since 2009, suggest that providing simple multicore scaling and a cost-effective path to growing performance may not be as insurmountable as [7] and many other computer architects opine.

Section 2 of this paper discusses the hybrid memory architecture underlying XMT. Section 3 briefly describes boosted decision trees, a timely application which we use to show the benefit of our hybrid memory architecture, as well as the results we obtained for this application. Finally, section 4 evaluates some choices we made in the design of XMT.

2 MEMORY ARCHITECTURE OF XMT

2.1 Our goal

One old insight of XMT is the need to support effectively in one architecture two memory paradigms: serial and parallel. Many programs consist of both serial sections of code and parallel sections, potentially with varying degrees of parallelism. Amdahl's Law implies that speeding up one section alone will be of limited benefit; to improve performance beyond a certain point, all sections must be sped up. In addition to the need for strong serial support for programs for which no parallel implementation is currently available, serial execution also shows up in more subtle ways in parallel programs. First, portions of some parallel programs may have limited parallelism, and in such cases it may be faster to execute those portions on a strong serial processor rather than underutilizing the parallel processors. Second, programs with fine-grained parallelism, even those with much available parallelism, need to switch between serial and parallel modes of execution frequently to orchestrate the spawning and synchronization of threads; for example, when parallelism is not represented by long running threads communicate infrequently, rarely or not at all, and lower overheads for switching to serial mode and back to parallel mode justify this.

From the memory architecture point of view, there is a tension between the goals of supporting serial and parallel computation. Serial code is more sensitive to memory latency than to bandwidth, as there is limited opportunity for a single thread to hide latency, while parallel code can issue many requests in parallel to hide latency. Reducing latency often requires bringing data closer to the processor, such as in a private cache. On the other hand, parallel computation often requires sharing data among processors, and protocols to maintain coherence among private caches scale poorly.

In light of this, we have developed a hybrid architecture with two components: (1) a "heavy" serial processor with a private writable cache and (2) a number of "light" parallel processors, each without a private writable cache. The two components are tightly coupled such that switching from serial to parallel and back can accomplished in time on the order of 10-100 cycles.

To develop this hybrid architecture, we first considered what limits performance from the algorithm side. Under the PRAM algorithmic model, the relevant factors are (1) work, the total number of operations to be performed and (2) depth, the length of the critical path of execution. From the memory architecture point of view, work comprises the amount of data read from and written to memory, and depth is dominated by the length of the sequence of round trips to memory (LSRTM). We then asked, for a given choice of workload, what is the best LSRTM that can be achieved from the parallel algorithm side. After performing this optimization by hand, we set out to automate this task. Through iteration of the hardware as well as the software development toolchain, we refined the automation of the process of achieving a given LSRTM.

2.2 Our design choices

Here, we describe our lead design choices for the two components of our memory architecture and how they work together. See fig. 1. This is just one possible set of choices we could have made; for a discussion and evaluation of design choices, see section 4.

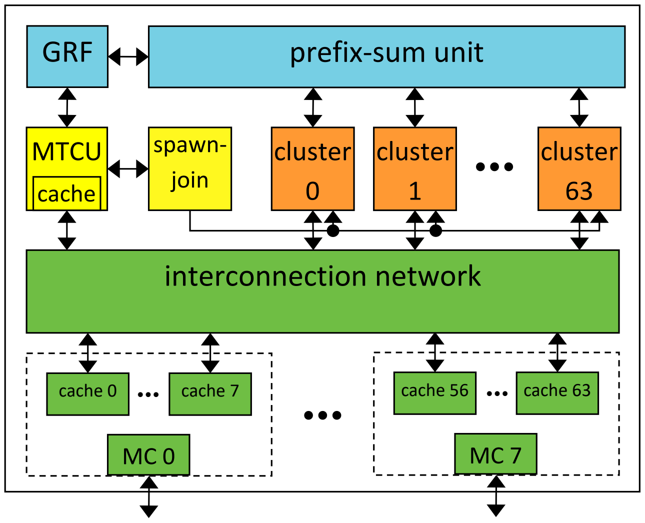

Figure 1: Block diagram of hybrid memory architecture of XMT. The serial portion comprises the MTCU, which includes a local cache. The spawn-join unit (yellow) is used for transitioning between serial and parallel mode from the control side, which is a bit suppressed in this current memory-centered paper. The parallel portion comprises the clusters (orange). The shared memory system comprises the interconnection network, shared caches, and memory controllers (green); it is used by both the serial and parallel portions. The global register file (GRF) and prefix-sum unit (blue) are used to coordinate concurrent execution of threads.

2.2.1 Serial mode. In serial mode, a single master thread control unit (MTCU) executes code. The MTCU is a standard serial processor core with its own private, writable cache. We choose a write-through no-write-allocate policy for the cache to reduce the potential for data to be brought into cache unnecessarily, which helps reduce the time needed to flush the cache. The MTCU private cache is connected to shared memory via a port on the interconnection network just like the TCUs.

2.2.2 Parallel mode. In parallel mode, a number of thread control units (TCUs) execute the program contained within the current parallel section of code (delimited by "spawn" and "join" instructions in XMT). TCUs are grouped into clusters (typically 16 TCUs per cluster) that share some resources including a single port to the shared cache.

TCUs lack private writable caches. Instead, several read-only memories are used to reduce the latency, and in some cases bandwidth, of accesses to shared memory:

- Each TCU stores a copy of the program in a local instruction buffer. This allows TCUs to run at their own pace rather than in lockstep.
- TCUs contain software-managed prefetch buffers, which reduce latency by allowing TCUs to send read requests to memory before they will be needed by the program and reduce LSRTM by allowing TCUs to issue reads back-to-back without waiting for them to complete one-by-one.
- Clusters contain read-only buffers, which are software-managed caches that allow TCUs to reuse data read by other TCUs in the cluster.

The shared cache is partitioned into cache modules, where each module is backed by a partition of the global memory space. Clusters communicate with the cache modules via an all-to-all interconnection network (ICN). For smaller configurations of XMT, the ICN is a mesh-of-trees network (MoT); for configurations where a pure MoT would be too large, a hybrid network is used instead where some of the middle layers of the MoT are replaced with layers of a butterfly network. All access by the TCUs to shared memory goes through the ICN.

Finally, the cache modules are connected to main memory (DRAM) via one or more memory controllers, which are evenly partitioned among the cache modules.

Requests by multiple TCUs in a cluster are queued, as are requests to the same cache module. Requests by the cache modules to the memory controllers are also queued. We hash memory addresses to spread memory accesses more evenly across the cache modules to reduce hot spots.

2.2.3 Transition from serial to parallel. When spawning threads, the MTCU first flushes its private cache to shared cache. This ensures that all data is available to the TCUs without the need for cache coherence protocols. Assuming that not too much data is brought into the local MTCU cache, the flush will be efficient. A possible optimization here would be to flush only those cache lines containing data that will be needed in parallel mode.

Then, starting immediately after the spawn instruction, the MTCU broadcasts the spawn block to the TCUs one instruction after another. Because each TCU has its own copy of the program and its own program counter, each thread can progress at its own pace.

2.2.4 Transition from parallel to serial. After all threads finish executing, TCUs wait for all outstanding requests to shared memory to complete, and then control returns to the MTCU. All local parallel memories (e.g., read-only buffer) are invalidated; no data needs to be written from local memory to shared memory since the local memories are read only.

3 APPLICATION: BOOSTED DECISION TREES

An increasingly-popular approach to machine learning is gradient boosted decision trees, as implemented by XGBoost [2]. XGBoost is designed to perform well on serial and parallel CPUs and has recently been extended to be supported by GPUs [9] as well. According to the authors of XGBoost, it has been used by many winners of machine learning competitions, including all of the top-10 winners of KDDCup 2015 as well as many top-3 winners on the popular machine learning competition website Kaggle (acquired by Google in 2017): 17 of the 29 challenge winning solutions published on Kaggle's blog during 2015 used XGBoost, compared with 11 that used deep neural networks. In some senses, Kaggle represents the marketplace for data scientists: companies often sponsor competitions on Kaggle to find solutions to problems of interest to them, and they also use Kaggle for recruiting data scientists, either by evaluating the Kaggle ranks of applicants to data scientist positions or by sponsoring competitions on Kaggle whose purpose is recruiting.

Reducing the training time would be beneficial to users of XGBoost. Indeed, speedups have been demonstrated using GPUs [9], and work continues on reducing times even on CPUs (e.g., the current beta version of the Intel Data Analytics Acceleration Library (DAAL) [8]). However, we conjecture that there is room for further speedups, and we have produced initial evidence to validate this conjecture. GPUs are tuned for approaches such as deep learning that consist mostly of regular operations with high computational intensity such as matrix multiplication and convolution. In contrast, XGBoost relies heavily on irregular operations with low computational intensity, such as sorting, compaction, and prefix sums with indirect addressing. Although support for irregular algorithms on GPUs appears to be improving, it still lags far behind support for regular algorithms.

3.1 High-level review of algorithm

A decision tree is a binary tree where internal nodes represent yes-or-no questions about an instance and leaf nodes represent the label to be reported for all instances that lead to that leaf. A simple type of decision tree is one in which each internal node asks whether a certain feature of the input is below or above some threshold, where the choice of feature and threshold are parameters of the model.

On their own, decision trees may be prone to overfitting. One approach to mitigate this is by limiting the depth, and thus the complexity, of the tree. The downside to this is that a single shallow decision tree is a fairly weak model. To compensate for this, multiple decision trees can be trained and their results averaged to produce a stronger model. XGBoost uses a boosted decision tree approach, in which trees are added one by one to refine the output produced by the trees in the model so far.

XGBoost uses a greedy approach to build each decision tree. To begin, XGBoost starts by creating a single leaf node and assigning all of the training examples to that leaf. XGBoost builds the tree by recursively splitting the examples at each leaf so as to produce the highest information gain, stopping when the gain falls below a specified threshold.

The majority of the time taken by XGBoost is spent searching for the best split point (a feature and its threshold) for each leaf. For each possible split, XGBoost looks at the left and right sides of the split and for each side computes a score representing how large of a refinement will be made by this split; the information gain is the sum of these two scores minus the score of the original, un-split node. A simple scoring function provided by XGBoost is to compute the square of the sum of the errors (signed differences) between the true output for each training example and the current prediction.

XGBoost makes use of the following insight: if the training examples are sorted in order of increasing value of a given feature, then all possible splits for that feature can be trivially found by walking through the list. Furthermore, the sums of errors can be updated while walking through the list simply by subtracting the error of the current element from the sum for the right side and adding it to the sum for the left side. This implies that the sums that are needed are the prefix-sums of the errors in this sorted order. Because a different sorted order is needed for each feature, the order in which the training examples are accessed is constantly changing, leading to an irregular memory access pattern.

3.1.1 Overview of parallel algorithm on XMT. Our parallel algorithm for XGBoost on XMT takes the serial algorithm and replaces each step with a corresponding parallel alternative:

- We sort the training examples using a shared-memory sample sort. In contrast to work on GPUs [9] that uses radix sort, this is less regular but provides more parallelism for some inputs. We do not rearrange the examples themselves in memory but instead maintain arrays of pointers to the examples in sorted order, one per feature. This results in more irregular memory access in later steps but saves work when splitting nodes.
- To compute the sums of errors, we use a parallel prefix-sums algorithm. This is similar to [9]; however, XMT can exploit more parallelism in this step than GPUs: on XMT, all TCUs can participate in computing the prefix-sums for a single feature whereas the GPU algorithm only uses a single thread block per feature.
- To find the split with the maximum score, we use a parallel reduction algorithm with maximum as the associative binary operator. Again, this is similar to [9] but without the limitation of one thread block per feature.
- To split each node and rearrange its associated examples accordingly, we apply parallel prefix-sums to perform compaction. In contrast, [9] employs two strategies to handle splitting: (a) for the first few levels of the tree, do not rearrange the examples. Instead, mark each example with the node it now belongs to after each split. (b) For deeper levels of the tree, rearrange the examples after each split using radix sort.

The XMT algorithm above and the GPU algorithm of [9] represent different trade-offs resulting from the memory architectures of the respective platforms. The XMT algorithm favors reducing algorithmic complexity (work and depth) at the expense of increased irregularity, maintaining pointers to examples and rearranging them as necessary to avoid idle threads in later steps. In contrast, the GPU algorithm maintains regularity as much as possible at the expense of increased work and depth by deferring irregular data movement until the overhead of skipping over examples that do not belong to the current node becomes prohibitive.

We also note that both the XMT and GPU algorithms involve numerous transitions between serial and parallel execution, as most of the above steps are executed many times and each execution of a step incurs multiple serial-to-parallel transitions. The impact of this is discussed in sections 4.4 and 4.5.

3.2 Method

We compare XMT to commercial CPU and GPU platforms for the machine learning approach of gradient boosted decision trees and obtain significant speedups.

Software To facilitate a fair comparison, we compared our code against the following:

(1) XGBoost, both serial and parallel CPU implementations [2]
(2) the GPU-accelerated version of XGBoost [9].

Here, we focus on the training step, as it is more time consuming than inference and the parallelism is more difficult to exploit. This is not to exclude inference: although inference on decision forests is embarrassingly parallel across the trees, we still would expect some benefit on XMT since the problem is irregular.

XGBoost is written in C++ (with GPU kernels in CUDA), but there is currently no C++ compiler available for XMT. Therefore, we needed to rewrite the XGBoost algorithm in XMTC, with a focus on computing the same result as the original XGBoost code while exposing some parallelism. This starting point may prove to be a disadvantage here, as XGBoost was designed with the strengths and weaknesses of multi-core CPUs and GPUs in mind, and after more extensive work, we may be able to get better speedups.

Computing platforms To obtain serial and parallel CPU performance results, we ran XGBoost on a modern Linux machine with two 8-core Intel Xeon E5-2690 processors (16 cores in total). To obtain GPU results, we ran XGBoost on an Amazon EC2 p3.2xlarge instance, which includes eight cores of an Intel Xeon E5-2686 v4 CPU and a Tesla V100 GPU (Volta microarchitecture), NVIDIA's most advanced GPU to date.

Results for XMT were obtained using XMTSim, a cycle-accurate simulator of the XMT architecture derived from a commitment to silicon of XMT using FPGA. XMTSim was configured to simulate an XMT processor that would use silicon area comparable to the Tesla V100 (16,384 TCUs, 32 MB shared cache) and also provide nearly the same bandwidth to DRAM (768 GB/s).

XGBoost configuration and dataset XGBoost on all platforms was configured to generate 120 trees with a maximum depth of 6.

The dataset used in this experiment was the Higgs boson dataset taken from the Kaggle machine learning challenge website. It consists of 250,000 training examples with 30 features each. This dataset was also used by the authors of XGBoost in their work.

3.3 Speedups

The results show that among the platforms above, XMT would outperform both the CPU and the GPU; see fig. 2.

As far as we found out, other work on parallel implementations of decision forests does not report direct comparisons to the best serial implementation. Work on training tree ensembles using MapReduce [10] did not report any speedup versus best serial due to lack of memory on the serial machine.

Work on boosted trees [11] achieved a self-speedup of up to 42× on a 48-core shared memory machine and up to 25× on a 32-core distributed memory machine, but no results are reported relative to best serial.

4 DISCUSSION OF DESIGN CHOICES

The above is one possible design choice for a hybrid memory architecture, which we made based on looking at various parallel workloads. Up to a point, the community has agreed on the transition from serial computing to parallel computing. Although no description is publicly available, NVIDIA GPUs appear to have a high-performance all-to-all network connecting parallel processors to shared cache. However, the CPU and GPU each have some separate memories, where data shared between them must be copied from one to the other. In addition to discrete GPUs, AMD also produces Accelerated Processing Units (APUs), which combine a traditional CPU and GPU on a single die, and Intel also produces CPUs with an integrated GPU. However, the balance of silicon area between CPU cores and GPU cores on these chips has so far not favored GPU performance as in discrete GPUs.

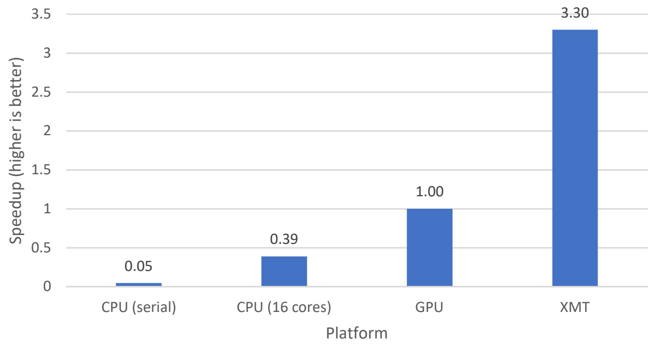

Figure 2: Speedups of XGBoost on various platforms relative to the most powerful NVIDIA GPU. XMT has a speedup of 3.3X while the CPU platforms have slowdowns (speedup <1X).

4.1 Evidence for advances in GPU memory architecture

Our discovery of recent changes in modern NVIDIA GPU memory architecture is a bit anecdotal. It began with our earlier attempts to run cycle-accurate simulations of programs running on modern GPUs. We used FusionSim [13], based on GPGPU-Sim [1], as a starting point and adapted the included configuration, which was designed to match the NVIDIA GTX 480 GPU (Fermi architecture), to attempt to match the NVIDIA Tesla M40. We used FusionSim rather than GPGPU-Sim alone since we sought to model the transitions between serial and parallel execution in addition to the parallel kernels themselves.

We ran a list ranking benchmark based on parallel pointer jumping as a (highly irregular) benchmark of three NVIDIA GPUs as well as FusionSim. Our goal was to develop a cycle-accurate simulation of the Tesla M40 GPU that we would then use for further work plans. However, we had to abandon our plans since we could not get FusionSim to match the actual performance of modern GPUs. The differences we observed can be seen in fig. 3. For large inputs sizes of 1 million elements or more, FusionSim matches the Tesla M40. However, we point out two discrepancies for lists smaller than this.

First, for small input sizes (less than 8000 elements), FusionSim underestimates the run time of the benchmark relative to all three of the actual GPUs. This implies that there are additional overheads for launching kernels on the actual GPUs that are not reflected in FusionSim.

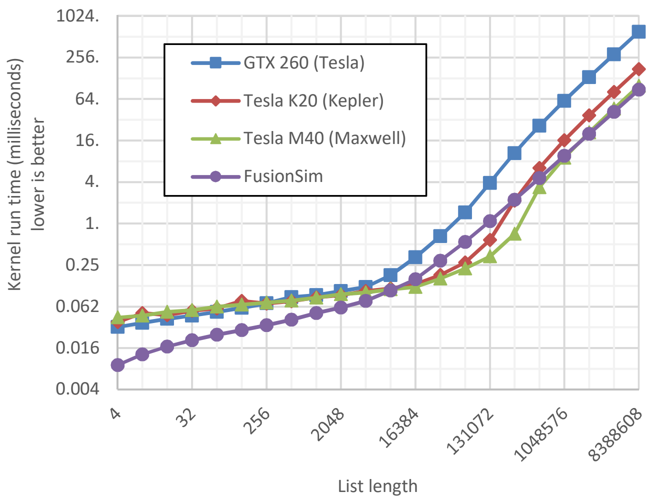

Figure 3: Cycle accuracy of FusionSim (GPGPU-Sim) relative to three NVIDIA GPUs running a list ranking benchmark.

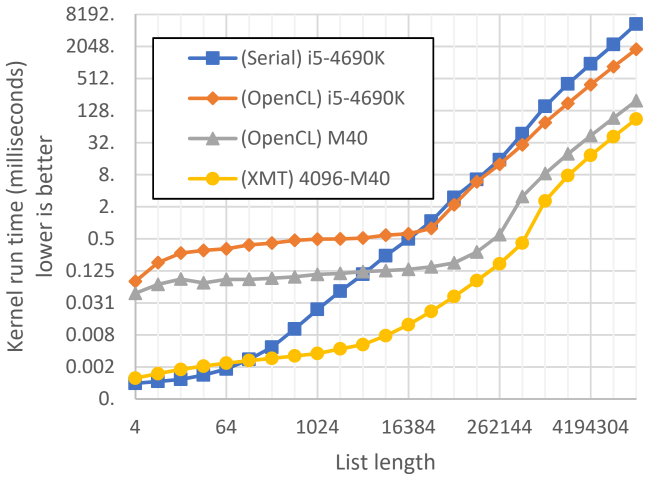

Figure 4: Time taken by various processors on a list ranking benchmark, including a serial CPU (Intel Xeon E5-2686 v4), an integrated GPU (Intel Core i5-4690K), a discrete GPU (NVIDIA Tesla M40), and a simulated XMT system configured to match the Tesla M40.

Second, the more recent Tesla K20 and M40 GPUs exhibit a steeper increase in runtime at around 250 thousand elements than at any other point, but FusionSim does not reflect this; FusionSim more closely follows the older GTX 260 in this respect.

In particular, the second observation above led us to suspect that NVIDIA made some improvements between the release of the GTX 260 in 2008 and the Tesla K20 in 2012. We could not make sense of the nature of this improvement based on published papers. In fact, we found it surprising given the well-cited keynote talk [4] with its claim: "locality equals efficiency"; how can parallel architectures that equate locality with efficiency (and minimizing reliance on non-local memories) provide such strong support for high rates of data movement? So, we felt that we need to dig deeper. To our surprise we found a patent [5] filed five years earlier, which went barely unnoticed in the literature suggesting that NVIDIA is indeed heading in a direction that seems a near opposite of [4]. That is, providing much better support for shared memory at the expense of local memories on its GPUs. Interestingly, [5] still suggests similar motivation to [4]; namely, that it would be better from an energy consumption point of view. However, we have not been able to find support in the literature for improved energy consumption as a result of trading local memories for shared ones. In fact, much of the architecture literature seems to continue being influenced by [4] and its call for limiting data movement. Indeed, when we then looked up information about the streaming multiprocessor in their P100 Volta, we didn't expect to find that even the register file is shared. It will be interesting to find out at the conference how representative is our anecdotal experience. Finally, the extent to which support for low-overhead transition between serial and parallel execution is being followed remains to be seen as GPUs continue to evolve.

4.2 Integrated vs. discrete GPUs

Some recent Intel processors have integrated GPUs that share their memory system with that of the CPU cores. Two examples are the Intel Core i5-4690K (with an Intel HD Graphics 4600 GPU) and the more recent Intel Xeon E3-1578L v5 (with an Intel Iris Pro Graphics P580 GPU). A notable difference between these two is that the P580 has 128 MB of eDRAM on the same package, which is used as a level 4 cache. The results we were able to get for the Intel Xeon E3-1578L v5 were a bit inconsistent, which we speculate may be due to the first generations of Intel GPUs with eDRAM not being fully optimized. Therefore, we discuss only the Intel Core i5-4690K in the following comparisons.

4.3 Performance on irregular algorithms: list ranking

We use the same list ranking benchmark as before to determine whether the integration of the GPU provides an advantage here and to see whether we can detect any improvement due to more recent Intel GPUs being more tightly integrated. See fig. 4. For the largest list (16 million elements), the i5-4690K achieves a speedup versus serial of 2.6X. The integrated GPU is outperformed by the more powerful discrete M40 GPU for all list sizes, which is expected since this benchmark involves little communication between the GPU and the CPU. Notably, XMT performs nearly as well as the serial CPU for small inputs while beating the M40 GPU even for large inputs.

4.4 Serial-parallel transition overhead

The GPU version of the boosted decision tree program as tested in section 3 has over ten thousand kernel launches, and the XMT version has nearly as many parallel sections. Here, we examine the overhead of this more closely under two runtimes: OpenCL and OpenGL

4.4.1 OpenCL. Figure 5 shows the overhead of switching from serial execution to parallel and back in terms of the time taken to launch an empty OpenCL kernel. Surprisingly, the discrete K20 and M40 GPUs outperform the integrated GPU even for small numbers of threads. XMT spawns threads faster than any GPU by over an order of magnitude when the amount of parallelism is low and remains faster than the GPUs even when much parallelism is available.

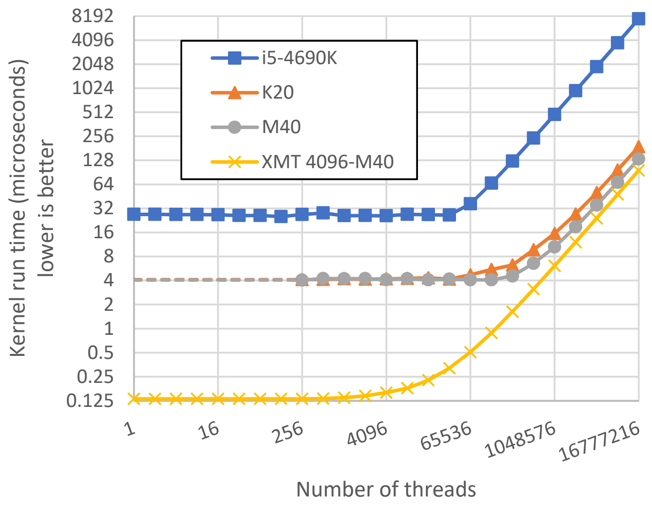

Figure 5: Time taken to launch a single empty OpenCL kernel (spawn block on XMT) with respect to the number of threads launched. This is a measure of the time required to transition from serial to parallel and back. For each platform, the run time starts increasing once the number of threads reaches the maximum the platform can run at a time. For the K20 and M40, we use a minimum of 256 threads (indicated by the dotted line) since these GPUs are not designed for fewer threads; in our tests, running this benchmark with fewer than 256 threads was slower than with 256 threads.

4.4.2 OpenGL. We suspected that the poor performance of the integrated GPU relative to the discrete GPUs may be due in part to the NVIDIA OpenCL runtime being more optimized than its Intel counterpart. In an attempt to avoid the overhead of OpenCL, we tested a short OpenGL graphics benchmark consisting of a single OpenGL shader program that combines two textures (essentially arrays of pixels) using a simple arithmetic operation. Because this is a graphics benchmark, we were limited to running on computers that were configured to allow using the GPU for rendering graphics rather than only for GPGPU computation. Hence, we do not have results for the NVIDIA K20, M40, or V100 GPUs.

Figure 6 shows that for inputs up to 2048 pixels, the Intel i5 processor with HD Graphics 4600 is faster than the discrete NVIDIA GTX 1060 GPU. Possible explanation for this advantage can be found in the Memory section of [3].

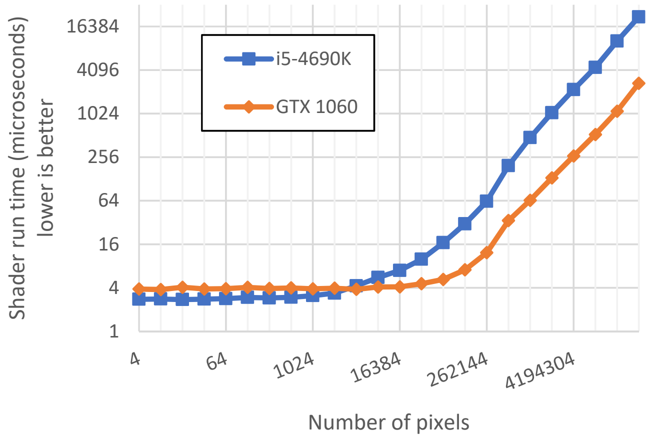

Figure 6: Figure: Time to execute a short OpenGL shader that applies a SAXPY operation (y := y+ax) to two textures x and y versus texture size in pixels (length times width). For inputs up to 2048 pixels, the integrated GPU of the Intel i5-4690K processor is faster, indicating lower overhead.

4.5 Sensitivity to serial-parallel transition overhead

To gain some understanding of the importance of low-overhead transition from serial to parallel in a complete application, we examine what would happen if this overhead were increased relative to baseline provided by the XGBoost results above. In fig. 7, we show the effect of increasing the spawn latency, which is the hardware portion of the transition overhead, from its original value of 23 cycles to various values up to and including 50,000 cycles. Latencies up to about 1000 cycles have little effect on speedup, but performance falls off beyond that point. For comparison, the typical GPU kernel launch latency is around 10,000 cycles. If the overhead for serial-to-parallel transition on XMT were as high as it is for the GPU, then XMT would perform no better than the GPU.

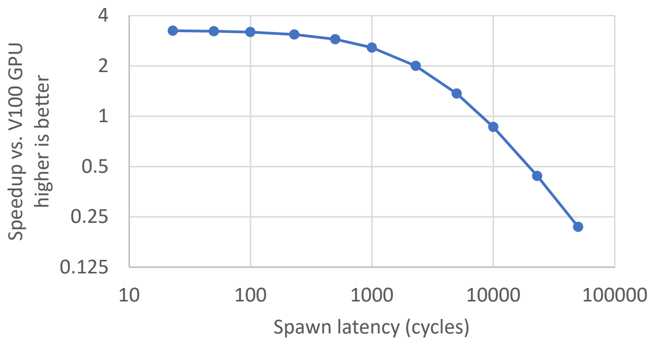

Figure 7: Effect of serial-to-parallel transition (spawn) latency on speedup (log-log axes). The leftmost point is the speedup for the latency

5 CONCLUSION

As their name suggests, streaming multiprocessor memory organizations have long provided strong support for moving data in and out of execution units. However, as long advocated by our XMT/PRAM approach, the need to better support irregular parallel algorithms led some successful GPU designs to increasingly move towards reliance on shared memories, breaking away with their past emphasis on local memories and locality at all cost. While this has led to marked improvements, their limited ability to support down-scaling of parallelism, especially for discrete GPUs, is hurting them significantly for supporting some full applications. The emphasis of some sections of the machine learning market on methods such as stochastic gradient descent (SGD), and their reliance on full matrix multiplication, for deep learning, appears to take a toll for other prominent market success stories in machine learning, such as the boosted decision trees application discussed in this paper.

However, our experience with XMT suggests that something bigger is at stake here. We demonstrated strong speedup on general-purpose applications, full support of the main theory of parallel algorithms and easy parallel programming; and, therefore, directions for finally providing simple multicore scaling and a cost-effective path to growing performance, finally overcoming what [7] and many other computer architects suggest is insurmountable.

REFERENCES

[1] Ali Bakhoda, George Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. https://doi.org/10.1109/ISPASS.2009.4919648

[2] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16*. 785–794. https://doi.org/10.1145/2939672.2939785

[3] Intel Corporation. 2015. The Compute Architecture of Intel Processor Graphics Gen9. https://software.intel.com/sites/default/files/managed/c5/9a/The-Compute-Architecture-of-Intel-Processor-Graphics-Gen9-v1d0.pdf

[4] W.J. Dally. 2009. *The end of denial architecture.* Technical Report. The International Symposium on Asynchronous Circuits and Systems (ASYNC). http://asyncsymposium.org/async2009/slides/dally-async2009.pdf.

[5] William James Dally. 2015. Unified Streaming Multiprocessor Memory. Patent No. US 9,069,664 B2, Filed Sep. 22nd., 2011, Issued June 30th., 2015.

[6] Fady Ghanim, Uzi Vishkin, and Rajeev Barua. 2018. Easy PRAM-based High-performance Parallel Programming with ICE. *IEEE Transactions on Parallel and Distributed Systems* 29 (Feb 2018), 377–390. Issue 2. https://doi.org/10.1109/TPDS.2017.2754376

[7] John L. Hennessy and David A. Patterson. 2017. *Computer Architecture, Sixth Edition: A Quantitative Approach* (6th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[8] Ying Hu, Oleg Kremnyov, and Ivan Kuzmin. 2018. Faster Gradient-Boosting Decision Trees. *The Parallel Universe* 33 (2018), 55–62. https://techdecoded.intel.io/resources/faster-gradient-boosting-decision-trees/

[9] Rory Mitchell and Eibe Frank. 2017. Accelerating the XGBoost algorithm using GPU computing. *PeerJ Computer Science* 3 (July 2017), e127. https://doi.org/10.7717/peerj-cs.127

[10] Biswanath Panda, Joshua S. Herbach, Sugato Basu, and Roberto J. Bayardo. 2009. PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce. *Proceedings of the VLDB Endowment* 2, 2 (Aug. 2009), 1426–1437. https://doi.org/10.14778/1687553.1687569

[11] Stephen Tyree, Kilan Q. Weinberger, and Kunal Agrawal. 2011. Parallel Boosted Regression Trees for Web Search Ranking. In *Proceedings of the 20th international conference on World wide web*. 387–396. https://doi.org/10.1145/1963405.1963461

[12] Uzi Vishkin. 2011. Using Simple Abstraction to Reinvent Computing for Parallelism. *Commun. ACM* 54, 1 (Jan 2011), 75–85. https://doi.org/10.1145/1866739.1866757

[13] Vitaly Zakharenko, Tor Aamodt, and Andreas Moshovos. 2013. Characterizing the Performance Benefits of Fused CPU/GPU Systems Using FusionSim. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. https://doi.org/10.7873/DATE.2013.148