Instruction Set Extensions for Photonic Synchronous Coalesced Accesses

Paul Keltcher, David Whelihan, Jeffrey Hughes
Massachusetts Institute of Technology Lincoln Laboratory
Lexington, MA.

***Abstract*— Microprocessors have evolved over the last forty-plus years from purely sequential single operation machines, to pipelined super-scalar, to threaded and SIMD, and finally to multi-core and massive multi-core/thread machines. Despite these advances, the conceptual model programmers use to program them is still that of a single threaded register file bound math unit that can only be loosely synchronized with other such processors. This lack of explicit synchrony, caused by limitations of metal interconnect, limits parallel efficiency. Recent advances in silicon photonic-enabled architectures [1, 5, 7] promise to greatly enable high synchrony over long distances (centimeters or more). In this paper, it is shown that global synchrony changes the way computers can be programmed by introducing a new class of ISA level instruction: the *globally-synchronous load-store*. In the context of multiple load-store machines, the globally synchronous load-store architecture allows the programmer to think about a collection of independent load-store machines as a single load-store machine. This operation is described, and its ISA implications explored in the context of the distributed matrix transpose, which exhibits a high degree of data non-locality, and is difficult to efficiently parallelize on modern architectures.**

Keywords—computer architectures; shared memory; photonics; instruction set; coalesced memory

I. INTRODUCTION

An Instruction Set Architecture (ISA) is the representation of an underlying computer architecture used by a programmer to realize application goals. The ISA exports the sum total of all capabilities of a computer to the programmer and embodies the way it is intended to be used in a collection of instructions accessible to a programmer. Despite the addition of performance-improving features such as super-scalar and caching, the programmer's mental model of a processor has changed little in 40 years. Therefore, the way computers are programmed has not changed significantly either. Unfortunately, the stalling of frequency scaling in the early part of the last decade, and the resultant shift from faster gates to more parallel gates has not resulted proportionally to increased performance in part because the sequential programming style useful in Instruction-Level Parallelism (ILP) exploiting super-scalar processors is at best ineffective in an explicitly parallel context, and at worst detrimental.

This work is sponsored by Defense Advanced Research Projects Agency (DARPA) under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government. Distribution Statement A. Approved for public release; distribution is unlimited.

As parallel shared memory computers have become larger, incorporating tens of independent super-scalar cores, latency in interconnection networks has become the dominant factor limiting performance. Computer architects have attempted to mitigate the effects of latency by [8]:

- Adding instructions to prefetch data or have more explicit control over the behavior of the memory sub-system.
- Increasing the number and proximity of caches to the execution core.
- Increasing threading, thereby hiding the effects of latency.

None of these remedies are particularly efficient from an energy perspective. All are the result of acceptance that electrical interconnect latency is an intractable problem. The ultimate effect imposed by this limitation is that processing hardware cannot be synchronized over long distances. This makes highly efficient parallel programming of many-core architectures difficult, and highly dependent upon architectural parameters.

When a fundamental limitation of underlying technology is reached, new technology must be introduced to move forward. In [5] the addition of silicon photonics to the architect's technological toolbox is shown to enable scalable parallel efficiency in dense processing loads with very low data locality. In that paper, the Synchronous Coalesced Access (SCA), which alleviates the effects of non-locality by reorganizing data in-flight in a photonic waveguide, is introduced. This capability is shown to greatly increase parallel efficiency by removing uncertainty within the interconnect and memory subsystem, permitting processors to operate in lock-step, with high efficiency.

This high degree of global synchrony permits a new paradigm in parallel programming: *the globally synchronous load/store*. The contribution of this paper is to present computer instructions that express global load-store behavior in a massively multi-core system in which individual processors do not need to understand the entire global load/store data access patter n. This capability and its role in the normal program flow is described in the context of a GPU-style architecture.

978-1-4799-1365-7/13/$31.00 ©2013 IEEE

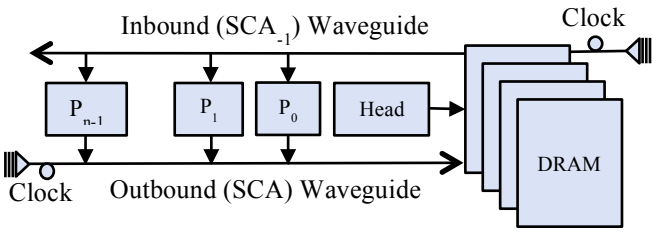

Fig. 1: P-Sync Architecture

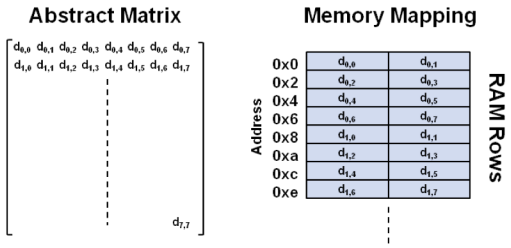

Fig. 2: Matrix Mapping

Section II briefly introduces the photonically-enabled P-sync architecture [5], and the SCA operation. Section III describes how multi-dimensional matrices are mapped to linear memory. In Section IV, new ISA extensions are defined, and their effects on the code stream for the challenging distributed matrix transpose operation are illustrated. Related work, specifically how existing parallel architectures work with distributed data, is discussed in section V, followed by conclusions in section VI.

II. PHOTONIC P-SYNC

In [5] a photonic network called PSCAN is described that utilizes chip-scale photonic networks for globally synchronous communication. Instead of using chip-scale photonics as a means of increased bandwidth density, photonics is utilized for global synchrony over long distances. This permits spatially separate processors to arbitrarily reorder data by synthesizing monolithic transactions in the photonic waveguide. Memory read/write operations are performed without any special buffering resulting in optimal use of channel and memory bandwidth and near 100% efficient use of computation.

The P-Sync architecture uses PSCANs, shown in Figure 1, where processors P_0 to P_{N-1} are "worker" processors and the "Head" node coordinates memory traffic. There is one PSCAN network to the shared memory for reads and one for writes. Like the GPU machine model, each worker processor is relatively simple and has a large instruction issue width. The differences with the GPU model is that P-Sync can coalesce an arbitrary amount of data from all processors to the global memory, whereas GPU coalescing across processors is potentially inefficient, unscheduled, and limited to small blocks of memory. The global DRAM is visible to all processors.

III. MATRIX DISTRIBUTION AND MAPS

Data reorganization occurs because multi-dimensional data structures must be processed along different abstract dimensions. Since computer memory is a linear resource, addresses increase sequentially, and there is an optimal way to access memory based on its internal structure. If a matrix is mapped to linear memory by sequencing each row in order, sequential operations on matrix elements are efficient. If, however, a processor needs to operate on columns of the matrix, the row ordered memory mapping is extremely inefficient.

Explicit whole structure data reorganizations such as transpose are commonly used to pre-stage data in memory in order to increase access efficiency over a set of operations defined by the application. Because PSCAN can synchronize all processors, it is possible for those processors to use a global map that defines exactly when each processor will write to the memory with the goal of synthesizing optimally structured memory accesses. Figure 2 shows an abstract matrix mapped to 1-D memory. The matrix is stored in row-major order, with each row sequentially stored contiguously in increasing memory locations. Because an entire memory line must be read whenever any element on that line is needed, accessing this matrix in row-order is maximally efficient. If, however, the columns must be operated on, only a part of each memory line is needed for any column. To get any two elements any matrix column together, two separate line reads would occur with only half of the retrieved data used.

IV. ISA EXTENSIONS FOR PHOTONICALLY INTERCONNECTED PROCESSORS

In this paper, a *map* defines a pattern with which multiple, spatially separate processors will write data to memory to realize an abstract structure. A map always specifies a contiguous access. If, in an MxN matrix, the number of rows M is equal to the number of processors P operating on the matrix, and each processor can hold all N elements of each row, a map for transpose is defined by these parameters:

base – The base address of the mapped write
P – The number of processors (or hardware threads) participating in the synthesized access
S – The number of bytes written by each processor during its portion of the map
B – The number of blocks of size *s*

The pseudo code that describes this mapping of local non-contiguous data held in processors to a global address space is shown here:

```
for i in [0 : B – 1]:
  for j in [ 0 : P – 1]:
    for k in [0 : S – 1]:
      processor[j].write (
          local_data[i*S + k],
          base + i*j + k)
```

To illustrate the use of new instructions in the context of matrix transpose, we adopt NVIDIA's PTX as an ISA foundation [6]. PTX is the intermediate assembly language used by NVIDIA processors. A PTX program is a thread in the data parallel SIMT (single instruction multiple thread) programming model and is agnostic of actual machine resources. Rather, the "thread" knows where it is relative to the data (vector, matrix, or 3D space). The run-time environment will parallelize the PTX application according to available resources. PTX allows the expression of parallelism without knowledge of available parallel resources. For this analysis the basic architecture is taken from Figure 1, where there is a coordinating processor ('head node') and worker processors. Two instructions are added, a "global" instruction that runs on the head node and a "local" instruction that runs on worker processors.

In this example, each thread will hold one row of the *NxM* matrix such that the number of hardware threads *P=M*. This analysis also assumes there is enough local memory to hold one row. In these new instructions, the head node initiates a coalescing read or write and the worker processors execute memory transactions relative to the control processor's initiation. To start each processor holds one row of the matrix in local memory and will write out the transpose. The following are the instructions for the *head node* processor:

```
  mov.u32  r, N
  mov.u32  rc, 0
    // set up a loop to go over the entire
    // matrix, one row at a time.

.loop:
  coalesce_sca      base_address, S, B
    // set up the coalescing write for all
    // worker processors to participate in.
    // Parameters are base address pointer
    // (transposed row destination), B (number
    // of blocks of size S) and S.  The
    // base_address is computed from the row
    // pointer, rc

  add.32  rc, rc, 1
  sub.32  r, r, 1
    // decrement loop count (r) and increment
    // row pointer (rc)

  bra  loop
    // continue until all rows have been
    // coalesced.
```

The code represents an SCA operation used to write a single matrix column back to memory, with N operations required to write the entire matrix back to memory in column major form. The coalesce_sca instruction is a blocking instruction that will not complete until all data is received from the worker processors. Whereas the coalesce_sca sets up a large contiguous block of memory, each worker processor writes to a different *n* byte space within that larger block. To transfer the data, the *worker processors* then execute the following code:

```
  mov.u32  r1, N         // loop count
  mov.u32  r2, [row]     // the size of a row
  mov.u32  r3, 0         // loop index

.loop:

    // compute local memory address based on
    // thread id, blocking (if blocked), r2,
    // and put in r5

  ld.local.u32  r4, local[r5];
    // read element out of local memory

  sca.b32       r4, r3;
    // participate in the global coalescing
    // store.  Each worker thread writes 4
    // bytes of data held in r4 into the
    // global SCA space at position r3.  This
    // is a blocking instruction and will not
    // proceed until this threads "time" comes
    // up on the photonic TDM waveguide.

  add.u32  r2, r2, 1;
  sub.u32  r1, r1, 1;
    // decrement loop count (r1) and
    // increment row pointer (r2)

  bra  loop;
    // continue until all 32bit elements have
    // been coalesced.
```

This assumes the multiple threads on one processor can coalesce their row reads from local memory, much the same as the GPU architecture coalesces row reads from multiple threads within a warp to the GPU global memory [10]. Unlike loads and stores in a CPU or GPU, the SCA instructions allow the hardware to order writes to memory across independent processors. Individually, the processors are writing transposed data with no spatial locality, but globally the memory write is perfectly contiguous within the sca_coalesce space. This globally synchronous load store architecture changes the way programmers can think about shared memory programming. What was, in the general purpose processor, a collection of independent load-store transactions, is instead presented as one single memory transaction in which individual processors order themselves relative to a global schedule.

V. Related Work

What separates this work from existing parallel architectures is the notion of combining memory traffic of multiple independent processors into a single efficient memory transaction. In this section we look at general purpose CPUs, GPUs, Cray XMT and Oracle Macro-Chip to understand how these architectures interface multiple independent load-store streams.

General purpose CPUs, in trying to be general purpose, take architectural innovations from other more special purpose processors. To deal with high latency memories modern general purpose machines manage latency by ISA additions such a prefetch, use caches and hardware prefetchers to reduce latency, amortize memory latency with vector instructions when possible and utilize limited hardware threading. While all of these innovations generally provide benefit to many single threaded applications, these constructs when applied to multi-core/multi-threaded applications with poor coordination between processors in a shared memory system can result in a performance penalty. The SCA instructions presented in the previous section, however, allow each processor to explicitly order themselves within a single larger transaction. This type of optimization is not possible with existing general purpose CPUs.

GPUs contain a large number of processing units which operate similar to SIMD units found in CPUs, are more special-purpose and excel when the application exhibits data parallelism. Prior to NVIDIA's Kepler processor[6] data sharing between threads within a warp required a store and load operation. Kepler added a Shuffle instruction that allows arbitrary permutations within a warp. Although this special instruction was added for communication between threads within a warp, GPUs have no synchronous, efficient global communication between warps. The programmer is required to explicitly move data between GPU devices via the CPU's memory, and like general purpose CPUs, multiple GPU processors cannot combine independent memory transactions from multiple processors into a single contiguous memory transaction. Still, if the data fits within the GPU (no off-chip interconnection requirements) and the application has sufficient parallelism (to amortize the on-chip interconnection limitations), performance can be quite high.

The Cray XMT [3] architecture does not fight the reality of high latencies within the interconnection network, but rather is architected to tolerate latency for parallel applications which exhibit a low degree of data locality. The XMT tolerates memory latency of large shared memory systems by using hundreds of threads per processor. While most threads are waiting for data, there should be at least one thread which can make forward progress. While this serves the machine well for sparse data like graph traversal [9] and sparse linear algebra [3], the XMT cannot optimize performance or efficiency for applications that have dense memory access patterns. Each thread in the XMT acts alone, so coordinating load-store instructions between threads into larger transactions for efficient memory usage is not possible.

The Sun/Oracle Macrochip [7] employs a point-to-point photonic interconnection between compute nodes on a single substrate. The goal of the macro-chip is to obtain performance like a traditional CMP, but with many more processors enabled by the high-bandwidth low-latency point-to-point interconnection between the cores on the virtual "chip". The compute nodes of the macro-chip are traditional general purpose processors, although any type of compute node could be imagined. To date there have been no proposed ISA additions to take advantage of photonics.

VI. Conclusions

This paper presents two new instructions, a global coalesce instruction and a local SCA instruction that allows programmers to express globally synchronous load-store communication across multiple processors. This globally synchronous load-store architecture permits programmers to take a collection of independent load-store processors and combine their transactions into a single monolithic memory transaction. This presents new programming possibilities for optimizing memory and network traffic, which are not possible with the existing single-threaded view load-store ISAs. Going forward, there are more instructions to investigate to further exploit the shared globally synchronous P-Sync architecture.

References

[1] N. Bliss, K. Asonovic, K. Bergman, L. Carloni, J. Kepner, and V. Stojanovic, "Photonic Many-Core Architecture Study," in *Proceedings of the Twelfth Annual Workshop on High Performance Embedded Computing (HPEC)* 2008.

[2] Shekhar Borkar, Pradeep Dubey, Kevin Kahn, David Kuck, Hans Mulder, Stephen Pawlowski, Justin Rattner, *"Platform 2015: Intel Processor and Platform Evolution for the Next Decade",* http://epic.hpi.uni-potsdam.de/pub/Home/TrendsAndConceptsII2010/HW_Trends_borkar_2015.pdf*, Page 6.*

[3] John Feo, David Harper, Simon Kahan, Petr Konecny, "Eldorado," in *Proceedings of the 2nd conference on Computing Frontiers, ACM,* May 2005.

[4] http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html

[5] David Whelihan, Michelle Beard, Jeffrey Hughes, Anna Klein, Sanjeev Mohindra, Julie Mullen, Eric Robinson, Scott Sawyer, Michael Wolf, Nadya Bliss, Jonnie Chan, Robert Hendry, Keren Bergman, Luca Carloni, "P-sync: A Photonically Enabled Architecture for Efficient Non-local Data Access," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2013

[6] http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[7] Pranay Koka, Michael McCracken, Herb Schwetman, Xuezhe Zheng, Ron Ho, Ashok Krishnamoorthy, "Silicon-photonic Network Architectures for Scalable, Power-efficient Multi-chip Systems," in *Proceedings of the 37th Annual international Symposium on Computer Architecture*, 2010.

[8] David Mizell, Cray Inc., "Introduction to the Cray XMT", wwwjp.cray.com/downloads/XMT-Presentation.pdf, 2010

[9] Georage Chin, Andres Marquez, Sutanay Choudhury, "Implementing and Evaluating Multithreaded Triad Census Algorithms on the Cray XMT," in *IEEE International Symposium on Parallel and Distributed Processings*, 2009

[10] Greg Ruetsch, Paulius Micikevicius, "Optimizing Matrix Transpose in CUDA", NVIDIA 2009