EVX: Vector Execution on Low Power EDGE Cores

Milovan Duric*, Oscar Palomar*‡, Aaron Smith†, Osman Unsal*, Adrian Cristal*‡§, Mateo Valero*‡, Doug Burger†
* Barcelona Supercomputing Center {first}.{last}@bsc.es † Microsoft Research {aaron.smith,dburger}@microsoft.com ‡ UPC § CSIC-IIIA

***Abstract*—In this paper, we present a vector execution model that provides the advantages of vector processors on low power, general purpose cores, with limited additional hardware. While accelerating data-level parallel (DLP) workloads, the vector model increases the efficiency and hardware resources utilization. We use a modest dual issue core based on an Explicit Data Graph Execution (EDGE) architecture to implement our approach, called EVX. Unlike most DLP accelerators which utilize additional hardware and increase the complexity of low power processors, EVX leverages the available resources of EDGE cores, and with minimal costs allows for specialization of the resources. EVX adds a control logic that increases the core area by 2.1%. We show that EVX yields an average speedup of 3x compared to a scalar baseline and outperforms multimedia SIMD extensions.**

I. Introduction

High performance has joined power-efficiency as a primary objective in mobile computing. Emerging applications such as media applications and physical simulations used in 3D games are compute intensive data level parallel (DLP) workloads that are increasingly utilized in mobile platforms. Achieving power efficient performance on these workloads is crucial for future mobile designs. Various techniques have been used to improve power efficiency such as clock gating, dynamic voltage and frequency scaling. Recent research explores yet another alternative in Explicit Data Graph Execution (EDGE) architectures, that break programs into blocks of dataflow instructions that execute atomically thus providing power efficient out-of-order execution (see [5], [6], [15] for details). Motivated by promising research results, in this paper we propose support for DLP acceleration in low power EDGE cores.

Programmable graphics processing units (GPU) [14], multimedia SIMD extensions and vector processors [4], [9], [17], all exploit DLP at various levels of performance and power efficiency. While GPUs can provide an order of magnitude increase in performance for highly parallel applications, their area overheads can be high for mobile designs. SIMD extensions are an inexpensive way to exploit DLP in low power processors, but they have overheads due to packing and unpacking operands that can exceed their benefit. The vector length is encoded in the ISA requiring new hardware, instructions and/or recompilation across design iterations for the same processor product family. Vector processors in the other extreme operate over dynamically large vectors [7], [8] and use sophisticated memory units to increase performance efficiency. Still, their design requires significant area costs making them historically utilized mostly in supercomputers [12], [17].

This paper contributes EVX, a vector execution model, that efficiently exploits DLP on general purpose EDGE hardware with minor changes to existing hardware. EVX leverages the dataflow execution model of EDGE architectures for efficient out-of-order vector execution. It provides: register- and streaming-based vector execution, sophisticated addressing modes and configurable vector length. EVX provides these advantages by specializing general purpose hardware and thus allows an efficient execution of parallel and non-parallel applications. This paper provides details about performance, power and area evaluation of EVX. EVX increases power consumption by 47.5%, and by achieving an average speedup of 3x on DLP workloads, EVX increases power efficiency 2.16x.

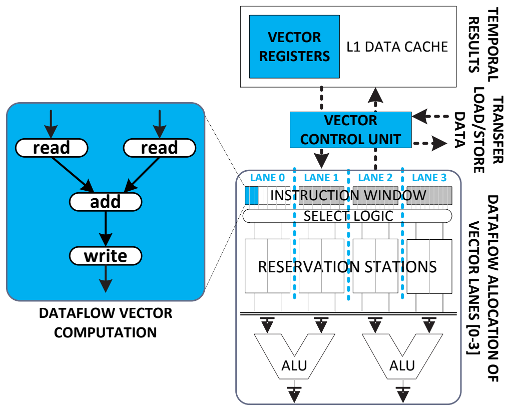


Fig. 1: EVX model on general purpose EDGE core.

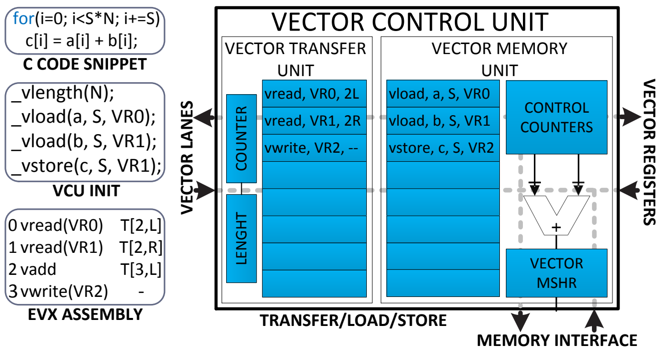
II. EVX: EDGE Vector Execution Overview

A. *Overview*

The goal of EVX is to provide the advantages of conventional vector architectures on the available hardware of lightweight processor. Figure 1 shows the EVX architectural model and hardware additions on an EDGE core. The EVX changes include: 1) Instruction window and reservation stations are banked, where each of these banks behaves as a vector lane [1]. The EVX computation executes in all vector lanes while providing higher throughput of DLP applications with little complexity. 2) An additional vector control unit (VCU) decouples the execution of vector memory instructions from computation to tolerate memory latency and provides sophisticated addressing modes. The vector memory operands are divided into slices. By transferring the available slices to the vector lanes, the VCU manages the reissue of EVX computation over different slices, without waiting for the entire memory operands. This way, EVX provides chaining of vector instructions at low hardware complexity, and while by reissuing computation it provides configurable vector length. And 3) Vector registers (VRs) repurposes some data cache resources to keep the temporal results of vector computation and hold memory operands.

EVX provides two execution modes to the programmer: register and streaming. In register mode, the VCU loads/stores data arrays to/from compute resources and exploits temporal locality through the VRs. In streaming mode, the VCU loads/stores data streams through VRs which behave as streaming buffers between the memory

This work has been partially funded by the Spanish Government (TIN2012-34557) and Microsoft Research.



```
for(i=0; i<S*N; i+=S)
  c[i] = a[i] + b[i];
```

```
_vlength(N);
_vload(a, S, VR0);
_vload(b, S, VR1);
_vstore(c, S, VR1);
```

```
0 vread(VR0)    T[2,L]
1 vread(VR1)    T[2,R]
2 vadd          T[3,L]
3 vwrite(VR2)   -
```

Fig. 2: EVX example and initialized vector control unit.

hierarchy and compute logic. EVX includes support for sequential, strided, 2-D strided, indexed addressing modes and reductions.

B. Vector Registers

Rather than introducing additional hardware structures, our implementation of VRs reuse a configurable subblock of the L1 data cache instead of a large vector register file. VRs have simplified accesses to data arrays that do not require any associative address lookup. The accesses are direct as in software managed memories [13]. Each VR allocates a configured number of cache lines, which are invalidated and flushed if needed before starting the vector execution. The cache reconfiguration is done sequentially, one cache line per cycle. When the VCU is configured for register mode, the VRs are used to store temporary results for EVX processing that spans multiple EDGE blocks. The vector length is limited to the maximum size of the available memory for VRs. However, when the VMU is configured for streaming mode, VRs behave as a circular buffer which allows for streaming an unlimited number of vector elements.

C. Vector Control Unit

The VCU contains the two major structures shown in Figure 2. First, the *Vector Memory Unit* (VMU) executes vector load and store instructions similarly to memory units in vector processors [2], [3], [16], [18]. The major difference is execution of memory instructions in slices to provide chaining and configurable vector length on general purpose hardware. Second, the *Vector Transfer Unit* (VTU) eliminates the additional control overhead and improves the EVX model, by transferring the data between the vector lanes and VRs.

1) Vector Memory Unit: Data parallel workloads often have regular memory access patterns that can be controlled using simple hardware support. The VMU exploits this advantage and leverages VRs to execute vector load and store instructions in a decoupled fashion. Each vector memory instruction uses one VR that holds the input/output memory operand. The VMU contains a queue of vector memory instructions that is initialized with access patterns. The 256-bit cache line size is used to easily slice large vector operands in VRs. The VMU issues the memory requests for a entire slice of a single memory instruction and then it executes the next instruction in round-robin fashion. Control counters keep track of vector elements processed by each instruction. The VMU uses vector miss-handler registers (VMSHR) to save the necessary information for each request and handle responses. The VMSHRs are simplified since addresses from different vector memory instructions are not compared. This allows for a higher number of outstanding requests and more efficient saturation of memory bandwidth.

2) Vector Transfer Unit: The VTU contains a queue of vector transfer instructions where each instruction is defined by the instruction type, target (in the case of a *vread*) and the associated VR. On a read, the VTU checks if the current slice is available and distributes it from the associated VR to vector lanes. On a write, the VTU forwards a slice of write request to the associated VR. Once all vector transfer instructions in the queue have issued, the EVX computation is complete for one slice of vector data. The computation reissues until the entire vector is processed.

D. EVX Mode

EVX mode is enabled on a block-by-block basis. Each EDGE block contains a header that encodes metadata about the block, including if block has vector instructions. Figure 2 shows an example of loop execution with EVX. Vector memory instructions (*vload/vstore*) initialize memory processing in VCU before the vector block executes. Vector block contains vector transfer (*vread/vwrite*) and compute (*vadd*) instructions. Vector transfer instructions execute in VCU and compute in disposable vector lanes. The multilane issue of vector compute instructions limits vector instruction blocks to the size of the instruction window in one lane. In this work we chose to have four vector lanes to issue a computation of 256-bit vector slices with 64-bit ALUs. We next discuss vector mode execution in the core pipeline.

Fetch: When a block with vector instructions is fetched, all instructions are mapped to the first lane of the instruction window. Scalar instructions issue only in first lane and vector instructions issue in all vector lanes. After the fetch stage completes, the list of vector transfer instructions is forwarded to the VCU. *Vector read* instructions only reference VRs and after forwarding to the VCU they are subsequently ignored by the issue logic. *Vector write* instructions leverage the issue logic, since they need to forward data to the VCU.

Execute: Vector instructions execute out-of-order in EDGE dataflow fashion. Vector compute instructions execute in four vector lanes, but the execution is interleaved if the number of ALUs is smaller than the number of lanes. The VCU executes vector memory instructions and transfers slices of vector data. Select logic reissues vector compute instructions when new data slices are available.

Commit: Since vector compute instructions may reexecute, and the number of iterations is controlled by the VCU, the VCU signals the control logic once all iterations are complete.

III. Evaluation

A. Methodology

We evaluate the baseline EDGE and EVX architectures using a detailed timing accurate simulator that models EDGE cores with the parameters shown in Table I. We developed McPAT [10] models starting from existing in-order models to estimate area, and runtime dynamic and leakage power in a 32nm technology used for low power devices. The simulator also models a multimedia SIMD extension for EDGE to compare to EVX. We also compare EVX to ARM NEON extension using simulations from Gem5 that is configured to match the parameters equivalent to EDGE core.

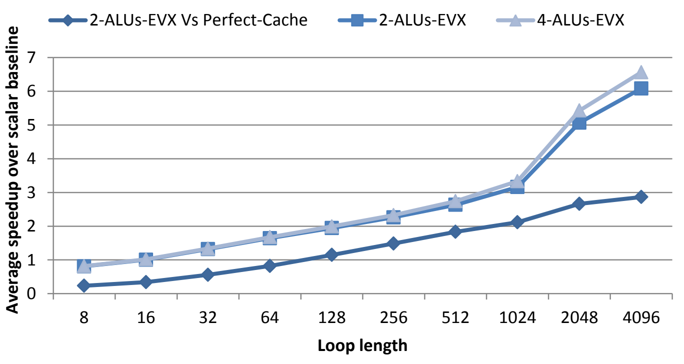
For our evaluation we select ten kernels from the Livermore Loops [11] shown in Table II. Each Livermore loop kernel utilizes a different memory access pattern and data parallel algorithm that we found suitable to explore the limits of our vector model. We hand-vectorized all the workloads using compiler intrinsics.

Component	Description
ALUs	2 Integer/FP
Register File	64 entries
Load-Store Queue	32 entries, unordered LSQ
L1 I-cache	32 kB, 3 cycles (hit)
L1 D-cache	32 kB, 3 cycles (hit)
L2	4 banks x 512 KB, 15 cycles (hit)
L1/L2 MSHRs	8 entries
DRAM	250 cycles
Branch predictor	OGEHL
On-chip-network	1 cycle/hop, Manhattan routing distance
Vector Registers	8 x 8192-bit
Vector-MSHR	64 entries

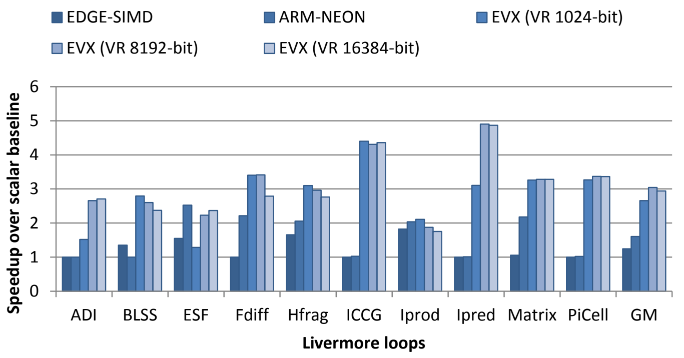
TABLE I: Simulator Configuration.

Name	Pattern	Dependencies	Vector Blocks	Stream
ADI	sequential/ strided	none	8	No
BLSS	sequential/ strided	reduction	1	Yes
ESF	sequential	none	3	No
FDiff	sequential	none	1	Yes
HFrag	sequential	none	1	Yes
ICCG	strided	none	1	Yes
IProd	sequential	reduction	1	Yes
IPred	strided	none	3	No
Matrix	strided	loop carried	1	Yes
PiCell	sequential/ indexed	none	5	Yes

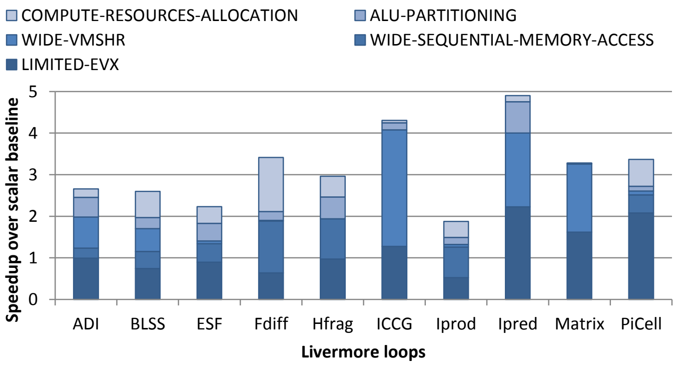
TABLE II: Workloads.



(a) Average EVX speedup for different loop lengths.



(b) EVX speedup for different VR sizes (loop length=1024).



(c) EVX performance breakdown (loop length=1024).

Fig. 3: EVX speedups and performance breakdown.

B. Results

Speedup for EVX and EDGE SIMD is reported against a scalar version running on the EDGE core. Speedup for ARM NEON is reported against the scalar version running on the same ARM core. In order to vectorize some kernels for EVX it is necessary to use multiple EVX compute blocks. The number of iterations in each kernel is configurable and referred to as loop length in the remainder of the evaluation. All kernels are evaluated after warming up the caches.

1) Performance: Figure 3a shows the speedup for a variable number of loop lengths ranging from 8 to 2048 elements. We evaluated EVX with two ALUs and with four ALUs. First, we discuss results with two ALUs. For short loop lengths (8 to 16 elements), there are no performance gains over the scalar version due to the initialization time required to setup the VCU and reconfigure the data cache. Initialization overhead is amortized around 16 elements and then vectorization provides a speedup that increases with loop length. This is explained by the reduction of repeated fetch, decode and issue of instructions in the vectorized loop, as well as the decoupling of memory and computation. For a loop length greater than 1024, the data sets become larger than L1 data cache size and EVX speedup increases from tolerating data cache misses. To measure the memory latency tolerance, we compare EVX to the scalar baseline with perfect data cache. The speedup of EVX with two ALUs is reduced when the scalar baseline uses a perfect data cache. Due to the improved performance of the baseline with a perfect cache, EVX requires more iterations to compensate its initialization overhead and it starts to achieve speedup with loop lengths over 64 elements. The VCU loads data more efficiently for long vectors, by packing and aligning vector elements in VRs, further keeping the ALUs busy in vector mode. Increasing the number of ALUs to 4, while not doing in line improvements of VCU does not provide performance benefits. This shows that VCU is balanced to efficiently feed two available ALUs.

Figure 3b shows the speedup for individual loops using a loop length of 1024 elements with different VR sizes and comparison to multimedia SIMD extensions. The EDGE SIMD extension operates on 64-bit words and uses two 64-bit SIMD units, the same as EVX. It supports unaligned accesses but it lacks sophisticated addressing modes which are needed to vectorize some kernels. However, the EDGE SIMD extension still increases the performance for kernels that use a sequential memory access (*ESF, Hfrag, Iprod*). In order to increase the applicability of SIMD, we have packed strided data into consecutive memory locations with scalar code. This code is amortized because the kernel iterates multiple times with the same data. However, the speedup is still limited (*BLSS*). Although a warm cache is advantageous for EDGE SIMD extensions, flexible

Structures	Baseline		EVX	
	Area	Power	Area	Power
Fetch (I-Cache, Block Predictor)	0.258	0.020	0.258	0.006
L1 D-cache subsystem (D-cache, LSQ, MSHR)	0.237	0.016	0.237	0.018
Execution resources (issue window, ALUs)	1.589	0.151	1.589	0.191
Vector Unit	-	-	0.045	0.009
Total	*2.093*	*0.187*	*2.138*	*0.224*
L2s, Memory Controller	2.079	0.055	2.079	0.133

TABLE III: Area (mm^2) and Power (W) estimates for EVX 1-core.

and efficient memory processing in EVX still provides better overall performance. ARM's NEON SIMD extension provides a more complete ISA, 128-bit wide SIMD words, as well as fused operations and performs better than EDGE SIMD. Still, it shows less applicability compared to EVX. The EVX speedup varies from 1.28x to 4.86x over the scalar baseline depending on the kernel and VR size. Geometric mean (*GM*) is 3.04. The kernels that exploit temporal locality in VRs (*ADI*, *ESF*, *Ipred*) improve performance with increased VR size. The kernels with streaming characteristics do not benefit from larger VRs (*Fdiff*, *Matrix*), since the VRs are only used as streaming buffers. In some kernels where the vectorized code is inside an outer loop, data cache reconfiguration is performed multiple times, which leads to performance degradation (*BLSS*, *HFrag*). The configurable size of the VRs allows EVX to adapt to the specific characteristics of the workloads.

Figure 3c shows the performance breakdown of EVX with different number of VMSHR entries and hardware optimizations. We have simulated a restricted EVX with several features disabled (**wide-sequential-memory-access**, **wide-VMSHR**, **ALU-partitioning** and **compute-resources-allocation**) to measure the impact of each one of them. The most simple EVX with all features disabled and a small VMSHR of 8 entries (**limited-EVX**) decouples memory and computation instructions and transfers multiple operands to the reservation stations in a single cycle. However, it provides limited speedup over the scalar baseline for most of the kernels or even performance degradation due to VCU initialization, with the exception of *ICCG*, *Ipred*, *Matrix* and *PiCell*. EVX with **wide-sequential-memory-access** generates a single request for a whole cache line for sequential memory patterns. This reduces the number of outstanding memory requests and address calculations. Kernels that use the sequential pattern benefit from this feature, yielding up to 2x speedup (*Fdiff*, *Hfrag*, *Iprod*, *ESF*) even with an 8-entry VMSHR. A larger VMSHR of 64 entries (**wide-VMSHR**) is benefitial for kernels that have strided patterns, since they produce more in-flight memory requests (e.g. *ICCG*, *Ipred*, *Matrix*), while other kernels do not make use of more than 8 entries. EVX with **ALU-partitioning** compute 64 bits of the vector data per lane regardless of the size of vector elements (e.g. 2x32bit, 4x16bit) and leverage the VCU to read packed compute vector operands. ALU partitioning increases the average speedup by about 10%. **compute-resources-allocation** feautre allows to reexecute compute instructions by refreshing the operands in the reservation stations. It yields 15% additional speedup by reducing control overhead, instruction fetch and issue pressure.

2) Power and Area Efficiency: Table III presents the area breakdown of different microarchitectural components in an EDGE core with support for EVX model. EVX increases the area of processor by 2.1% for the VCU, while assuming banked instruction window and reservation stations as a baseline. Table III shows the average total power consumption, while running scalar and EVX versions of Livermore loops. Reexecution of dataflow in multiple vector lanes reduces power consumption in the fetch unit, but increased dynamic activity of ALUs and memory system dissipate additional power. EVX incurs in 47.5% of average total power increase (19.8% excluding the L2 cache), and while providing an average speedup over 3x it significantly increases the power efficiency of EDGE cores (measured in performance per watt) by 2.16x in average.

IV. Conclusion

New technologies in the computer industry aim to deliver a high-quality user experience while running data parallel, multimedia applications seamlessly across all market segments, including low power mobile devices. Efficiently exploiting DLP from these applications requires improvements to current DLP acceleration models. In this work, we introduce vector execution model for low power cores as one such potential technology for enhancing the quality of future processors by efficiently exploiting DLP. Future work will look at scaling the performance and efficiency of EVX beyond the single core.

References

[1] K. Asanovic, "Vector microprocessors," Ph.D. dissertation, University of California, Berkeley, 1998.

[2] S. Ciricescu *et al.*, "The reconfigurable streaming vector processor (RSVP^{TM})," in *Proc. of the 36th MICRO*, 2003, pp. 141–150.

[3] R. Espasa and M. Valero, "Decoupled vector architectures," in *Proc. of the 2nd HPCA*, 1996, pp. 281–290.

[4] R. Espasa, M. Valero, and J. E. Smith, "Vector architectures: past, present and future," in *Proc. of the 12th ICS*, ser. ICS '98, 1998, pp. 425–432.

[5] M. Gebhart *et al.*, "An evaluation of the trips computer system," in *Proc. of ASPLOS'09*, March, pp. 1–12.

[6] C. Kim *et al.*, "Composable lightweight processors," in *Proc. of the 40th MICRO*, 2007.

[7] C. Kozyrakis and D. Patterson, "Overcoming the limitations of conventional vector processors," in *Proc. of the 30th ISCA*, 2003, pp. 399–409.

[8] R. Krashinsky *et al.*, "The vector-thread architecture," in *Proc. of the 31st ISCA*, 2004, pp. 52–63.

[9] Y. Lee *et al.*, "Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators," in *Proc. of ISCA '11*, pp. 129–140.

[10] S. Li *et al.*, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. of the 42nd MICRO*, 2009, pp. 469–480.

[11] F. M. McMahon, "The Livermore FORTRAN kernels: A computer test of numerical performance range," LLNL, Tech. Rep. UCRL-55745, source code from http://www.netlib.org/benchmark/livermorec.

[12] S. Nakazato, S. Tagaya, N. Nakagomi, T. Watai, and A. Sawamura, "Hardware technology of the sx-9," *NEC Technical Journal*, no. 4, pp. 15–18, 2008.

[13] NVIDIA, "NVIDIA's Next generation CUDA compute architecture: Kepler GK110," White Paper, 2012.

[14] J. Owens *et al.*, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[15] A. Putnam *et al.*, "Dynamic vectorization in the e2 dynamic multicore architecture," *SIGARCH Computer Architecture News*, pp. 27–32, 2011.

[16] F. Quintana, J. Corbal, R. Espasa, and M. Valero, "Adding a vector unit to a superscalar processor," in *Proc. of the 13th ICS*, 1999, pp. 1–10.

[17] R. M. Russell, "The cray-1 computer system," *Commun. ACM*, vol. 21, no. 1, pp. 63–72, Jan. 1978.

[18] R. Sasanka *et al.*, "ALP: Efficient support for all levels of parallelism for complex media applications," *ACM TACO*, vol. 4, no. 1, Mar. 2007.