The Case for Malleable Stream Architectures

Christopher Batten,[1] Hidetaka Aoki,[2] Krste Asanović [3]

[1] Massachusetts Institute of Technology, Cambridge, MA, USA – cbatten@mit.edu
[2] Hitachi, Ltd., Tokyo, Japan – hidetaka.aoki.rt@hitachi.com
[3] University of California, Berkeley, CA, USA – krste@eecs.berkeley.edu

Stream workloads vary widely, as do proposed stream architectures. We argue that stream processors should prioritize efficient temporal and spatial data-parallel execution, while not ignoring support for temporal and spatial kernel-parallel execution. We introduce a new malleable stream architecture with data- and kernel-parallel mechanisms that can be reconfigured as needed by stream applications.

1. The Landscape of Stream Workloads

Many applications can be expressed as *streams* of elements flowing between computational *kernels*. Although stream programs are more structured than general-purpose programs, they include surprisingly diverse forms of parallelism and communication patterns. For example, consider the toy application in Figure 1. Kernels A–C exhibit *data-level parallelism* (DLP) meaning the kernel execution for one element is independent with respect to other elements. Note that DLP kernels can still contain data-dependent control flow which can significantly complicate implementations. Kernel D is not data parallel due to the feedback loop. This stream graph also exhibits two forms of *kernel-level parallelism* (KLP). Kernels A and B are *task parallel* meaning they are on parallel branches of the stream graph. Although kernel D depends on C, they are still *pipeline parallel* since the kernels can execute in parallel on successive elements.

Stream applications also vary in their communication patterns. For example, although each stage of an FFT application is completely parallel it includes a more complicated butterfly communication pattern. Moving elements between a centralized stream buffer and copies of a DLP kernel can require global communication. Stream peeking, where a DLP kernel references elements past the head of a stream, can create complicated communication patterns between copies of the DLP kernel. A study by Gordon *et al.* showed significant variation in DLP and KLP (both task and pipeline parallelism) as well as communication patterns (both in the static graph and through stream peeking) across 12 stream applications [1].

2. The Landscape of Stream Architectures

A given stream workload can be executed in many different ways, as shown in Figure 2. The temporal DLP (T-DLP) approach executes the same kernel on a single processing element (PE) in sequence (Fig. 2a). T-DLP amortizes control and synchronization overhead over many executions of the same kernel. The temporal KLP (T-KLP) approach executes different kernels on a single PE in sequence (Fig. 2b). T-KLP can exploit producer-consumer locality by blocking streams in software-managed local memories [2] or in standard caches [4]. The spatial DLP (S-DLP) approach executes the same kernel on parallel PEs at the same time (Fig. 2c-d). As with T-DLP, this approach allows for significant amortization of control overhead. The spatial KLP (S-KLP) approach executes different kernels on parallel PEs (Fig. 2e-f). For pipeline-parallel kernels in an S-KLP approach we can increase efficiency by using an explicit PE-to-PE network [5].

Streaming workloads often have such large quantities of parallelism that performance is primarily limited either by on-chip energy consumption or by off-chip memory bandwidth. *Our position is that programmers and architects should first leverage energy-efficient DLP execution whenever possible.* DLP execution mechanisms, especially traditional pipelined vector instructions (T-DLP), provide the lowest energy per operation in control and datapath structures, and can move and synchronize data streams in large blocks. DLP (without data-dependent control flow) is also trivially load balanced as opposed to the problematic load balancing in KLP approaches.

In spite of DLP's advantages, KLP is still valuable. A pure DLP approach will have poor utilization for stateful kernels. Even in the absence of stateful kernels, a pure DLP approach may require so much buffering or off-chip memory traffic

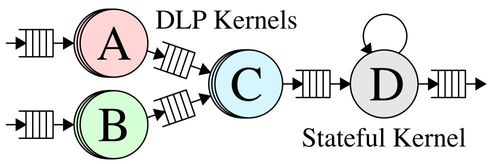

Figure 1: Example Stream Application

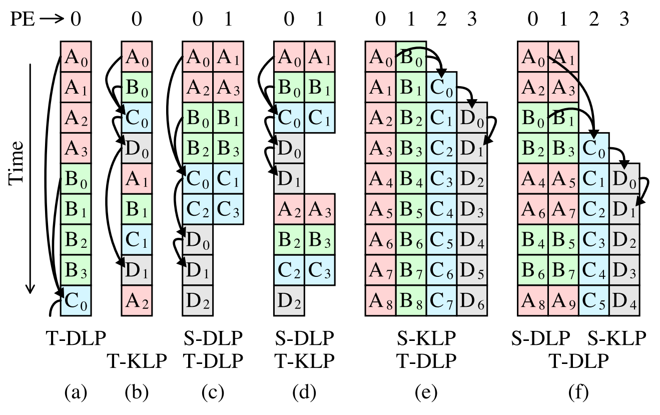

Figure 2: Execution Approaches – A_i = execution of A for stream element *i*. PE = processing element. T-DLP, T-KLP, S-DLP, S-KLP = temporal/spatial, data-level/kernel-level parallel execution.

	Number of Cores	S-DLP Mechanisms	T-DLP Mechanisms	S-KLP Mechanisms	T-KLP Mechanisms	Comments
Multi-Core x86	<10	1×128b *	n/a	Coherent Cache	Cache	Assuming 128 b SSE
STI Cell BE	1+8	1×128b *	n/a	Inter-core DMA	Local Store	Inter-core DMA on ring network
Tilera TILE64	64	1× 32b *	n/a	Static Mesh NoC	Cache	Hardly any DLP mechanisms
SPI Storm-1	1	16× 32b *	n/a	n/a	Local Store	Vector lanes *and* subword SIMD
NVIDIA GT200	30	8× 32b	4	n/a	DRAM	Cores must execute same kernel
Intel Larrabee	>10	16× 32b	n/a	Coherent Cache	Cache+Stream	SW control of caches
MIT Scale	1	4× 32b	1-32	n/a	Cache	Configurable vector length
Maven	≈100/n	n× 64b	1-32	Coherent Cache	Cache+Stream	Malleable stream architecture

Table 1: Comparison of Various Stream Architectures – A *core* is a control processor with vector lanes and/or subword SIMD units (denoted by *) for *spatial DLP amortization*. *Temporal DLP amortization* indicates how many elements per lane (one element per cycle) can be controlled with a single SIMD instruction. *Spatial KLP mechanisms* exploit producer-consumer locality between kernels running on different cores while *temporal KLP mechanisms* exploit this locality by blocking streams between kernels running on the same core.

that we cannot fully utilize all of the PEs. The large buffers may also detrimentally increase the per-element latency in real-time applications. *Thus, our position is that programmers and architects must still be able to exploit KLP, but after DLP.* A related concern is that DLP kernels with significant data-dependent control-flow may not map well to SIMD execution resources, and so sufficient MIMD execution resources must also be provided.

Table 1 lists the mechanisms used to exploit DLP and KLP in several modern processors. Most processors either take a DLP-focused approach (Storm-1, GT200), or a KLP-focused approach (Cell, TILE64). Although it is possible to exploit DLP on a TILE64 processor it is difficult to do so *efficiently* since there are hardly any DLP execution mechanisms. On the other side of the spectrum, a GT200 can only exploit DLP and cannot support a S-KLP approach. The GT200 supports T-KLP by wastefully buffering streams in DRAM. Notice that most DLP mechanisms provide only S-DLP; they do not allow one vector/SIMD instruction to control many cycles worth of execution even though this is easier to implement than S-DLP and more energy efficient.

3. Maven: A Malleable Stream Architecture

We are currently developing the Maven processor, a malleable array of vector-thread engines suitable for both stream workloads and general-purpose programs (see Figure 3). Each Maven core includes a small control processor (CP), an L1 cache, and a *vector-thread* lane. Vector-threading (VT) is a new architectural technique which attempts to efficiently intermingle vector and threaded execution on the same hardware resources [3]. Maven VT lanes are much simpler than earlier VT implementations, and they better amortize issue logic and dependency checking overhead across vectors to enable a very efficient T-DLP approach. Maven VT lanes also support conditional control flow within DLP kernels while maintaining vector amortization where possible.

The Maven CPs are interconnected through a dedicated control network which enables one CP to send vector commands to a neighboring core's VT lane, providing an efficient S-DLP mechanism. Essentially, Maven can be viewed as a "sea-of-lanes", with 10-100's of individual vector lanes that

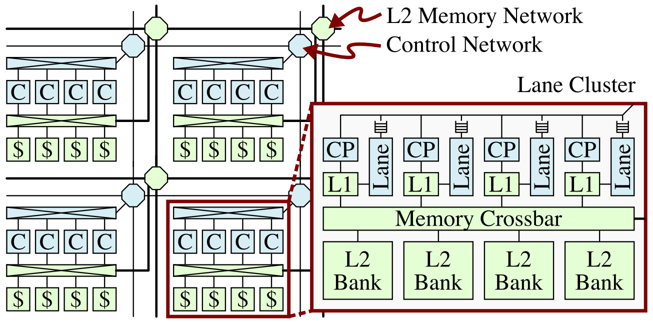

Figure 3: The Maven Processor – Only a portion of the sea-of-lanes is shown (C = Core, $ = L2 Bank, CP = Control Processor)

can be *ganged* together to form *vector-thread units* (VTUs) of various sizes. A Maven processor can be configured as many small VTUs, a few large VTUs, or combination of large and small VTUs. This flexibility enables the Maven processor to efficiently implement both DLP and KLP execution approaches.

Maven's L1 caches include stream mechanisms to exploit producer-consumer locality, and they are interconnected with the L2 cache banks through an on-chip mesh network. The L2 banks form a large shareable on-chip storage to support T-KLP execution.

We believe this emphasis on a prioritized set of DLP-before-KLP mechanisms will provide a flexible yet efficient substrate for future parallel workloads.

References

[1] M. Gordon et al. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *Architectural Support for Programming Languages and Operating Systems*, Oct 2006.

[2] B. Khailany et al. A programmable 512 GOPS stream processor for signal, image, and video processing. *Int'l Solid State Circuits Conference*, Feb 2007.

[3] R. Krashinsky et al. The vector-thread architecture. *Int'l Symp. on Computer Architecture*, Jun 2004.

[4] J. Leverich et al. Comparing memory systems for chip multiprocessors. *Int'l Symp. on Computer Architecture*, Jun 2007.

[5] D. Wentzlaff et al. On-chip inteconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, Sep 2007.