Multithreading vs. Streaming

David Tarjan*
Univ. of Virginia, Dept. of Computer Science
Charlottesville, VA 22904
dtarjan@cs.virginia.edu

Kevin Skadron*
Univ. of Virginia, Dept. of Computer Science
Charlottesville, VA 22904
skadron@cs.virginia.edu

Categories and Subject Descriptors
C.1.4 [**Parallel Architectures**]

General Terms
Performance, Design

Keywords
Multithreading, memory staging, multicore, manycore, programming, streaming

1. INTRODUCTION
The recent, industry-wide shift to multicore chips has been driven by a combination of the ILP and power walls. Individual cores struggle to improve single-thread throughput without unreasonable power dissipation. The natural solution is to integrate more cores in each generation. However, memory-bandwidth requirements scale up accordingly and relative DRAM latency (in clock cycles) continues to rise.

These trends require new architectures that maintain memory performance by maximizing efficient use of bandwidth and overlapping computation with memory access. Software-managed memory hierarchies and streaming have been widely recognized as one way to accomplish this (e.g. [3]), but there is less recognition that hardware techniques can be equally effective. In particular, deep multithreading with hardware stream aggregation and sorting (as used in graphics processors for example) accomplishes this. These two approaches represent endpoints of a spectrum of architectures for bandwidth and latency management. The problem with these endpoints is that they are best suited to specific categories of applications. Future manycore architectures must flexibly support both paradigms.

This paper briefly outlines the pros and cons of each approach and then advocates unification in the form of a multithreaded organization with fine-grained control over each processing element's local store.

2. Streaming
Streaming as a memory performance paradigm is often confused with streams (typically FIFOs with optional peek) as a programming language construct. The latter is useful for expressing data parallelism. Streaming as a memory performance paradigm (which we will henceforth call *memory staging*) is quite different. Although streams are a nice abstraction for programming memory-staging, they are not strictly required. Memory staging explicitly breaks programs into gather and compute phases (with an optional scatter phase). The gather phase generates a list of memory addresses to be read and stores the list in on-chip memory. This list may require an additional sorting stage to optimize DRAM locality, or this may be done directly by hardware. Then, while this list is sent to the memory controller, other computation uses the processing elements (PEs). When the gathered data is ready, its compute phase begins. Output may require another sorting stage for DRAM locality, or may be chained to a subsequent kernel. The memory staging paradigm is most effective with a scratchpad because this ensures that gathered data cannot accidentally be evicted due to cache conflicts. On the other hand, memory staging requires at least double buffering (one region for data actively worked on by computation, at least one region for memory transfer). With a scratchpad to stage data, memory staging does not need more than one thread context per PE, because data access latencies are deterministic. The addressing logic to access the scratchpad can also be much simpler, requiring none of the address translation and permission checking which is needed when accessing the address space of normal memory. Memory staging works well with a variety of architectures for the multiple processing elements, and the deterministic data access latencies makes SIMD execution especially efficient when data parallelism is present.

Memory staging works well when data access patterns are regular and the live state associated with outstanding memory references is small. Memory staging loses efficiency when data access patters are irregular. This either causes the gather kernel to load redundant data, or requires a cache implemented in software, with the associated software overhead [3]. Furthermore, when the amount of computation to be done on a block of data is not fixed (due to data dependent branching and looping for example), the programmer or compiler have a hard time trying to find the ideal block size to balance out memory transfer latency with the computation latency of any block. This can lead to underuse of the staging memories or stalling of the computation. A further concern with staging paradigms is that either the programmer or the compiler *must* manage the staging. This may limit purely hardware based on the staging paradigm to application domains with sufficiently regular data access patterns and large enough volumes to justify specialized hardware.

Cell BE [5] is perhaps the most visible commercial product today following the memory staging paradigm. In Cell BE, the staging area is the per-SPE local store, with the L2 used for inter-core communication and capturing temporal reuse.

3. Multithreading
A long-standing hardware technique to overlap computation with latency is to support multiple thread contexts per PE, with per-PE

*This work was conducted while the authors were on sabbatical/internship with NVIDIA Research.

MSPC'08, March 2, 2008, Seattle, WA, USA.

storage consisting of all threads' register context plus any cache shared among the threads. When one thread stalls on a memory reference, another thread runs. Multithreading also allows simpler pipelines, since pipeline bubbles—for example, due to control hazards, for example, or multi-cycle access to the register file—can simply be covered by switching to another thread. The chip may consist of many such multithreaded PEs. When sufficient parallelism is present, caches are not needed to reduce average latency. Instead, caches are used to multiply bandwidth. By exploiting temporal and spatial locality, caches reduce demand for precious pin bandwidth. Multithreaded hardware can still sort and batch reference streams for DRAM locality using hardware tables; the multithreading covers the additional latency due to the batching.

The Tera MTA [2] was an early example of this extreme approach, with 128 threads per PE and no caches whatsoever. Scientific workloads often exhibit sufficient parallelism to occupy 128 threads. Graphics processors are another example, with a large number of threads per PE (24 in the GeForce™ 8800). Unlike the MTA, GPUs do use caches, but as bandwidth multipliers, not to reduce average latency that must be covered, and—in the case of the NVIDIA CUDA™ architecture and the GeForce 8800's parallel data cache—for memory staging. The Sun Niagara architecture is a third interesting example. Here the degree of multithreading is only 4-8 [1, 4], presumably based on expected workloads with high degrees of temporal locality in the secondary cache and hence low average latency.

Multithreading with caches works well in exactly the cases mentioned above where memory staging loses efficiency. Multithreading is also efficient when the live state associated with each data element is large, and when applications exhibit task instead of data parallelism; and the multithreaded paradigm may be a more natural target for porting legacy applications. Multithreading suffers from some inefficiencies, however: large register files (with one context per thread), live state in the register file and local store sitting idle while waiting for data from memory; and the risk that poor scheduling among threads may fail to capture available locality. Another concern for multithreading is the possibility that more and more threads per PE will be needed to cover growing memory latencies (although this is less of a concern now that CPU frequencies are rising more slowly).

4. Toward a Common Platform

Some applications realize better memory performance with memory staging, others with multithreading [6]. Indeed some applications may go through phases where one or the other paradigm is preferred. Yet the need for economies of scale and standardization argues for a common hardware platform to support both memory staging and multithreading.

Multithreaded hardware with local storage, such as GPUs, can fairly easily be repurposed for memory staging as needed. Instead of running a single kernel at a time with integrated gather-compute, these two phases can be separated and staged with software pipelining. Gather and compute kernels run concurrently as separate threads on the same PE, with one set of threads gathering while another computes with previously gathered data. The programming model can by default assume multithreading with integrated gather-compute (e.g., NVIDIA's CUDA programming environment), but allow programmers the option to specify separate gather and compute kernels. The compiler or runtime would ideally spare the programmer from the need to select the optimal stagger of the gather and compute kernels needed for optimal staging.

Availability of a local store with scratchpad semantics (e.g., the GeForce 8800 parallel data cache) helps, but caches with sufficient associativity can realize many of the same benefits. The chief concern is for the local store to have enough capacity for the requisite double buffering.

Starting from the opposite approach, subdividing the large local register file (LRF) of a core which implements memory staging would make it possible to have multiple thread contexts, each with their own private set of registers.

Adding extra storage and logic to the scratchpad memory so that it could act as a cache would be more expensive, but would be a worthwhile investment for many workloads.

Work by Erez et al. [3] shows that many memory staging applications end up replicating caches in software, where the caching scheme is particular to the application. Multithreaded applications that cannot be blocked into the cache may in turn lose temporal locality if interleaving of thread execution is non-deterministic. Giving both approaches new primitives to better emulate or control caches (for example switching local stores from caching to scratchpad semantics, or giving fine-grained software control over replacement policies in caches) will help provide a single platform that achieves the best possible memory performance.

Determining the exact semantics of when and how to switch between modes is an open question, as is if being able to use both modes simultaneously would be advantageous.

Continued research is needed to understand how to size the various resources appropriately, and how best to provide appropriate mechanisms for control over on-chip storage.

5. ACKNOWLEDGMENTS

We would like to thank Mattan Erez and Bill Dally for interesting discussions on the topic of memory staging and Doug Voorhies for feedback which improved the quality of the paper. We also thank the anonymous reviewers for their helpful comments.

6. REFERENCES

[1] K. Aingaran, P. Kongetira and K. Olukotun. Niagara: a 32-way Multithreaded Sparc Processor. *IEEE Micro*, 25(2), Mar.-Apr. 2005.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. *Proc. ICS,* 1990.

[3] M. Erez, J. H. Ahn, J. Gummaraju, M. Rosenblum, and W. J. Dally. Executing Irregular Scientific Applications on Stream Architectures. *Proc. ICS,* 2007.

[4] T. Johnson and U. Nawathe. An 8-core, 64-thread, 64-bit Power Efficient Sparc SoC. *Proc. ISSCC,* 2007.

[5] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, Introduction to the Cell Multiprocessor. *IBM J. Res. & Dev.* 49(4/5), 2005.

[6] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis. Comparing Memory Systems for Chip Multiprocessors. *Proc. ISCA*, 2007