**ADVERTIMENT**. La consulta d’aquesta tesi queda condicionada a l’acceptació de les següents condicions d'ús: La difusió d’aquesta tesi per mitjà del servei TDX (www.tesisenxarxa.net) ha estat autoritzada pels titulars dels drets de propietat intel·lectual únicament per a usos privats emmarcats en activitats d’investigació i docència. No s’autoritza la seva reproducció amb finalitats de lucre ni la seva difusió i posada a disposició des d’un lloc aliè al servei TDX. No s’autoritza la presentació del seu contingut en una finestra o marc aliè a TDX (framing). Aquesta reserva de drets afecta tant al resum de presentació de la tesi com als seus continguts. En la utilització o cita de parts de la tesi és obligat indicar el nom de la persona autora.

**ADVERTENCIA**. La consulta de esta tesis queda condicionada a la aceptación de las siguientes condiciones de uso: La difusión de esta tesis por medio del servicio TDR (www.tesisenred.net) ha sido autorizada por los titulares de los derechos de propiedad intelectual únicamente para usos privados enmarcados en actividades de investigación y docencia. No se autoriza su reproducción con finalidades de lucro ni su difusión y puesta a disposición desde un sitio ajeno al servicio TDR. No se autoriza la presentación de su contenido en una ventana o marco ajeno a TDR (framing). Esta reserva de derechos afecta tanto al resumen de presentación de la tesis como a sus contenidos. En la utilización o cita de partes de la tesis es obligado indicar el nombre de la persona autora.

**WARNING**. On having consulted this thesis you’re accepting the following use conditions: Spreading this thesis by the TDX (www.tesisenxarxa.net) service has been authorized by the titular of the intellectual property rights only for private uses placed in investigation and teaching activities. Reproduction with lucrative aims is not authorized neither its spreading and availability from a site foreign to the TDX service. Introducing its content in a window or frame foreign to the TDX service is not authorized (framing). This rights affect to the presentation summary of the thesis as well as to its contents. In the using or citation of parts of the thesis it’s obliged to indicate the name of the author

# Specialization and Reconfiguration of Lightweight Mobile Processors for Data-Parallel Applications

Milovan Duric

Department of Computer Architecture

Universitat Politècnica de Catalunya

A document submitted as

*PhD Thesis*

December, 2015

**Advisor**: Oscar Palomar
**Director**: Mateo Valero

To my parents for their unwavering support in all my endeavors.
Without you this would not have been possible.

*Mojim roditeljima za njihovu beskrajnu podršku u svim mojim nastojanjima. Bez vas ovo ne bi bilo moguće.*

# Acknowledgements

I would like to acknowledge my advisors Mateo Valero, Adrin Cristal, Osman Unsal and Oscar Palomar for giving me the opportunity to attend PhD studies Universitat Politècnica de Catalunya in Barcelona, and for their guidance, confidence, and patient demeanor during the studies. They are the reason for my success on the professional plan, but also the reason for all the good time I had with my friends and my wife in this beautiful city.

I would like to thank Doug Burger and Aaron Smith for inviting me to the unforgettable three month internship at Microsoft Research Redmond. They provided me with their invaluable support and help, both during my stay in Redmond and after I returned to Barcelona and continue working with them on various projects related with my studies. They showed me how hard work can and does pay off, how hard work provides huge satisfaction if you are building something useful, and how team work is the key for being successful.

I would also like to acknowledge all my great friends and colleagues from Barcelona Supercomputing Center (BSC) that supported me during my PhD studies and that shared with me their insight and their expertise. Many thanks go to Srdjan Stipic, Sasa Tomic, Vasilis Karakostas, Daniel Nemirovsky, Ferad Zyulkyarov, Gulay Yalcin, Azam Seyedi, Nehir Sonmez, Vesna Smiljkovic, Vladimir Gajinov, Adri Armejach, Nikola Markovic, Milan Stanic, Ivan Ratkovic, Timothy Hayes, Javier Arias, Petar Radojkovic, Nikola Rajovic, Ivan Tanasic and many many others that I missed adding. I sincerely thank all you guys for your company during my PhD and for all the great time we had together.

Finally, I would like to thank my family for all their love and encouragement. I deeply thank my parents, Radomir and Dragica, my sister Andriana and most of all my loving, encouraging, and patient wife Tamara whose faithful support during the final stages of this PhD is so appreciated.

This work is supported by the cooperation agreement between the Barcelona Supercomputing Center, National Supercomputer Facility and Microsoft Research and by the Ministry of Science and Technology of Spain and the European Union.

# Abstract

The worldwide utilization of mobile devices makes the segment of low power mobile processors leading in the entire computer industry. Customers demand low-cost, high-performance and energy-efficient mobile devices, which execute sophisticated mobile applications such as multimedia and 3D games. State-of-the-art mobile devices already utilize chip multiprocessors (CMP) with dedicated accelerators that exploit data-level parallelism (DLP) in these applications. Such heterogeneous system design enable the mobile processors to deliver the desired performance and efficiency. The heterogeneity however increases the processors complexity and manufacturing cost when adding extra special-purpose hardware for the accelerators. In this thesis, we propose new hardware techniques that leverage the available resources of a mobile CMP to achieve cost-effective acceleration of DLP workloads.

Our techniques are inspired by classic vector architectures and the latest reconfigurable architectures, which both achieve high power efficiency when running DLP workloads. The high requirement of additional resources for these two architectures limits their applicability beyond high-performance computers. To achieve their advantages in mobile devices, we propose techniques that: 1) specialize the lightweight mobile cores for classic vector execution of DLP workloads; 2) dynamically tune the number of cores for the specialized execution; and 3) reconfigure a bulk of the existing general purpose execution resources into a compute hardware accelerator. Specialization enables one or more cores to process configurable large vector

operands with new special purpose vector instructions. Reconfiguration goes one step further and allow the compute hardware in mobile cores to dynamically implement the entire functionality of diverse compute algorithms.

The proposed specialization and reconfiguration techniques are applicable to a diverse range of general purpose processors available in mobile devices nowadays. However, we chose to implement and evaluate them on a lightweight processor based on the Explicit Data Graph Execution architecture, which we find promising for the research of low-power processors. The implemented techniques improve the mobile processor performance and the efficiency on its existing general purpose resources. The processor with enabled specialization/reconfiguration techniques efficiently exploits DLP without the extra cost of special-purpose accelerators.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The technology scaling of CMOS transistors[52] has allowed for an exponential growth of transistor density over the last four decades. By using the extra transistors to implement more complex computer architectures (e.g. pipelining, superscalar, out-of-order, caching), each new generation of a processor family has provided more computational work per clock. At the same time, the technology improvement has enabled achieving higher frequencies. While increasing the processor complexity and frequency to improve the performance, power density has been increasing as well, further making processor cooling highly expensive and difficult. Power density problems have forced processor manufacturers to introduce multiple cores to scale performance in a power-efficient way by limiting clock frequency and core design complexity. Consequently, chip multiprocessors (CMP) in the last decade have penetrated various market segments from servers to low power smart mobile devices. At the same time, mobile devices are becoming ubiquitous and the mobile market segment is starting to dominate the entire computer industry. Customers demand that each new generation of mobile devices significantly improves upon the previous generation. They seek low-cost, high-performance and energy-efficient mobile devices, which execute sophisticated mobile applications. The costs, performance and efficiency requirements tailor the design of future mobile processors.

The mobile devices of today enable new compelling user experiences. Users access their mobiles through more natural interfaces such as speech and video gesture. Mobile applications manage an ever-increasing amount and quality of mobile data like: photos, videos, and a world of content available in the cloud. Although computationally demanding, the abundant level of parallelism in modern mobile software offer potential for power-efficient acceleration. By exploiting the available parallelism it is possible to address the major considerations in the design of future mobile processors.

Parallelism exists on different levels: instruction, data and task (ILP, DLP, TLP). ILP allows for simultaneous execution of multiple instructions (operations) of a single instruction stream. TLP allows for simultaneous execution of multiple independent tasks, where each task is part of parallel program that executes its own instruction stream (thread). DLP allows for simultaneous execution of the same operation on multiple data elements as indicated by a single instruction. Today's mobile CMPs efficiently exploits ILP in a single out-of-order mobile core and TLP with multiple low power cores. On the other hand, taking all the advantages of ample DLP in mobile applications is still challenging on CMPs without dedicated hardware. To harness this extreme potential and improve efficiency of CMPs, various solutions have been deployed particularly in the form of accelerators and offload engines inside heterogeneous system architectures for low power devices.

Heterogeneous system architectures [6, 25, 42, 43] introduce an advanced processor design that incorporates various processing units in a single chip. Figure 1.1 shows a layout of heterogeneous architecture built in the Qualcomm Snapdragon 810 processor for low-power mobile mobile devices. The Snapdragon processor includes general purpose cores with different power and performance characteristics (CPUs), a graphics processing unit (GPU), a digital signal processor (DSP), an image signal processors (ISPs), and a dedicated multimedia subsystem and other specialized hardware. Such complex design with many incorporated units allows to chose the proper substrate for each workload depending on its characteristics and available parallelism. The extra dedicated units (e.g. multimedia accelerators) improve power efficiency either by exploiting the parallelism or by using synthesized hardware to perform frequently repeated computation of a particular workload. On the other hand, the incorporation of dedicated resources incurs additional area overheads and increases the design complexity and verification cost of the processors. Although silicon area seems to be inexpensive nowadays,

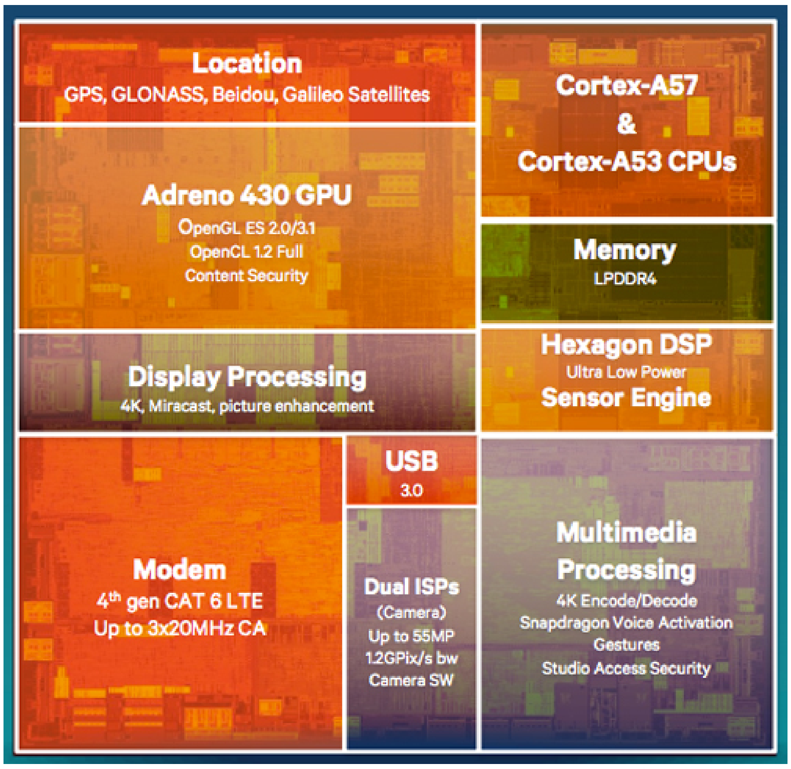


Figure 1.1: An example of the heterogeneous system architecture in the Snapdragon 810 commercial processor. It has general purpose cores and various dedicated units incorporated on a single chip.

mobile processor manufacturing costs are comparatively more sensitive to area. In particular, the low end market segment of mobile processors with large production numbers and slim margins is greatly affected by additional area costs. For example in commercial Snapdragon processor showed in Figure 1.1, the low power ARM cores approximately cover less than 20% of the total area footprint. In the Snapdragon chip, the dedicated units incur extreme area overheads compared to general purpose cores. Minimizing the additional area for various accelerators directly affects the final cost of future mobile processors. At the same time, it is also interesting to think about the extra power required to run the dedicated processing units. Even when not using them, they dissipate leakage power, and if they are power gated, there is a significant performance overhead for powering these units on and off.

To improve power efficiency without significant area and power additions, we propose a novel design methodology for future mobile processors. This design methodology avoids the processor heterogeneity and yet allows us to build a high-performance, power-efficient mobile processor. Our processors dynamically specialize and reconfigure the existing hardware resources available in general purpose cores to exploit the DLP in mobile workloads. Such methodology should: (1) incur minimal area and power overheads to the existing execution substrate and (2) deliver desirable power and performance improvements on the lightweight mobile processor without dedicated accelerators.

There are several existing techniques, which also improve efficiency of general purpose cores, e.g. clock gating, power gating, dynamic voltage and frequency scaling. These techniques are applied to the existing hardware resources and incur negligible additions including for mobile processors. However, they do not sufficiently increase the mobile processors efficiency neither exploit the available parallelism in mobile software like our specialization and reconfiguration techniques. Recent research to improve performance efficiency by exploiting the parallelism explores yet another research alternative through dynamic CMPs [22, 32, 36, 41]. They compose simple lightweight cores into a bigger and more powerful core to tailor the execution substrate for a particular workload. By composing or decomposing the available cores to run a single- or multi- threaded applications, dynamic CMPs provide a flexible and power-efficient execution substrate that exploit ILP and TLP, but not DLP. As far as we know, there is no work to date that has investigated accelerating DLP workloads on dynamic CMPs.

While these existing cost-effective techniques and innovative architectures improve the efficiency of mobile CMPs, they still do not exploit the high potential of available DLP in mobile applications. Classic vector processors [1, 19, 30, 44, 63, 75], multimedia SIMD extensions [8, 20, 55, 59, 68, 73], GPUs [46, 53, 54, 56, 77] and various compute hardware accelerators are different architectural models that are typically employed to exploit DLP. Each of these models performs DLP operations at different granularities. Vector processors and multimedia extensions leverage a single-instruction multiple-data (SIMD) model to increase the power-efficiency of DLP applications. Vector processors increase power efficiency by using a single vector instruction to operate over configurable large vector operands.

The configurable vector length reduces fetch, decode and issue power dissipation. By issuing more in-flight requests to load/store large operands, vector processors efficiently tolerate memory latency and increase their performance. Although efficient on DLP applications, the area costs of modern vector processor [44] limit their utilization especially in lightweight mobile devices. On the other hand, multimedia extensions use specialized SIMD hardware on general purpose cores and thus feature less area overhead. Compared to vector processors, multimedia extensions provide limited improvements of applications with DLP phases. GPUs extend the SIMD model, by combining it with many threads to further increase the efficiency of SIMD execution. Such execution model is typically referred to as SIMT (single-instruction multiple-threads). GPUs provide remarkable improvements of highly parallel applications, but their architectural model seems to perform less efficiently than the model of classic vector processors [44]. Hardware accelerators improve the power-efficiency of compute intensive DLP workloads in the most efficient way, by using a fixed-function or dynamically configurable hardware that performs a specific computation over many elements. Their applicability is limited, which makes the overhead of incorporating them more noticeable in mobile processors.

In this thesis, we intent to exploit the available resources of a mobile CMP to achieve a cost-effective acceleration of DLP workloads and increase the processor efficiency. Our research avoids the high requirement of additional resources for classic vector processors, or hardware accelerators. Instead of incorporating such power-efficient though complex offload engines to a mobile processor, we dynamically specialize the processor for classic vector execution or reconfigure it into a compute hardware accelerator. Such tailoring of the processor execution substrate improves its efficiency with minimal hardware additions.

Rather than building the mobile processor from the scratch, we consider the existing architectures for low power devices. Most of the low power devices nowadays use ARM [62] and X86 Atom processors [33]. They have been greatly improved over a last couple of years, including upgraded to 64-bit architectures. On the other hand, this research proposes low power CMPs based on an Explicit Data Graph Execution (EDGE) architecture [7, 21, 69]. As opposed to high-performance mobile ARM and X86 cores that use a power-hungry out-of-order microarchitecture to discover data dependencies dynamically, EDGE processors leverage an EDGE compiler to encodes data dependencies statically. The compiler encodes dataflow within the EDGE instructions and thus avoids complex hardware structures typically found in out-of-order cores. It makes EDGE cores power-efficient and cost-effective.

EDGE CMPs incorporate dynamic capabilities that compose their lightweight cores to increase flexibility and performance efficiency. Previous research [22, 36] of CMPs with composable EDGE cores has shown promising results. Motivated by these results, we have chosen the composable EDGE cores as a research vehicle where we investigate our ideas. And even though we chose such an unconventional baseline, we need to emphasize that the ideas proposed in this thesis can be applied to a diverse range of general purpose processors available nowadays. Therefore, we see these ideas as competitive approaches to improve and build a real product, a mobile CMP that dynamically reconfigures into a vector or a compute hardware accelerator.

We summarize the proposed ideas into the following list of high-level contributions presented in this thesis:

- **EDGE core Specialization for Power-Efficient Vector Execution - EVX**, a hardware specialization technique that efficiently exploits DLP on a modest EDGE core. EVX avoids area and complexity of classic vector processors by specializing the existing resources of the EDGE core. It yet provides the advantages of vector architectures such as: register- and streaming-based vector execution over configurable large vector operands, sophisticated addressing modes and memory latency tolerance. It allows an efficient execution of parallel and non-parallel applications on an EDGE core without offloading them to an accelerator.

- **CMP Specialization for Dynamically-Tuned Vector Execution - DVX**, a novel technique that specializes a general purpose CMP into an efficient DLP accelerator. As opposed to adding a dedicated accelerator to a general purpose CMP, DVX leverages the existing CMP resources to avoid the area overheads of the specialized hardware. DVX leverages and extends the EVX design to utilize the additional cores in CMPs. It allows a processor to dynamically configure the allocation of compute and memory resources, which are specialized to process vector elements in parallel. The thesis explains an architectural model of DVX that is applicable to most commercial CMP nowadays, various design options and one particular DVX implementation on a dynamic EDGE CMP.

- **High-Performance EDGE Vector Co-processor - VCOP**, a novel vector design that combines classic high-performance vector architectures, EDGE and GPUs to efficiently accelerate DLP workloads. We build dedicated VCOP to compare it to a DVX enabled EDGE CMP, and we find VCOP as power-efficient high-performance accelerator. We believe that the results motivate further research of VCOP, as a modern and competitive vector processor beyond low power and mobile devices.

- **Configurable Compute Fabric of General Purpose Resources**, an accelerator of compute intensive DLP workloads that reduces most of the overheads imposed by conventional accelerators. Instead of using extra execution resources, the compute fabric is composed of the execution resources available in a CMP. Such accelerator does not incur a fixed-function hardware or configurable functional units to the chip and yet efficiently accelerates compute-intensive DLP workloads.

- **Reconfiguration of Cores for Mobile Devices**, a novel design of general purpose cores, which adapt their resources to workload demands. One or more cores on-the-fly are composed into a big processor or reconfigured into an accelerator. Reconfigurable cores build a novel high-performance and low-cost mobile processor which: 1) maximizes computational capabilities of the existing general purpose cores; 2) minimizes the amount of extra hardware for accelerators.

# Chapter 2

# Background on DLP Accelerators and Composable EDGE Cores

In this chapter we explain more details related with the existing architectural models that exploit DLP, their advantages and limitations. We discuss about composable EDGE cores that we use as a research vehicle for our investigation, including the cores architecture and their dynamic features.

## 2.1 DLP Accelerators

### 2.1.1 Classic Vector Processors

Classic vector processors are known to be an efficient substrate to execute DLP applications and achieve high performance [44]. They have been used for decades in supercomputers to accelerate various scientific applications with a large amount of DLP [1, 13, 30, 63, 75]. Recently, the fast evolution of GPU hardware [46, 77] have motivated the manufacturers to use GPUs instead of vector processors as accelerators for high performance DLP workloads. Nowadays the computer industry produces only one supercomputer based on vector architectures [79]. At the same time the ample DLP in commodity software motivates further research of advanced vector architectures beyond supercomputers [2, 17, 38, 39, 40, 44, 61]. Multimedia speech, image or video processing repeat identical operations over the streams of sound samples, pixels and video frames. The DLP available through

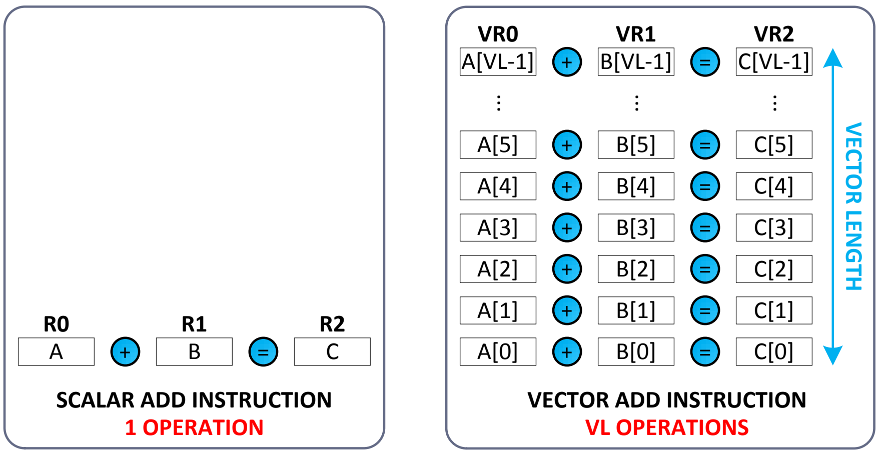


Figure 2.1: The execution model of a scalar add and a vector add instruction. The vector add instruction performs VL operations over vector operands, whereas the scalar add perform only one operation.

these operations especially fit to vector execution, which revives the popularity of classic vector processors.

The classic vector architectures such as the Cray-1 [63] use large register files to hold sets of data elements (vectors). Each vector instruction operates on large, configurable-size vector operands in those registers, as opposed to scalar instructions in a general purpose processor that operate over a single element. Thus each vector instruction exposes more parallelism and reduces the power hungry structures that extract the parallelism dynamically (e.g. hardware for out-of-order execution).

Figure 2.1 shows the execution model of an add instruction in a general purpose scalar and a special purpose vector processor. The scalar add instruction leverages scalar registers (R0, R1, R2), which hold single elements. The scalar instruction controls one add operation over elements A and B to produce a single result C. One other hand, the vector add instruction manages large vector registers (VR0, VR1, VR2), which hold vector operands. The number of elements per register is dynamically defined up to their size, depending on the vector length (VL). One vector add instruction performs VL operations to provide an efficient addition of two vectors.

The vectors are placed somewhere across the memory (assuming no caches) and the large vector registers are used both to hide long memory latency and exploit temporal data locality. Vector memory instructions are deeply pipelined by consecutively issuing memory requests for data of a large vector register. The memory latency is paid only once for the first element and amortized over many data in the vector register. After an initial overhead, vector memory instructions are able to move one memory word between registers and memory at each cycle. Such memory processing is one of the most important advantages of vector architectures. It maximizes the usage of available memory bandwidth and increases the utilization of compute resources in vector processors.

Vector architectures include several hardware structures, which are not found in general purpose processors. We find the following structures as the most relevant to achieve an efficient DLP acceleration.

- *Vector registers* are sized to hold a large number of vector elements, instead of single values like scalar registers of general purpose processors. Hence, each vector instruction encodes a large amount of parallelism through many independent operations over the elements in vector registers. To enable simultaneous execution of different vector instructions on different functional units, vector registers need to provide multiple read and write ports. Large size vector register files with many ports are one of the most complex and power-hungry structures in vector processors.

- *Vector functional units* are deeply pipelined, so that they can start a new operation on every cycle. To perform more than a single operation per cycle and improve the performance, vector processors generally incorporate parallel pipelined functional units, called *vector lanes*. In this case, vector registers are partitioned across the lanes. It increases performance of vector processors by enabling many operations of a single vector instruction to be pipelined among the parallel units at a low design complexity.

- *Vector memory units* load or store vector operands in a pipeline to efficiently tolerate memory latency. Besides sequential memory access, they support complex addressing modes: strided access that loads/stores vector elements with unit or non-unit stride between the elements; indexed access that gathers/scatter vector elements by using their indices. Such sophisticated memory units can operate over consecutive and non-consecutive vector elements, as well as sparse arrays. It increases the applicability of vector processors by allowing for vectorization of a wide range of DLP workloads.

- *Vector mask registers* are used to support conditional execution of each operation in a vector instruction. Masking of vector operations enables vectorization of loops with conditional computation and further increases the applicability of vector processors.
- *Vector length registers* are used to control the vector length of a DLP computation, since the length may not be known at compile time. By dynamically sizing the number of vector elements, vector processors reduce the number of control instructions and increase their efficiency.

On the top of basic vector hardware many improvements have been proposed. The complexity of the vector register file has been reduced by having separate clusters of registers for different functional units and thus reducing the number of ports [38]. Besides register-based vector execution, streaming-based vector processors have been proposed [10]. They leverage vector registers to buffer streaming data between the memory and execution units. Streaming based vector processor do not capture the temporal locality in vector registers. Thus they allow for an efficient processing of unlimited streaming data within a single vector instruction. For non-streaming workloads, vector processors have been extended with caches to capture data locality and minimize initial memory overheads. Decoupled execution of memory and compute instructions avoids memory overheads, by loading vectors ahead of computation [16]. Work on out-of-order execution of vector instructions shows even better results [18]. Combining vector and threaded execution increases the flexibility of vector processor and exploit TLP and DLP on a unique parallel substrate [40].

Specialized vector processors deliver an auspicious substrate for DLP workloads with many advantages over general purpose processors. They expose an extremely high amount of parallelism with a single vector instruction, by operating over many independent elements. It reduces fetch, decode and issue overheads, and simplifies the control logic of processor front-end. Sophisticated vector memory units further increase the utilization of vector processor resources and thus its efficiency. Parallel functional units exploit ample DLP and yield a high performance of vector processors.

On the other hand, vector processors have certain limitations. They incur remarkable area overheads due to large vector registers and multiple vector functional units. As such, vector processors have been historically used in supercomputers rather than as general purpose processing elements in commodity processors. Recent interests in running mobile parallel applications with vector processors have leaded to various low power vector designs for embedded systems [10, 34]. Embedded vector designs have been made in the form of dedicated accelerators, with hardware additions limited to DLP applications. Non-parallel workloads make these accelerators idling most of the time and thus their heavy hardware yet unavailable in mobile processors. What is needed is a microarchitectural support that offers the advantages of vector processors on the existing resources of mobile CMP. Specializing the mobile CMP into an efficient vector-based accelerator would avoid extra area costs, while running parallel and non-parallel applications on the unique substrate.

### 2.1.2 Multimedia SIMD Extensions

Multimedia SIMD extensions have been used over a decade to exploit DLP in commodity processors, while exploiting very much of the existing hardware [8, 20, 55, 59, 73]. Pioneer multimedia extensions exploited sub-word parallelism for narrower elements than 32-bit processors were optimized for. By partitioning the existing functional units and data in scalar registers, the pioneer extensions were able to perform simultaneous operations on 8- or 16- bit vectors. Operating over multiple operands in parallel within a single SIMD instruction has increased the performance of DLP workloads at a low hardware complexity. With the time SIMD width has grown, while incorporating wider functional units and wider SIMD registers. Nowadays, the SIMD width goes up to 512 bits [67].

Multimedia extensions increase the performance of DLP workloads on general purpose cores. They incur moderate area overheads for SIMD registers and functional units, while leveraging most of the existing processor resources. Compared to more complex and larger hardware additions like vector processors, multimedia extensions cost less and they are easier to implement. Therefore, they have been widely applied in commercial processors, including those for mobile devices.

Multimedia extensions successfully accelerate DLP workloads with narrow data operands or short vectors. On the other hand, they do not fully exploit the maximum potential of highly parallel mobile workloads. The moderate size of SIMD resources deliver significantly less performance and efficiency than the heavier hardware of vector processors. Beyond the performance and efficiency, multimedia extensions lack a few other features found in vector processors. The most important are the following:

- Multimedia extensions operate over small operands and the operand size is encoded within a SIMD instruction (e.g. 4 element of 32-bit), as opposed to large dynamically sized vector operands. It leads to the addition of hundreds of instructions across different multimedia extensions for the same processor product family. Additional fetch, issue, decode and control overhead is necessary to iterate the SIMD instructions sized for small operands in order to process large vectors.

- SIMD memory instructions offer limited support for non-sequential memory accesses. Intel has recently introduced only support for indexed access [67]. Processors with multimedia extensions issue a modest number of memory requests, as opposed to sophisticated vector memory units that pipeline a large number of memory requests. It may incur overheads due to packing and unpacking SIMD operands, which can exceed the benefits of wider registers and functional units.

- Multimedia extensions lack masked operation of a SIMD instruction. It reduces the applicability of SIMD extensions to DLP workloads without conditional computation in loops.

Employing the missing vector features and increasing the width of SIMD resources tailor the design of future SIMD extensions. Such extensions will perform like classic vector architectures within general purpose processors, which may provide more efficient and powerful DLP acceleration.

### 2.1.3 Graphics Processing Units

Graphics processing units (GPUs) have been invented as a dedicated hardware accelerators for 2D and 3D graphics, images and videos [47, 51]. GPUs promoted the use of windows-based operating systems, graphical user interface, video games and other graphics features. They have evolved into more programmable highly parallel multiprocessors [46, 54, 77]. The fixed-function GPU hardware has been replaced with the programmable pipeline, which is exposed through parallel programming languages. Hence, GPUs nowadays extend their applicability beyond the graphics processing. They allow for acceleration of highly parallel and compute intensive DLP applications.

Modern GPUs provide an extremely parallel execution substrate with hundreds of floating point functional units. In the most basic form, GPUs have multiple cores and multiple banks of DRAM memory to support sufficient memory bandwidth for each core. The GPU cores combine the SIMD and multithreaded execution models (Nvidia calls such cores streaming multiprocessors). Each core has a lot of parallel functional units organized in lanes, as opposed to vector processors that have a few lanes with deeply pipelined units. The number of registers in GPU cores is also impressive and they are partitioned across the lanes. It enables the GPU cores to execute many GPU threads, without an expensive thread context switch. Each thread uses a subset of GPU registers that holds a state of the thread. The threads are organized in groups, which execute in lockstep by using SIMD based hardware (Nvidia calls such groups warps). For each thread in the group, the same instructions issue and execute at the same time in different lanes. An address coalescing hardware intents to fuse parallel memory requests from multiple lanes into a single request to reduce memory overheads and efficiently exploit available bandwidth.

Large amount of DLP in parallel programs is exploited through many GPU threads. The threads are divided into thread blocks, and the blocks into the groups that executes in lockstep. GPUs employ a hardware thread block scheduler that assigns thread blocks to the multithreaded GPU cores. Each core then utilizes a hardware thread scheduler to pick which group of threads from the assigned block to run each cycle. By picking whatever group that is ready to go, the GPU core hides the long latency to DRAM increases the utilization of the compute GPU resources. Therefore, the GPU area is spent on the large number of registers to hold the state of many threads instead of on large caches as happens in general purpose multicore processors. Massive multithreading to hide the latency is also in contrast with vector processors, which pay the latency for the first memory operation and then intent to pipeline the rest of accesses.

GPUs provide a dynamic runtime hardware that allows for execution of DLP workloads with complex control flow. It enables the group of GPU threads to execute instructions such as branch, jump, call or return in each thread. The threads of the group may diverge when some lanes branch to some address, while the others not. This flexibility has the consequence of providing independent control flow for each thread in the group. Nevertheless, only one instruction can execute across the lanes. Runtime conditional execution on GPU hardware is similar to what vector processors do at compile time by using scalar instructions and masked operations. While the divergence of GPU threads makes programming easier, the efficiency and performance results of GPU acceleration might be insufficient, when control flow is too complex.

Various innovations have been deployed to improve the basic form of GPUs. Dual thread scheduler with two dispatch units per GPU core increase the utilization of the core compute resources [77]. Even further improvements in utilization of GPU resources are achieved through multitask execution on a single GPU [54, 72]. Fast double precision floating point arithmetic and error correction code (ECC) are some of the features that provide more general purpose computing on GPU [76]. Cache memories are added on GPUs to capture data locality. They save many local variables that are shared between threads, but also other temporal data for function calls, stack frames or register spilling. Integrated GPU and CPU on the same chip [6, 27] allows for a fast offloading of parallel workload onto the GPU accelerator.

Modern GPUs are used in many supercomputers to increase the performance of high performance computing applications with substantial DLP. Recent low power heterogeneous architectures use GPUs as a way to accelerate mobile DLP applications. While GPUs can provide an order of magnitude increase in performance for highly parallel applications, their area overheads can be high for mobile processor systems. Moreover, further generalization of GPU hardware leads to less efficient graphics processing, which significantly affects the power efficiency of mobile processors. And although substantially reduced the penalty of moving data between the CPU and GPU on the same chip, not highly parallel applications yet incur the overheads for repetitive data movement operations. For such applications, the acceleration on general purpose cores may be more efficient.

### 2.1.4 Hardware Accelerators

Besides SIMD extensions and GPUs, various hardware accelerators are employed to accelerate compute intensive and data parallel applications. Accelerators yield high performance and efficiency within a fixed-function hardware logic that performs a specific computation over large amount of data. Instead of executing a single operation over many elements like vector processors, hardware accelerators synthesize the entire compute region with many operations. By streaming the data through the fixed-function hardware, accelerators avoid control complexity. They do not have per instruction overheads such as fetch, decode and complex out-of-order issue nor the register file to hold the temporal results. Such hardware make the accelerators extremely efficient, but limited to particular workloads.

By getting a set of accelerators and general purpose cores to work together in the same chip, heterogeneous computing utilizes additional transistors in a more efficient manner. On the other hand, accelerators have certain disadvantages. They increase processor complexity, hardware design and verification costs, and finally chip area. In a world of mobile processor, the design and verification costs are amortized through a large production of chips, but area overheads preserve a significant portion of the cost. Although the fixed hardware requires modest resources, different applications employ different accelerators, which may incur too much area overhead.

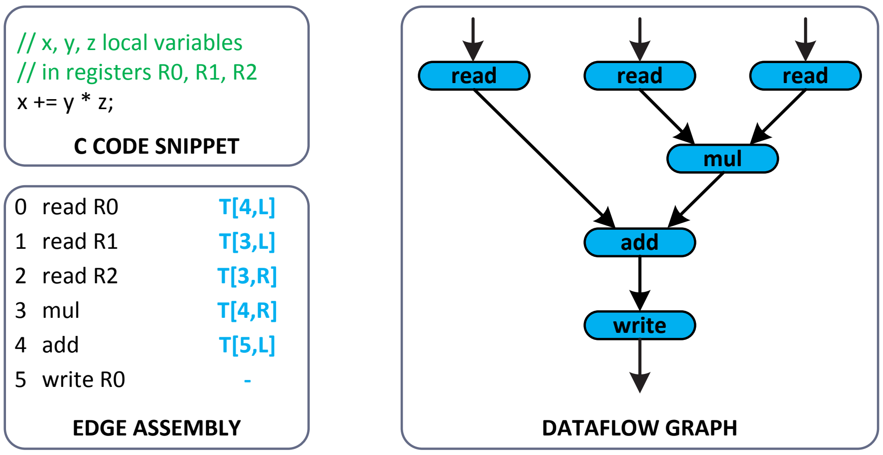
To provide non-specialized accelerators that can be configured to any functionality, various designs propose reconfigurable hardware accelerators [11, 12, 23, 24, 50, 57, 58]. They still outperform CMPs with general purpose cores, SIMD extensions and GPUs across a wide range of applications. The reconfigurable accelerators moderate the disadvantages of accelerators, by matching the compute and performance requirements of diverse workloads. To provide a suitable substrate for various applications, the reconfigurable accelerator requires more compute resources. Such substrate is still area dominant in a domain of low power mobile processors. On the other hand, existing homogeneous mobile computing without accelerators do not provide performance and efficiency amenable by modern mobile applications and their customers. Therefore, the trade-off between performance and cost in mobile devices seems like an interesting design problem for computer architects.

## 2.2 Composable EDGE Cores

### 2.2.1 EDGE Architecture

Explicit Data Graph Execution (EDGE) architectures use compilers to divide a program into blocks of instructions that execute atomically [7, 69]. The atomic instruction blocks (AIB) consist of a sequence of dataflow instructions, where the instructions explicitly encode *producer-consumer* relationships. An EDGE ISA directly expresses the dataflow graph with the encoded relationships between EDGE instructions. The compiler generates dataflow statically and thus avoids power hungry hardware structures that discover data dependencies dynamically. The EDGE ISA exposes more concurrency and increases power efficiency by allowing for an efficient out-of-order execution of EDGE instructions.

EDGE instructions inside the AIB communicate directly. Each instructions leverages two reservation stations, which hold its left and right operands respectively. Producer instructions encode targets that route their outputs to appropriate reservation stations of consumer instructions. The hardware uses the producer instruction's targets to deliver its output directly to the reservation stations of consumer instructions, instead of writing it back to a register file shared among instructions such as in conventional ISAs. Register operations in EDGE architecture are used only for handling less-frequent inter-block communications, and



```
// x, y, z local variables
// in registers R0, R1, R2
x += y * z;
```

C CODE SNIPPET

```
0  read R0        T[4,L]
1  read R1        T[3,L]
2  read R2        T[3,R]
3  mul            T[4,R]
4  add            T[5,L]
5  write R0         -
```

EDGE ASSEMBLY

Figure 2.2: An example of the EDGE assembly and dataflow graph of a short C code snippet. EDGE instructions encode the dataflow, by adding the targets of *consumer* instructions in the code of *producer* instructions.

registers to keep the temporary results between the AIBs instead of caches and memory. Each AIB reads temporary results of previous AIBs from the registers to the reservation stations of its consumer instructions that operate over the results of the previous AIBs. Similarly, the AIB writes new temporary results back to the registers to be consumed by the following blocks. Instructions in the AIB execute in dataflow order, where an instruction becomes ready to issue when its inputs arrive to the reservations stations.

Figure 2.2 shows a C code example, the EDGE assembly of the compiled C code and corresponding dataflow graph. The C code example contains a few operations over local variables (x,y,z), which are assumed to be saved as temporal results of previous EDGE AIBs in registers R0, R1, R2. The dataflow graph shows data dependencies defined by the compiler and encoded through EDGE instructions. The EDGE assembly has instructions that read and write these temporal results between reservation stations and registers (e.g. instruction *0 read R0 T[4,L]* - read the data from register 0 and send it to the left reservation station of the instruction 4). Except for reads or writes, other instructions communicate through the reservation stations (e.g. instruction *3 mul T[4,R]* - performs a mul operation once its inputs arrive to the reservation stations and then sends its output to the right reservation station of the instruction 4).

The EDGE architecture avoids many inefficient and power-hungry structures typically used in today's out-of-order processors. EDGE cores do not incorporate hardware structures such as: per instruction register renaming, multiported register file, complex bypass network, and complex wakeup logic. The EDGE ISA restricts the AIBs in several ways to further simplify the hardware of EDGE cores that maps the blocks to the execution substrate. AIBs are variable-size: they contain between 4 and 128 instructions and may execute at most 32 loads and stores. The hardware relies on the compiler to break programs into blocks of dataflow instructions and assign load and store identifiers to enforce sequential memory semantics [69]. To improve performance, the compiler uses predication to form larger blocks filled with predicated instructions [70]. Some instructions (e.g. conditional) produce a predicate instead of a value. The predicate is broadcast to all instruction and predicated instructions execute only if they receive a matching predicated value. Finally, EDGE ISA simplifies hardware that detects when an AIB has finished and commits its atomic execution. The architecture relies on the compiler to ensure that a single branch is produced from every AIB, and to encode all the register and memory outputs of each AIB. The compiler appends a *header* to each AIB, where it encodes the set of register writes for temporary and store identifiers for permanent results of the AIB execution.

### 2.2.2 Dynamic EDGE CMP

Our baseline dynamic EDGE multicore 2.3 consists of low power, high performance, decentralized processing cores connected by an on-chip network as in [36]. The baseline design provides the benefits of other tiled architectures - namely simplicity, scalability, and fault tolerance. Each core contains an instruction window and two reservation station buffers sized for 128 instructions, along with a 64-entry register file and a 40-entry load/store queue (LSQ). Such microarchitecture design is chosen to support the execution of EDGE AIBs with up to 128 instructions. To enable the processing of 64-bit operands the cores incorporate 64-bit ALUs, as well as size their reservation stations buffers and registers to 64 bits. Each ALU supports integer and floating point operations. Along with single

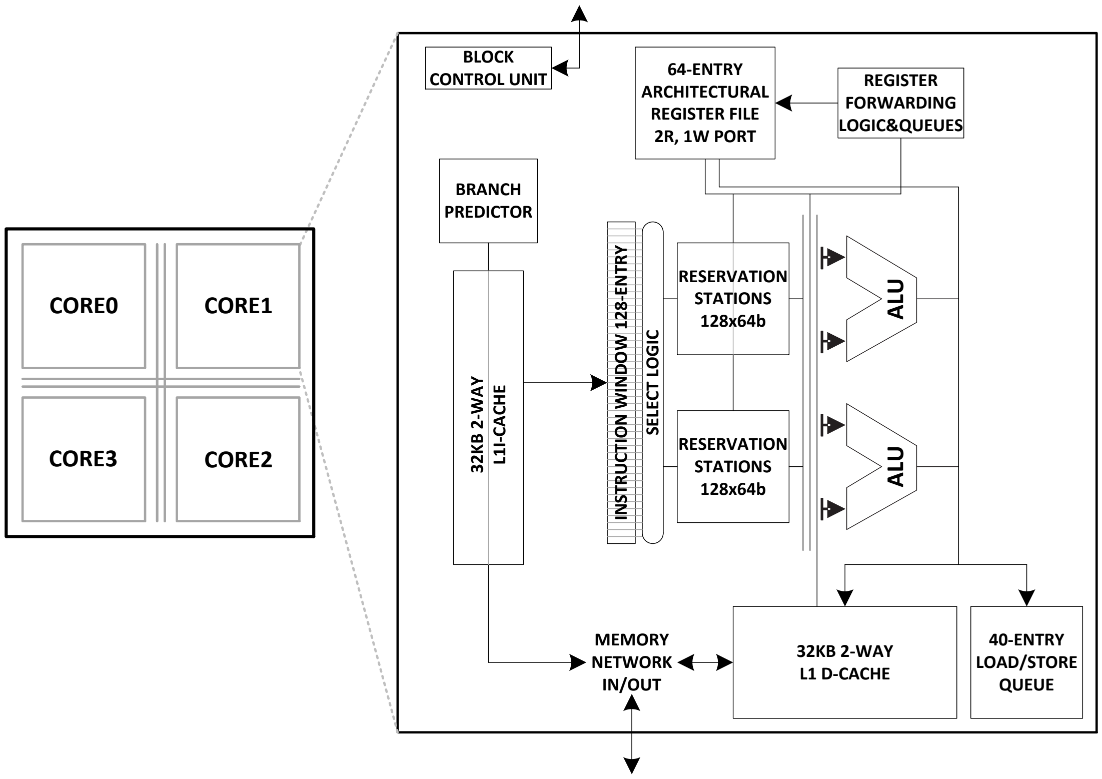

CORE0
CORE1
CORE3
CORE2
BLOCK CONTROL UNIT
64-ENTRY ARCHITECTURAL REGISTER FILE 2R, 1W PORT
REGISTER FORWARDING LOGIC&QUEUES
BRANCH PREDICTOR
32KB 2-WAY L1I-CACHE
INSTRUCTION WINDOW 128-ENTRY
SELECT LOGIC
RESERVATION STATIONS 128x64b
RESERVATION STATIONS 128x64b
ALU
ALU
MEMORY NETWORK IN/OUT
32KB 2-WAY L1 D-CACHE
40-ENTRY LOAD/STORE QUEUE

Figure 2.3: The microarchitecture details of a 4-core EDGE CMP. Each core has an instruction window with 2 reservation station buffers sized for 128 instructions, register file, 2 ALUs and separated instruction and data caches.

64-bit operations, ALUs support sub-word SIMD operations. They support simultaneous eight 8-bit, four 16-bit or two 32-bit operations per single ALU. Each core has separated L1 instruction and data caches. The L2 cache is shared by all cores and contains multiple banks to provide high memory bandwidth (e.g. a 4-core CMP contains 4 banks of L2 cache).

To improve performance of the accesses to the memory system, the LSQ enables unordered, speculative issue of load instructions that are predicted to be independent of previous stores. The LSQ buffers store instructions until their AIB commits to support recovery of speculative loads. When a store instruction arrives to the LSQ, it checks if there is any violating younger load that should have waited for the store. The LSQ leverages load/store identifiers encoded by the compiler to detect if sequential memory semantics have been violated. When a violating load is detected, its corresponding AIB is flushed and reexecuted. The AIB results are discarded, but the state of instruction window is retained to avoid the duplicated fetch and decode of the flushed AIB. When the AIB reexecutes, the LSQ enforces load/store ordering defined by the compiler to ensure progress until the AIB commits. When the AIB commits, the LSQ issues the store instructions to the caches, thus committing the results to memory.

The core executes a single AIB of EDGE instructions within a simple pipeline that resembles pipelines of in-order processors. The instructions execute in the dataflow order defined by the compiler. The cores use a modest wakeup mechanism, where an instruction becomes ready at the moment when all its inputs arrive. The instruction scheduler picks at most two register and two compute or memory instructions to execute at each cycle. The scheduler uses only the position of instructions in the AIB (age) to prioritize their execution, while also checking if there are structural hazards (ports or ALUs availability). The commit stage checks when the AIB has produced all its outputs that are encoded by the compiler and commits them to the registers and memory. While the AIB is committing its outputs, the next AIB is being fetched and decoded. When the commit phase is completed, the next AIB starts its execution.

Such execution of a single AIB per core limits the maximum performance of EDGE cores. Since each core contains the instruction window and reservation station buffers sized for 128 instructions, any AIB without that many instructions underutilizes the core's resources. Moreover, the atomic execution of the blocks limits the instructions to partially commit and release the available resources for new upcoming instructions. It incurs an overhead between the execution of two AIBs, either to read temporary results from the registers or to load new data from the memory. The speculative execution of multiple smaller AIBs per each core may avoid these overheads, but it requires a complex hardware support to control such speculative execution. This support is a trade-off between extra performance and core's complexity, which is not investigated in this work.

#### 2.2.2.1 Composing Cores

A key characteristic that distinguishes dynamic multicores from other architectures is the ability to dynamically adapt the architecture for a given workload

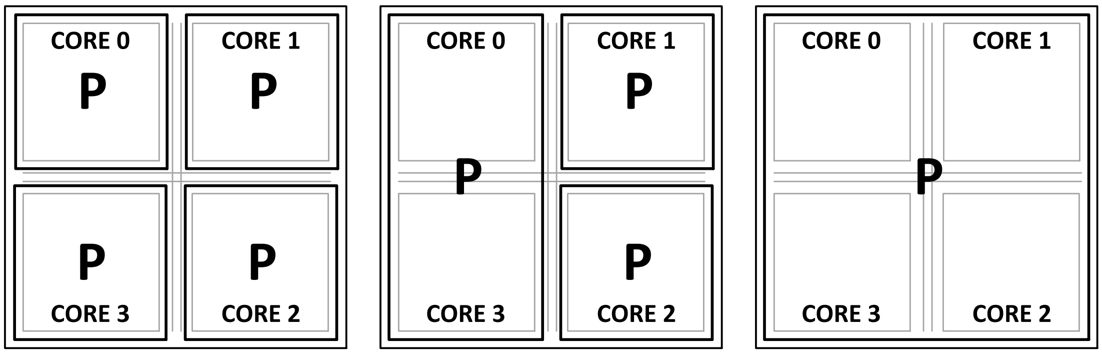


Figure 2.4: An example of dynamic configurations in a 4-core EDGE CMP. The CMP composes one or more cores into a wide-issue superscalar processor.

by composing cores. Rather than fixing the size and number of cores at design time, one or more *physical* cores can be grouped together at runtime to form bigger, more powerful *logical* cores. For example, serial portions of a workload can be handled by composing every physical core into one big logical processor that performs like a wide- fetch, decode and issue superscalar core. Alternatively, when ample thread-level parallelism is available, the same large logical processor can be decomposed so each physical processor can work independently to execute instruction AIBs from independent threads. Figure 2.4 shows the examples of various dynamic configurations of a 4-core EDGE CMP.

Each AIB is mapped to a single physical core and when composed into a logical core, the architecture uses additional physical cores to execute speculative (predicted) AIBs. Whenever the non-speculative AIB commits, it sends the commit signal along with the exit branch address to all other cores in the logical processor. Speculative AIBS on the correct path continue to execute, while AIBs on misspeculated paths are flushed. Composing is always done at block boundaries and is initiated by the runtime system. Decomposing cores makes the cores inactive and requires flushing the dirty lines of each cache being dropped from the logical processor, as well as updating the cache mapping. Dirty cache lines in the remaining cores are written back only when are evicted.

Logical cores interleave accesses to registers and memory among their physical cores, which provides a logical core with the fused memory resources of all the composed physical cores. For example, a logical core composed of two physical cores uses an additional bit of the address to choose between the two L1 caches and two LSQs, doubling the L1 cache and LSQ capacity. The register files are similarly interleaved, but since only 64 registers are exposed by the ISA, the additional register file capacity is power gated to reduce power consumption.

To enable the speculative execution of predicted AIBs, each core incorporates a next-block predictor [36]. The most of the predictor state is designed to be distributed among the cores, which increases the capacity of the predictor when more cores are composed. The microarchitecure of each core needs to buffer the results of speculative AIBs until they commit or flush. It requires buffering of memory store and register write instruction results. The LSQ buffers the stores and enables the execution of speculative memory instruction in the predicted AIBs. To support the speculative execution of register instructions, the register file in each core incorporates a write buffer, which holds register writes of speculative AIBs. When an AIB commits its buffered writes update the register file, which keeps the architectural state. Otherwise, if an AIB is flushed the writes are discarded. The write buffers have the complete knowledge of expected writes for each AIB, since the compiler encodes all register writes in the header of the AIB. By using the compiler information, the write buffers are capable of forwarding speculative registers values to speculative AIBs that attempt to read them. When the register file receives a read request it forwards the register value to the requesting core, if the requested value is available in register or the buffer. If the value is not available, the read request is stalled until the value arrives. As a result of this implementation, the speculative AIBs waiting for the temporary results of the previous AIBs can make progress without waiting for all the AIBs to become non-speculative and commit. It reduces the overheads between the execution of speculative AIBs and provides higher performance.

# Chapter 3

# EDGE Core Specialization for Power-Efficient Vector Execution

## 3.1 Overview

This chapter proposes a hardware specialization technique that enables vector execution of DLP workloads on a general purpose core. The specialization provides the advantages of classic vector processors (see Section 2.1.1) on the existing core's resources. While accelerating DLP workloads, the enabled vector execution increases the core performance and efficiency. We use a modest low power core based on an EDGE architecture (see Section 2.2.1) to implement our technique, called EVX. Unlike most existing DLP accelerators that utilize additional hardware and increase the complexity of mobile processors, EVX leverages the available resources of the EDGE core. It incurs minimal additions to dynamically specialize the EDGE core into an efficient DLP accelerator.

EVX leverages the existing core's compute logic to perform computation over large vector operands. The computation is based on the vector compute instructions, which statically encode compute dataflow. The dataflow computation allocates the compute resources and repeats over all the elements of the vector operands. EVX adds a dedicated vector control unit to execute vector memory instructions decoupled from the computation and increase the performance of EVX. The memory instructions utilize sophisticated access patterns, which enable vectorization of a wide range of DLP workloads. The vector control unit loads/stores the vector operands and transfers their elements to/from the compute logic that performs vector computation. As in classic vector architectures, EVX operates over configurable large vector operands. Instead of including a complex vector register file, EVX reconfigures a configurable part of the L1 data cache to hold such operands.

This chapter describes the core's microarchitecture modifications for EVX; the extension of the EDGE architecture with new vector instructions and registers; and the details of EVX implementation on an EDGE core. This is followed by an evaluation of performance, power and area of EVX, and a comparison with the baseline EDGE core as well as with EDGE and ARM Neon SIMD extensions. The chapter ends with related vector work and concluding remarks.

## 3.2 EVX Microarchitecture Modifications

Figure 3.1 shows the microarchitecture of a dual issue EDGE core and the same core with enabled EVX. The EVX modifications and additions that enable execution of vector instructions are:

- The existing cores compute hardware is banked to allow for higher vector computation throughput on the EDGE core. The banking is applied to instruction window and reservation stations, while distributing the functional units among the banks. Each of these banks behaves as a vector lane [3]. Vector compute instructions execute in each lane to increase the computation throughput with little complexity. An atomic block with the vector compute instructions (vector AIB) allocates the lanes and repeats such parallel and atomic execution of its vector instructions multiple times to compute large vector operands. The repeated executions of the vector instructions compute the vector operands slice-by-slice and the size of the slice matches the size of the vector lanes. This resembles the execution of vector instructions in classic vector processors with a few lanes, where each lane performs many pipelined operations to process large vector operands.
- A dedicated vector control unit (VCU) is incorporated to execute vector memory instructions and decouple them from the computation. The VCU

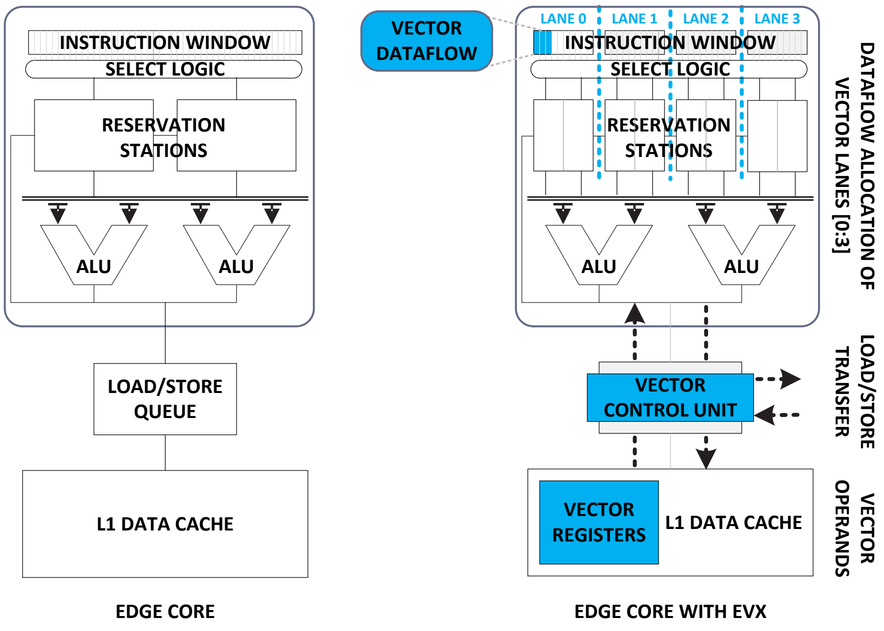


Figure 3.1: The microarchitecture details of dual issue EDGE core and the core with enabled EVX. EVX partitions compute resources to resemble parallel vector lanes; the dataflow computation executes multiple times in all vector lanes to process large vector operands; the dedicated vector control unit performs the execution of memory instructions and transfers the vector operands slice-by-slice between the compute lanes and vector registers in the data cache.

execution of the memory instructions has many advantages. The memory instructions operate over large vector operands, while the VCU pipelines many memory requests to load/store such operands. The pipelined execution of load/store instructions reduces memory latency. The VCU avoids load/store queue and issues memory requests directly to the memory hierarchy (L1, L2 or DRAM memory) to increase memory bandwidth. The vector memory instructions executed by the VCU bring up sophisticated addressing modes, which enables the vectorization of various DLP workloads. Decoupling compute and memory execution allows the vector load instructions to go ahead of the computation and eventually overlap the memory access time. The vector memory operands can be configurable large, which makes them inappropriately sized to be computed on the available core's hardware. To overcome that limitation, the large operands are divided into slices and then computed on the lanes of general purpose hardware, slice-by-slice. The VCU transfers the available slices of the vector operands to the vector lanes and thus manages the multiple executions of the vector compute instructions. The compute instructions perform their operations over different vector slices, when they are available instead of waiting for the entire vector operands. This way, EVX chain vector memory and compute instructions at low hardware complexity [29].

- A part of the L1 data cache memory is dynamically reconfigured into vector registers (VRs) to hold large vector operands. It allows EVX to operate over dynamically sized operands placed in the VRs, but also to keep the temporary results of vector computation.

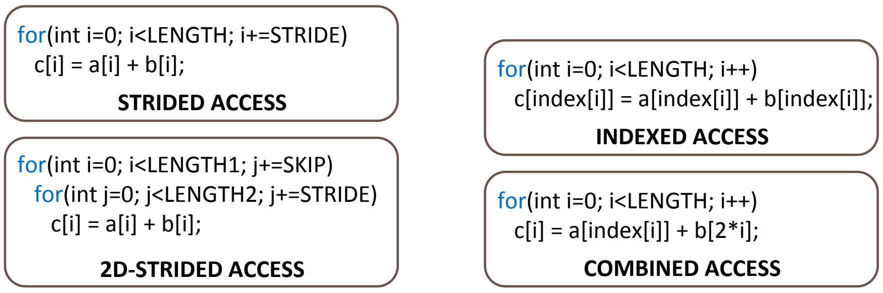
VCU is the only new hardware structure that needs to be added to the general purpose core to enable EVX. By loading/storing dynamically sized vector operands and transferring slices of such operands to/from the existing core's compute logic, the VCU allows for "EVX with two different modes". These modes resemble *register* and *streaming* vector execution. In register like mode, the VCU loads/stores *data arrays* and transfers them through the compute resources; EVX exploits temporal locality in the VRs, which in register mode hold memory operands as well as temporary compute results of long vector computations. In streaming like mode, the VCU loads/stores *data streams* through the VRs, which in this mode only buffer the input/output data between the memory hierarchy and compute logic. The microarchitecture is not particularly set for any of these so-called modes. The different ways of modifying a workload to use EVX tailor the mode to match workload demands (e.g. whether a workload performs a few vector compute operations directly and only over memory operands in the VRs or it performs the more complex computation over memory operands in the VRs and temporary compute results stored in the VRs).

## 3.3 EVX Architecture Extensions

To enable operations over vector operands, EVX extends the EDGE architecture. It adds three new classes of vector instructions to the EDGE ISA:

- *Vector compute instructions* partially compose vector AIBs, which perform vector computation. EVX provides a vector equivalent of each scalar EDGE logic/arithmetic instruction. A single bit of each instruction defines whether it performs scalar or vector operations. The vector compute instructions encode the dataflow in the same way as scalar EDGE instructions. Each vector compute instruction encodes one or two consumer instructions to which it sends the computation result. As opposed to a scalar EDGE instruction that performs a single operation, a vector compute instruction executes multiple times and performs multiple operations over many vector elements. The number of vector elements is dynamically configured and does not affect the instruction's encoding. The size of the elements varies between 8, 16, 32 and 64 bits, and it is configured as same as the number of elements. Such design does not require recompilation of vectorized applications when changing the implementation of EVX, as long as each implementation can operate over a configurable number of dynamically sized vector elements.

- *Vector register instructions* transfer slices of vector operands, by performing read (*vread*) or write (*vwrite*) vector register accesses. The register instructions complement the vector compute instructions when composing the vector AIBs. They connect the compute instructions with their input and output vector operands in the VRs, by encoding dataflow connections between them. The vector operands in the VRs can be either the source or destination operands of the vector memory instructions or the temporary vector results between the vector AIBs. The vread instructions read slices of the input vector operands from the encoded VRs and transfer the slices to the encoded consumer instructions; the consumers are the vector compute instructions that performs operations over the slices. The vwrite instructions write slices of the output vector operands to the encoded VRs; the



```
for(int i=0; i<LENGTH; i+=STRIDE)
  c[i] = a[i] + b[i];
```

**STRIDED ACCESS**

```
for(int i=0; i<LENGTH1; j+=SKIP)
  for(int j=0; j<LENGTH2; j+=STRIDE)
    c[i] = a[i] + b[i];
```

**2D-STRIDED ACCESS**

```
for(int i=0; i<LENGTH; i++)
  c[index[i]] = a[index[i]] + b[index[i]];
```

**INDEXED ACCESS**

```
for(int i=0; i<LENGTH; i++)
  c[i] = a[index[i]] + b[2*i];
```

**COMBINED ACCESS**

Figure 3.2: Examples of loops, which utilize diverse access patterns. The incorporated vector memory instructions support addressing modes for such patterns to increase the applicability of EVX.

slices are the results of the compute instructions, which encode the vwrite instruction as their consumers.

- *Vector memory instructions* load (*vload*) or store (*vstore*) configurable vector operands. The vector memory instructions do not encode the dataflow like the other scalar and vector EDGE instructions. Instead of dataflow, the memory instructions only encode the VRs that are the destination of load memory operations (vload) or source of store memory operations (vstore).

  To increase the performance of the vector execution, the vector memory instructions obey relaxed (incomplete) memory ordering. Such ordering assumes no memory dependency between the vector load/store instruction. The load instructions do not necessarily see the results of previous store instructions. The store instructions do not need to commit their results in order. The non-speculative store instructions can partially write their operands to memory, as soon as parts of their operand are available. The user (compiler/programmer) has to manage the execution of memory dependent instructions by separating them through memory barriers as in [28]. The barrier suspends the execution of the memory instructions after the barrier, until the memory instructions before the barrier commit their results. In this work we do not use nor further investigate such barriers, because we only accelerate various DLP workloads that process independent vectors.

The memory instructions support sophisticated addressing modes: *sequential*, *strided*, *2D-strided*, *indexed* and *masked* addressing modes. The instructions with sequential addressing mode (also called unit strided access) utilize only the base address of vectors to load/store consecutive vector elements. The strided addressing mode utilize a stride parameter to describe the distance between iterations of the inner loop and access the vector elements by using the distance value. The 2D-strided addressing mode along with the stride includes a skip parameter, which describes the distance between iterations of the outer loop [10]. The indexed and masked addressing modes utilizes the values of one VR like indices/mask for the vector elements (e.g. A[B[i]], where the values of B have to be in the VR).

EVX extends the architecture with support for partial vector reductions, which reduce large vectors to a subset of vector elements instead of a single scalar value. The reductions do not incorporate new instructions. They partially reduce vectors by using the already introduced vector compute instructions, while the existing scalar instructions finalize the reduction process and produce the single-value reduction result.

Along with the new instructions, EVX extends the EDGE architecture with new special-purpose registers:

- *Vector length register* specifies the number of elements in each vector operand, which is used by vector compute or memory instructions.
- *Element size register* defines the size of elements in vector operands. It complements the length register, when controlling the execution of vector compute and memory instructions, which dynamically configure the number and size of elements in their operands.
- *8 memory description registers* describe the type of each vector memory instruction and its access pattern for the decoupled execution. The type of instruction specifies whether it loads (*vload*) or stores (*vstore*) a vector operand. The access pattern includes: a vector base address and a combination of parameters such as stride, skip or index VR. Per each vector AIB, EVX supports the execution of maximum 8 vector memory instructions

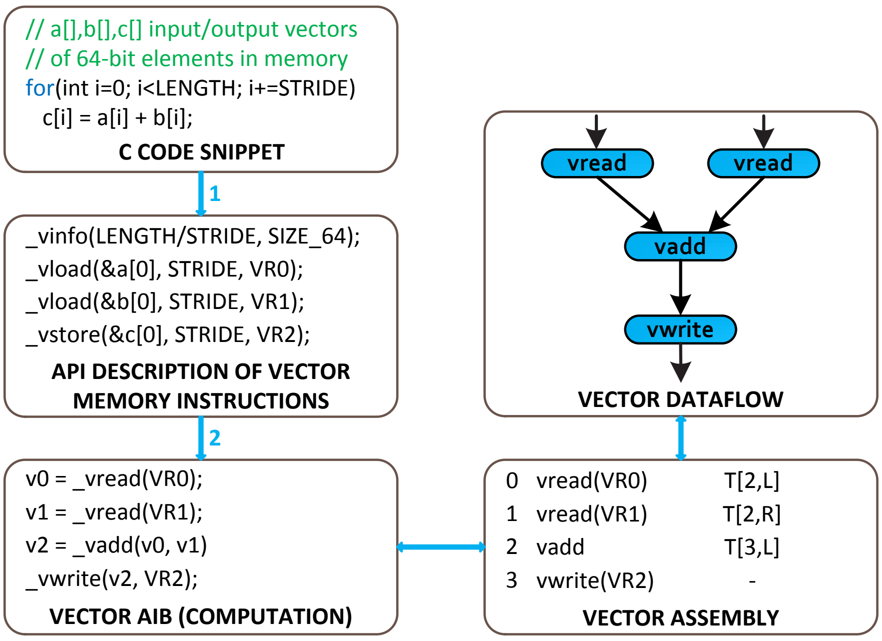
and 8 memory description registers are provided to describe each instructions. We chose such memory capabilities, because our analyses of various workloads find it suitable to vectorize most of their DLP code regions.

- *8 vector registers* hold configurable large vector operands. The number of VRs is chosen to provide at least one operand storage for each of the 8 memory instruction. The size of VRs is configurable, since they are made by dynamically reconfiguring a part of L1 data cache memory. It allows for keeping of extremely large vector operands in the VRs. To avoid the microarchitecture reconfiguration overhead, the architecture permits a fixed partitioning of the memory between the data cache and VRs. In that case, the VRs are allocated by a special system function and the VRs memory cannot be used for the caching before the VRs are deallocated. This thesis does not investigate this advanced architecture capability.

- *Reduction vector register* holds the initial reduction value (e.g. 0 when summing the values) and the reduction result. It is VR8, since the numbers 0:7 are reserved for conventional VRs. The reduction register is sized to hold one slice of vector operand; and the size of vector slice depends on the particular EVX implementation.

- *4 dedicated scalar registers* hold scalar values, which are used to facilitate mixed vector-scalar operations. They are encoded as same as the VRs, by using the numbers 9:12.

The EVX architecture extension provides a very comprehensive set of vector instructions and addressing modes to enable vectorizaion of a wide range of DLP workloads. We believe that a modern compiler could exploit the advantages of the EVX extension to auto-vectorize most of the workloads [37]. However, in this work we utilize an in-house EDGE C/C++ compiler [69] that does not enable the auto-vectorization option. To write the vectorized workloads, we have extended the compiler with new vector instructions. We have developed an API that describes the decoupled execution of vector memory instructions, initializes values of the scalar registers as well as vector length and element size registers. The API is translated into the register instructions, which write the values into



```
// a[],b[],c[] input/output vectors
// of 64-bit elements in memory
for(int i=0; i<LENGTH; i+=STRIDE)
  c[i] = a[i] + b[i];
C CODE SNIPPET
```

```
_vinfo(LENGTH/STRIDE, SIZE_64);
_vload(&a[0], STRIDE, VR0);
_vload(&b[0], STRIDE, VR1);
_vstore(&c[0], STRIDE, VR2);
API DESCRIPTION OF VECTOR
MEMORY INSTRUCTIONS
```

```
v0 = _vread(VR0);
v1 = _vread(VR1);
v2 = _vadd(v0, v1)
_vwrite(v2, VR2);
VECTOR AIB (COMPUTATION)
```

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

Figure 3.3: An example of the loop vectorization for EVX. Decoupled vector memory execution is described though the provided API. The vector AIB is written by using intrinsics of vector compute and register instructions, which later encode dataflow of the vector execution.

associated control registers. We have also provided support for various intrinsics that describe vector AIBs. The AIB consist of vector compute and register instructions. The AIB reads input slices of the vector operands from the VRs, performs the atomic computation over them and writes the output slices to the VRs. The atomic vector computation is repeated over different slices of vector operands until computing the entire operands. The programmer writes the vectorized code by describing the vector memory instructions followed by the vector AIB that performs the computation.

The Figure 3.3 shows a simple vectorizable loop and its step-by-step vectorization. The first step of the vectorization describes vector memory instruction that load/store the loop's memory operands. The memory description initializes the memory control registers, which orchestrates the execution of the vector memory instructions. The API functions are used to describe the memory execution, as well as to initialize the vector length and size of vector elements. Each vector memory instructions is described with: type (load or store), start address ('a', 'b' or 'c'), stride between the elements (STRIDE) and the associated vector register (VRx). All instructions share the vector length (LENGTH/STRIDE, because iterations do not process consecutive vector elements) and size of vector elements (SIZE_64). The second step of the vectorization describes the vector AIBs that performs the loop's computation by repeating atomic executions of its vector instructions. The AIB is written with the provided intrinsics of vector instructions. The AIB consists of the vector compute (vadd) and register(vread, vwrite) instructions. Figure 3.3 shows the assembly translation of the vector AIB and its dataflow encoded in the vector EDGE instructions. The vread instructions read slices of the vector operands from the encoded VRs (VR0, VR1) and forward them to the left and right operand of the vadd instruction respectively. The vadd performs *add* operation over the input slices and forwards its result to the vwrite instruction that writes it to the encoded VR2. The AIBs repeats the atomic execution of its vector instructions until computing the configured number of vector elements (LENGTH).

As it is shown in the example from Figure 3.3, the vector memory instructions are described apart from the vector AIBs, by writing their description to the memory control registers. The vector AIBs are composed of the vector compute and register instructions; and they also include a small set of scalar instructions, which control the further flow of execution (e.g jump to the next block). To differentiate the vector AIBs from the scalar EDGE AIBs, a single bit in the block header encodes if a block contains the vector instructions or not.

## 3.4 EVX Implementation

EVX computation is enabled on a block-by-block basis. The memory instructions execute in the VCU, decoupled from the vector computation; and they can start the execution as soon as they have their description available in the memory control registers. The vector AIBs perform computation of vector operands in the lanes of the existing core's compute logic; and utilize the VCU only to read and write slices of the vector operands from/to the VRs. Such EVX design decouples the execution of the vector compute, register and memory instructions and we discuss the implementation of the execution for each class of the instructions.

In the rest of this section, we first explain the execution of the vector AIBs in the core pipeline. Next, we provide a detailed description of the additional vector control unit and its logical structures, which decouples the execution of the vector memory and register instructions. We further discuss the reconfiguration of L1 data cache resources into the VRs, show an example of execution in the EVX enabled core and finally describe the support for sophisticated EVX addressing modes and partial reductions.

### 3.4.1 Vector AIBs in Core Pipeline

The vector AIBs are fetched and decoded like the scalar AIBs of EDGE instructions, see Chapter 2. The EVX design allows for executing a mix of scalar and vector instructions. The scalar instructions execute in one vector lane, once per fetched AIB. On the contrary, the vector instructions execute in each lane of the compute logic and repeat such execution by operating over different slices of vector operands. Such multi-lane issue of the vector instructions limits the size of the vector AIBs to the size of instruction window in one lane (bank). For example, an implementation of EVX with four lanes limits the size of the vector AIBs to 32 instead of from 128 instructions (the 128-entry instruction window is partitioned to 4 banks with 32 instruction in each bank).

In this work we do actually chose to have four vector lanes to simplify the transfer of the vector slices between the VRs and the compute lanes. The four lanes enable operations over 256-bit vector slices (four 64-bit lanes), which is the size of one cache line; please note that a number of cache lines is used for the VRs. It allows for simple read/write access to the VRs, while transferring the entire cache lines between the registers and the four lanes. Each line of the VR holds one slice of the vector operand and each slice has a specific number of vector elements that depends on their size. For example, a 256-bit wide vector slice contains eight 32-bit elements or thirty-two 8-bit elements. To perform the computation over such slices, the vector compute instructions leverage the existing 64-bit ALUs that support the 8-, 16-, 32- or 64-bit sub-word SIMD operations. In this way,

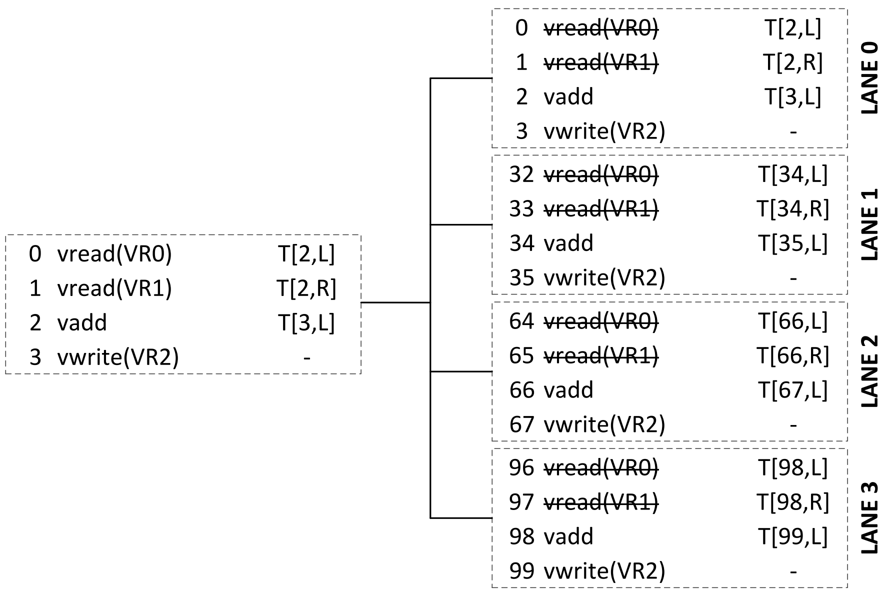


Figure 3.4: The EVX 4-lane issue of vector compute instructions. The instructions are replicated in each lane to increase the computation throughput.

the execution of one vector compute instruction in four vector lanes can processes up to sixty-four 8-bit elements of its vector operands.

**Fetch and Decode:** When an AIB is fetched from the L1 instruction cache, the block header is examined to determine if the block contains the vector instructions. If so, then all instructions are mapped to the first lane of the instruction window and all the vector instruction are replicated across the lanes. The replication of vector instructions involves adjusting the instruction targets by a fixed offset to specify the consumer reservation station in the same lane (+32, +64 and +96 for the lanes 1, 2 and 3 respectively). While decoding the instructions, the vector register instructions (vread/vwrite) are forwarded to the VCU, because it performs the register operations. Vector read instructions only encode the dataflow connection between input VRs and consumer compute vector instructions; and after forwarding to the VCU, the vector reads are subsequently ignored by the issue logic (nullified). On the contrary, the vector write instructions encode the output VRs and they need to forward the computed vector slices from the producer compute instructions to the VCU; the vector writes leverage the issue logic and remain active in the instruction window.

In the example shown in Figure 3.4 instructions 0 to 3 are vector instructions and their instruction number, mnemonics, and targets are presented. As these four vector instructions are decoded, they are replicated across the lanes and the replicated targets are updated to encode consumer instructions in the same lane. For example, instruction 2 is a vector add that targets the left operand of instruction 3 (T[3,L]). The vector add is replicated in lane 2 as instruction 34 (2 plus a fixed offset of 32) and the replicated target is updated to encode the target 35 (T[35,L]). Similarly, the instruction 35 is the replicated instruction 3 in lane 2. The instructions 0 and 1 are vector read instructions and instruction 3 is vector write. As these instructions are decoded, they are forwarded to the VCU.

**Execute and Memory:** Instructions execute out-of-order in dataflow fashion according to their dependencies. 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. In the dual-issue EDGE core that we use as our baseline, it takes two cycles to issue one compute instruction in all lanes. Since four vector lanes share two ALUs, instructions from lanes 0 and 1 issue first, followed by lanes 2 and 3. The vector write instruction's format encodes the destination VR and has a single input operand for the data. Vector write instructions execute only in lane 3 (lanes 0 to 2 are ignored by the issue logic), because the issue schedule makes it the last lane in which the write instruction receives its input data. When the write in lane 3 receives its data, the data from all the lanes is forwarded to the VCU that writes it to the VR. Simultaneously with the execution of vector compute instruction in the pipeline, the VCU executes decoupled vector memory instructions and transfers slices of the vector operands between the VRs and reservations stations of the compute instructions. Select logic repeats the execution of the resident compute instructions, when new vector slices are available and the instructions have their operands ready in the reservation stations. This way, each vector compute instruction executes as many times as necessary to process all the elements of its vector operands.

In the example in Figure 3.4 the VCU executes vector read instructions 0 and 1, by reading 256-bit vector slices from VR0 and VR1 and transferring them to the left and right reservation stations of the vector add in four lanes (64 bits per lane respectively). Once the vector adds receive their input operands, the adds

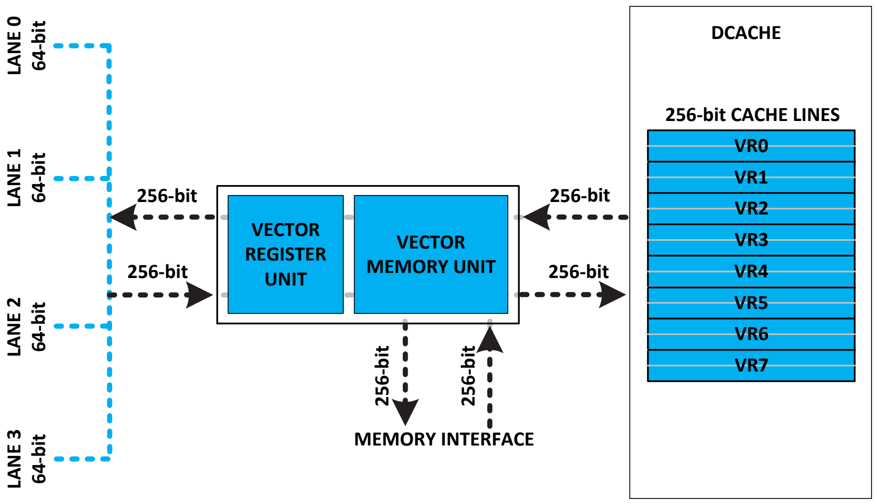


Figure 3.5: The vector control unit executes the new vector memory and register instructions. It has a dedicated sub-unit for each class of the instructions.

from lane 0 and 1 issue to the ALUs followed by the adds from lanes 2 and 3. The adds in lanes 0 and 1 will complete first, while sending their results to the vector writes in the same lanes. But only the vector write in lane 3 will execute after it receives its input operand; and at that time it will forward the input operands from each lane, by sending a 256-bit slice to the VCU.

**Commit:** Compared to scalar EDGE AIBs that commit when they produce all their scalar results and resolve the next block PC, the vector AIBs have one additional restriction on commit. The vector compute instructions in the vector AIBs execute multiple times to process large vector operands. The executions are controlled by the VCU that transfers slices of the vector operands from the VRs to the compute lanes and back. This way, the VCU commits the results for each atomic execution of the vector compute instructions to the VRs; and once all the executions are complete, the VCU signals the control logic that the vector AIBs are ready to commit.

### 3.4.2 Vector Control Unit

The VCU( shown in Figure 3.5) is a dedicated hardware logic added to the EDGE core to enable EVX on its general purpose resources. The VCU is composed of two logical sub-units. The first sub-unit is the *Vector Memory Unit* (VMU). It loads/stores large vector operands in a similar fashion to the memory units in classic vector processors [10, 16, 61, 66]. The major difference is the execution of vector memory instructions in slices to allow for slice-by-slice computation of the vector operands on the existing general purpose hardware. The second sub-unit is the *Vector Register Unit* (VRU). It enables the computation over the large vector operands by transferring their slices between the VRs and the compute vector lanes in the core pipeline.

#### 3.4.2.1 Vector Memory Unit

The VMU executes vector load and store instructions in a decoupled fashion. It exploits programmable memory access patterns of DLP workloads and with minimal hardware additions performs vector memory execution. Each vector memory instruction uses one VR to hold its input/output memory operand and one memory control register that describes its access pattern by using one of the supported addressing modes. The size of memory operands is dynamically configured. Instead of computing such large operands at once, the 256-bit cache line size is used to slice the operands and compute them slice-by-slice in the four vector lanes of the general purpose hardware. To improve the performance of such computation, the VMU issues the memory requests for a slice of one memory instruction and then it executes the next instruction in round-robin fashion.

The decoupled execution of vector memory instructions at the slice granularity enables overlapped execution of vector memory and compute instructions. The memory instructions load or store slices of 256 bits and the transfer logic forwards the slices in-order, once they are ready to the reservation stations of the compute instructions. Instead of waiting for the memory instructions to complete, the compute instructions can execute whenever they have ready slices of their operands in the reservation stations. Such overlapped execution resembles the chaining of conventional vector instructions, where the output of one instruction is bypassed (chained) to the input of a following instruction (e.g. one load/store unit bypasses its output to one compute unit). The slice-by-slice execution of vector memory instruction in a round robin fashion by the single VMU is yet more

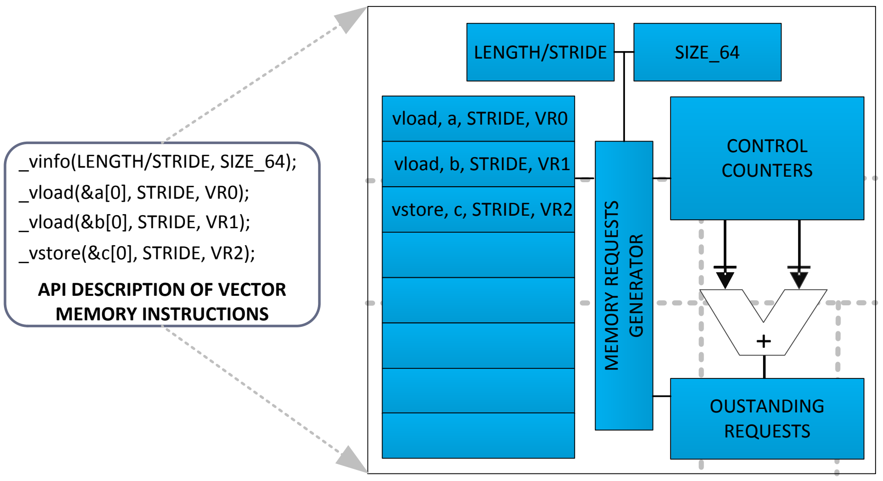

_vinfo(LENGTH/STRIDE, SIZE_64);
_vload(&a[0], STRIDE, VR0);
_vload(&b[0], STRIDE, VR1);
_vstore(&c[0], STRIDE, VR2);
API DESCRIPTION OF VECTOR MEMORY INSTRUCTIONS
LENGTH/STRIDE
SIZE_64
vload, a, STRIDE, VR0
vload, b, STRIDE, VR1
vstore, c, STRIDE, VR2
MEMORY REQUESTS GENERATOR
CONTROL COUNTERS
+
OUSTANDING REQUESTS

Figure 3.6: The vector memory unit executes the vector load and store instructions in a decoupled fashion by using the information of their access patterns.

flexible for chaining memory and compute instructions. It enables bypassing the values of each memory instruction to different compute instructions through the vector register unit, as explained later.

The VMU (shown in Figure 3.6) executes vector memory instruction, which obey the relaxed memory ordering. Therefore, the VMU does not require a complex load/store queue to dynamically discover memory dependencies and forward data between dependent load-store instructions. Instead of comparing the addresses of each load request to the addresses of each outstanding store request, the VMU issues memory requests directly to the memory hierarchy. This simplifies the VMU implementation and improves the memory efficiency by allowing for a large number of outstanding memory requests. To save the requests, the VMU incorporates a lightweight vector outstanding requests handler, which saves per request necessary information to handle their responses. To execute vector memory instructions in a round robin fashion, the VMU incorporates control counters, which keep track of vector elements processed by each memory instruction. A few more structures are added to build the VMU. We further describe each structure with more details:

*Queue of vector memory instructions* holds active memory instructions. The queue is visible in the EVX architecture as *the memory description registers*. Each of 8 memory description registers (an entry of the queue) can hold the type and memory access pattern associated with one active memory instruction as shown in the example in Figure 3.6. The description registers are initialized by executing a set of scalar (register) instructions that write the description of the active memory instructions into the associated description registers; and their execution must precede the execution of the vector AIB that utilizes their operands.

*Length register* configures the number of vector elements in the vector operands. The register is visible in the EVX architecture and its initialization must precede the vector AIB that utilizes the length value. The length value is utilized to control the execution of the vector memory instructions by counting the number of loaded/stored elements.

*Size register* configures the size of vector elements in the vector operands. As the length register, the size register is visible in the EVX architecture and its initialization must precede the vector AIB that utilizes the size value. It is utilized by the VMU, when generating the load/store requests for the elements of vector operands.

*Control Counters* are used to track the vector elements processed by vector memory instructions. Each memory instruction has a counter that points to the next vector element in the operand to be processed, as well as to its position in the VRs. When the element is processed (issued load/store memory requests), the counter is incremented. When all the counters reach the total number of vector elements to be processed (value of the length register), the vector processing is complete and the notification signal is sent to the core's control unit.

*Memory request generator* calculates the addresses of the vector elements and issues memory requests. It incorporates an extra adder unit for the address calculation and one address register per vector memory instruction. The adder is shared between the memory instructions in a round-robin fashion. It either increments a current vector address from the address register with a stride value (unit stride, non-unit stride or skip) for an instruction with strided based access pattern; or adds an index value to a vector base address for an instruction with indexed access pattern. At each cycle, the generator is able to produce a new address and issue a new load/store request to memory. After a 256-bit slice of the load/store operands is processed, the generator picks the next ready instruction from the queue of vector memory instructions to execute in the next cycles. To improve the EVX performance and power efficiency, the generator produces a single request per cache line of a vector operand with sequential memory access. In this case, the generator increments the address register with the size of a cache line (32 in our implementation with 256-bit cache lines), while the counter of the instruction elements increments its value by the number of elements in the cache line (e.g. 8 for 32 bit elements).

The generator can send memory requests either to the L1/L2 cache memory or the DRAM memory controller. In this work we chose to send the requests to the L2 cache banks to avoid having two copies of the vector data in the L1 cache (reconfigured VRs memory and regular L1 data cache memory). Such decision enables higher memory bandwidth for vector memory instructions by issuing memory requests directly to multiple banks of the L2 cache memory.

*Outstanding requests handler* saves the necessary information for each load-/store request and handles the responses arriving from the L2 cache. The outstanding requests handler does not consider the general coalescing between different vector memory instructions to simplify its implementation; and the relaxed memory ordering allows for such implementation. The handler considers the coalescing that occurs only when the addresses from consecutive memory requests correspond to the same cache line. It avoids multiple accesses to the same cache line in a case of strided accesses with small stride values (e.g. one memory access instead of four when loading/storing four 32-bit elements with stride of 2). Such simplified handler does not require a CAM memory to hold per request addresses neither it enables accesses to its entries by searching for the response addresses. To avoid use of a CAM memory, the vector memory instructions requests and responses are appended with the handler entry number. In the case of coalesced requests of the same cache line, there is a linked chain of the entries and they are processed sequentially, when the response arrives. When issuing a load request, the requests handler keeps only the position of the vector data in the requested cache line, the VR and its position (pointed by the instruction counter). When a load response arrives, the corresponding entry of the handler is accessed by using the entry number included in the request and its response. The requested data from the cache line attached to the response is read and placed into the corresponding position of the destination VR. Vector store requests leverage the handler in a similar fashion. Instead of waiting for the requested data like load requests, store requests wait for the store acknowledgments, which notify the store instructions that their result have been committed to memory.

In this section we have briefly described the implementation of VMU and its decoupled execution of vector memory instructions. To show the advantages of implemented decoupling, we can qualitatively compare the decoupled memory execution against the classic memory execution in today's commercial general purpose processors. Today's processors execute speculative (predicted) memory instructions out-of-order, ahead of the computation to saturate memory bandwidth and tolerate memory latency. The number of outstanding memory requests in these processors depends on the number of outstanding memory instructions; and it is limited by the size of the instruction window, the number of free physical registers and the number of entries in a complex load/store queue that checks for memory dependencies. On the other hand, the decoupled memory execution in the VMU does not have such limitations. The VMU executes vector memory instructions, where each instruction loads/stores large vector operands and issues a large number of memory requests. Instead of issuing memory requests through the load/store queue as it performs the classic memory execution, the VMU avoids such queue and issues memory requests directly to the memory hierarchy (L2 cache banks). It enables the VMU to sustain more outstanding memory requests, which in return saturates the memory bandwidth more efficiently and increases the performance of the entire system.

#### 3.4.2.2 Vector Register Unit

The VRU (shown in Figure 3.7) manages the transfer of 256-bit slices of the vector operands in the VRs to/from the allocated compute vector lanes. By transferring the slices to the compute logic and back, the VRU controls the repeated executions of the resident vector compute instructions. To enable such transfer operations, the VRU utilizes the following structures:

*Counter of compute iterations* keeps track of the number of executions of vector compute instructions. The counter is incremented per each computed

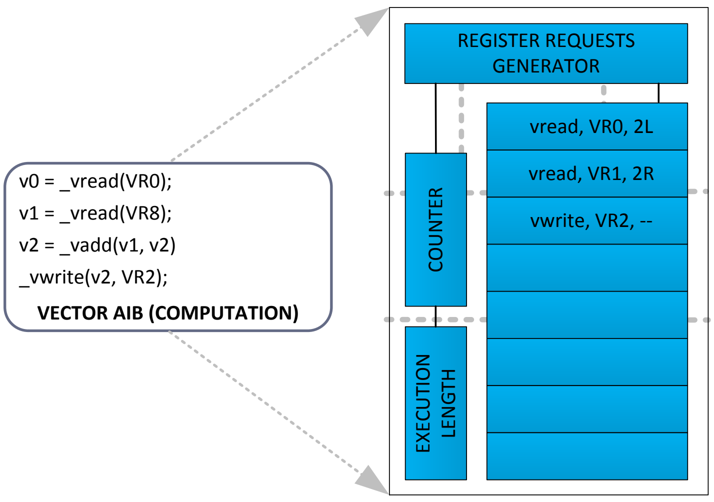


Figure 3.7: The vector register unit executes the incorporated vector read and write instructions. They transfer slices of the vector operands between the VRs and the vector lanes.

slice of the vector operands (when all read/write register instructions execute). It is used to point to the current slice of each vector operand that is accessed by a vread or vwrite instruction, as well as to count the number of executions.

*Execution length register* configures the total number of atomic executions of the vector instructions in the vector AIB. This register is not visible in the EVX architecture extension. Its value is initialized with the vector length (number of vector elements), when writing the length value into the length register of the VMU. The length value is later modified into the number of atomic executions, by discarding the least significant bits of the register. The actual number of the discarded bits depends on the size of vector elements. The VRU utilizes the execution length register to detect when the entire vector operands are processed in order to stop further transferring of vector slices.

*Queue of vector register instructions* holds active vector register instructions (vread/vwrite). Unlike the queue of vector memory instructions, the queue of vector register instructions is not visible in the architecture ISA. It is initialized by forwarding the register instructions when a vector AIB is fetched and decoded (see Section 3.4.1). Each register instruction is defined by the instruction type (read/write), target (in the case of a *vread*) and the associated VR as shown in the example in Figure 3.7. The instruction connects the vector operand in the VRs to the input or output of a vector compute instruction. When the VR holds the source or destination operand of a vector memory instruction, the register instruction chains either one vector load to one vector compute instruction (*vread*) or one vector compute to one vector store instruction (*vwrite*). To enable chaining of each memory instruction, the queue of vector register instructions is sized equally as the the queue of vector memory instructions; and it holds up to 8 vector register instructions.

*Register request generator* produces read/write requests, while managing the repeated executions of vector compute instructions. It is able to produce one request at each cycle. On a read, the generator checks if the current slice of the input vector operand is available, reads it from the associated VR and distributes it to the vector lanes. On a write, the generator forwards the current slice of the output vector operand that is forwarded from the compute lanes to the associated VR. Once all vector register instructions in the queue have fired, one atomic execution of the vector instructions in the current AIB has completed. The generator repeats the producing of read and write requests until the instructions performs all its executions and compute all the elements of its vector operands; at this moment, the counter of compute iterations reaches the value of the length register and the generator stops.

### 3.4.3 Vector Registers

Rather than introducing additional hardware structures, our implementation of VRs (shown in Figure 3.8) repurposes a configurable part of the L1 data cache memory. Such implementation avoids the inclusion of a vector register file, unlike conventional vector architectures which require including one in the design; and its structure is typically large and complex. Our VRs in the L1 data cache have simplified accesses to the cache data arrays, which do not require any associative address lookup. Since tags are not compared, the accesses to the VRs are direct as in software managed memories[14, 54]. Each VR allocates a previously configured number of cache lines. The lines are invalidated and flushed if needed before starting the vector execution. The cache reconfiguration is done sequentially,

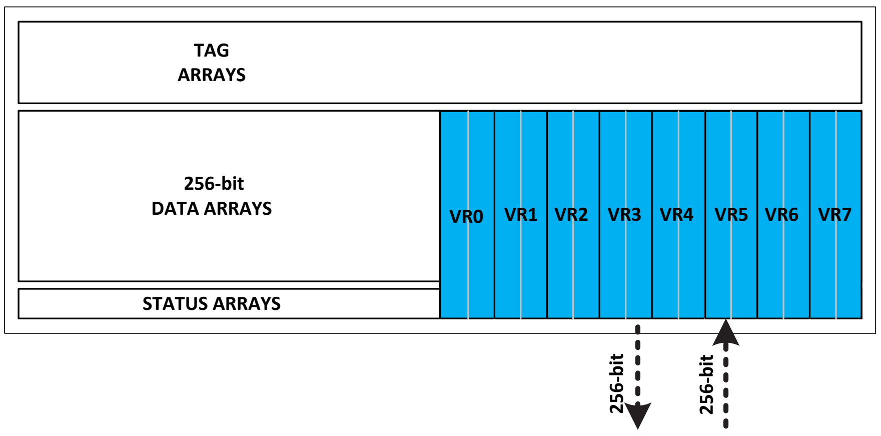


Figure 3.8: The VRs allocate a part of the L1 data cache. Each register uses only the data and status cache arrays, and they are accessed directly without tag comparisons.

one cache line per cycle and the reconfiguration impact is studied in Section 3.6. When the applications are highly parallel, the VRs can be allocated just once at the beginning of the application to minimize the reconfiguration impact. Any cache line not controlled by the VRs is still utilized as a regular data cache line. When the processor does not perform vector processing, all the lines of the L1 data cache can be used for data caching.

The number of the VRs is 8, which is defined by the EVX architecture. The size of the VRs can be dynamically configured to match workload requirements. When EVX performs in register mode, the VRs along with memory operands store the temporary vector results of EVX processing that spans multiple vector AIBs; and the maximum vector length of the EVX computation is limited by the maximum size of the L1 data cache memory reserved per each VR. When EVX performs in streaming mode, the VRs behave as circular streaming buffers and they are utilized only to buffer memory operands. Unlike register mode, streaming mode allows for computing of an unlimited number of the vector elements by streaming them through the compute lanes.

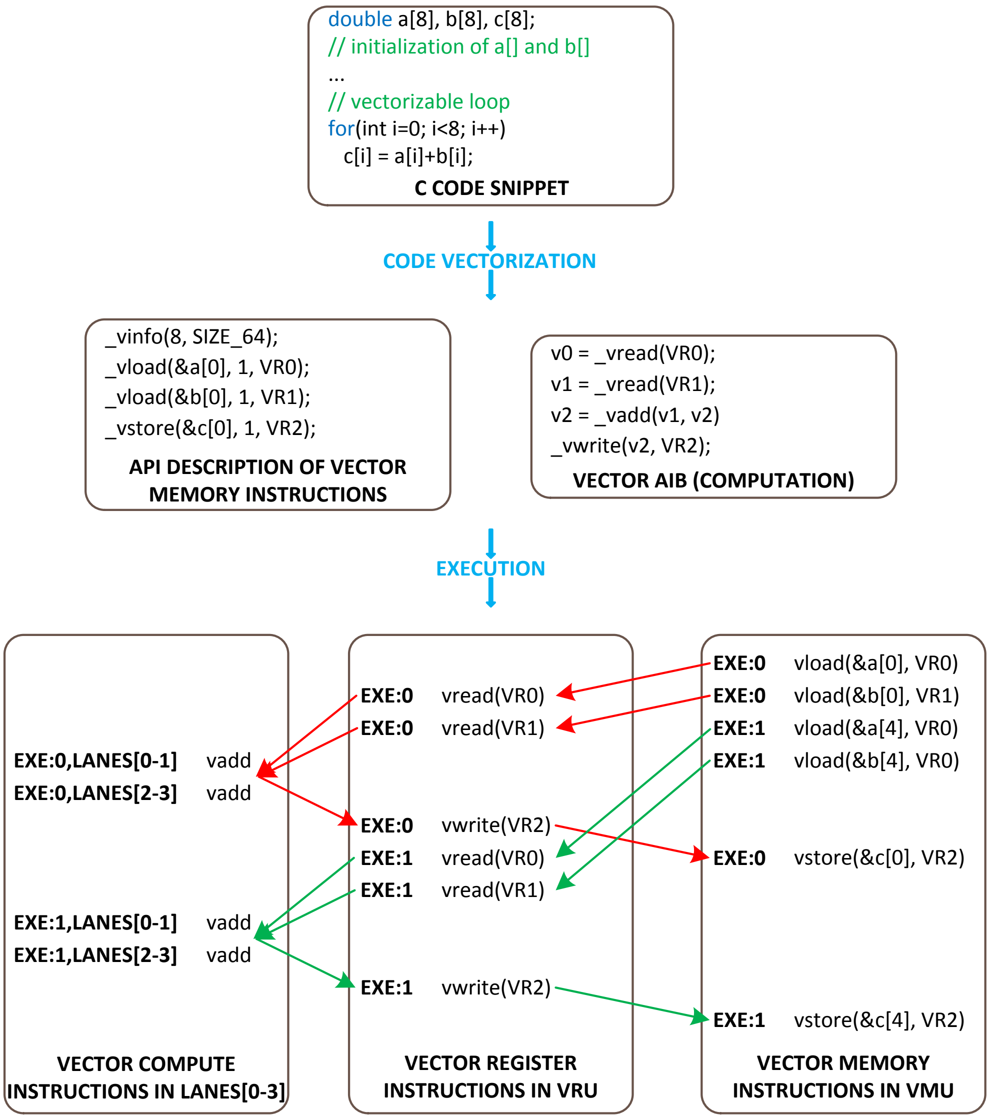


Figure 3.9: An example of EVX. The vector compute, register and memory instructions execute in a decoupled fashion. All the vector instructions repeat their executions 2 times (red arrows show the flow of **EX:0** and green arrows of **EX:1**) over 256-bit slices until the vectors of eight 64-bit elements are processed.

### 3.4.4 EVX Example

Figure 3.9 shows an execution example of EVX. The example includes the code of a simple vectorizable loop, its vectorization and the execution of vector instructions. The loop performs the addition of 8 consecutive vector elements from vectors a and b and stores their result sequentially in vector c. The vectorization is similar to the vectorization shown in Figure 3.3. It describes the patterns for the vector memory instructions and the computation of the vector AIB. The execution of all the vector instructions is divided into three independent streams: compute, register and memory; and each stream utilizes the different execution substrate specialized for one class of the vector instructions.

The vector memory instructions perform the loop's memory execution in the VMU by using the initialized memory patterns. In this example we assume that the VMU initialization has been performed and do not show it in the execution diagram. The memory instructions execute slice-by-slice in round robin fashion; the slice size matches the size of one 256-bit cache line. The vload instructions execute as soon as their patterns are initialized. The vstore instructions execute when the slices of their data operands becomes available. The VMU first picks the vload instructions to execute. They sequentially load vectors a and b into VR0 and VR1. As in classic vector processors with caches [17], the VMU loads the entire 256-bit cache lines and alternates between instructions at each line. Similarly, the VMU stores cache lines from VR2 to vector c, when the 256-bit slices arrive to the register. Each load and store instruction issues two memory requests to load or store eight 64-bit elements of its vector operand (while assuming the input/output vectors align to the cache line size).

The vector AIB performs the loop's computation over slices of the loop's memory operands. The AIB is composed of vread/vwrite (register) and vadd (compute) instructions. The vread instructions chain the outputs of the vload instructions (VR0 and VR1) to the inputs of the vadd instruction. Similarly, the vwrite instruction chains the output of the vadd instruction to the input of the vstore instructions (VR2). The register vread/vwrite instructions execute in the VRU and the vadd instruction in the lanes of existing core's compute logic. The vread instructions read 256-bit slices of the vector operands from VR0 and VR1. As soon as they are available, the slices are read and transferred to the left and right reservation stations of the vadd instruction. The vadd instruction executes whenever it has new slices of its operands ready. Its single execution performs the addition of 256-bit slices and produces a 256-bit result. By leveraging two 64-bit ALUs, the execution of the vadd instruction takes two consecutive cycles (this is assuming a very specific implementation with 1-cycle adders). The vadd forwards its result to the vwrite instruction, which writes it into VR2 to be stored to the memory by the vstore instruction. The vector AIB repeats the execution of its vector until the configured number of vector elements is processed. In this example, the vector instructions executes two times to process the vector operands of eight 64-bit elements on the four 64-bit vector lanes. The dataflow between the compute, register and memory instructions is shown with red arrows for the first execution of the vector instructions (*EX:1*) and green arrows for its second execution (*EX:1*). When all the instructions finish their executions, the vector AIB commits and deallocates the lanes of the compute core's resources.

### 3.4.5 Implementation of Indexed Memory Accesses

The VMU enables indexed memory accesses. The memory instructions with indexed accesses use one VR to hold vector data operand and one VR to hold index values for address calculation. The VR with index values can be shared across multiple memory instructions. The indices can be either computed by using vector operations or loaded from memory. The VMU generates independent load/store memory requests per each vector element, by summing index values and the vector base address in order to calculate the addresses of the indexed elements. Figure 3.10 shows an example of the VMU indexed access. VR0 holds the indices and VR1 holds a memory operand. The VMU uses the indices in VR0 to gather vector elements from the memory into VR1 (indexed load) or to scatter them from VR1 into the memory (indexed store).

This implementation of indexed accesses can be easily extended to support the double indexed accesses. For example, the output of one indexed access can be the input of another access (A[B[C[i]]]). The instructions with the double indexed access utilize two extra VRs to hold their indices. In this thesis, we do not vectorize applications with such access patterns nor further investigate them.

### 3.4.6 Implementation of Masked Memory Accesses

Similarly to the implemented indexed access, we implement the execution of vector memory instruction with masked accesses. One VR holds mask values and

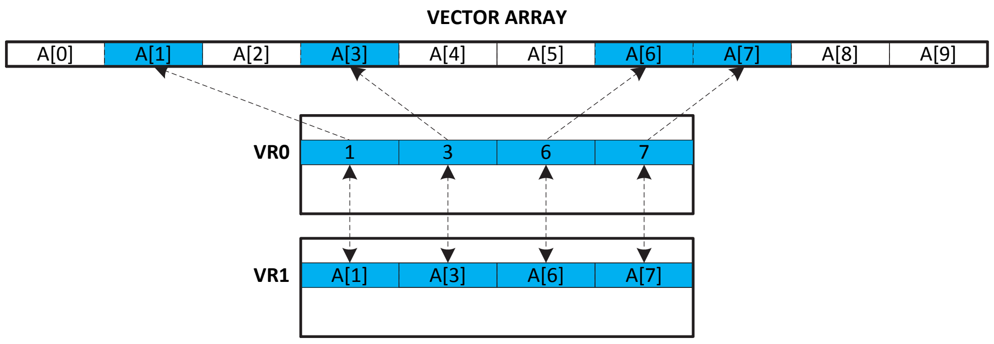


Figure 3.10: An example of the indexed vector accesses. VR0 holds indices and VR1 the vector indexed elements, which can be gathered from/scattered to memory.

the memory instructions perform independent memory accesses for each element, which corresponding mask has a positive value. The masked execution is necessary to vectorize loops with conditional code. Before executing such loops, the condition has to be evaluated per each vector iteration in order to produce a vector mask (vector of boolean values that specifies whether the condition is satisfied or not). The vector mask has to be written into the VR, which is used by the vector memory instructions with masked accesses. The VMU issues memory requests based on the mask values in the VR. It processes conditional vector accesses in a fully decoupled fashion, while loading vector data for which the condition is satisfied (mask value is "true") and storing their results to memory in the same way. The vector lanes are not aware of the conditional execution. They process only the elements of vectors on a correct conditional path, which increases the efficiency of vector computation.

The masked memory accesses allow for masked vector execution. Such implementation remarkably differs from the traditional implementations of masked vector execution in classic vector processors. Instead of utilizing the mask register to enable/disable compute operations over different vector elements of the vector operands like in traditional designs, our implementation loads/stores the vector elements depending on the corresponding values in the mask register. It avoids the hardware additions, which are required to mask the compute operations (e.g. a dedicated mask register that controls the sophisticated issue of the compute vector operations). While simplifying the implementation, the masking of memory operations per each vector element can have positive as well as negative impact on the efficiency of the masked vector execution. On one hand, when only a few elements of the vector operands have "true" corresponding mask values, this simplified design avoids the unnecessary memory accesses by masking vector memory requests. On the other hand, when only a few elements of the vector operands have "false" corresponding mask values, this design incurs the additional memory accesses by issuing independent load/store requests for different elements of the vector operands. Considering its implementation costs along with diverse efficiency impacts, in this thesis we chose the design that masks the vector memory requests and we do not additionally investigate the traditional approaches that mask the compute operations.

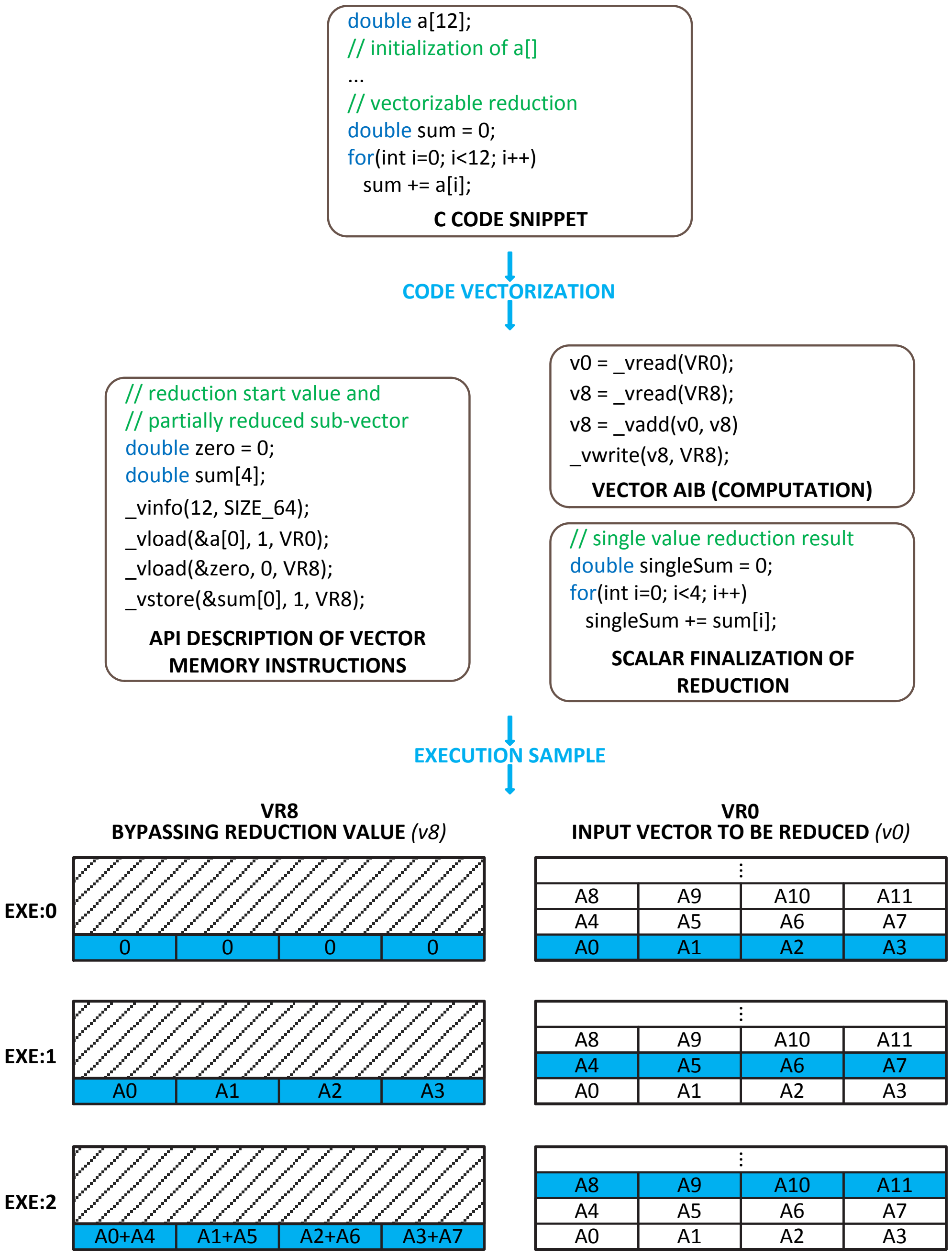
### 3.4.7 Implementation of Reductions

The reductions are implemented at low cost, by using the EVX microarchitecture without special hardware additions nor modifications. The implementation of reductions basically utilizes one vector AIB to perform operations that partially reduce one or more input vectors. The repeated executions of the vector compute instructions in the AIB over different 256-bit vector slices reduce the large input vectors by passing (chaining) the output slice of one execution as the input slice of the next execution. The last execution of the vector compute instructions produces a 256-bit result; and that is the size of the vector slices. The input vectors may be either the temporary results of vector computation in the VRs or the operands of vector load instructions.

The reduction vector register (VR8) is used to bypass the temporal reduction value between the repeated executions of the vector instructions in the vector AIB. The register is sized to hold a 256-bit slice of vector data and allocates only one 256-bit line of the data cache memory. The vector load and store instructions that operate over VR8 do not utilize the vector length register, because the number of elements in their operands matches the size of VR8. Similarly the vread/write instruction always reads/writes the same cache line allocated by VR8.

VR8 must be initialized with the reduction start value before the reduction starts. The vector load instruction is used to load the start value. The VMU executes such load instruction by reading a single value from memory and replicating it over VR8. For example, when accumulating a vector of 64-bit elements, the 256-bit storage of VR8 is typically initialized with four 64-bit zero values. By executing vread and vwrite instructions over VR8, the VRU reads the 256-bit value from VR8 into the vector lanes and then writes the result of the reduction vector AIB back to the register. For the first execution of vector compute instructions, the VRU reads the reduction start value. The result of the first execution is written back to the VR, and transferred to the vector lanes in the following execution. The input vectors that are being reduced are transferred slice-by-slice for each execution, since they are hold in the regular VRs. The vector instructions repeat their executions until the entire input vectors are reduced. The reduction result is a 256-bit vector slice; and it is stored to memory by a vector store instruction. The VMU issues this store instruction when the vector instructions finish all their executions and the reduction result is written into VR8. The scalar instructions are used to finalize the reduction process, by reducing the 256-bit result in the memory to a single value.

The cost-effective implementation of reductions proposed in this thesis has various advantages as well as disadvantages, when comparing it against traditional implementations of reductions in classic vector processors. Instead of reducing large vector operands to a single value like in classic designs, our implementation partially reduces large vectors to a 256-bit vector slice. It avoids the complex cross-connections between the compute vector lanes, which are required to finalize the reduction and produce a single reduced value. In our implementation, the scalar instructions are used instead of the complex hardware additions to finalize the reduction process. It minimizes the implementation overheads, but incurs the extra performance penalty that is required to execute the additional instructions. Also, it is important to note that this design decision exposes to the programmer/compiler the size of the vector slice. Such design requires recompilation of the code for different generations of the same processor family, where other sizes of the vector slices could be preferable. Although it yields the minor performance improvements, in this thesis we chose the cost-effective design for



```
double a[12];
// initialization of a[]
...
// vectorizable reduction
double sum = 0;
for(int i=0; i<12; i++)
  sum += a[i];
```

```
// reduction start value and
// partially reduced sub-vector
double zero = 0;
double sum[4];
_vinfo(12, SIZE_64);
_vload(&a[0], 1, VR0);
_vload(&zero, 0, VR8);
_vstore(&sum[0], 1, VR8);
```

```
v0 = _vread(VR0);
v8 = _vread(VR8);
v8 = _vadd(v0, v8)
_vwrite(v8, VR8);
```

```
// single value reduction result
double singleSum = 0;
for(int i=0; i<4; i++)
  singleSum += sum[i];
```

Figure 3.11: An example of the vector add reduction. A vector add instruction performs slice-by-slice reduction of the vector operand in VR0. VR8 bypasses the output of the vadd execution to the input of its next execution. The example shows the states of the VRs before the vadd executions: 0, 1 and 2.

reductions; and we do not further investigate the trade-offs between such design and the traditional implementations of reductions.

Figure 3.11 shows an example of the vector reduction by using EVX. The example shows the code of a simple vectorizable reduction, the code vectorization and the execution sample of the EVX reduction. The reduction accumulates (adds) elements of the input vector (a[12]). Its vectorization describes the vector memory instruction, the vector AIB that performs reduction compute operations and the scalar code that finalizes reduction. The memory instructions load the input vector into VR0 and the reduction start value into VR8 ( (replicated zero values), as well as store the final reduction result from VR8 to memory. The vector AIB is composed of vector instructions, which read slice-by-slice of VR0 and the current value of VR8, perform their addition and write the result back to VR8. The scalar code accumulates the four 64-bit elements of a 256-bit reduced vector slice into a single value. The figure shows three states of the VRs, each one before different consecutive executions of the vector instructions (*EX:0, EX:1, EX:2* ). After two such executions, VR8 has the value of two accumulated slices from VR0. The vector instructions repeat their execution three times to accumulate 12 elements of its input vector.

### 3.4.8 Implementation of Vector-Scalar Operations

The EVX model leverages four dedicated 64-bit registers to hold scalar operands for mixed vector-scalar operations. Scalar values that are not known at compile time or that do not fit in the immediate field of a vector compute instruction are stored to these registers. If the scalar value is less than 64 bits, it is replicated across the scalar register, so that the register holds the entire 64-bit operand for one vector lane. For example, two 32-bit scalar values make one operand for the 64-bit vector lane, and the lane performs subword compute operations over mixed 32-bit vector-scalar elements. The initialization of scalar values must precede a vector AIB, which has vector instruction that operates over the scalar values. The vector instructions accesses the scalar values from the dedicated registers by using vread instructions that read the dedicated scalar instead of the VRs. The VRU reads the 64-bit scalar value and writes it across all lanes (replicate) for

| Component | Description |
|---|---|
| ALUs | 2 Integer/FP |
| Reservation Stations | 128 x 2(left/right) x 64-bit |
| Register File | 64 entries |
| Load-Store Queue | 32 entries, unordered LSQ |
| L1 I-cache | 32 kB, 1 cycles (hit) |
| L1 D-cache | 32 kB, 1 cycles (hit) |
| L2 | 1 bank x 512 KB, 15 cycles (hit) |
| L1/L2 MSHRs | 8 entries |
| DRAM | 250 cycles |
| Branch Predictor | OGEHL |
| Vector Registers | 8 x 8192-bit (128 elements of 64-bit) |
| Vector-MSHRs | 32 entries |

Table 3.1: Simulator configuration.

each execution of the vector instructions. To avoid read and write operations of the same 64-bit value at each new execution, the VRU can be optimized to signal the core's issue logic that replicated values are ready for the next execution of the vector compute instructions.

## 3.5 Experimental Setup

### 3.5.1 Simulator

We evaluate an EDGE core and the proposed EVX technique by using a detailed, timing, in-house simulator from Microsoft Research. The simulator is written in SystemC to accurately model the baseline and extended EDGE cores with the parameters shown in Table 3.1. They include an in-house model of the memory hierarchy (L1, L2 and DRAM memories). We developed McPAT [45] models to estimate the area, and runtime dynamic and leakage power of one EDGE core in a EDGE CMP with and without EVX. The models assume 32nm low power technology. They are based on the existing in-order McPAT models, that have been extensively modified and extended to match the design of the EDGE cores. We chose to extend the McPAT in-order models, because the EDGE cores avoids most of the structures typically used in out-of-order processors.

| | Name | Pattern | Dependencies | Vector AIBs | Stream |
|---|---|---|---|---|---|
| Livermore Loops | 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 | none | 1 | Yes |
| | PiCell | sequential/indexed | none | 5 | Yes |
| SDVBS | Disparity | sequential/ strided/masked | none | 10 | Yes |
| | Tracking | sequential/ strided | none | 16 | Yes |

Table 3.2: Workloads.

The EDGE simulator models a multimedia SIMD extension for EDGE architecture. We use it as a reference point to compare with EVX along with the scalar execution on the EDGE core. We also compare EVX to ARM NEON extension by using simulations from Gem5[5] simulator, which is configured to match the parameters equivalent to the EDGE core. We chose NEON to represent the state-of-the-art of SIMD extensions for commercial low power architectures. Although we are aware that fair comparison of two different architectures is difficult, we do this to show the applicability of EVX/NEON over different DLP kernels. On the other hand, we do not compare EVX with dedicated DLP accelerators, because it seems to be the unfair comparison; these accelerators require including the large specialized hardware additions, while EVX requires including only the simple additions to the general purpose EDGE core.

### 3.5.2 Benchmarks

For our evaluation we select ten kernels from the Livermore Loops Benchmark [48] and two entire applications from the San Diego Vision Benchmark Suite (SD-VBS) [74]. The Livermore Loops have been used for decades and they are still used for evaluating the processor efficiency on DLP workloads [15]. The SDVBS workloads have been chosen as they represent emerging mobile applications with significant power/performance constraints. The characteristics of the selected workloads and their vectorized versions are shown in Table 3.2.

Each Livermore Loops kernel utilizes a different memory access pattern and data parallel algorithm, which we found suitable to explore the advantages and limitations of our vector model. Their computations are not complex, which allows for in time reasonable vectorization without extra compiler support. The kernels use sequential, strided or indexed memory access patterns. Some of them also include vector reductions (*Iprod, BLSS*).

The SDVBS workloads have loops with sequential, strided and masked memory accesses. *Disparity* map is a memory-intensive application that processes a pair of stereo images taken from slightly different positions. The algorithm operates on each pixel and offers a large amount of DLP even with small inputs. *Feature tracking* is a memory and computationally intensive application that extracts motion information from a sequence of images. The major algorithm phases are coarse grained, and the amount of DLP is smaller but scales with the input image size.

### 3.5.3 Methodology

We hand-vectorized all the workloads by using the API to describe the EVX memory instructions and the compiler intrinsics to write the vector AIBs. Additionally, we have profiled both applications by using GProf tool to find their most time-consuming functions/loops before vectorizing them.

In order to vectorize some kernels from the selected Livermore Loops it is necessary to use multiple vector AIBs. Some of the kernels, which are vectorized with a single AIB (*BLSS, Fdiff, Hfrag, Iprod, Matrix*) or multiple data-independent AIBs (*PiCell*) benefit from *EVX streaming execution mode.* They do not exploit temporal locality, while performing simple computation over large amount of data. The kernels, which are vectorized with multiple data-dependent compute AIBs (*ADI, ESF, Ipred*) benefit from *EVX register execution mode.* They perform more complex computation that spans multiple vector AIBs and exploit temporal locality between these blocks. The VRs hold the temporary results between the AIBs, which limits the number of executions for the vector instructions in one AIB to the size of the VRs. The vectorized SDVBS loops perform simple independent computations, which fit in one vector AIB. These loops benefit from *EVX streaming execution mode*. The temporal locality of SDVBS workloads is only exploited through the L2 cache, which saves the output data of vector AIBs and thus captures the temporal locality between different computations.

All the selected workloads perform the computation over 32-bit data elements, either mixed integer and floating point elements (SDVBS) or floating point elements only (Livermore Loops). This enables the VRs to hold as twice as more vector elements than what they are sized for, by packing them (e.g. VRs size to hold 128 elements of 64 bit hold up to 256 elements of the loops). In the same way, the 32-bit element size allows for processing of 2 vector elements per each 64-bit ALU operation.

The selected Livermore Loops kernels are evaluated after warming up the caches, which makes fair the comparison of EVX and scalar execution without memory prefetching. On the contrary to the selected Livermore Loops, we show the SDVBS results while running the entire applications without warming up the caches beforehand. To show an upper bound of EVX benefits against more advanced scalar execution when data prefetching is present, we include the results of the scalar execution with perfect L1 caches.

The number of iterations in each Livermore Loop kernel is configured to 1024, unless the impact of this number is evaluated in the experiment. For the SDVBS applications, the number of loop iterations depends the granularity of their algorithms and size of on their inputs. We use two input image sizes in our experiments: SQCIF (128x96) and QCIF (176x144). While such sizes are not particularly large by modern standards, they allow us to show the EVX improvements on the selected DLP workloads with an acceptable simulation time. Larger image sizes would offer even more data parallelism and as a consequence the higher EVX performance.

We evaluate the performance of the EVX enabled and baseline EDGE core when executing the selected Livermore loop kernels and SDVBS applications. We also compare the speedup achieved by using EVX, the EDGE SIMD extension

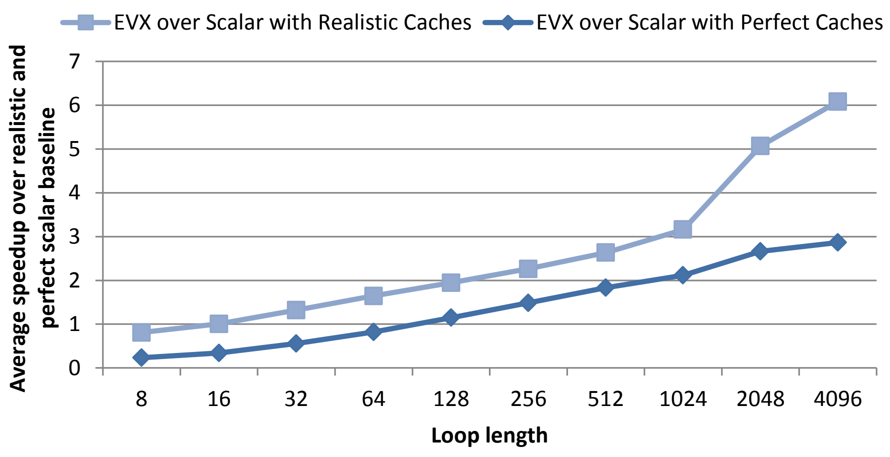


Figure 3.12: Average EVX speedup for different loop lengths.

and ARM Neon SIMD extension for the Livermore Loops kernels. The speedup for EVX and EDGE SIMD is reported over scalar version of kernels running on the EDGE core. Similarly, the speedup for ARM NEON is reported against the scalar version of kernels running on the same ARM core. The kernels for the Neon extension are vectorized by using gcc autovectorization combined with ARM NEON intrinisics, which maximize Neon performance and make the fair comparison with hand-vectorized EVX results.

## 3.6 Results

### 3.6.1 Performance

Figure 3.12 shows the average EVX speedup for the Livermore Loops for a variable number of iterations ranging from 8 to 2048 elements. The number of iterations in each kernel is configurable and referred to as loop length in the remainder of this evaluation. To show the tolerance to memory latency, the speedup of EVX is presented over the scalar baseline with both realistic and perfect data caches. Please note that EVX in both cases accesses the realistic data cache. In following, we first discuss the results with realistic memory system. For short loop lengths (8 to 16 elements), there are no performance gains over the scalar version. It happens due to the initialization startup overhead required to setup the VCU (to describe the memory access patterns, the length of vector computation and the size of its elements) as well as overhead required to reconfigure the data cache. The overheads are amortized around 16 elements, when EVX provides a speedup that further increases with loop length. The speedup increases with the size of vector operands for various reasons. First, the reduction of per instruction fetch decode and issue overheads in the vectorized loop increases, which improves the relative performance. Second, the decoupled execution of vector memory instructions in EVX yields more benefits for larger vectors due to amortizing startup memory overheads over many vector elements. For a loop length greater than 1024, the speedup increases even more (3.16-6x for loop lengths of 1024-4096). At this length, the input data sets of the selected kernels often become larger than the L1 data cache size and the performance of scalar loops decreases because of data cache misses. On the other hand, the decoupled EVX memory execution loads/stores vector data from/to the L2 cache, while avoiding cache penalties. The scalar execution with perfect data cache reduces the EVX speedup by eliminating data cache misses. EVX over such improved scalar execution requires more iterations to compensate its initialization overhead. In that case, it starts achieving speedup with loop lengths over 64 elements. The speedup is moderated, but still high for long loop lengths (about 2-2.9x for lengths of 1024-4096). Besides reduced per instruction overheads, EVX achieves this speedup by efficiently packing and aligning 32-bit vector elements in the VRs. This increases the computation throughput by computing two 32-bit elements per each EVX compute operation in 64-bit ALU. On the contrary to EVX, scalar execution processes extended 32-bit compute operands in each ALU, thus underutilizing the capability of ALUs. Later in this section we compare EVX with the EDGE SIMD extensions that pack two 32-bit elements per each ALU operation, and we also present a breakdown of the speedup achieved with EVX to determine the specific impact of this feature.

Figure 3.13 shows the impact of the size of the VRs on speedup achieved with EVX. We have experimented with the VRs configured to hold 16, 128 and 256 vector elements of 64-bits and report the speedup obtained with the selected Livermore Loops (loop length of 1024). Note that the configured VRs hold twice more 32-bit elements packed in them (e.g. VR sized for 128 elements of 64 bits

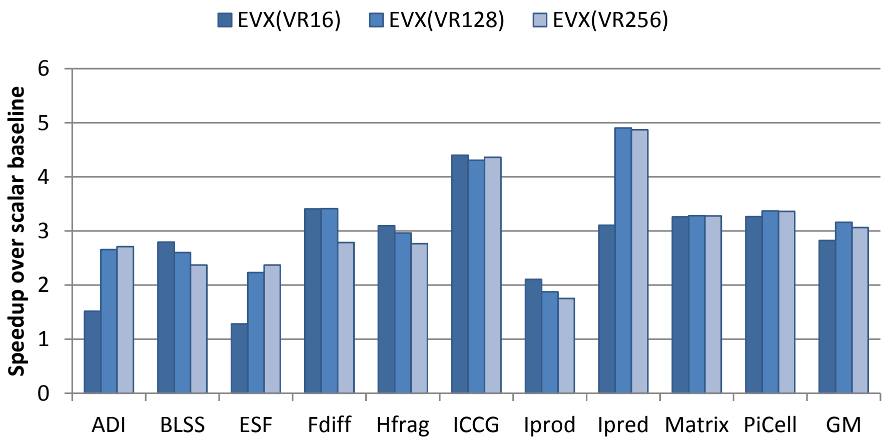


Figure 3.13: EVX speedup for different VR sizes (loop length=1024).

holds 256 elements of 32 bits). The average speedup with the VRs sized for 16 elements is 2.82x. The speedup increases to 3,16x when configuring the VRs to hold 128 elements, but further decreases to 3x when configuring the VRs for 256 elements. The kernels that exploit temporal locality between different vector AIBs in the VRs (*ADI, ESF, Ipred*) improve their performance, while increasing the size of the VRs. The size limits the maximum length of the computation for each vector AIB (the size of vector operands), because the VRs hold the temporary results between different AIBs. For example, the VRs sized for sixteen 64-bit elements limit the size of the vector operands to thirty-two 32-bit elements. For these kernels, *strip mining* is used to iterate their loops 1024 times. This limitation makes the startup overheads still significant and limits the EVX performance. Increasing the size of the VRs to hold 128 instead of 16 elements yields up to 75% of extra performance (*ADI*) by amortizing the startup overheads. Further increasing the size provides modest performance improvements, since the startup overheads are already negligible at that moment and exploiting more temporal locality in the registers does not change the execution time. The workloads with streaming characteristics do not benefit from larger VRs (*Fdiff, Matrix*), since the VRs are only used as streaming buffers. The increasing of the VRs sometimes leads to performance degradation, due to unnecessary reconfiguration of the large data cache memory (*Fdiff*) for such registers. In some kernels where the vectorized code is inside an outer loop, data cache reconfiguration is

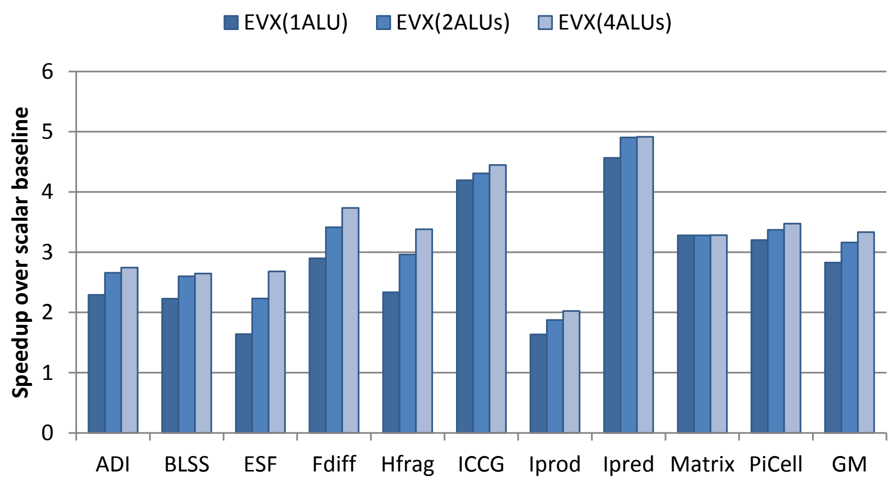


Figure 3.14: EVX speedup for different number of ALUs (loop length=1024).

performed multiple times, which leads to even bigger performance degradation (*BLSS, HFrag*). The VRs sized for 128 elements shows the best overall results and we use it as default configuration for the rest of our experiments. It is yet interesting to note that the configurable size of the VRs allows EVX to adapt its hardware resources to the specific workload requirements (e.g. highly streaming workloads prefer small VRs to buffer the streaming data); and this feature would make possible to choose a different VR size for each benchmark.

Figure 3.14 shows the individual EVX speedup on the selected Livermore Loops kernels with different number of ALUs for the EVX enabled design: 1, 2 and 4. The loop length is 1024. The default hardware configuration of EVX uses the two existing ALUs of the EDGE core. Increasing the number of ALUs does not provide significant performance benefits. The average EVX speedup with four ALUs increases only by 5% compared to its default configuration. This happens because the selected kernels utilize memory intensive DLP algorithms with moderate computation. For such workloads, the extra ALUs without in line improvements of the EVX memory execution generally do not yield significant performance. It yet varies across different workloads. For the kernels with non-sequential memory access, the EVX memory execution matters more than its computation. In this case, EVX cannot provide enough data to saturate the extra units and its speedup does not change (*BLSS, Ipred, Matrix*). For the kernels that

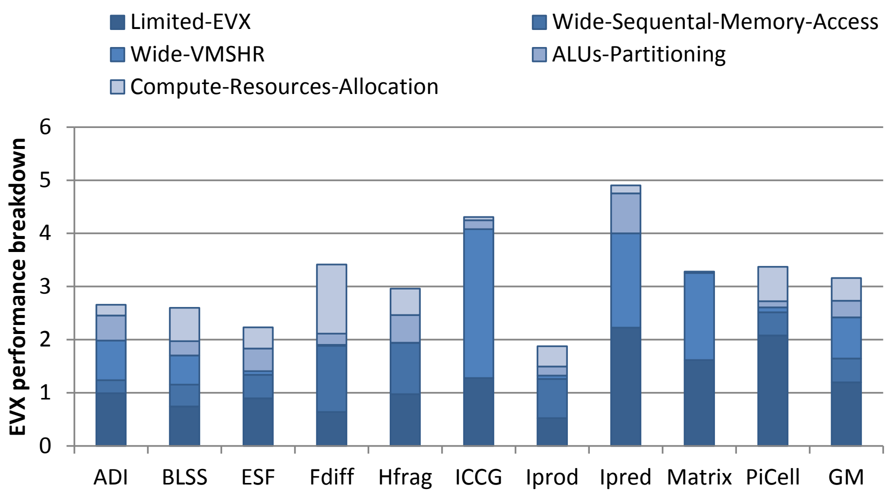


Figure 3.15: EVX performance breakdown (loop length=1024).

utilize sequential memory access, EVX can exploit extra units and its speedup marginally increases (*ESF, Fdiff, Hfrag*). Such results show that our design of the EVX hardware for memory execution (VMU) is balanced to efficiently feed the two available ALUs. When using only one ALU for EVX instead of the available two, EVX reduces its average speedup by 11%. The performance loss with one active ALU goes up to 26% for the workloads with sequential memory access (*ESF, Fdiff, Hfrag*) and workloads with sufficient compute operations (*ADI, BLSS, Ipred*). These results are quite interesting and motivates future research of the ultra low power EVX design. This design could shut one ALU off the circuit (e.g. power-gating) to save the energy and yet allow for the EVX performance improvements on the memory intensive DLP applications with modest computation. In this work, we do not investigate the power additional optimization and we use the existing 2 ALUs for each EVX enabled computation.

Figure 3.15 shows the performance breakdown of EVX by incorporating one by one different hardware features of EVX to measure the impact of each one of them. We have evaluated several EVX features: *wide-sequential-memory-access, wide-VMSHR, ALU-partitioning and compute-resources-allocation.* We explain the benefit of each feature and discuss its impact on the performance. The most simple EVX with all the features disabled and the small VMSHR of 8 entries (*limited-EVX*) decouples the execution of the vector compute, register and memory instructions; they execute simultaneously on the specialized hardware substrate specialized to reduce memory latency. It yet provides limited speedup over the scalar baseline for most of the kernels or even performance degradation due to the startup overheads. The exception exists in some kernels where such configured EVX achieves notable speedups (*ICCG, Ipred, Matrix, 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 32 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 the other kernels do not make use of more than 8 entries. EVX with partitioning of ALUs compute 64 bits of the vector data per each lane regardless of the size of vector elements (e.g. two partitions of 32-bit operands or four partitions of 16-bit operands) and leverage the VCU to read packed compute vector operands. ALU partitioning increases the average speedup by about 10%. The allocation of core's compute resources allows to reexecute the resident compute instructions by refreshing the operands in the reservation stations. It yields 15% additional speedup by eliminating the bookkeeping instructions as well as the instruction fetch and decode overheads.

Figure 3.16 shows the reduction of instruction fetch, decode and commit overheads in the general purpose pipeline with EVX over to the same overheads with scalar execution. EVX utilizes the incorporated vector instructions, where each one operate over configurable large vector operands; this reduces the average fetch and decode overheads by 50 times. For the kernels that repeat their computation in outer loops (*Matrix*) these overheads are reduced by over 170 times and for the partially vectorizable kernels(PiCell) the reduction is diminished to about 3x. The average reduction of per instruction commit overheads is even higher (over 66x), because EVX atomically executes the fetched and decoded vector compute instructions in the pipeline and its dedicated control hardware (VCU) commits their results to the memory off the pipeline.

Figure 3.17 shows the reduction of instruction execute overheads in the general purpose pipeline with EVX compared to the overheads with scalar execu-

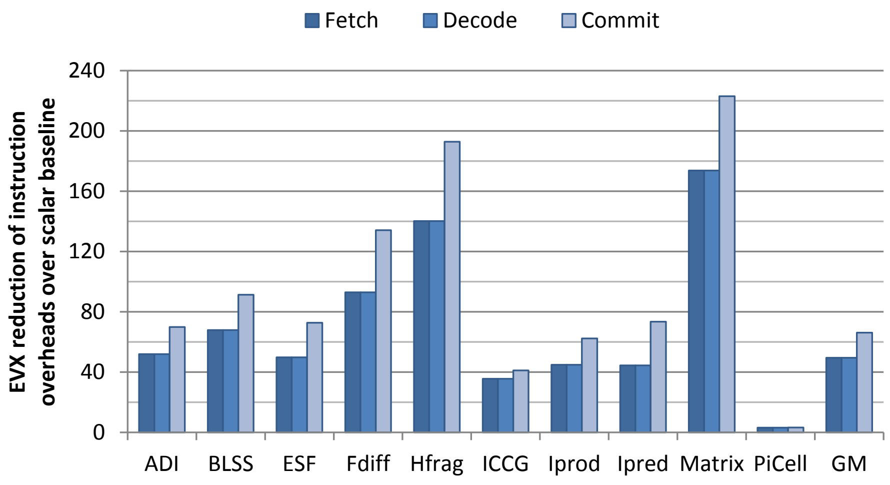


Figure 3.16: EVX reduction of instruction fetch, decode and commit overheads in the general purpose pipeline (loop length=1024).

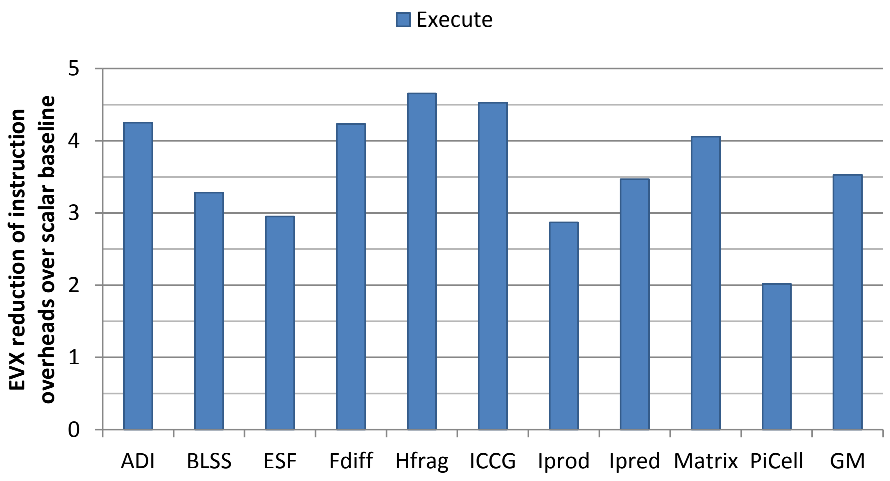


Figure 3.17: EVX reduction of instructions execute overheads in the general purpose pipeline (loop length=1024).

tion. This reduction exists because of various reasons. First, scalar execution of compute instructions performs operations over the 32-bit operands extended to match the size of the 64-bit ALUs, while EVXin the pipeline simultaneously performs two compute operations over packed 32-bit operands per each ALU access. Second, scalar execution utilizes the pipeline to execute the both compute and memory instructions, while EVX utilizes the dedicated hardware (VMU) to perform the specialized execution of the vector memory instructions off the general

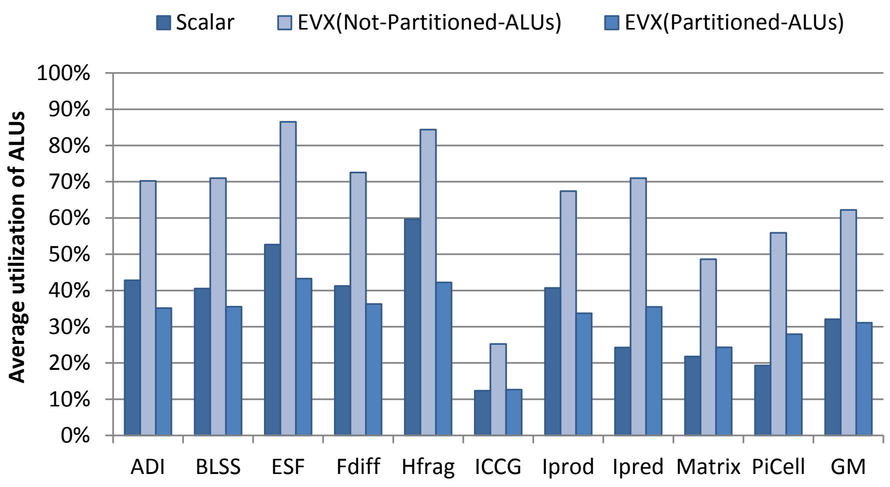


Figure 3.18: Average utilization of ALUs with scalar execution and EVX that uses two different implementations of ALUs (loop length=1024).

purpose pipeline. Third, EVX performs the computation over the configurable large vector operands and thus avoids the execution of bookkeeping instructions (e.g. instructions that increment the counters of a loop and check various loop's conditions). Our results show that EVX in average reduces the number of executed operations in general purpose pipeline by over 3.5 times. For the kernels with large number of memory operations (*ADI, Fdiff, Hfrag, ICCG*), EVX reduces the execute overheads up to 4.5x. Such reduction of execute operations in the pipeline improves the EVX performance as well as its efficiency.

Figure 3.18 shows the average utilization of the ALUs with scalar execution and EVX that uses the ALUs implemented without and with partitioning. The different implementations of the ALUs do not have the impact on their utilization with scalar execution, because scalar compute instructions always operate over 64-bit operands by extending the smaller ones. The utilization of the ALUs with scalar execution varies between the kernels, while depending only on the number of misses in the L1/L2 cache memories; compute and memory intensive workloads have similar utilization of the ALUs, because memory instructions use the ALUs to compute the memory addresses. The kernels with more misses in the L2 cache significantly underutilize the ALUs due to waiting for their operands (*ICCG, Matrix, PiCell*). The other kernels performs more efficiently. In total,

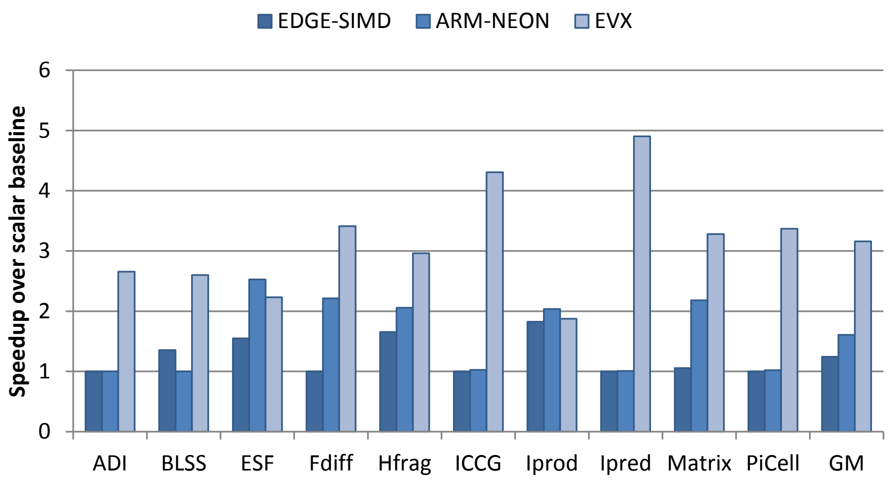


Figure 3.19: Relative speedup of EVX and SIMD extensions (loop length=1024).

scalar execution exploits the ALU resources about 32% of the cycles. EVX increases this utilization by over 2 times, when the ALU partitioning is disabled. The specialized memory execution in the VMU tolerates the memory latency and delivers more data to the ALUs than scalar execution. On the contrary to scalar execution where the utilization of the ALUs depends on the number of the cache misses, the utilization with EVX depends on the efficiency of such memory execution. For example, EVX on the kernels with strided access patterns and large number of cache misses loads/stores vector operands less efficiently and utilizes the ALUs less(*ICCG, Matrix*) than on the kernels with sequential access patterns (*ESF, Fdiff, Hfrag*). The partitioning of ALUs reduces their utilization by 50%, which is the utilization approximately the same like with scalar execution (31% of the cycles). It is interesting to note that EVX with such (in this thesis default) implementation of the ALUs does not increase their utilization, but increases the efficiency of the ALU accesses by simultaneously operating over different elements in each partition of the ALUs.

Figure 3.19 compares the speedup achieved with EVX and several multimedia SIMD extensions. The EDGE SIMD extension operates on 64-bit words (two 32-bit elements) and uses two 64-bit SIMD units, as same as EVX. It supports unaligned memory accesses, but lacks more sophisticated addressing modes that are needed to vectorize some workloads; neither it enables the operand shuffle operations that can workaround to deal with the lack of the addressing modes. The extension does not incorporate extra wide SIMD reservation stations nor extra compute SIMD resources. Such design allows for the fair comparison of EVX and EDGE SIMD extension, since both of them utilize no complex hardware additions. The results show that EDGE SIMD extension increases the performance for workloads with sequential memory access, which do not require shuffling between compute operands (*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 some kernels iterates multiple times with the same data (*BLSS*), but the speedup is still limited. Although the cache is warmed up for both approaches, the SIMD extension should benefit more of such cache than EVX by loading/storing the SIMD operands directly from/to the L1 data cache. Yet, the efficient EVX memory execution applicable to more benchmarks yields higher overall performance for EVX. Compared to the EDGE SIMD, ARM's NEON SIMD extension provides a more complete ISA (e.g. shuffle operations), 128-bit wide SIMD words, as well as fused multiply-add operations. On the ARM based core NEON performs better than EDGE SIMD extension on the EDGE core and worse than EVX enabled EDGE core. Only for a few kernels, NEON has higher speedup than EVX. It happens because the data for these kernels are already in the cache and the kernel's execution time is too small to mitigate the EVX startup overheads (*ESF, Iprod*). While the EDGE SIMD extension improves the execution of the selected Livermore loops kernels on the EDGE core by 20%, ARM NEON extension improves their execution on the ARM core by 60%, due to more complete ISA and wider SIMD registers. The EDGE SIMD accelerates 3 of 10 selected kernels, while NEON accelerates 5. EVX provides the most complete ISA and allows for vectorization of each kernel.

Figure 3.20 shows the EVX speedup over the scalar baseline for the selected SDVBS applications and their input images. *Disparity* has the larger amount of DLP, because it processes individual pixels of the input images. Therefore EVX on this application shows the speedup of 12.8x and 16.5x, when it uses sqcif and qcif input images respectively. On the contrary to *disparity*, the *tracking* with the selected input images of the same quality has less DLP due to its coarse-grain processing algorithms. For this reason, EVX on the *tracking* application yields the

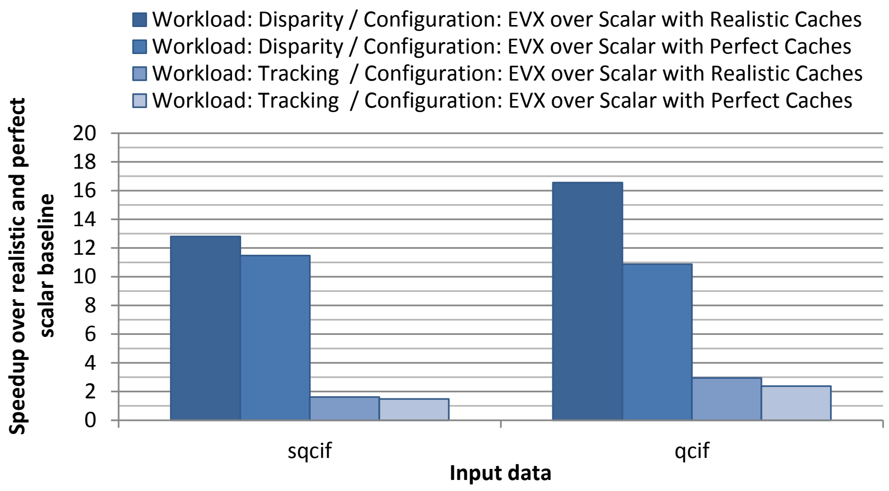


Figure 3.20: Disparity and feature tracking results.

comparatively less speedup of 2.95x; and it has the same order of magnitude like the speedups on the selected Livermore Loops. EVX speedups on the *disparity* application are over 3 times higher than the speedup on each other evaluated workload and we next discuss various features of EVX that enable such results:

- *Disparity* loops for qcif inputs have about 176x144 iterations, which makes extremely large vector operands with more than 25,000 elements. The EVX streaming mode repeats the execution of the vector instructions until all the elements of such large operands have been processed. This reduces the repeated fetch, decode and commit instruction overheads. Our results show that EVX on the *disparity* application reduces these overheads by over 100 times over the scalar baseline.

- The vectorization of both inner and outer loops by using 2-D strided memory accesses eliminates large amount of bookkeeping instructions. What remain of the inner and outer loop after vectorization is only vector memory and compute instructions; this eliminates over 90% of the total instructions from the original scalar version of the code. EVX also reduces the number of executed compute and memory operations by using various features of its specialized execution. Namely, vector memory instructions exploit wide memory accesses to load/store 256-bit cache lines of sequential vector operands. This reduces the number of memory requests performed. Vector compute instructions benefits from partitioning of the ALUs, while operating over 32-bit vector elements. Each execution of one vector compute instruction operates over a pair of 32-bit operands. This feature reduces the number of compute operations by 50%. This is advantageous for EVX, since it performs an extremely reduced number of the compute and memory operations. Our results indicate over 30 times less executed on the EVX enabled core compared to its scalar baseline.

- The decoupled EVX memory execution over the operands with more than 25,000 elements makes the EVX startup overheads negligible, as well as overlaps the latency of memory accesses with the computation of such large vector operands.

To compare the EVX memory tolerance against scalar execution with sophisticated prefetching, Figure 3.20 also includes the speedup over the scalar execution on the EDGE core with perfect cache. The speedup is reduced significantly for the *disparity* application with qcif inputs. When increasing the image size, the amount of temporal data increases and then it does not entirely fit into the L2 cache. It causes additional L2 cache misses and data replacement. The scalar execution with perfect cache does not suffer from such misses, whereas EVX reduces its performance. Nevertheless, the speedup of EVX in this case is still over 10x, due to extremely large reduction of the executed operations and fetch-decode overheads. On the other hand, *tracking* application further has high temporal locality, because its temporal data fit into the L2 cache. The amount of L2 cache misses is small, as well as the EVX degradation of speedup when comparing it to the scalar baseline with the perfect cache.

### 3.6.2 Area and Power

Table 3.3 presents the area breakdown estimation of different microarchitectural components in an EDGE processor with one modest dual issue core. The table includes the results for the baseline core and the enabled EVX core with VRs configured for 32 and 128 64-bit elements. Note that the VRs do not change the

| Structures | Baseline EDGE | EVX | |
|---|---|---|---|
| VR Size | - | 32 | 128 |
| Fetch, Decode, L1I | 0.258 | 0.258 | 0.258 |
| Issue | 0.005 | 0.005 | 0.005 |
| LS Unit, L1D | 0.237 | 0.237 | 0.237 |
| RegFile, ResStations | 0.061 | 0.061 | 0.061 |
| ALUs | 1.463 | 1.463 | 1.463 |
| Vector Control Unit | – | 0.045 | 0.045 |
| Core Total | 2.023 | 2.068 | 2.068 |
| L2 Cache | 2.079 | 2.079 | 2.079 |
| *Processor Total* | *4.102* | *4.147* | *4.147* |

Table 3.3: Area breakdown ($mm^2$) estimates of a 1-core EDGE and EVX enabled processor at 32nm.

area of the EVX core, because they repurpose the L1 data cache resources. EVX also leverages the existing core's compute resources and it increases the total core area only by 2.2%. The extra area is required for the VCU, while assuming ALUs with partitioning in the baseline core. The VCU utilizes modest hardware resources such as: a small set of counters and control registers, non-associative vector outstanding registers and one adder unit for address calculation. These resources have minimal area impact even in the lightweight EDGE core.

Table 3.4 shows the estimation of average power consumption in a 1-core EDGE baseline and EVX enabled processor with VRs configured for 32 and 128 64-bit elements, while running scalar and vectorized versions of the selected Livermore Loops krenels. The loop length is 1024. EVX reduces power consumption in the fetch and decode pipeline stages. It happens because EVX fetches and decodes vector AIBs once, but repeats execution of their vector instructions many times to process large vector operands. On the other hand, EVX increases power consumption in the issue pipeline stage (instruction window and instructions select logic), because the unmodified issue stage on EVX enabled core checks the readiness of each valid instruction in the instruction window; and EVX replicates the vector instruction in each lane of the instruction window, while increasing the number of valid instructions. Note that EVX can optimize the issue logic to check only the instruction mapped to the first lane of the instruction window,

| Structures | Baseline EDGE | EVX | |
|---|---|---|---|
| VR Size | - | 32 | 128 |
| Fetch, Decode, L1I | 0.020 | 0.006 | 0.006 |
| Issue | 0.097 | 0.134 | 0.139 |
| LS Unit, L1D | 0.016 | 0.004 | 0.004 |
| RegFile, ResStations | 0.006 | 0.005 | 0.006 |
| ALUs | 0.045 | 0.044 | 0.045 |
| Vector Control Unit | – | 0.036 | 0.038 |
| Core Total | 0.184 | 0.231 | 0.237 |
| L2 cache | 0.055 | 0.127 | 0.134 |
| *Processor Total* | *0.239* | *0.358* | *0.371* |

Table 3.4: Average power ($W$) estimates of a 1-core EDGE and EVX enabled processor at 32nm.

since they always execute in the same order (first the instructions in the lanes 0 and 1 and then the instructions in the lanes 2 and 3). In this thesis we do not further investigate such optimization. The dynamic activity of the ALUs along with total accesses to the reservation stations buffers are approximately the same on the baseline and EVX enabled cores (see Figure 3.18). As consequence, the power consumption in these structures is the same for both scalar and vectorized execution. The accesses to the VRs of the repurposed L1 data cache resources increase the data cache power consumption. The decoupled EVX memory execution more efficiently saturates the memory bandwidth and increases power dissipation in the L2 cache. In total, EVX increases the average power consumption of the EDGE processor by about 50% and 55%, when configuring the VRs for 32 and 128 elements respectively; and while providing over 3x of the average speedup, EVX significantly increases the power efficiency of the processor.

## 3.7 Related work

Classic vector architectures enable high-performance and power-efficient data parallel computing by operating over large vector operands. They require large vector registers to hold such operands, a large number of memory banks to increase memory throughput, as well as a large number of parallel functional units (lanes) to increase computation throughput [3, 19, 63]. Due to such complex design, they have been utilized like commercial processor in supercomputers, rather than in embedded devices. Adding a vector unit to superscalar processor [61] have been proposed to improve the performance of DLP workloads, by introducing functional units and a large vector register file into a general purpose core. However, the complexity of the proposed addition limited its applicability in commercial microprocessors. On the contrary, EVX minimizes the additional complexity of vector processors by leveraging the existing core functional units to perform the vector computation and data cache resources to hold vector operands. Although avoiding the complex vector-specific hardware additions, EVX yet resembles the classic vector execution on the lightweight EDGE core, which maximizes the core's power efficiency.

Classic vector architectures have been extended and improved in several ways. Decoupled vector architectures [16] use queues to decouple the scalar, vector arithmetic and vector memory execution in a traditional vector design. The results show that decoupling provides superior memory latency tolerance compared to a pure in-order vector implementation. EVX also incorporates the decoupled vector memory execution in the extra vector control unit, which loads/stores vector operands. Work on out-of-order vector architectures [18] shows even better performance than decoupled vector execution. For such execution, it requires a larger vector register file and register renaming logic. EVX leverages existing dataflow mechanisms of the EDGE architecture for out-of-order execution of its compute instruction. CODE [39] is a more recent vector architecture that incorporates decoupled vector execution. It is based on a clustered decentralized design. Such design allows for non-complex scaling of its resources including the vector register file and the number of functional units. EVX does not decentralize nor scales the existing core units. Its decoupled memory execution feeds only the units available on the core, while the vector registers dynamically configure their resources.

Various designs propose accelerators for embedded processors, which are based on the conventional vector execution. The Reconfigurable Streaming Vector Processor (RSVP) [10] proposes a streaming vector coprocessor for accelerating data streaming applications. RSVP uses input/output stream units which are programmed by using flexible memory patterns. Instructions are executed in a dataflow fashion as a set of ordered, dependent computations for each vector element. RSVP is an efficient solution for data streaming applications, however the lack of vector registers prevents exploiting data locality and executing non-streaming DLP workloads efficiently. The Imagine Streaming Processor (ISP) [34] uses streaming registers to keep the results between the clusters and its clustered design offers a high scalability and arithmetic rates. RSVP and ISP are specialized hardware accelerators. On the other hand, EVX offers the specialized execution on general purpose hardware and execute parallel and not parallel code without offloading onto the accelerators.

There is an another recent research alternative that combines general purpose and vector execution on the general purpose hardware to efficiently support of all levels of parallelism (ALP) [66]. It uses a conventional superscalar multicore to exploit ILP/TLP/DLP, while focusing on memory system to increase the performance of DLP multimedia applications. The ALP uses streams/registers in data cache, which load/store large vector operands similarly to EVX. The existing issue/rename logic is extended to use slices of these registers as a source or destination of each vector compute instructions. On the other hand, EVX uses the extra hardware unit that transfers the the slices of vector operands between the compute instructions and the vector registers to avoid more complex issue/dispatch overheads. EVX executes vector compute instruction out-of-order by using statically generated dataflow, whereas ALP dynamically discovers data dependenices and checks the operand slices availability.

Previous work in exploiting DLP in the TRIPS EDGE architecture [64, 65] extends a massive unicore processor for high performance devices to execute vector EDGE instructions. On the contrary, EVX works on a modest dual issue core for low power devices. Although targeting different segments of computer industry, the both approaches have some similarities as well as differences. Like the re-execution of vector instruction in our work, the instruction revitalization is utilized to support vector operations in a TRIPS extended processor. EDGE AIBs allocates large amount of compute resources in the TRIPS processor and repeat their execution over different data. The TRIPS vector execution does not decouple memory accesses through the extra memory unit like EVX. It utilizes a separate thread to orchestrate the memory execution. Instead of using the L1 cache for the large vector registers, the L2 cache is accessed as a software managed cache and DMA is used to access memory. While bypassing the L1 cache, the TRIPS vector execution does not exploit temporal locality in fast memory levels. On the other hand, EVX reconfigures the part of L1 cache into the VRs to hold temporal data between the blocks of vector instructions.

Nowadays, most commercial processors incorporate still limited multimedia SIMD extensions [8, 20, 55, 59, 73] to accelerate DLP workloads. Compared to conventional vector processors, SIMD extensions operate over modest size vector operands and offers the reduced support of sophisticated addressing modes. EVX overcome these limitation through its decoupled and sophisticated memory execution, including the vector registers in data cache that hold configurable large vector operands. Moreover, EVX can be complemented with the SIMD resources such as wide registers and ALUs to increase its computation throughput.

## 3.8 Summary

In this chapter, we have introduced one potential technology to enhance the quality of mobile processors by efficiently exploiting DLP on its lightweight general purpose cores. Our technology specializes the substrate of a low-power EDGE core for vector execution (EVX) like in classic vector processors. EVX executes the new incorporated vector instructions, which operate over configurable large vector operands instead of scalar elements. The vector compute instructions allocates the existing compute resources and repeat their execution over different vector elements. The vector memory instructions loads/stores vector operands through an extra vector dedicated hardware, which saturates memory bandwidth and increases the performance of the existing compute resources. The dedicated memory execution enables more sophisticated addressing modes and increases the applicability of the EVX technology.

Our results show that enabling EVX on the EDGE core requires only 2.2% of the extra core area. Like classic vector processors, EVX greatly reduces instruction fetch, decode and commit overheads. It overlaps the latency of memory accesses with the computation of large vector operands. At the same time with the ALU partitioning, EVX twice more efficiently utilizes the available execution resources in the EDGE core. On the set of the selected Livermore loops, EVX yields over 3x of the speedup with 55% higher power consumption. The running of entire SDVBS compute vision applications shows even better results.

We believe that high-performance and power-efficient design along with a minimal area overheads evaluated in this chapter makes our specialization technique very advantageous for processors in mobile devices. Motivated by these results, we further looked at scaling the performance and efficiency of the specialization beyond a single core. In the next chapter we introduce a vector execution that dynamically specializes one or more cores of a CMP for DLP acceleration.

# Chapter 4

# CMP Specialization for Dynamically-Tuned Vector Execution

## 4.1 Overview

This chapter proposes a cost-effective technique that dynamically specializes the available resources of a low power chip multiprocessor (CMP) into an accelerator of DLP workloads. In the same way as EVX specializes a single general purpose core (see Chapter 3), this new technique specializes a configurable number of the CMP's cores to mimic the functionality of a vector processor. Such specialization enables dynamically-tuned vector execution (DVX) of data parallel workloads on the general purpose substrate without a dedicated accelerator.

Previous chapter proposes the EVX technique that specializes a modest low power core for classic vector execution (see Chapter 3). In this chapter, we enhance the proposed execution to dynamically allocate compute and memory resources beyond single core. Unlike classic vector processors that use a fixed number of vector lanes, DVX tunes the size of its execution substrate to a particular workload phase. For this dynamic feature, DVX introduces lightweight threads of vector instructions controlled by hardware. The threads scale the execution of vector instructions over multiple cores with minimal performance overheads. The control hardware divides the vector operands into equal partitions of vector elements for vector instructions in each thread and orchestrates threaded execution.

This chapter starts with the DVX specialization technique presented on the level that is applicable to most commercial architectures nowadays. It is followed with one particular implementation that narrows this technique down to the EDGE architecture and describes the DVX implementation on a 4-core dynamic EDGE CMP. Since DVX has evolved from EVX and has been implemented on the EDGE processor, these two DVX sections have some things in common with EVX design although they include the advanced DVX features. We further introduce a more conventional approach of DLP acceleration within heterogeneous processor system that we use to compare against the DVX implementation. In this approach we add a dedicated DLP accelerator based on the DVX ISA and special-purpose hardware resources incorporated onto to the EDGE CMP. We next evaluate performance, power and area results of DVX and the dedicated accelerator with a comparable amount of resources. The chapter ends with related approaches and concluding remarks.

## 4.2 Specialization of CMPs for DVX

DVX specialization technique allocates the general purpose resources to perform computation over vector operands. The specialization utilizes minimal hardware additions that control such computation and efficiently load/store vector operands. The additions dynamically tune the allocated resources to match diverse workload requirements. We explain each of these specialization steps: the allocation, the addition of resources, the operating modes, as well as tuning of the resources. We next discuss the specialized execution of DLP workloads with vector reductions and the execution while occurring interrupts.

### 4.2.1 DVX Allocation of the Existing CMP Resources

DVX specializes the general purpose cores to execute vector instructions that perform compute, register and memory operations over configurable large vector

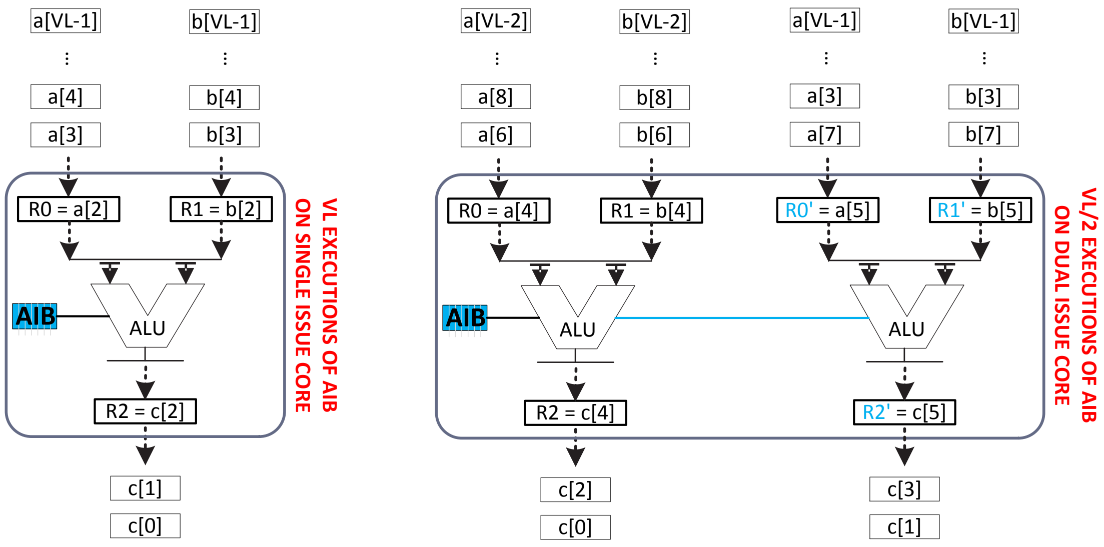


Figure 4.1: An examples of DVX computation on single-issue and dual-issue cores. One AIB allocates the core's compute resources and repeatedly executes over slices of the large vector operands.

operands. Atomic Instruction Blocks (AIB) with the vector compute instructions allocate the compute resources of one or more cores to perform computation over the vector operands. One instance of such vector AIB is fetched and decoded once, but repeatedly executed multiple times to compute the large operands. The vector operands are accessible only outside the AIB, as the block input or output operands. The vector compute instruction inside the AIB process the operands slice-by-slice in a pipeline, as it is shown in Figure 4.1. By processing large operands in slices (sub-vectors), each new execution of the vector AIB requires only the available general purpose hardware. The hardware processes vector slices in the same way as scalar operands and does not require any modification. The number of executions is dynamically configured for each vector AIB, to resemble the execution of classic vector instructions that perform one operation over a configurable number of vector elements.

In the same way as in an EVX enabled EDGE core, vector instructions in DVX cores operate over slices of vector operands. The slices are sized to match the width of available compute resources in general purpose core. One way to increase the parallelism inside a vector AIB as well as computation throughput of the block executions is to divide the compute resources into multiple banks and execute the AIB in each bank on different data. Figure 4.1 shows 1-wide and 2-wide executions of the AIB. 1-wide execution uses one bank of compute resources. 2-wide execution leverages two banks, and each vector instruction in the AIB issues in lock step across the banks to process 2-wide vector slices. This hardware modification limits the size of vector AIBs to fit in a single bank, but enables execution of vector AIBs over wider vector slices at no additional costs. The banking of compute resources resembles vector lanes that share available ALUs. An alternative way to increase the computation throughput inside the AIB is through wide SIMD extensions. The SIMD extensions can process wider vector slices within a single execution of the AIB. However, they incorporate extra SIMD registers and ALUs to the general purpose core. In this work we chose to bank resources to avoid adding the extra SIMD resources.

Vector AIBs can be created at the ISA level (programmer by writing *begin/end* directives or compiler as in the EDGE architecture), while statically defining the size and structure of the AIB. The maximum size of the AIB is limited to fit onto available execution resources. If the resources are banked, the size of the AIB is limited by the size of one bank. An alternative way to dynamically create the AIB would require extra hardware support to enable for block sizing at runtime. This would allow an unchanged application binary to run across various designs of the same processor family. However, in this work, we chose to create the vector AIBs at the ISA level to minimize the complexity and additions of our vector design.

### 4.2.2 DVX Addition of Dedicated Resources

To minimize hardware additions like in EVX, vector AIBs in DVX execute on the existing issue logic, reservation stations (physical registers) and ALUs of the CMP substrate, as shown in Figure 4.2. The reservation stations keep the temporal results between the instructions in a vector AIB. The vector registers (VRs) are an addition to the general purpose core, and they only hold the input/output operands of the AIB. This approach significantly reduces the size of a vector register file, the power-hungry structure that limits the applicability of vector design beyond supercomputer processors. *Vector control unit (VCU)* is a DVX

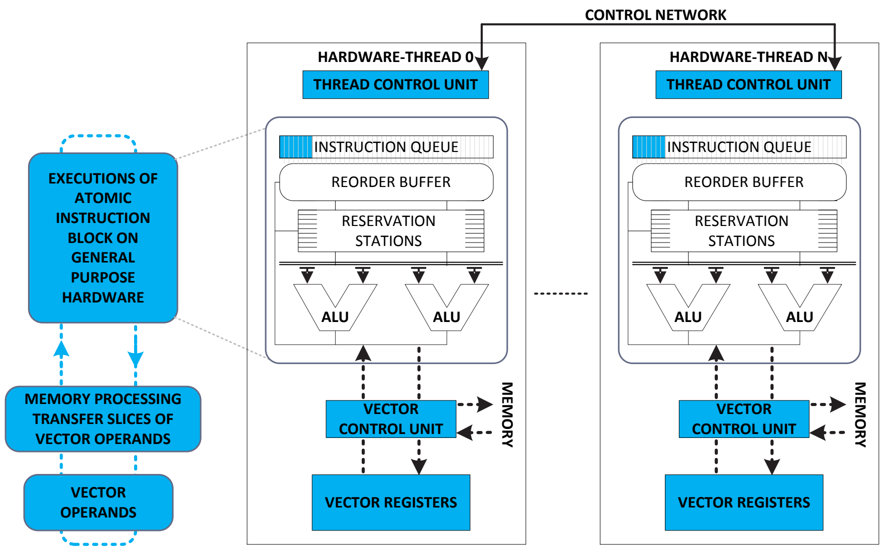


Figure 4.2: The specialized DVX on general purpose CMP utilizes a vector AIB to allocate the compute hardware and perform operations over vector operands. The VCU loads/stores the operands and transfers their slices trough the allocated resources. The TCU tunes the amount of allocated resources.

dedicated unit added to the general purpose core to manage DVX. The VCU decouples the execution of vector memory instructions from the executions of vector compute instructions on the allocated resources. The decoupled memory execution is specialized to tolerate memory latency and provide sophisticated addressing modes required in diverse DLP workloads. While using the VRs to hold large memory operands, the VCU performs the specialized memory execution in a fashion similar to classic vector designs [16, 61]. Beside sophisticated memory processing, the VCU reads/writes slices of large vector operands in the VRs and transfers them per execution of compute AIBs. Vector AIBs allocate the available resources and perform only vector computation; while the specialized memory processing in the VCU increases the utilization of the allocated resources, as well as power efficiency of DVX.

### 4.2.3 DVX Operating Modes

As same as EVX on the EDGE core, DVX on the the general purpose CMP delivers two operating modes and they can be dynamically configured. When the vector computation spans multiple vector AIBs, DVX performs similar to a conventional, register-based vector architecture; the AIBs process large vectors, while VRs hold vector memory operands and capture the data locality between the AIBs. For simple vector processing that fits into a single vector AIB, DVX performs as a streaming-based vector architecture; the AIB processes vector streams and VRs only buffer streaming memory operands. Register and streaming modes are controlled by the programmer/compiler, which specializes the DVX allocated resources for specific application characteristics.

### 4.2.4 DVX Tuning of Resources

Besides configuring the operating modes like in EVX, DVX brings up new dynamic feature that allows for scaling the vector execution over multiple cores in the CMP. This way, DVX dynamically tunes the amount of its resources to different workload requirements. A direct approach to scale DVX would be to leverage software runtime support with conventional threads, which execute the vector instructions across multiple cores. This approach incurs startup overheads to create the threads and replicates the execution of bookkeeping instructions in each core. By using the thread synchronization through shared memory flags, it is possible to reduce startup overheads, but not to avoid the execution of bookkeeping instruction on each core. To maximize performance and avoid issues of the existing software approaches, DVX introduces lightweight threads controlled by hardware that scale the execution of vector instructions over multiple cores. The hardware threads avoid startup as well as bookkeeping overheads. The DVX specialization technique adds a simple dedicated *thread control unit (TCU)* to each core of the CMP and an additional on-chip-network between the cores to start, stop and control the threads with vector instructions.

A single core in a DVX enabled CMP executes parallel and non-parallel phases of the application. It executes scalar (e.g. bookkeeping instructions) and vector instructions. Since it controls the entire execution, the thread running on this core is termed as *master*. The other threads are started on other cores through the TCUs, when requested by the master to improve the vector processing, as *slaves*. The slave threads are lightweight. The TCU disables the execution of common bookkeeping code redundantly executed in each slave thread, which executes only vector instructions. Vector operands are divided into equal partitions of vector elements to distribute the work among the vector instructions in each thread. Each VCU in connection with TCU on the same core determines the partition for their core and loads/stores vector elements of that partition. Each slave thread speeds up the vector processing by executing the same vector compute and memory instructions on its partition of vector operands. By starting slave threads, DVX uses additional cores to increase the number of vector lanes at runtime. It increases the number of vector elements processed in parallel and improves performance of highly parallel applications.

Before starting slave threads, cores have to be idle (not running any application). The TCU of the master keeps track of idle cores, which can start slave threads. The number of potential slave threads can be configured by the programmer, but starting slaves depends on a core's availability. The advanced alternative is hardware support that defines the number of slave threads at runtime, by using various metrics such as: number of long latency instructions in a vector AIB, vector length or number of available cores. In this work, we only explore a basic approach where the programmer defines the number of slave threads and assumes that all cores are available for vector execution.

### 4.2.5 DVX and Reductions

DVX on general purpose cores partially support reduction of large vector operands, without extra hardware additions nor modifications. The repetitive executions of one vector AIB over different vector slices reduce the large input vectors, by passing the output slice (result) of one execution as the input slice for next execution. The last execution of the block produces a final result. The size of the result depends on the computation throughput on allocated compute resources (e.g. 1-wide or 2-wide reduced vector slice). The VCU stores the result to memory and general purpose code is used to reduce it to a single value.

When scaling DVX over multiple cores, each core executes a thread of vector instructions that reduces its partitions of vector operands. The VCU in each core stores its reduction result to memory. Instead of enabling complex communication between the cores to reduce multiple reduced slices, the general purpose execution performs the final step of the reduction and produce a single scalar value. Such design incurs performance overheads to load/store slices of different cores and finalize the reduction computation with scalar instructions. It is trade-off between performance and hardware complexity and while focusing on the segment of lightweight processors in this work we choose to avoid extra complexity.

### 4.2.6 DVX and Interrupts

The execution of classic vector instructions over large operands can incur significant commit/squash overheads when servicing interrupts. An interrupt routine either needs to wait for a "long" time until a current vector instruction commits all its results or need to discard "large" temporal results of the vector instructions. Vector AIBs operate over slices of vector operands. We leverage that to support a mechanism that services interrupts leaving a well-known state without having to discard any temporary result nor wait until whole execution finishes.

Without interrupts, a vector AIB allocates the core resources for many repetitive executions. When the AIB finishes all its executions, the resources are deallocated and the next AIB or following scalar instruction can continue execution. In the case of interrupt or exception, the AIB can commit its current execution (the AIB operations over the current slice of vector operands) and deallocate the resources for the service routine that handles the interrupt or exception. The PC of the AIB is saved to memory and the block is restored when the routine finishes its execution.

Although we do not implement the servicing of interrupts, we believe that it would be possible to improve the performance of DVX, by allowing the VCU to continue vector execution in the "background" while the service routine executes in scalar fashion. This would require the service routines with only scalar instructions and keeping the context of the VRs and VCU while the routines execute. If the code of routine has vector instructions, the state of the VCU as well as the state of the VRs have to be saved to memory before the routine starts its execution. The saving of large context from the VRs and VCU may incur unacceptable overheads before starting the routine. To avoid that problem, the vectorization might be applied only to service routines, which can tolerate timing overheads for the later benefits of vector execution. In this work, we only discuss possible solutions for servicing of interrupts and we do not further investigate their detailed design nor implementation.

## 4.3 Implementation of DVX

We implement DVX on a 4-core composable CMP based on an EDGE architecture. Two features of the EDGE-based CMP simplify our DVX implementation. First, the architecture already executes AIBs with EDGE instructions. The block model itself makes it easy to extend the core with DVX support for comparatively less overhead than a conventional general purpose core. Second, the EDGE CMP offers an on-chip-network that DVX leverages to control DVX threads. Additionally, the EDGE CMP has one feature that increases the efficiency of DVX. Namely, the statically encoded dataflow within the vector compute instructions in the EDGE AIBs enables power-efficient out-of-order execution of the vector compute instructions.

In the rest of this section we first explain the details of the DVX implementation. This implementation utilizes one core for non-parallel scalar execution and a configured number of EDGE cores for parallel vector execution. We next show an example of such implemented DVX. In the end, we discuss the advanced implementation that utilizes a configurable number of cores for both scalar and vector executions.

### 4.3.1 DVX on dynamic EDGE CMP

DVX extends the previous EVX implementation that specialize one EDGE core(see Chapter 3) for vector execution to dynamically scale the execution over the other cores available in the EDGE CMP. Figure 4.3 shows an EDGE processor and a block diagram of one core with support for DVX. The major changes over the baseline core are highlighted in the figure.

In the same way as in the EVX implementation, vector AIBs in the DVX implementation allocate the EDGE core's compute resources for the slice-by-slice computation over configurable large vector operands. The instruction window and reservation stations buffers are additionally divided into four banks to resemble parallel vector lanes and allow for processing of 4-wide slices per each execution of a vector AIBs. In our implementation on the EDGE cores with the 128-entry instruction window, this design limits the size of vector AIBs to have maximum 32 instructions (as same as one bank of the instruction window). Two general purpose ALUs are shared among the four lanes to perform the compute operations in consecutive cycles. The VCU is added to load/store the large vector operands to/from the VRs decoupled from the computation on the allocated resources. It issues memory requests directly to the L2 cache banks to increase the memory bandwidth; and transfers slice-by-slice of the vector operands from the VRs to the compute resources and back. The VRs hold the vector memory operands, as well as temporary vector results that may exist between dependent vector AIBs. In this implementation, 8 VRs repurpose a configurable amount of the L1 data cache resources to keep the DVX area overheads minimal, like in software managed caches of commercial GPUs [54]. Additionally, 4 dedicated scalar registers are incorporated for mixed vector-scalar operations. Along with slices of the vector operands in the VRs, the VCU transfers the scalar operands to the compute resources for each new execution of a vector AIB.

Apart from the evolved design from the EVX implementation, the DVX implementation adds the TCU that starts/stops slave threads by sending thread control messages via the existing interconnect on the dynamic EDGE CMP. Before executing a vector AIB, both VCU and TCU must be initialize with the DVX parameters. The parameters consist of: vector length, memory access patterns (e.g. vector base address, size of elements and stride between the elements) and number of cores to be used for DVX. One or more scalar AIBs with EDGE instructions need to be executed to initialize the dedicated DVX units (VCU/TCU) with these parameters (to initialize their control registers).

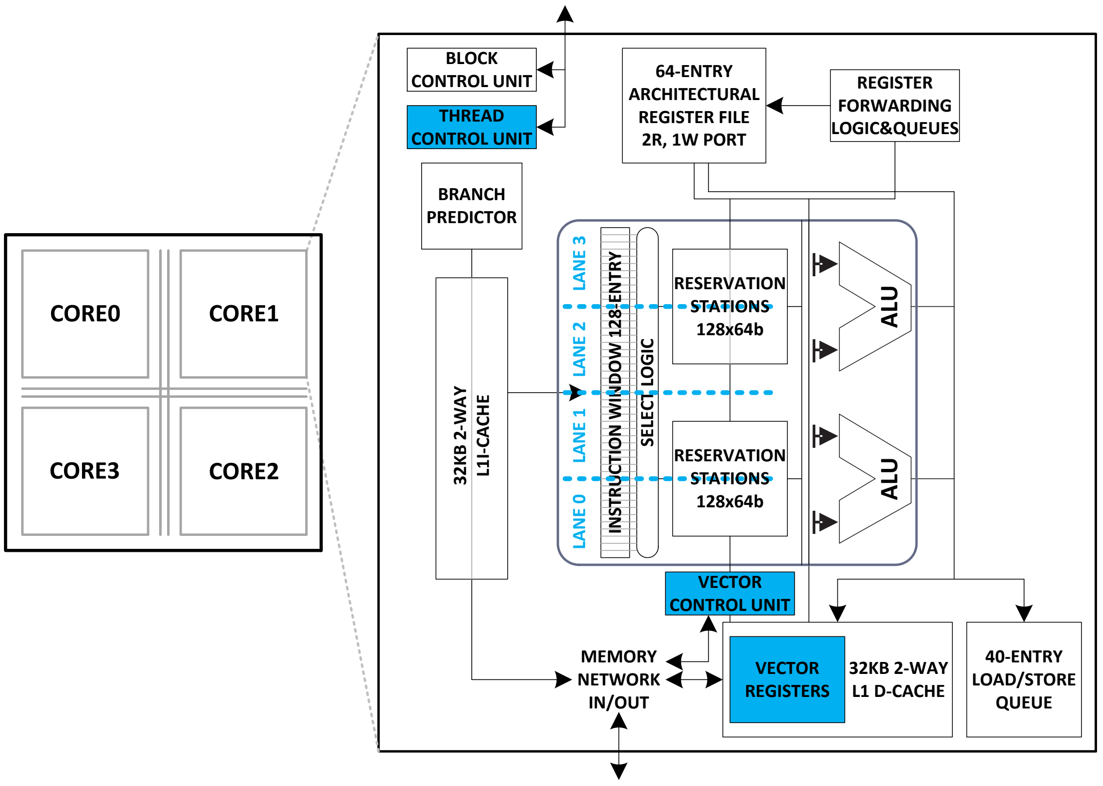


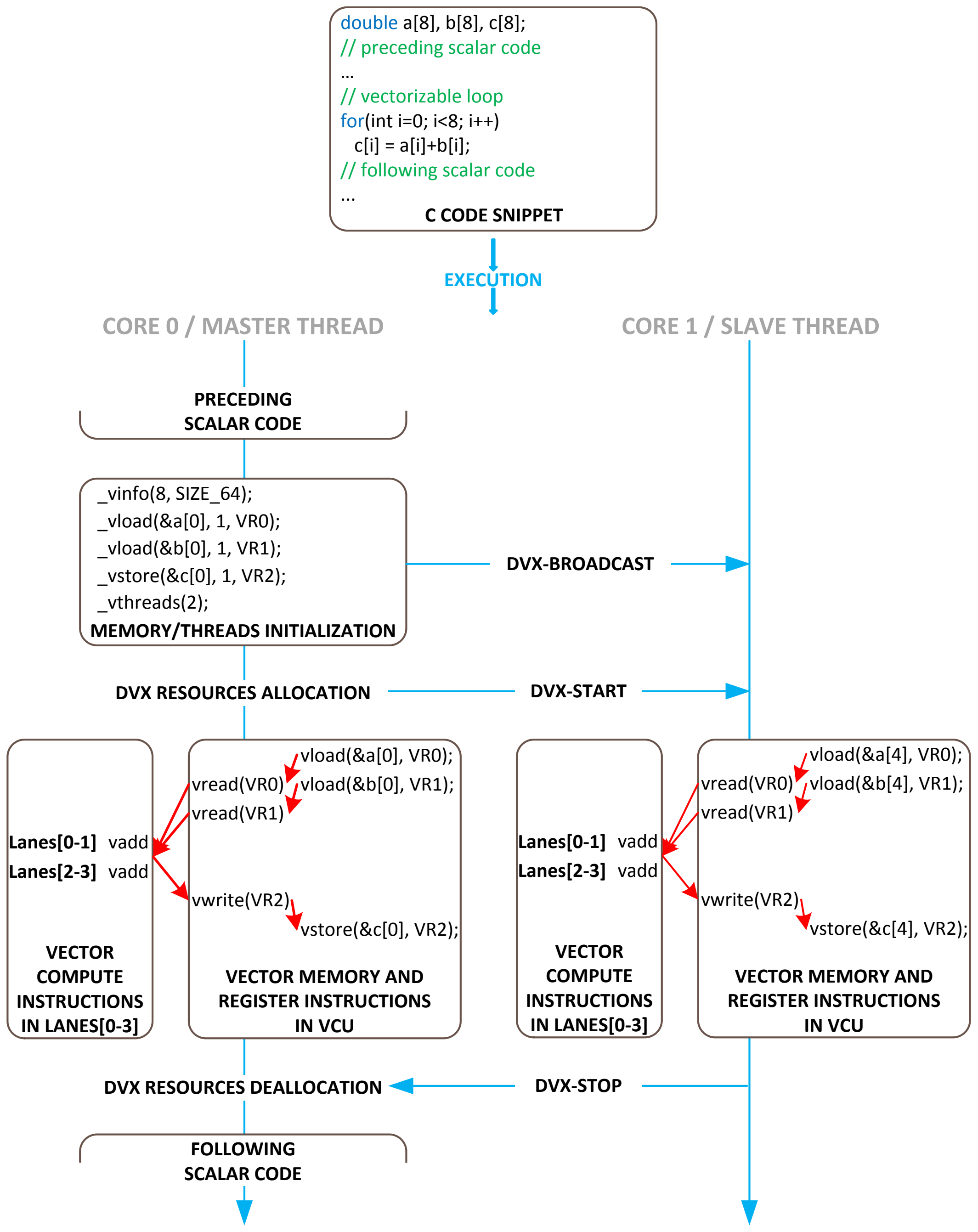
Figure 4.3: The DVX implementation on dynamic EDGE CMP. It banks the core compute resources to increase DVX compute throughput and repurposes the L1 data cache resources into the VRs to avoid extra area overheads.

When multiple cores are used only for parallel vector execution, the master threads executes the scalar AIBs on a single EDGE core and initializes the dedicated DVX units. When the master thread fetches a vector AIB, it allocates the compute resources on the configured number of adjacent cores by starting slave threads. The master thread broadcasts a *DVX-START* request with the address of the AIB to be executed by the slave threads. Each slave thread fetches the AIB and executes it on its partition of vector elements. The threads execute the same AIB, but not necessarily in lock step. The divergence between the cores occurs at the instruction level granularity inside the AIB or at the level of AIB executions. When the slave thread finishes its executions, it sends a *DVX-STOP* message to the master thread. The master threads commits the AIB when it receives all the completion messages. The slave threads are then terminated, cores are deallocated and the master thread initiates the execution of the next AIB on a single core. The next block can be a scalar EDGE AIB that continues non-parallel execution of the workload or an AIB that initializes the dedicated DVX units for the following vector computation.

While executing the AIBs with the DVX parameters, the master thread broadcasts the parameters to the VCUs/TCUs of all the cores that are going to be used for the following vector processing. The parameters are broadcast before the DVX compute block is fetched and any of the slaves threads started. When the VCU of one core receives the parameters, it calculates the start addresses for its partition of vector elements by leveraging the available core's ALUs. As soon as the addresses are calculated, the VCU starts executing the vector load instructions to prefetch as much as possible of the input vector data. On other hand, vector store instructions are initiated when the slave threads start and slices of the store operands in the VRs become available.

The distribution of vector operands among the cores is simple and straightforward. It divides the operand into equally sized partitions of vector elements, based on the number of cores (e.g. vector of 1024 elements is divided to 256-element partitions for 4 DVX cores). In the case when the total number of vector elements is not a multiple of the number of cores, the core with the largest identifier takes care of the remainder. Such distribution of vector operands enables DVX to efficiently increase the amount of the allocated resources without further communication between the cores. Additionally, the issuing of DVX memory requests directly to the L2 cache banks avoids possible overheads of data movement operations between the L1 data caches in multiple cores allocated for DVX.

The VCU calculates the base addresses for its partitions of vector operands without additional software nor hardware support. It utilizes a simple algorithm that increments the operand base address by the offset value for the previous partitions; and the offset is calculated by multiplying the core identifier, the number of elements in the partition, the stride between the elements and the size of each element. The VCU leverages the existing ALUs to perform this calculation. The rest of the DVX parameters in the VCU remain unchanged (e.g. the stride between vector elements, the size of elements).



```
double a[8], b[8], c[8];
// preceding scalar code
...
// vectorizable loop
for(int i=0; i<8; i++)
  c[i] = a[i]+b[i];
// following scalar code
...
```

```
_vinfo(8, SIZE_64);
_vload(&a[0], 1, VR0);
_vload(&b[0], 1, VR1);
_vstore(&c[0], 1, VR2);
_vthreads(2);
```

Figure 4.4: An example of 2-core DVX that performs the addition of 8-element vector operands. Each core executes the same vector compute, register and memory instructions over 4-element partitions of the vector operands.

### 4.3.2 DVX Example

Figure 4.4 shows an example of the loop in general purpose C-code and its DVX execution. The loop contains a simple DLP kernel, which performs the addition of 8-element vectors a and b and stores the result to vector c. The code of the loop is modified (vectorized) to use DVX by describing the decoupled DVX memory execution, a preferred number of specialized cores and the vector AIB that performs the loop's computation. One core is running the master thread and executes both the scalar and vectorized parts of the code. It executes the code that initializes patterns of vector memory instructions and the number of DVX threads. Two memory instructions are described to load vectors a and b to VR0 and VR1, and one instruction to store its operand from VR2 to vector c. Each instruction utilizes sequential memory access pattern and operates over eight 64-bit element vectors. When the master thread commits the AIBs with the DVX parameters, it broadcasts all the parameters to other free cores that are going to run slave threads. This example defines DVX with two threads and assumes that one extra core is free to take over the execution of slave thread.

When the master thread fetches the vector AIB, it sends a DVX-START message that creates slave thread on another core. By using the vector AIB, each thread allocates its core's compute resources and executes the same vector compute instructions over the 4-element partitions of 8-element vector operands. The AIB contains the vector compute instructions (*vadd*), as well as the vector register instructions that connect the DVX computation on general purpose resources and the DVX decoupled memory execution (*vread(VR0), vread(VR1), vwrite(VR2)*). The dedicated VCUs execute the initialized memory instructions and utilize the register instructions to transfer the slices of vector operands between the VRs and allocated resources. In our particular implementation with 4 vector lanes (4 banks), the AIB executes only 1 time (1x4) on each core to process its 4-element partition of vector operands. When both threads finish the execution of the vector AIB and the master thread receive the DVX-STOP message, only the master thread continues the execution of the scalar AIBs. The core that executed the slave thread remains idle until the master thread fetches the next vector AIB.

### 4.3.3 DVX and Composing Cores for Scalar Execution

The dynamic EDGE CMP can tune its resources by composing one or more cores into a logical processor, which executes single-threaded applications (see Section 2.2.2.1). Instead of using one core to run the master thread, the composed logical processor executes the master thread over multiple cores to improve performance of scalar phases in DLP applications. In this case, one core executes the oldest AIB with scalar instructions, while the others execute predicted scalar AIBs speculatively. The AIBs that initialize the DVX parameters for the VCUs/TCUs can execute speculatively as well. These AIBs broadcast their results to other cores in the processor at their commit time to avoid speculative initialization of these units. The VCU of each core in the processor starts executing the vector load instructions as soon as their parameters are initialized. The vector store instructions on the contrary wait until their store operands in the VRs become available and non-speculative; at this moment they start storing them to the L2 cache memory.

The core that first fetches a vector AIB continues the execution of the master thread alone. That core immediately flushes the younger speculative blocks on the other cores and makes them free to start slave threads with the same AIB. The master thread may start its execution speculatively, because the vector compute block may be predicted; in this case, DVX execution in each thread is speculative and vector results are buffered in VRs. If the vector AIB is flushed, the results are discarded from the VRs and the VCUs restart the execution of their vector load instructions. If the master thread becomes non-speculative, all the slave threads become non-speculative; at that moment, the vector store instructions in each core can start sending their results to the L2 cache memory; each AIB running in slave thread commits when it completes its execution, whereas the AIB in the master thread additionally waits for all slave threads to finish the execution. When all slave threads finish, the master continues scalar execution of the scalar AIBs on the composed cores, until the next vector AIB is fetched.

## 4.4 Conventional alternative to DVX

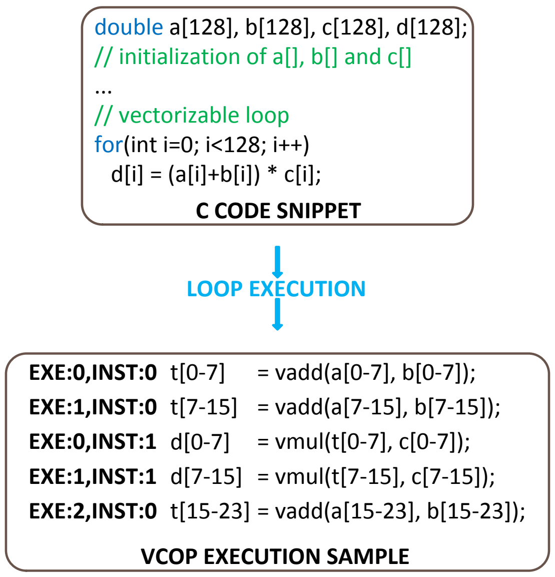
To compare to DVX, we designed a dedicated DLP accelerator with the DVX ISA and special-purpose hardware resources. This accelerator is incorporated into an EDGE CMP in a form of offloading engine. It combines the EDGE architecture with designs of classic vector (co)processors (VCOP). As opposed to specializing general purpose cores for DVX, our VCOP incorporates a significant amount of extra resources that are exclusively used for DLP acceleration. The comparison of DVX to such VCOP shows different trade-offs between dynamic specialization of general purpose cores and static specialization of the hardware in a more complex heterogeneous processor.

In the rest of this section, we first describe the VCOP design, its implementation and integration with the host EDGE processor. In the end, we summarize the differences between the DVX and VCOP designs and resources.

### 4.4.1 Classic VCOP in EDGE Fashion

VCOP is designed to incorporate the advantages of classic vector and EDGE architectures on the same substrate. To improve performance like advanced vector designs [16, 61], VCOP decouples the execution of vector compute and memory instructions. Since VCOP is incorporated into an EDGE CMP, it is design to be similarly efficient. For this reason, VCOP leverages the EDGE mechanism for out-of-order execution of vector compute instructions.

VCOP executes EDGE-based vector AIBs like DVX enabled EDGE cores. A producer vector instruction encodes its targets, which may be a left or right operand of consumer vector instructions. Instead of using a classic vector register file, each compute instruction uses two vector reservation stations buffers, which are sized to hold large vector operands. The producer instruction writes its results to the reservation stations of their consumer instructions. The instructions execute when they have their operands ready, in dataflow order, and the order is defined statically within the instruction encodings. Besides two reservation stations buffers for each compute instruction, VCOP utilizes a set of vector registers (VRs) and a modest set of dedicated scalar registers. The VRs hold temporary vector results between the vector AIBs and the input/output operands of the



```
double a[128], b[128], c[128], d[128];
// initialization of a[], b[] and c[]
...
// vectorizable loop
for(int i=0; i<128; i++)
  d[i] = (a[i]+b[i]) * c[i];
```

```
EXE:0,INST:0  t[0-7]   = vadd(a[0-7], b[0-7]);
EXE:1,INST:0  t[7-15]  = vadd(a[7-15], b[7-15]);
EXE:0,INST:1  d[0-7]   = vmul(t[0-7], c[0-7]);
EXE:1,INST:1  d[7-15]  = vmul(t[7-15], c[7-15]);
EXE:2,INST:0  t[15-23] = vadd(a[15-23], b[15-23]);
```

Figure 4.5: An example of the enabled execution model in VCOP. The instructions operate over consecutive 8-wide slices of their vector operands in VCOP.

vector memory instructions. The scalar registers hold the operands for mixed vector-scalar operations.

On the contrary to DVX, the VRs and reservation station buffers in VCOP are equally sized. It allows VCOP to perform computation of large vector operands without transferring the operands from the VRs to the reservation stations first as in DVX. In VCOP, each vector compute instruction dependent on an operand in a VR utilizes the operand directly from the VR. The compute instruction multiplexes the input to the ALU from the VRs, instead of its reservation station. This design resembles the chaining of vector memory and compute operations typically found in classic vector processors. We show the benefits of the enabled chaining in Section 4.6. In a similar way, the vector compute instruction can use its operand directly from the scalar registers when it operates over mixed vector and scalar operands.

The size of the vector operands is dynamically configured and it is only limited by the size of the VRs/reservation station buffers. Each compute instruction performs the configured number of operations over the elements of its vector operands. The operations execute in lockstep on each ALU available in VCOP, like in classic vector designs. To maximize the performance of out-of-order vector EDGE computation in VCOP and allow for more competitive comparison to DVX, VCOP enables a more sophisticated execution model than classic vector processors. Namely, the instructions in VCOP do not wait until the entire operands arrive to their reservation stations nor execute over the entire vector operands like in classic vector processors. Instead, vector instructions in VCOP issue multiple times and each time they operate over ready slices of their vector operands. The size of the slices matches the size of the compute resources used for one lockstep execution (e.g. the size of 256-bit slices matches the size of eight 64-bit ALUs). Note that this execution model resembles the execution model of modern GPUs, where a group of threads (e.g. 32 threads in NVIDIA GPUs) execute in lockstep on multiple ALUs as vector instructions in VCOP. While different groups in GPUs execute out-of-order without dependency, in VCOP there is one restriction that simplifies its implementation; each vector instruction performs the encoded compute operations sequentially over consecutive slices (slice[0], slice[1], slice[2]...) of its operands and sends slices of the computed result to the reservation stations of the encoded consumer instructions. The select logic checks if an instruction has the ready slice of all its operands instead of the entire operands to mark the instruction ready for the execution. It allows for forwarding of incomplete vector results between dependent instructions in a vector AIB, and resembles chaining of vector instructions at low implementation cost. The compute process repeats until all the instructions in the AIB perform the configured number of operations and the block produces its vector results. The example of the proposed execution model is shown in Figure 4.5. In this example, two vector instructions (*vadd, vmul*) repeat their operations over consecutive slices of their vector operands. The operations of the vadd and vmul instructions are chained, by forwarding the incomplete results of the vadd instruction to the reservation station of the dependent vmul instruction. We evaluate the benefits of our enabled execution model in Section 4.6.

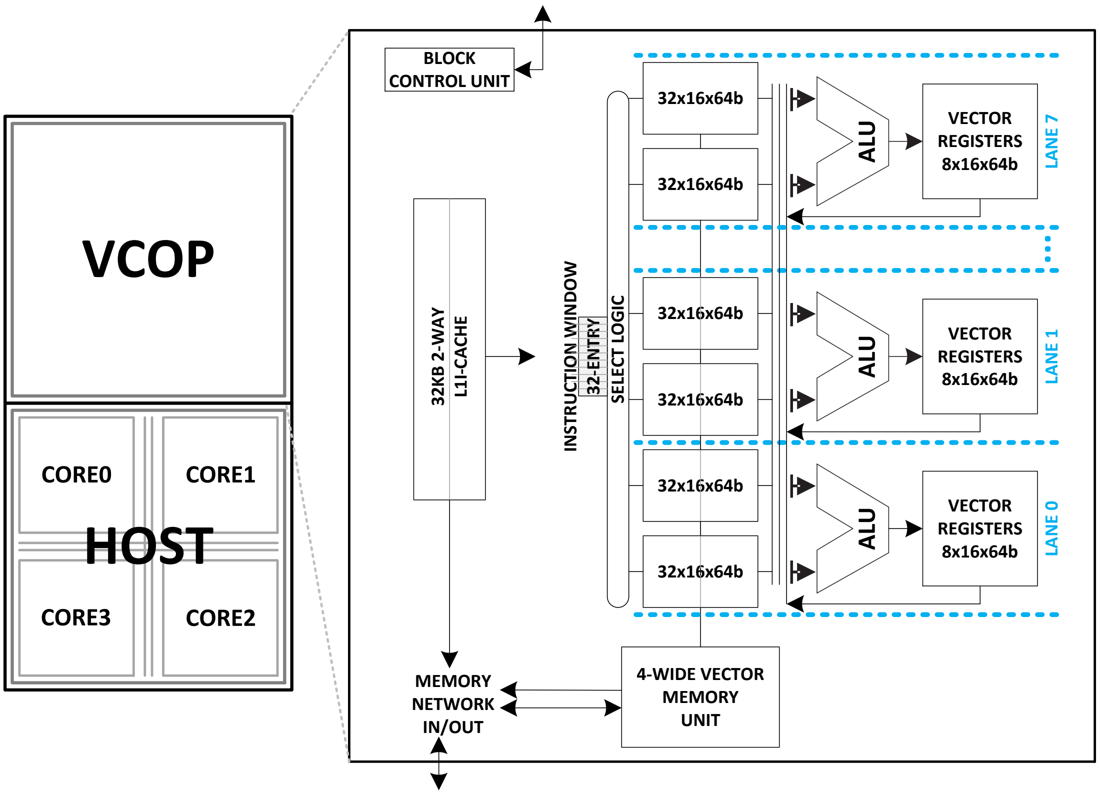


Figure 4.6: Microarchitectural components of the EDGE VCOP. It has the instruction window with two vector reservation stations buffers sized for 32 instructions and eight vector registers. Out-of-order vector computation leverages the eight ALUs in lockstep. The vector memory unit decouples the vector memory execution from the computation and issues up to 4 memory requests per cycle.

### 4.4.2 VCOP Implementation

To make VCOP implementation 4.6 fairly comparable to DVX on an EDGE CMP, we limit the size of vector AIBs that execute on VCOP to have a maximum of 32 instructions. It makes the instruction window in VCOP sized to hold 32 instructions and two reservation stations buffers sized to hold 32 vector operands (32 active instructions in the instruction window of VCOP, each one may have up to two vector operands). To enable the execution of vector instruction over slices of their vector operands, each instruction in the instruction window has a pointer that sequentially iterates its slices. The select logic utilizes this pointer for each active instruction in the instruction window to check if the instruction has its pointed slice of all its operands ready; if this is the case, the logic selects and schedules the instruction to issue its operation over the slice of its operands.

To be competitive to the DVX enabled CMP, VCOP incorporates 8 conventional VRs and 4 scalar register. It also adds the same amount of execution resources as DVX leverages from the CMP. VCOP uses eight 64-bit ALUs, which support sub-word SIMD operations over 8-,16- ,32- or 64-bit elements. This corresponds to the 2 ALUs per core of the 4-core EDGE CMP that we use for evaluating DVX (see Section 4.6). VCOP issues vector compute instructions in a lockstep to each ALU, while simultaneously performing eight 64-bit operations over 256-bit slices of vector operands. The reservation stations and the VRs are banked across the ALUs to reduce the number of ports and their complexity.

The execution of vector memory and compute instructions is decoupled to increase efficiency like in DVX enabled cores and this design decision differs significantly from the non-decoupled memory execution in modern GPUs. Vector memory unit (VMU) in VCOP executes the memory instructions and uses the conventional VRs to hold their operands. It simultaneously executes up to 4 memory instructions and allows for issuing of 4 load/store memory requests per cycle. The memory instructions in VCOP support sophisticated addressing modes like memory instructions in DVX enabled cores. Similarly, the host processor initializes the VMU with the access patterns for each memory instruction, before executing the vector AIBs in VCOP.

VCOP supports partial reductions of large vector operands. It uses a single 256-bit register that holds the temporary results of 8 lockstepped 64-bit ALU operations over slices of vector operands. While pipelining the operations over large vector operands to the lockstepped ALUs, VCOP uses the value from the register as an input of each subsequent operation and stores the result back to the register. It repeats the same process until the entire vector operands are computed and produces a final 256-bit reduction value. To eliminate complex inter-ALU connections, VCOP the same as DVX cores stores the final reduction value from the register to the memory; and the host processor finalizes the reduction process by using scalar instructions.

### 4.4.3 Integration of VCOP with EDGE CMP

The heterogeneous EDGE processor incorporates VCOP to a host EDGE processor to offload and accelerate DLP workloads. The host is a general purpose processor, which executes non-parallel phases of applications and offloads the parallel vectorized phases onto the VCOP. The host processor executes instructions that initializes and controls the vector execution on VCOP. When requested by the host, VCOP executes vector compute and memory instructions.

To provide maximal performance achievements of the acceleration, the host processor and VCOP are tightly integrated and they share the L2 cache memory. VCOP executes vector load/store instructions by issuing memory requests directly to the L2 banks shared with the host processor. It avoids the overheads, which would exist for data movement operations between the host and VCOP in the case when they have the separated memory resources.

The host executes EDGE-based scalar AIBs. When it fetches a vector AIB, the host sends the address of the block to VCOP. It overtakes the execution of the vector AIB and notifies the host when it completes the execution. The vector execution is constrained to the boundaries of the AIB, as opposed to execution in classic vector processors that can start executing the following vector instructions as soon as there is space in the instruction buffer. In order to tolerate this limitation, host processor continues executing beyond the current vector AIB instead of waiting VCOP's response to commit the block. By overlapping the execution of the vector and the following scalar AIBs, the host processors can execute instructions that initializes VCOP with the access patterns for the memory instructions in the next vector AIB. The host continues execution until it fetches the vector AIB that has to be offloaded to VCOP. When VCOP finishes the execution of the vector AIB, it starts executing the next vector AIB without initialization nor startup overheads. We evaluate the benefits of the overlapped execution in Section 4.6.

Although in this work we model only a simplified heterogeneous processor system without virtual memory nor interrupts, there is missing an important point of the proposed overlapped execution design. Since an exception can happen in the vector AIB (e.g. page miss), the overlapped execution of the host processor

|  | Description |  |
|---|---|---|
|  | DVX | VCOP |
| Design principle | morphing general purpose cores to vector cores | classic based vector core adopted to the EDGE |
| Vector buffers (registers/stations) | hold operands between instruction blocks | hold operands for each instruction |
| Buffers resources | repurposing data cache | banked vector register file |
| Compute resources | independent ALUs in multiple EDGE cores | lockstepped ALUs in single VCOP core |
| Execution of compute instructions | repeated executions over consecutive vector slices | repeated executions over random vector slices |
| Execution of memory instructions | decoupled, one instruction per cycle in multiple cores | decoupled, multiple instructions per cycle in a single core |
| Addressing modes | strided, indexed, masked |  |
| Reduction | partially supported |  |

Table 4.1: DVX/VCOP design overview.

and VCOP is unsafe. We call this "unsafe execution". To preserve the consistent architectural state in the case of exception during the unsafe execution, the host processor buffers the results of the following AIBs without committing them, until VCOP finishes the execution of the vector AIB. At this moment, the host processor commits the results of the unsafely executed AIBs and returns back to the normal "safe execution".

### 4.4.4 DVX/VCOP Resources and Design Comparison

DVX leverages the existing general purpose resources of EDGE cores to perform vector execution of DLP workloads without offloading onto the accelerator. On the contrary to DVX, VCOP is a dedicated DLP accelerator added to an EDGE CMP to offload the DLP workloads. VCOP is sized to be comparable to a 4-core EDGE CMP specialized for DVX. VCOP has the same amount of compute and memory resources as 4 DVX cores and combines the same memory bandwidth, by sharing the L2 cache banks with the CMP.

Table 4.1 summarizes the differences and similarities of DVX and VCOP designs. On the contrary to the implemented DVX that repurposes the L1 data resources into the VRs, VCOP design includes more sophisticated resources such as a large vector register file and reservation stations that hold vector operands for each vector instruction; such classic design does not require transferring of vector slices from/to the VRs for each vector instruction on the allocated general purpose resources of DVX cores. VCOP also incorporates multiple lockstepped ALUs organized in vector lanes, while DVX leverages the ALUs available in the EDGE cores. Although utilizing radically different design approaches, both VCOP and DVX execute compute vector instructions over slices of vector operands. DVX repeats the execution of the compute instructions over consecutive slices of vector operands and VCOP over independent slices like groups of threads on GPUs. DVX scales the execution of vector compute instructions over the ALUs in one or more DVX cores, whereas VCOP executes the compute instructions in lockstep on the ALUs in a single core. Both DVX and VCOP decouple the execution of vector memory and compute instructions. DVX scales the execution of each memory instruction over multiple cores, and each core issues independent memory requests. To equally saturate the memory bandwidth, VCOP simultaneously executes different memory instructions and issue their memory requests in parallel. DVX and VCOP support the same addressing modes as well as partial reductions of the large vector operands.

## 4.5 Experimental Setup

### 4.5.1 Simulator

We evaluate an EDGE CMP, the proposed DVX technique and the alternative VCOP design by using a detailed, timing, in-house simulator from Microsoft Research. The simulator is written in SystemC to accurately model the baseline EDGE cores, DVX enabled cores and VCOP with the parameters shown in Table 4.2. We experiment with DVX dynamic configurations with 1, 2 and 4 cores and with VRs sized to hold 32 or 128 elements of 64 bits per each register. We experiment with VCOP static configurations that have 4 and 8 vector lanes and the VRs sized to hold 32 and 128 elements of 64 bits per register. We investigate the area, the runtime dynamic and leakage power consumption of DVX and

| Component | Description | |
|---|---|---|
| | DVX core | VCOP core |
| ALUs | 2 Integer/FP | 8 Integer/FP |
| Reservation Stations | 128 x 2(left/right) x 64-bit | 32 x 2(left/right) x [128-512] x 64-bit |
| Register File | 64 entry | - |
| Load-Store Queue | 32 entry unordered | - |
| L1 I-cache | 32 kB, 1 cycles (hit) | 32 kB, 1 cycles (hit) |
| L1 D-cache | 32 kB, 1 cycles (hit) | - |
| L1/L2 MSHRs | 8 entry | 8 entry |
| DRAM | 250 cycles | 250 cycles |
| Branch Predictor | OGEHL | - |
| On-Chip-Network | 1 cycle/hop | 1 cycle/hop |
| Vector Registers | 8 x [32—128] x 64-bit | 8 x [128—512] x 64-bit |
| Vector-MSHRs | 32 entry | 4 x 32 entry |
| Total Cores | 4 | 1 |
| L2 | 4 banks x 512 KB, 15 cycles (hit) | |

Table 4.2: Simulator configuration.

VCOP by using McPAT [45] area and power models that we developed for a general purpose EDGE CMP, the CMP with DVX and the heterogeneous CMP with VCOP. We extended the existing in-order McPAT models to build the models of EDGE processors with DVX and VCOP, since the EDGE architecture avoids most of the structures typically required for out-of-order execution. The models assume 32nm low power technology.

### 4.5.2 Benchmarks

For our evaluation we use the same Livermore Loops kernels like for for the evaluation of EVX (Table 3.2), while excluding the selected applications from the San Diego Vision Benchmark Suite. The reason for this is an unacceptable simulation time when running the Vision Benchmark applications.

### 4.5.3 Methodology

Running various kernels shows the benefits and limitations of our vector model, as well as our more conventional approach. The number of iterations in each kernel is configured to 1024, which provides enough large vector operands to saturate vector execution on various DVX and VCOP configurations. The kernels operate over 32-bit elements and they are evaluated after warming up the caches to avoid DRAM memory access overheads of scalar, DVX and VCOP executions. To saturate the resources of VCOP's memory unit that execute up to 4 different memory instructions at each cycle, we apply loop-unrolling optimization to all the kernels running on VCOP with less than 4 memory instructions. Note that the kernels running on DVX enabled cores do not require such optimization, because DVX memory units at each core can execute the same memory instruction over different partitions of vector operands.

To simplify evaluation of DVX and VCOP, in this work we do not investigate the execution that dynamically changes the number of cores (e.g. one core for scalar and one or more for vector execution phases). We configure the number of cores for each kernel at the beginning of its execution, and do not change the number of cores at runtime. The entire kernels, including their non-vectorizable parts such as the initialization of the VMUs/TCUs are executed with multiple cores (see Section 4.3.3). The warming up of the caches is performed with 1, 2 or 4 cores for 1-,2- or 4- core DVX configurations respectively; and 4 cores are used to warm up the caches in the heterogeneous processor with VCOP. It enable higher and competitive performance of DVX and VCOP, by avoiding possible overheads for data movement operations from a single bank of the L1 data cache (warming up writes the data to the L1 cache) into the L2 cache banks (DVX/VCOP loads the data from the L2 cache).

## 4.6 Results

### 4.6.1 Performance

Figure 4.7 shows the speedup of DVX over the previously proposed vector execution for an EDGE core (EVX), when it uses a custom low level fork/join implementation of threads to scale it over multiple cores. The fork/join functions avoid runtime overheads, including the synchronization with locks to provide the fastest thread control. Each slave thread just spins, by reading a shared variable

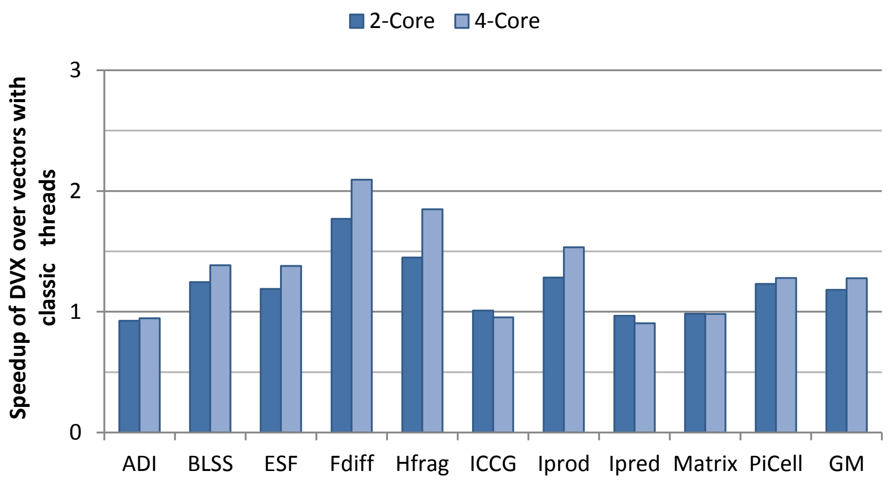


Figure 4.7: Speedup of DVX over EVX that uses conventional threads to scale over multiple cores in a CMP.

to start the vector execution, when the master thread has work to spawn across the cores. In the same way, slave threads notify back the master thread when they finish execution. The master thread controls the execution of the EVX slave threads, as well as performs the vector computation. Although inefficient due to actively reading the shared variable, the custom fork/join implementation is the fastest approach to be compared to DVX with hardware controlled threads. Note that such fast scaling with conventional threads requires all the threads to be created in advance as well as to be ready for the the vector execution. The overhead of using shared variables (and associated coherence protocol messages and movements of the shared variable between the L1 caches) to synchronize the master and the slave threads is quite significant for short kernels (*Fdiff, Hfrag, Iprod*); and increasing the number of used cores shows even more benefits of DVX. More complex kernels execute for longer time and thus they have no significant performance impact when using threads. Moreover, some kernels (*ADI, ICCG, Ipred*) with large amount of data in the L1 caches show a very slight slowdown of DVX compared to the scaled EVX with threads. This happens because in EVX, slave threads are spinning on the cores without data in the L1 data cache; and while leveraging the cache for the VRs, the slave threads avoid flushing the dirty lines back to the L2 level like DVX cores. The design decision for the hardware

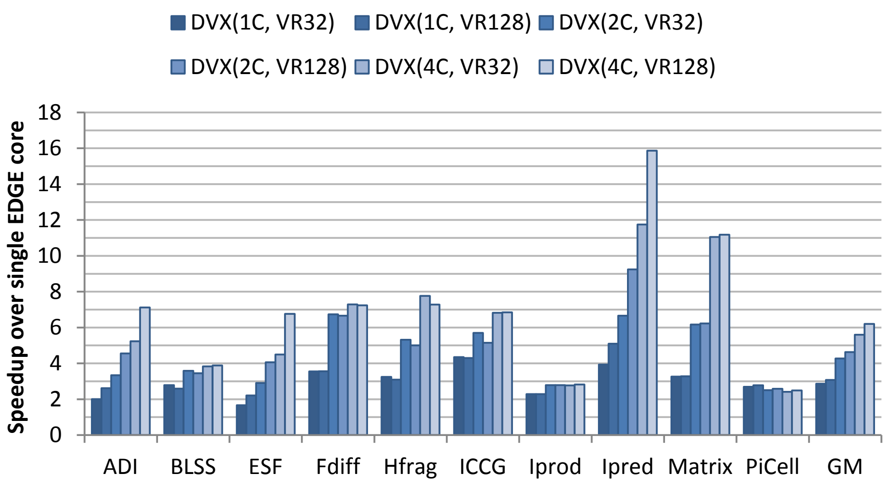


Figure 4.8: Speedup of DVX with different dynamic configurations.

orchestrated DVX threads allows by 27% higher average performance. The DVX threads do not require inefficient spinning nor significant timing overheads to start their execution. DVX dynamically creates slave threads and outperforms the limited EVX approach with statically created threads of vector instructions.

Figure 4.8 shows the speedup of diverse DVX configurations over the general purpose execution on a single EDGE core. DVX is dynamically configured to use 1, 2 or 4 cores and the VRs to hold 32 and 128 elements of 64 bits (or 64 and 256 elements of 32 bits). While increasing the size of the VRs, DVX with 1, 2 and 4 cores improves the performance for the kernels that exploit temporal locality between different vector AIBs in the VRs (*ADI, ESF, Ipred*). The more detailed impact of the VRs size on the performance of the specialized vector execution on an EDGE core is discussed in Section 3.6). In this section we further focus on configuring the different number of the cores for the specialized vector processing. For both sizes of the VRs, DVX significantly scales its performance when increasing the number of cores. For example, with the VRs sized to hold 128 elements, 4-core configured DVX outperforms 1-core DVX by over 2x in average and it outperforms 2-core DVX by about 1.33x; 4-core DVX achieves up to 16x of the maximum speedup (*Ipred*) and 6.2x of the average speedup. The kernels with sufficient amount of parallel compute or memory operations (*ADI, ESF, Fdiff, Hfrag, ICCG, Ipred, Matrix*) scale their performance when increasing

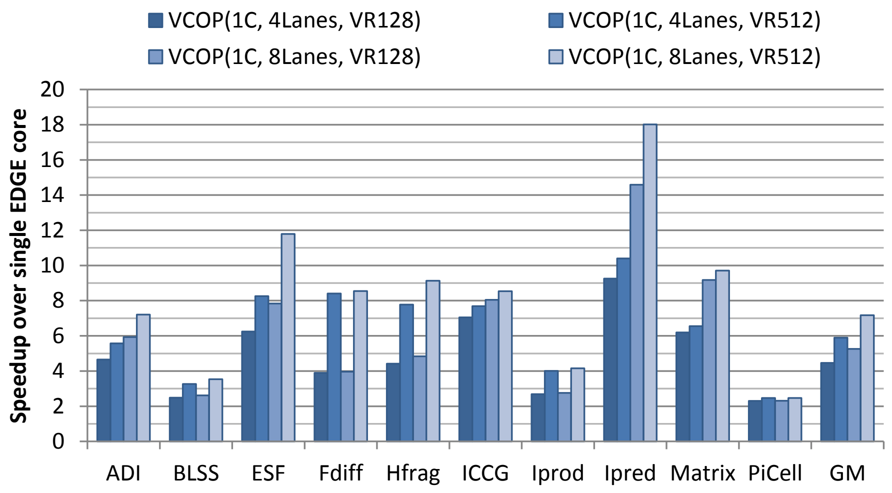


Figure 4.9: Speedup of VCOP with different static configurations.

the number of cores for DVX. On the other hand, reductions and not entirely vectorized kernels do not yield such performance when increasing the amount of allocated DVX resources. Reduction kernels (*Blss*, *Iprod*) achieve up to 4x of the speedup by exploiting the DVX reduction mechanism on each core. Their performance does not greatly scale with the number of cores, because each core reduces only its partition of the vector operands and general purpose execution produces the final result. *PiCell* is an example of a not completely vectorizable kernel, which cannot exploit the additional DVX resources.

Figure 4.9 shows the speedup of various configurations of VCOP over a general purpose execution on a single EDGE core. As opposed to DVX configurations, in this experiment we statically change the VCOP parameters. We experiment with different number of vector lanes and sizes of the VRs, as well as the sizes of the reservation stations in VCOP. The speedup is included for 4- and 8- lane VCOP with VRs/stations sized for 128 (VCOP-128) and 512 elements (VCOP-512). The 4-lane VCOP matches the amount of compute and memory resources of 2 DVX cores, whereas 8-lane VCOP utilizes the same amount of resources as 4 DVX cores. Increasing the size of the VRs yields over 32% and 36% of extra performance in 4-lane and 8-lane VCOP respectively. The reason for such great improvements is the reduction of startup (memory latency) overheads between the vector AIBs that execute on VCOP. The overheads exist because *strip-mining*

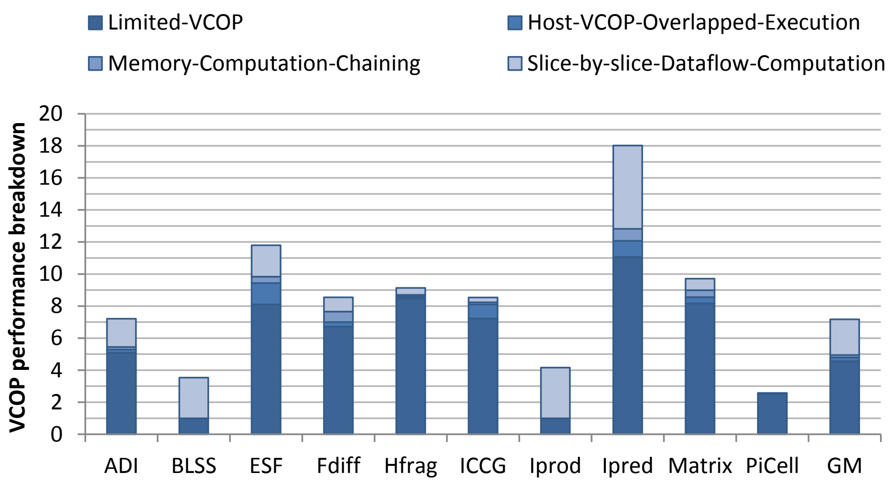


Figure 4.10: Performance breakdown of VCOP.

is required to process 1024 iterations on VCOP with the small VRs. Although the host processor overlaps the initialization of the memory unit in VCOP, the startup latencies are quite remarkable in kernels without complex vector computation to compensate them (*Fdiff, Hfrag*). Increasing the number of vector lanes improves the performance of VCOP for kernels with sufficient amount of DLP (*ADI, ESF, ICCG, Ipred, Matrix*). The average performance improves by 18% and 22% for VCOP-128 and VCOP-512 respectively. In the same order, they achieve an average speedup of 6.2x and 7.2x. In the rest of our experiments, we only utilize 8-lane VCOPs to have the equal amount of the compute and memory resources like the 4-core EDGE processor with DVX.

Figure 4.10 shows the performance breakdown of VCOP-512 by incorporating one by one different hardware features of VCOP to measure the impact of each one of them. We have evaluated several VCOP features: *host-VCOP-overlapped-execution, memory-compute-chaining and slice-by-slice-dataflow-computation.* We explain the benefit of each feature and discuss its impact on the performance. The most restricted VCOP (*limited-VCOP*) with all these features disabled achieves 4.55x of the average speedup. VCOP with overlapped execution of instructions on VCOP and its host processor minimizes startup overheads of vector AIBs on VCOP; it improves the VCOP performance by an extra 5%. The chaining of the vector memory and compute operations enables the vector computation over

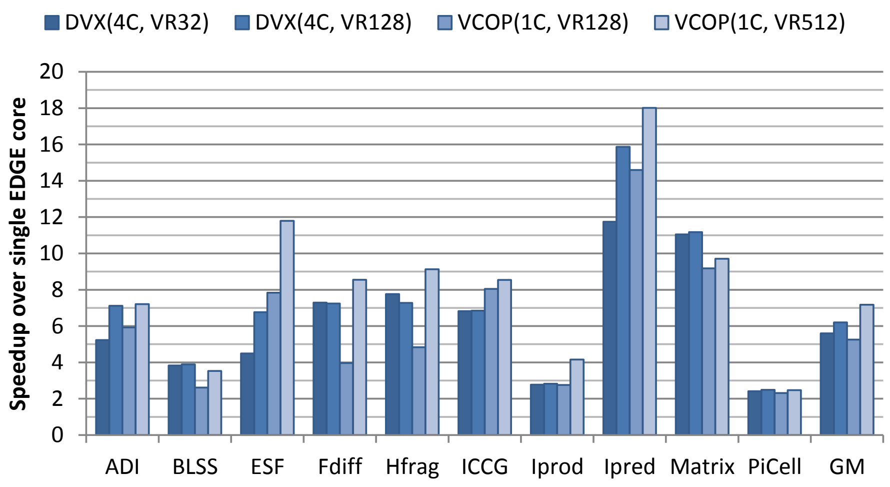


Figure 4.11: Relative performance comparison of DVX and VCOP.

dependent partially available memory results; and enables the vector memory execution over dependent partially available compute results. It yields 3% of the additional VCOP performance improvements. Instead of operating over the entire vector operands per each issue of a vector compute instruction, the instructions in VCOP can repeat their issue by performing the compute operations over ready slices of the vector operands. This computation of the large vector operands in VCOP enables performing partial reductions at low implementation costs(*BLSS, Iprod*), and increases the average VCOP performance by over 45%.

Figure 4.11 shows the performance comparison of DVX configured to use 4-cores and VCOP with the same amount of compute and memory resources. The specialized VCOP with the VRs sized to be the same as the VRs on 4 configured DVX cores achieves only 1.16x of the speedup over DVX enabled general purpose cores. The selected kernels with their diverse DLP characteristics emphasize the advantages and limitations of DVX and VCOP:

- The kernels with a single vector AIB show the benefits of the *streaming mode*. 4-core DVX with 32-element VRs outperforms VCOP-128 (*BLSS, Fdiff, Hfrag*) by avoiding the VCOP startup memory overheads. Its impact is especially remarkable in kernels without complex vector computation (*Fdiff, Hfrag*). *ICCG* is an exceptional streaming kernel that repeats the processing in an outer loop. While decreasing the vector length in each iteration of the outer loop, the streaming benefits becomes unimportant, as well as the advantages of DVX streaming mode.

- *Configurable large VRs* improves the performance of DVX for kernels that use multiple vector AIBs (*ADI, ESF, Ipred*), because the VRs exploit the temporal locality between the vector AIBs in the *register mode*. Increasing the size of VRs in VCOP is beneficial for all the kernels, because VCOP lacks the *streaming mode*. VCOP outperforms 4-core DVX configuration with the equivalent VRs for the non-streaming kernels, but requires extremely large VRs of 512 elements to be competitive to DVX for the streaming kernels. With such VRs in VCOP, 4-core DVX outperforms VCOP for only two kernels *BLSS, Matrix*.

- *Vector-scalar operations* show the advantage of VCOP in kernels where they are dominant (*ESF, Ipred*). VCOP leverages conventional approach where vector instructions for these operations use combined vector and scalar registers. On the other hand, DVX transfers slices of vector operands or single scalar values through the allocated general purpose resources and loses performance while waiting for the scalar value in each execution.

- DVX on each core processes its partition of vector elements, while allowing vector memory instructions to equally distribute memory requests to the L2 cache banks. On the other hand, VCOP uses loop unrolling inside the vector AIB to saturate the resources of the vector memory unit and in the some cases *BLSS, Matrix* the saturated unit cannot equally exploit the L2 cache banks.

- *Dynamic configuration of hardware resources* shows the advantage of DVX for the kernels without sufficient amount of DLP (*PiCell*). Such kernels usually fit in the 1-core DVX configuration. DVX dynamically tunes its resources to match an amount of DLP offered by each workload. On the other hand, VCOP potentially underutilizes its statically allocated resources on the workload without enough DLP to saturate them.

| Structures | DVX (4C) | VCOP (1C) | |
|---|---|---|---|
| VR Size | 32 & 128 | 128 | 512 |
| Fetch, Decode, L1I | $0.258 \times 4$ | 0.158 | 0.158 |
| Issue | $0.005 \times 4$ | 0.001 | 0.001 |
| LS Unit, L1D | $0.237 \times 4$ | – | – |
| RegFile, ResStations | $0.061 \times 4$ | 1.230 | 5.625 |
| ALUs | $1.463 \times 4$ | 5.852 | 5.852 |
| Vector Control Unit | $0.045 \times 4$ | 0.224 | 0.224 |
| DVX/Host Cores | 8.272 | 8.092 | 8.092 |
| VCOP Core | – | 7.465 | 11.860 |
| L2 Cache | 7.467 | 7.467 | 7.467 |
| *Processor Total* | *15.738* | *23.024* | *27.419* |

Table 4.3: Area ($mm^2$) estimates of a 4-core processor with DVX and VCOP at 32nm.

### 4.6.2 Area and Power

Table 4.3 presents the area breakdown of different microarchitectural components of a 4-core EDGE CMP with support for DVX and heterogeneous design of a 4-core CMP as a host for various configurations of VCOP. The fetch and decode stages in VCOP have a smaller area compared to DVX-enabled general purpose cores. One of the reasons is having the replicated fetch and decode stages for DVX in each core, as opposed to VCOP that uses a single front-end logic for multiple vector lanes. Note that the cores specialized for DVX already exist in the typical mobile processors (including their fetch and decode stages), while the VCOP accelerator utilizes smaller and yet additional front-end resources. Another reason is the size of the instruction window in DVX cores and VCOP. Compared to the general purpose cores, VCOP also avoids the L1 data caches; it issues memory requests directly to the L2 cache and has a single instruction cache to fetch vector AIBs. The execution resources of VCOP are equivalent to the resources available in 4 EDGE cores; but a wide vector register file banked per vector lanes is an additional structure of VCOP that is avoided by DVX. Table 4.3 summarizes the total area of a 4-core processor with DVX and VCOP designs. Host cores are identical to DVX cores without a vector unit. Both processors include a multi-banked L2 cache memory of the same size (2MB). The area size

| Structures | DVX (4C) | | VCOP (1C) | |
|---|---|---|---|---|
| VR Size | 32 | 128 | 128 | 512 |
| Fetch, Decode, L1I | 0.020 | 0.019 | 0.004 | 0.003 |
| Issue | 0.412 | 0.419 | 0.026 | 0.027 |
| LS Unit, L1D | 0.015 | 0.015 | – | – |
| RegFile, ResStations | 0.012 | 0.014 | 0.086 | 0.371 |
| ALUs | 0.164 | 0.166 | 0.166 | 0.172 |
| Vector Control Unit | 0.093 | 0.100 | 0.032 | 0.037 |
| DVX/Host Cores | 0.716 | 0.732 | 0.146 | 0.128 |
| VCOP Core | – | – | 0.314 | 0.611 |
| L2 Cache | 0.387 | 0.419 | 0.355 | 0.397 |
| *Processor Total* | *1.104* | *1.151* | *0.815* | *1.136* |

Table 4.4: Average power ($W$) estimates of a 4-core processor with DVX and VCOP at 32nm.

of the processor that adds VCOP exceeds the size of the processor specialized for DVX from 46% to 74% depending on the size of VRs in VCOP.

Table 4.4 shows the average power consumption of DVX on 4 EDGE cores and VCOP, while executing the set of selected DLP kernels. VCOP uses only one host core to control the vector execution and the three remaining cores are assumed to be perfectly power-gated. VCOP-128 consumes less power than the CMP with both DVX configurations. It happens mainly due to savings in instruction fetch, decode and issue stages. Besides single fetch and decode across vector lanes, VCOP issues each instruction over the slice of vector operands to its centralized vector lanes on single core. On the contrary to VCOP, DVX issues instructions over slices of the vector operands in 4 different cores and incurs additional power overheads compared to the more conventional VCOP design. VCOP-512 yields an additional performance improvement over VCOP-128 and two DVX configurations, but at the cost of higher power consumption mainly in the large VRs.

Figure 4.12 shows the energy-delay product (EDP) of a 4-core processor with enabled DVX. The results are normalized to EDP achieved by the general purpose execution on a single EDGE core. Average EDP of DVX scales down with the number of cores used in configuration for the kernels with sufficient amount

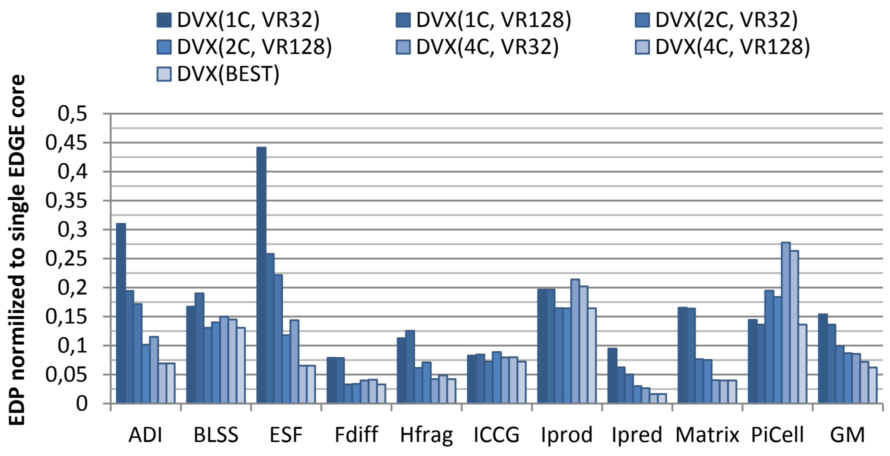


Figure 4.12: Energy-delay product (EDP) for DVX.

of DLP; and each configuration shows the substantial EDP reduction with respect to general purpose execution. A key advantage of DVX is the flexibility to dynamically tune the amount of resources for each kernel. To show the potential benefits of this feature, we include the configuration with lowest EDP for each kernel in Figure 4.12 (*DVX(BEST)*). 4-core DVX often achieves the best EDP results, since the number of iterations in each kernels is 1024 and thus the DLP offered is high enough to be exploited with 4 cores. Reduction kernels (*BLSS, Iprod*), kernels with modest compute or memory parallelism (*Fdiff, ICCG*) and not entirely vectorizible kernels (*PiCell*) yet have the insufficient amount of DLP to saturate the 4-core DVX configuration. DVX(BEST) provides better results for these kernels by utilizing 1- or 2-core DVX configurations. Compared to DVX with fixed 4-core configuration, DVX with the dynamically tuned resources improves the average EDP by 14%.

Figure 4.13 shows the EDP comparison of the DVX enabled processor and the processor with incorporated VCOP. The results for both processors are normalized to EDP achieved by the general purpose execution on a single EDGE core. Figure 4.13 includes the EDP results of 4-core DVX, which provides compatible performance and resources with VCOP. VCOP in average provides the best EDP results, because it achieves higher speedups at lower power overheads. While VCOP-128 increases the area of the 4-core processor by 46%, it achieves an average EDP higher by 3% over 4-core DVX configuration with the same size of the

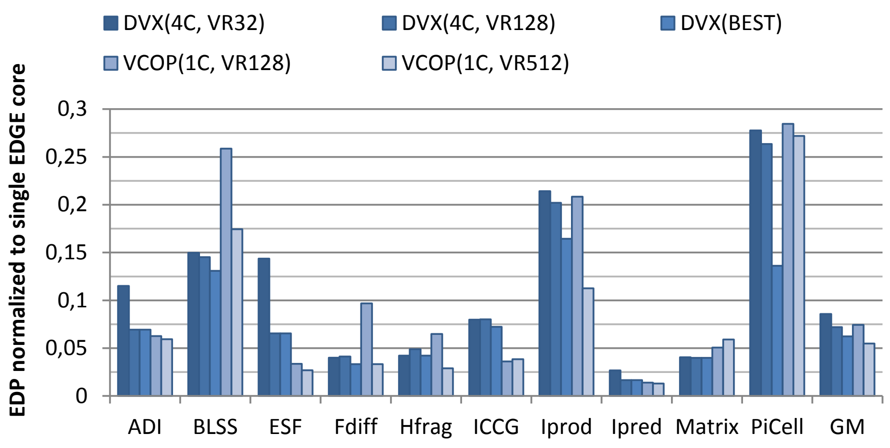


Figure 4.13: Energy-delay product (EDP) comparisons of DVX and VCOP.

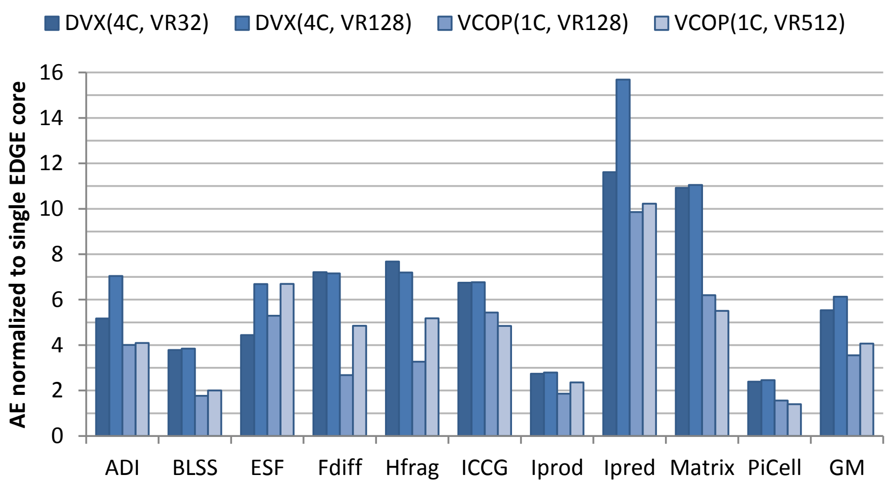


Figure 4.14: Area efficiency (AE) comparison of DVX and VCOP.

VRs. VCOP-512 has 25% lower average EDP than 4-core DVX, but with an area increase over 74%. For some of the kernels with an insufficient amount of DLP (*BLSS*, *PiCell*), 4-core DVX provides comparable or even better EDP results. The best DVX configuration over these kernels further achieves even better EDP results, by utilizing the number of cores suitable to exploit the amount available parallelism. On the contrary to such flexible substrate, VCOP does not allow for the tuning of resources and potentially underutilize them for the applications without abundant DLP.

Figure 4.14 shows the area efficiency (performance per $mm^2$, the higher is better) of a 4-core processor with enabled DVX and the processors with incorporated VCOP. The results are normalized to the general purpose execution on a single EDGE core, but counting the total area of the 4-core processor. DVX increases the area efficiency by about 6.1x due to large speedup on the existing CMP resources. On the contrary to such area efficient design, VCOP yields only 16% higher speedup than DVX and increase the area efficiency by 4x; it provides cost-inefficient performance improvements due to its excessive area footprint for the dedicated resources that are added to the lightweight EDGE CMP.

## 4.7 Related work

The same as EVX (see Chapter 3), its extended DVX version proposed in this chapter has similarities to various designs of classic based vector architectures such as [3, 17, 63]. The classic vector processors execute vector instructions, which operate over large configurable-size operands placed in complex vector register files. The vector instructions expose more parallelism than instructions that operate over scalar operands, which increases the efficiency of vector execution. To improve their performance, vector processors incorporate multiple functional units (vector lanes) for high compute bandwidth and multiple memory banks for high memory bandwidth. Although vector processors yield high performance and power efficiency, the large and complex parallel hardware resources have limited their applicability mainly to the domain of supercomputers. It has been suggested to add a vector unit to a superscalar processor [61] to investigate their applicability in the domain of general purpose cores. This increases the performance of the cores on DLP workloads, but incurs significant overheads due to the additional large vector register file and multiple functional units. DVX aims to achieve such improvements and yet avoid adding the complex vector-specific hardware resources. Instead of adding extra resources, DVX specializes the existing resources of a general purpose CMP into a vector-based DLP accelerator.

During decades of their utilization, classic vector architectures have been significantly improved [4, 16, 18, 39, 40] and DVX enables most of these improvements. We next discuss the most relevant improvements of classic vector processor. Then we summarize the previous work that investigates vector execution on the EDGE architecture and finally we differentiate DVX from the modern architectural models that exploit DLP in commercial processors.

Decoupled vector architecture [16] utilizes different queues to decouple scalar, vector compute and memory execution. The results show that decoupling tolerates memory latency and improves performance of in-order vector implementation. Work on out-of-order vector architectures [18] shows even higher performance at the cost of a larger vector register file. CODE [39] proposes a clustered vector architecture that leverages decoupling as well. Its clustered design further allows for easy scaling of the vector register file size and the number of functional units. For maximal performance on a general purpose CMP, DVX enables out-of-order execution of vector compute instructions and decoupled execution of vector memory instructions. To keep the footprint of vector hardware small in DVX, the L1 data cache is used to implement the register file and the EDGE to implement out-of-order execution of vector instructions.

The latest vector processors have been proposed based on the vector-thread architecture [4, 40]. This architecture scales vector execution across multiple vector cores by leveraging a control processor and associated hardware to start vector threads across the vector cores. The vector instructions are grouped in *Atomic Instruction Blocks* (AIB) and by sending a vector-fetch command, each vector core fetches and executes AIBs over different vector elements. Thus, fast hardware controlled threads on vector cores resemble dynamic vector lanes. The vector-thread architecture further improves the conventional vector design by enabling independent control flow in each vector-thread to exploit irregular DLP code. The control processor handles the lock-step execution of vector-threads when control flow does not diverge and synchronization between threads when control diverges. DVX resembles the vector-thread execution model by using hardware threads to scale its execution over the cores in CMP. DVX avoids the resources of control processor and keeps exploiting the simple homogeneous CMP substrate. Instead of adding general purpose capabilities to a specialized vector processor like the vector-thread architecture, DVX specializes the resources of an existing general purpose CMP.

The most similar research alternative to DVX combines general purpose and vector execution on a conventional superscalar multicore processor to enable efficient support of all levels of parallelism (ALP) [66]. Like DVX, ALP focuses on the memory system to increase the performance of DLP multimedia applications. It loads/stores large vector operands to/from streams/registers in data cache; and extends the existing issue/rename logic to use slices of these registers as a source or destination of each vector compute instructions. DVX, the same as its predecessor EVX avoids such issue/dispatch overheads by using extra hardware that transfers slices of the vector operands from memory to the vector compute instructions and back. While ALP relies on the conventional threads to scale the vector execution over multiple cores, DVX introduces faster hardware-managed threads to dynamically tune the hardware resources to a specific application phase.

Previous work in exploiting DLP on EDGE architectures extends an ultra-large unicore TRIPS processor [65] with vector-like execution. The processor targets high performance single-threaded execution with an extensive amount of compute resources. The vector support leverages EDGE AIBs to allocate the resources and repeat their execution over different elements. The L2 is accessed as a software managed cache to provide additional memory bandwidth, while an extra thread is expected to orchestrate the memory execution. The proposed vector execution on the large TRIPS processor lacks the flexibility to tune the amount of resources to different applications or workload phases like DVX.

Modern commercial general purpose processors incorporate multimedia SIMD extensions [8, 20, 55, 59, 73] and GPUs[46, 47, 51, 54, 56, 77] to accelerate DLP phases of various applications. Compared to vector processors and DVX, SIMD extensions achieve limited performance improvements. On the other hand, today's GPUs provide more powerful execution model for DLP workloads than SIMD extensions. Highly parallel applications increase their performance by using GPUs as their offload accelerators. The offloading incurs overheads for data movement operations between CPUs and GPUs; and the overheads have been greatly reduced by integrating the GPU and CPU into a single heterogeneous chip processor [6, 27]. Yet, the overheads in applications that frequently alternate parallel GPU and non-parallel CPU execution still have significant impact on the performance. DVX-enabled CMP avoids such overheads by executing parallel and non-parallel code regions on the general purpose CMP. Additionally the

DVX enabled CMP dynamically tunes its resources to different workloads phases and types of parallelism.

## 4.8 Summary

In this chapter, we have proposed dynamically-tuned vector execution (DVX) by specializing one or more available cores in a CMP into a DLP accelerator. DVX extends the previously proposed EVX technique to utilize the available resources in CMPs and scale the specialized vector execution over multiple cores. It increase the flexibility of DLP acceleration by dynamically tuning the amount of the specialized resources to different workload requirements. For such tuning, DVX incorporates lightweight threads controlled by hardware to scale the execution of vector instructions over multiple cores.

We have evaluated DVX on a 4-core EDGE CMP and show that it extends the processor area by only 1.1%. While executing the set of selected Livermore loops, 4-core configured DVX yields over 6.2x of the average speedup over the general purpose execution on a single EDGE core. Additionally, 4-core DVX outperforms EVX when it uses a low level fork/join implementation of threads to scale over 4 cores by 27%. DVX improves the energy-delay product of the CMP over 14 times. On the contrary to the cost-effective DVX implementation, the conventional heterogeneous system approach with the same CMP and a dedicated accelerator seems to be much more area extensive. The accelerator adds an amount of execution resources equal to what DVX already leverages from the CMP itself. It increases the area footprint over 74% and greatly affect the cost of the lightweight EDGE processor. DVX avoids such costs and yet gains over 86% of the speedup obtained with the accelerator.

The favorable DVX results evaluated in this work makes it a promising alternative to high-performance DLP accelerators in commercial mobile processors. Following work will look further at dynamic specialization techniques of general purpose cores for mobile devices. In the next chapter we investigate one such technique that reconfigures the mobile cores into diverse hardware accelerators.

# Chapter 5

# CMP Reconfiguration into Accelerators

## 5.1 Overview

This chapter proposes general purpose mobile cores that adapt the underlying hardware to a given workload. Existing mobile processors need to utilize more complex heterogeneous substrates [6, 25, 42, 43] to deliver the demanded performance on diverse mobile applications. They incorporate different cores and specialized accelerators. On the contrary, our processor utilizes only lightweight composable cores and dynamically provides an execution substrate suitable to accelerate a particular workload. Composability avoids static placement of the hardware resources into different cores and allows a workload to dynamically optimize its execution substrate by composing one or more cores into a large processor. Our approach goes one step further than composability and on-the-fly reconfigures one or more mobile cores into accelerators. Such accelerators improve performance and efficiency with minimal hardware additions.

The accelerators are made of general purpose ALUs reconfigured into a compute fabric and a general purpose pipeline that streams data through the fabric. To enable reconfiguration of the ALUs into the fabric, the floorplan of a 4-core processor is changed to place all the ALUs in close proximity on the chip. A configurable circuit-switched network is added to connect the ALUs of each core into the compute fabric. The network permits the ALUs to be reconfigured to perform various commonly repeated computations, instead of executing general purpose instructions. The pipeline used by the accelerators can be configured as well, by composing one or more cores into a large pipeline. It allows for tuning the accelerator memory system by incorporating extra resources that exploit higher memory bandwidth. Moreover, such configurable and yet general purpose pipeline of the accelerators increases their applicability by arranging the data that has either regular or irregular access patterns.

This chapter introduces reconfiguration of cores into the accelerators. It briefly describes connection of the processor's ALUs into the compute fabric, as well as the necessary hardware additions and modifications. It further explains the integration of the fabric into the general purpose pipeline, the way it configures and utilizes the fabric. It is followed with a description of the speculative computation in the fabric used by configured accelerators. Next is evaluation, which shows the performance benefits of reconfiguration and the impact of composability. The chapter ends with related work and concluding remarks.

## 5.2 Reconfiguration of Cores

The reconfiguration of the composable cores extends their capabilities beyond general purpose processing. The bulk of the resources can be allocated and used either as general purpose processors or accelerators. Each application running on this processor dynamically tunes the amount of allocated resources and their configuration to achieve the desired performance and efficiency. For example, one application may be executed by using a set of cores for general purpose processing and another one accelerated by using the remaining cores in the CMP reconfigured into an accelerator. The operating system (OS) would be in charge of controlling the sharing of resources among different applications. To simplify our studies and avoid side effects, in this work we only explore reconfiguration without interacting with the OS. We assume that all cores are available for a single workload and avoid the OS to control the sharing of resources.

The reconfiguration feature makes an accelerator of one or more general purpose cores. The accelerator is composed of the general purpose ALUs reconfigured to perform like a compute fabric and the general purpose pipeline that streams compute data through the fabric. The reconfiguration feature is implemented on a 4-core CMP. The floorplan design of the CMP is slightly modified to provide a suitable execution substrate for the compute fabric. The original design assumes 4 identical tiles, one per core. The new floorplan places the existing ALUs of each core close to each other by shifting them from the original location to the appropriate corner, as shown in Figure 5.1. A configurable circuit-switched network is added to connect all the ALUs into the fabric. When configured, the network dynamically provides a specialized datapath that couples the ALUs of the fabric to be used by the accelerators. This is further explained in Section 5.3.

Although in this work we focus on a 4-core CMP, the proposed design can further scale beyond four cores. One way to scale it is to create a larger compute fabric, by extending the switched network to connect ALUs in a CMP that incorporates more than four cores. This network extension includes longer metal interconnects and repeaters that are positioned along the network at various points to enable connections between ALUs on not-neighboring cores. With such network, the resources of the compute fabric scale by increasing the number of cores incorporated on a chip. Instead of connected the resources of many cores into a large fabric, each group of four neighboring cores could be also reconfigured into a different accelerator, where each one has its private fabric. Such design enables simultaneous acceleration of different compute regions in a dynamically configured hardware pipeline, which resembles the proposed design of Multicore Accelerator [58]. To simplify our research, in this work we investigate one compute fabric that executes one at most the commonly repeated region of compute instructions. The compute instructions are mapped to the fabric to configure the appropriate datapath among the ALUs. By using the configured datapaths, the fabric avoids passing the data via a complex power-hungry register file that is shared between different instructions. Once configured, the fabric may receive and process new data like a stream, over and over without reconfiguration. If the amount of data that is processed by the fabric is large, reconfiguration significantly increases the processor efficiency by eliminating per instruction overheads such as fetch, decode and register file accesses, as in [26]. To avoid any propagation delays in the configured fabric, the accelerators execute multiple iterations

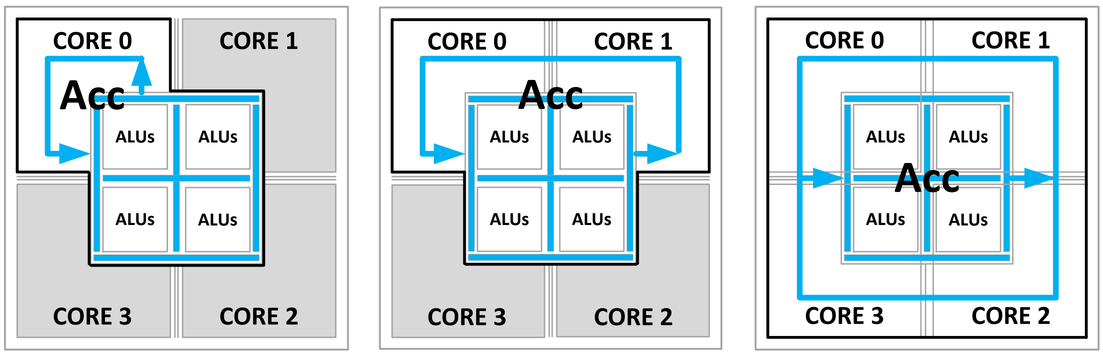


Figure 5.1: The dynamic reconfiguration of a composable 4-core CMP into various accelerators. The accelerators are composed of the existing ALUs connected into compute fabric and one or more cores that stream data through the fabric.

of the configured computation through the fabric in a pipelined fashion. Different stages of the fabric simultaneously compute different values. The network controls the progress of data through the connected ALUs and maximizes performance gains.

The fabric is easily integrated into the existing general purpose pipeline of the composable cores. The accelerators use the pipeline to execute memory instructions as well as to send and receive data processed by the fabric. Since the fabric performs most of the computation, the pipeline mainly executes non-compute instructions. This increases the efficiency of memory processing, by using most of the pipeline resources to saturate the available memory bandwidth. Composing cores further increases the amount of resources available for memory processing and allows for issuing more in-flight memory requests. Figure 5.1 shows various examples of the general purpose cores reconfigured into an accelerator. The accelerators leverage the general purpose pipeline that composes 1, 2 or 4 cores to stream data through the fabric.

Ideally, the reconfiguration feature would benefit from compiler modifications to statically provide the configuration for the fabric and optimize the workload to use such configured fabric. The compiler has to be extended to recognize a compute code region that may be optimized by using reconfiguration of the CMP resources into an accelerator (e.g. using trace based static profiling). The compiler will then automatically embed the instructions that maps the computation onto the compute fabric and produce the other instructions used by accelerators: load, store and passing of the data though the fabric. Alternatively, the compiler may be arranged to use programmer hints or annotations to indicate to a code that is likely to be optimized and suggest the preferred configuration (e.g. number of composed cores used by an accelerator). In this work, we do not explore such sophisticated compiler modifications. To simplify our research, we only leverage an in house compiler [24] that translate by a programmer defined compute region into the configuration for the fabric and we change the code by hand to utilize the configured fabric.

## 5.3 Compute Fabric

The compute fabric (Figure 5.2) is made by reconfiguring the ALUs of a CMP. These ALUs perform general purpose compute operations. Each ALU supports various operations, by using a set of functional units (FU)[1] and control logic to select which one is dispatched to perform the specific operation. Only a single FU per ALU is active simultaneously at most. The ALUs in one core are connected with a shared bypass network. The network enables the ALUs to operate conventionally, while bypassing the data between the ALUs and a shared register file. To support reconfiguration, we add a configurable circuit-switched network that connects all the FUs of a 4-core CMP. The network enables reconfiguration of the ALUs into the compute fabric, in which a subset of FUs is configured to perform a computation of frequently repeated compute instructions. One FU performs one compute instruction and the network passes the data between the FUs that perform dependent instructions. It permits using multiple FUs per ALU simultaneously, as opposed to conventional operation dispatch. By overlaying the switched network over the conventional bypass network, the FUs may be dynamically toggled to operate either conventionally by performing general purpose instructions; or as part of the compute fabric that performs a commonly repeated computation.

[1]In this work functional units refer to ALU sub-units such as shifter, adder or multiplier.

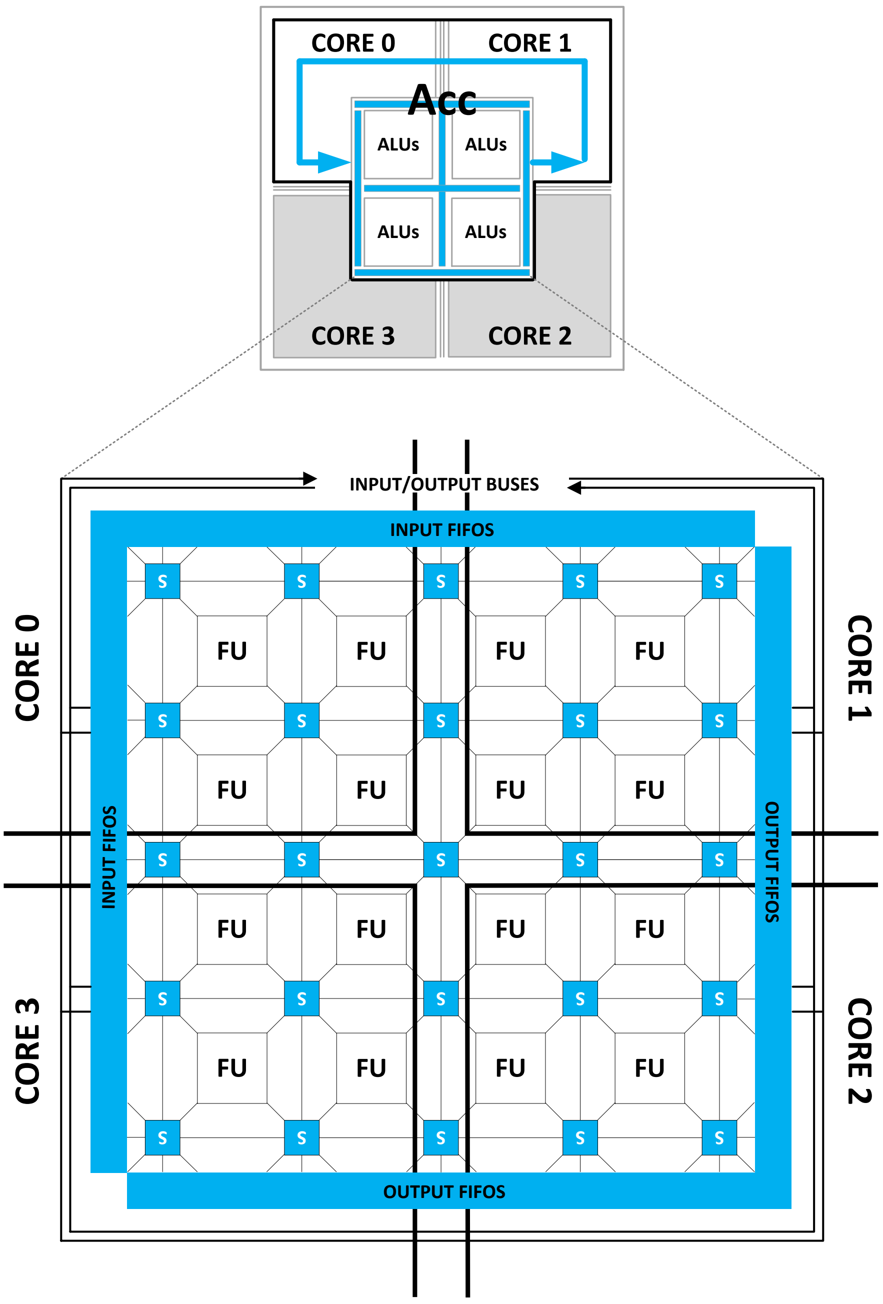


Figure 5.2: The implementation of the compute fabric on a 4-core CMP. It adds a circuit switched network to couple the existing functional units of four neighboring cores. The network permits the units to be reconfigured into the fabric.

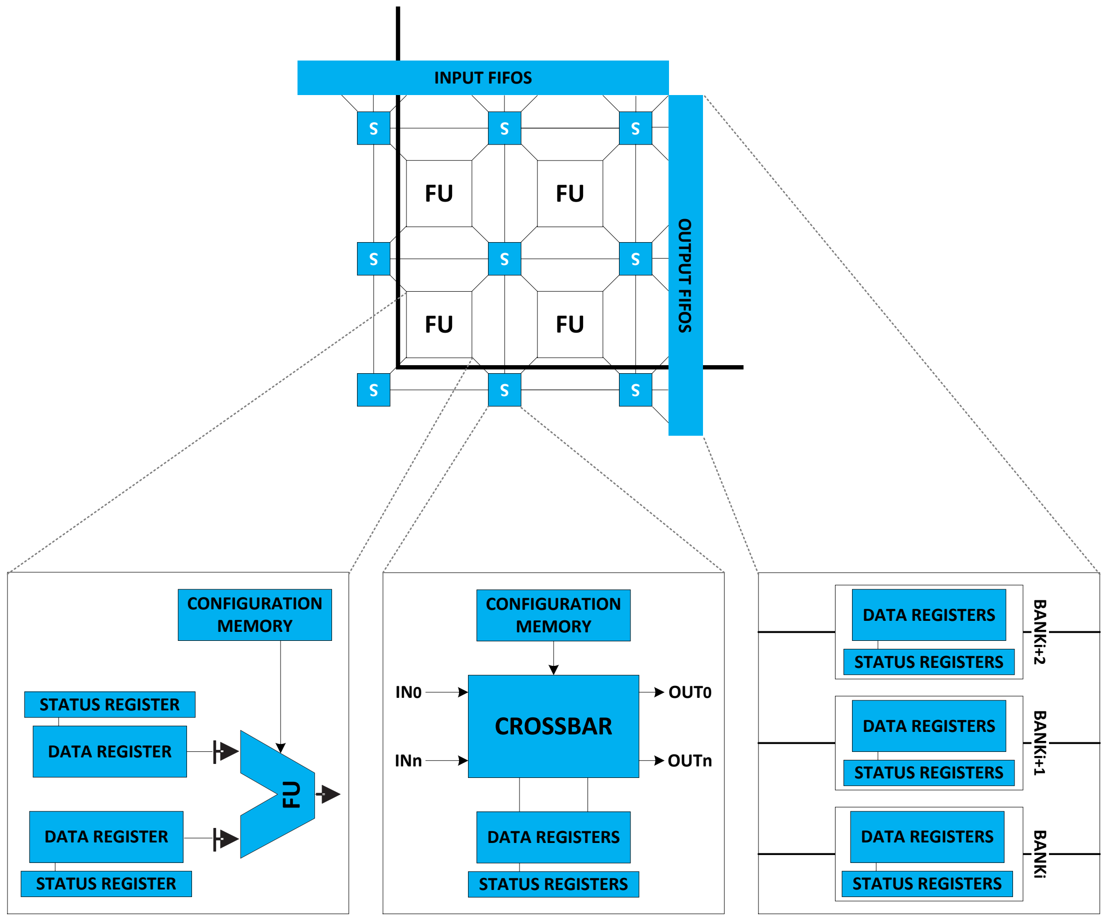


Figure 5.3: The microarchitecture details of each node in the fabric.

The network is based on a lightweight 2D mesh of nodes, as shown in Figure 5.2. Each node of the mesh may be: a) an extended FU b) a configurable switch or c) a fifo queue that holds input or output values, and their implementations are shown in shown in Figure 5.3. Each FU extends the compute logic by incorporating two data registers and two status registers. The data registers hold the values that are input operands of the operation to be computed by the FU. Each status register indicates whether the content of the associated data register is valid or not. Each FU is connected to four neighboring switches from where it receives its inputs and sends outputs. The switches are connected to fifo queues, FUs or other switches. Each switch includes: a multiplexer, an associated configuration memory, data and status registers. The multiplexer provides an output by selecting from two or more inputs according to its configuration bits, which are stored in the configuration memory. The data register holds the data that passes through a switch, before the switch is able to forward it to the next node. The switch is configured by writing configuration bits into the configuration memory. The configuration bits are transmitted by reusing the network datapaths, like in [24]. When configured, the switches provide the specialized datapath between the fifo queues and various FUs. The fifo queues hold input or output data values and each fifo is directly connected to one switch. The input values from each fifo are inserted into the the connected switch. Such inserted data goes through various FUs and switches on the configured datapath towards the fifos that hold output values of the configured computation.

Computation on the datapath may be driven either *statically* or *dynamically*. Statically driven computation uses a schedule provided by a specialized compiler to issue the operations. The compiler calculates the delays of each operation and generates the static execution schedule for the fabric. Input values are usually injected into the fabric at once and results are available after a known delay provided by the compiler. Such scenario simplifies the design of the network, which requires nothing beyond connections and routing control. On the other hand, dynamically driven computation requires including data and status registers in the network nodes to dynamically arrange the execution schedule. The schedule is arranged basing on the dataflow control of values arriving to the FUs. The FUs perform an operation when all its inputs are indicated as ready and the next node on the path is free to receive a new value. Dataflow control enables performing computation in a pipelined fashion on the fabric simply by streaming ready values into it. On the other hand, statically driven computation requires a complex static scheduling to pipeline the computation. The static scheduling may be jeopardized by events such as cache misses, which result into variable latencies hard to predict at compile time. This resembles the choice between in-order and out-of-order instruction issue. In-order issue logic is much simpler than out-of-order, but it relies heavily on the compiler to produce highly optimized schedules; this is specially complex due to variable latency instructions (e.g. loads that miss in the cache). Unlike the complexity of the hardware for out-of-order execution, the complexity of the hardware required to implement dynamically driven computation in our case is small. Therefore, in our work we chose dynamically driven computation. It provides a more flexible substrate with the cost of only small hardware additions and does not require complex compiler support to generate the execution schedule.

By configuring the datapath, the fabric allocates a set of the FUs available on the CMP, possibly leaving a number of them off the datapath. The dynamically driven computation uses the allocated FUs only when a FU has ready input values. Hence, the fabric may not utilize all the allocated FUs every cycle. There is a bubble in each cycle that a FU, allocated or not, is not occupied. On the other hand, an accelerated workload may still use some general purpose compute instructions, which are not mapped to the compute fabric (e.g. instructions that reduce results generated by the fabric). These instructions may "fill the bubbles" and utilize the FUs during such cycles, while increasing the utilization of the FUs. To achieve this, the wakeup and select logic is extended to check if there is a pipeline bubble and issue a compute instruction if the bubble exists. The extended logic interacts with a dynamically controlled dataflow in the fabric to check if a particular FU is unoccupied. When a general purpose compute instruction is issued into an unoccupied FU, the fabric is not affected nor are any structural hazards created. This extension to the processor logic enables the FUs to operate in a hybrid mode, by alternating between the operations of general purpose instructions and configured computations in the fabric. The hybrid mode enables acceleration of workloads, whose computation do not entirely fit onto the fabric and the remaining compute instructions need to dispatch general purpose operations to the FUs. This increases the utilization of the FUs and provides more flexible compute substrate used by the accelerators.

## 5.4 Integration of Reconfigurable Fabric

The fabric is integrated into the existing general purpose pipeline of the composable cores, like it is shown in Figure 5.4. The fabric performs like a deeply pipelined execution unit with long latencies. The fifo queues in the fabric facilitate the integration. The pipeline sends the input values to the fifos that hold the inputs (*input fifos*) and receive the output values from the fifos that hold

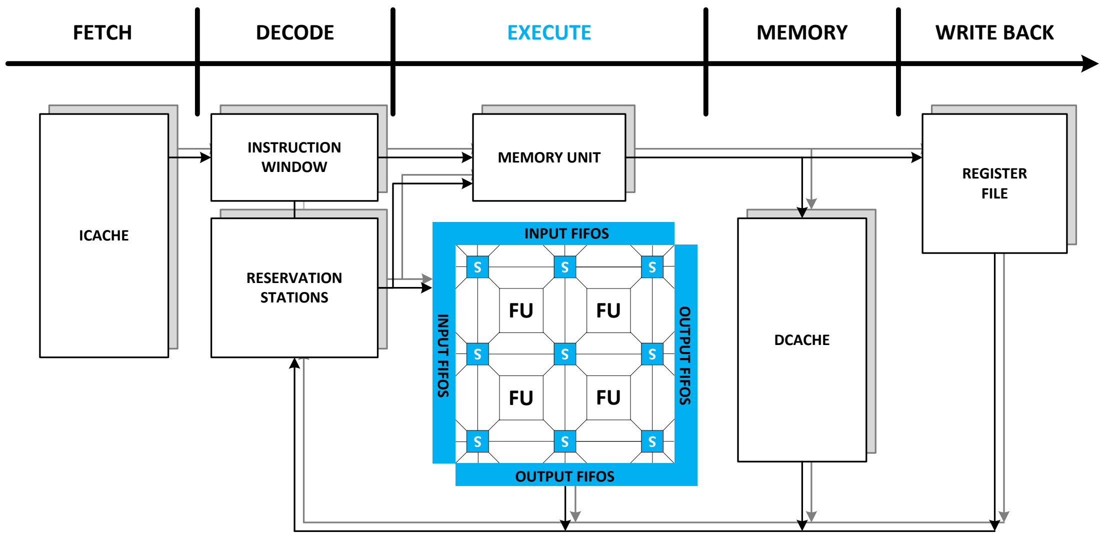


Figure 5.4: An example of the simplified compute fabric integrated into the pipeline of composable cores. The fabric affects only the pipeline execute stage.

outputs (*output fifos*). The pipeline is connected with the input fifos by using the input bus between the reservation station buffers and the write ports of the input fifos. Each entry of the input fifos is updated by writing value to the entry of the fifo encoded by the fifo and entry identifiers appended to the value. A similar connection is incorporated between the pipeline and output fifos, except the output bus connects the reservation stations to the read ports of the output fifos. Inside the fabric, each fifo queue is directly connected to the east, west, north or south switch node. The point-to-point connections between the fifos and the switches manage the progress of input and output values when they arrive to the particular node, as it is already explained in Section 5.3.

Since the general purpose pipeline can compose one or more cores, the fabric has to be connected to each of the cores in such pipeline. Each core has its input and output buses, which connect the core's reservation station buffers to the input and output fifos of the fabric. There are different options for designing the connection buses between the buffers and the fifo queues: each input/output bus can share the read/write ports of the fifo queues implemented as one bank of the fifo entries; the buses can use the independent ports added to the bank to increase the input/output data throughput; or the buses can share the ports of the banked fifo entries (e.g. by having independent bank of the entries per each fifo). To increase the data throughput and performance without requiring extra read or write ports, in this work we chose to have one dedicated bank per each fifo of the fabric as shown in Figure 5.3. We show the benefits of such design decision in Section 5.7.

The general purpose pipeline of the composed cores executes AIBs of EDGE instructions. The AIBs include instructions that send the data to the input fifos or receive it from the output fifos (*send/receive* instructions). Send/receive instructions pass the data between the pipeline and fabric, by accessing the fifo queues. Before the fabric is utilized by the pipeline, it has to be configured to perform the desired computation.

### 5.4.1 Fabric Configuration

The fabric is configured by executing a number of *configuration* instructions, which may occupy one or more AIBs. Each configuration instruction has an immediate field that holds the identifier of a given node in the switched network and the data used to configure that node. Rather than broadcasting this data through the network until it reaches its destination node, the data is sent to a fifo queue connected to a row or a column where the destination node resides. From the fifo queue, the data is inserted into the associated node and each node forwards the data to the next node, until the data reaches its destination. North nodes forward the data to south nodes and west nodes forward the data to east nodes. By configuring various switched nodes, the fabric forms a specialized datapath that connects a heterogeneous set of FUs that is used for the computation. The number of blocks used for the configuration defines the configuration overhead, which is analyzed in section 5.7.

Configuration overhead is mitigated through repeated execution of the configured computation. If the entire application exploits a single configuration of the fabric, configuration overhead is amortized and does not remarkably affect performance. On the other hand if the application exploits different configurations of the fabric, the configuration phase is executed a few times, which may reduce the performance improvements. For such cases, an additional memory (RAM) could be used to keep the various configurations of the fabric. By storing the configuration into the specialized memory, the configuration overhead would be reduced. Instead of executing one or more blocks with configuration instructions, the memory would instantly send the configuration data to the fabric. In our experiments we do not use such memory, but we see its potential benefits for different applications, which could share the fabric configurations to accelerate common critical functions (e.g. FFT could be used either for graphics or speech recognition applications).

### 5.4.2 Fabric Utilization

The fabric is utilized by the pipeline, which executes AIBs appended with send/receive instructions. AIBs are composed of EDGE instructions, which explicitly encode the dataflow within targets in producer instructions. Send instructions act like producers, by forwarding input values into the fabric. The targets of a send instruction route the value to the appropriate input fifo queues of the fabric to be inserted into the right place of the configured datapath. The value is provided by a memory or register-read instruction executed in the general purpose pipeline. Receive instruction act like producers as well, by forwarding output values from the fifo queues encoded in the instructions to their specified consumers.

We rely on the general purpose pipeline to access memory and arrange input and output values. Such design may accelerate the compute regions with any kind of memory access patterns, regular, irregular or a combination. To efficiently arrange the operand values and maintain the correctness of the targeted application, the pipeline leverages the commonly available units of the memory subsystem such as LSQs, low-latency caches and hardware prefetchers. Instead of modifying the pipeline configuration, composing cores tunes the pipeline resources (e.g. data cache size) on-the-fly to further increase the efficiency of memory accesses, which is evaluated in section 5.7. After the value has been loaded by the general purpose pipeline, a send instruction is used to insert the value into the fabric. Similarly, a receive instruction is used to collect the output value produced by the fabric, which is later stored to memory. Alternatively, the value can be read from/written to a general purpose register.

Each send/receive instruction is associated with a fifo queue, which is encoded in the instruction. When fetching an AIB with send/receive instructions, the core allocates one entry in each input fifo and each output fifo queue. If there is no free entry per queue to be assigned to the AIB, the core stalls the block execution due to structural hazard. When an AIB commits, the entries assigned to the block are released and they can be used by the next AIBs. When the entry is assigned to an AIB, the send instructions in the block execute by writing input data to the entry of their encoded input fifo queues. The receive instructions in the block execute by reading the output data from the entry in their encoded output queues. If a value requested by a receive instruction is not available in the entry (not computed by the fabric), the output queue stalls the receive request until its entry receives the requested value.

The send/receive instructions included in the AIB control the entire iteration(s) of the configured computation. The AIB sends all the values to be computed and receives all the results. To increase performance, the AIBs with limited communication with the fabric (e.g. compute intensive regions) can apply *loop unrolling* to control multiple iterations of the computation. The send/receive instructions in the AIB are replicated per each iteration of the loop. An identifier specifies the iteration and the ordering of the send/receive requests in the fifo queues. The iteration identifier implies the same semantics with respect the ordering of send/receive requests as load-store identifiers for the ordering of memory instructions. The block header is extended to encode the number of iterations in the AIB, which allows a core to allocate multiple entries of each fifo for such AIB. While executing a given send or receive instruction, its iteration number is used to access the appropriate entry in the fifo queue and maintain correctness of the computation.

If the configured computation produces results after a long delay, *software pipelining* may be applied to relieve the problem. In this optimization, multiple AIBs are used. The first AIB only sends the data to be received by next block. Each of the following AIBs then sends data to be received by the next AIBs and receives the computed results of the data sent by previous AIBs to avoid long delays. The last AIB only receives the results of the data sent by previous AIBs and does not send any data. We evaluate the impact of these two optimizations in Section 5.7.

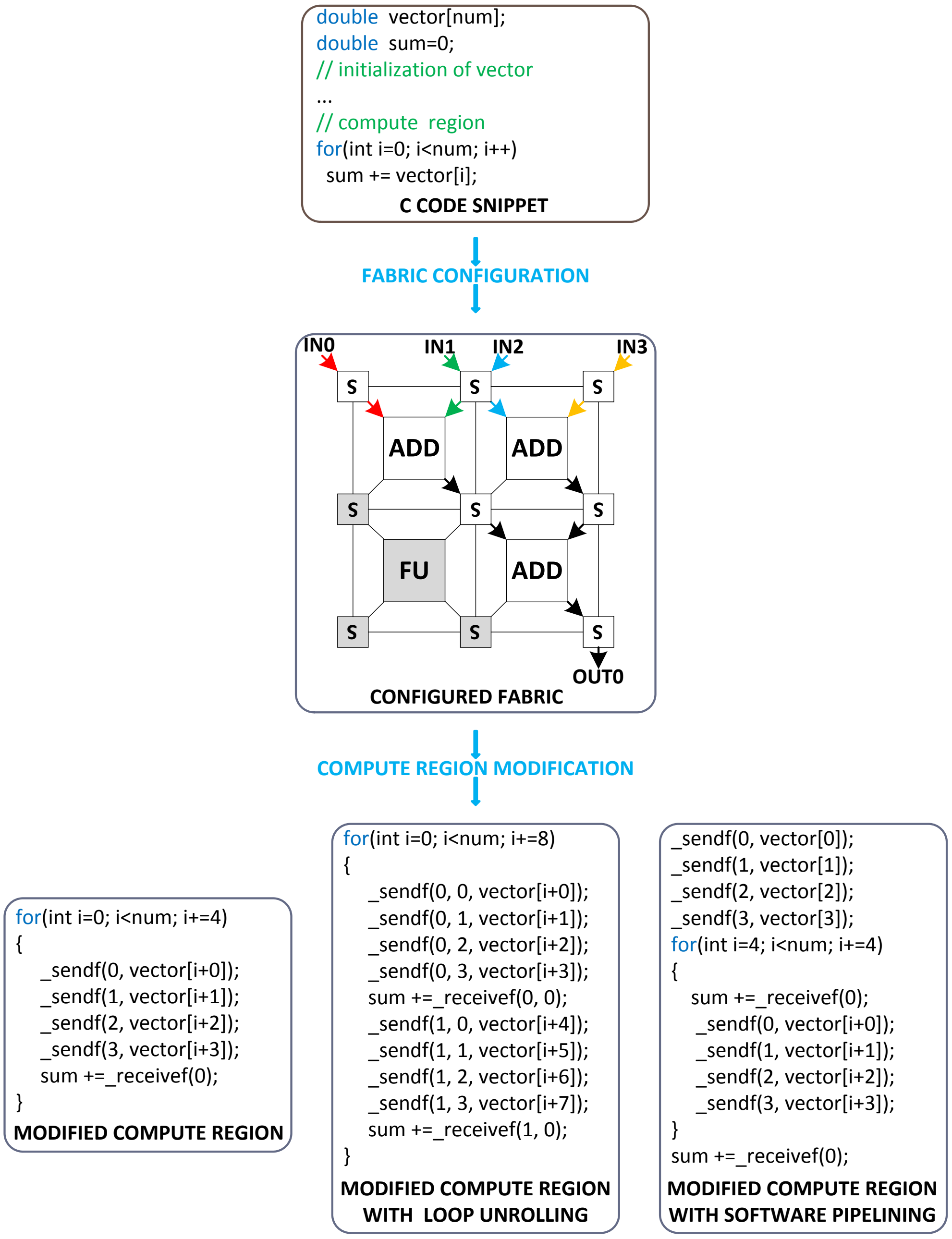
It is possible to further improve data throughput and performance of the accelerator by integrating a programmable memory controller to load input and store output values. Such controller performs memory accesses according to the access patterns, when they are known at compile time. Due to the additional complexity of integrating the programmable memory controller, we do not investigate such approach in this work.

### 5.4.3 Example of Acceleration

Figure 5.5 shows step-by-step acceleration of a simple compute code region. The first step of the acceleration configures a simplified fabric to perform the computation of the compute code region. The second step modifies the original code region to utilize the configured fabric. We also present two optimized versions of the code that utilizes the fabric, one using loop unrolling and the other using software pipelining.

The compute code region iterates a loop that adds two input values. The fabric is first configured to perform as much as possible computation of this loop. When configured, the fabric performs addition of four input values and produces one output result. The code is next modified to be accelerated. The original code of the loop contains compute and memory instructions. The modified code removes compute instructions that are executed by the fabric and has only one remaining compute instruction in the loop to accumulate the results computed by the fabric. The modified code has all the memory instructions from the original loop to load input and store output values and additionally appends send/receive instructions that pass the values through the fabric.

The modified code is written by using send/receive intrinsics, which specify the associated input/output fifo queue, the value per send/receive instruction and the loop iteration identifier if using loop unrolling. The modified code without optimizations describes an AIB, which controls one iteration of the computation within the fabric (an addition of 4 input values that produces as a result one output value). Loop unrolling optimization increases the size of the AIB to control



```
double  vector[num];
double  sum=0;
// initialization of vector
...
// compute  region
for(int i=0; i<num; i++)
  sum += vector[i];
```

```
for(int i=0; i<num; i+=4)
{
   _sendf(0, vector[i+0]);
   _sendf(1, vector[i+1]);
   _sendf(2, vector[i+2]);
   _sendf(3, vector[i+3]);
   sum +=_receivef(0);
}
```

```
for(int i=0; i<num; i+=8)
{
   _sendf(0, 0, vector[i+0]);
   _sendf(0, 1, vector[i+1]);
   _sendf(0, 2, vector[i+2]);
   _sendf(0, 3, vector[i+3]);
   sum +=_receivef(0, 0);
   _sendf(1, 0, vector[i+4]);
   _sendf(1, 1, vector[i+5]);
   _sendf(1, 2, vector[i+6]);
   _sendf(1, 3, vector[i+7]);
   sum +=_receivef(1, 0);
}
```

```
_sendf(0, vector[0]);
_sendf(1, vector[1]);
_sendf(2, vector[2]);
_sendf(3, vector[3]);
for(int i=4; i<num; i+=4)
{
   sum +=_receivef(0);
    _sendf(0, vector[i+0]);
    _sendf(1, vector[i+1]);
    _sendf(2, vector[i+2]);
    _sendf(3, vector[i+3]);
}
sum +=_receivef(0);
```

Figure 5.5: An example of the acceleration of a compute code region by using the fabric. The fabric is configured to perform the computation of the code region and the code region is modified to utilize such configured fabric. The modified code can be further optimized to increase the performance.

two iterations of the configured computation (an addition of 8 input values that produces two output values). Software pipelining optimization describes an AIB, which receives the outputs of the previous compute iteration and sends the inputs for the following compute iteration. Such optimized AIB must be preceded by the code that sends the inputs of the first iteration and followed by the code that receives the outputs of the last iteration.

## 5.5 Squashing and Speculative Execution

Managing values that pass through the reconfigurable compute fabric is quite challenging in conventional out-of-order processors. It is important to keep the original order in which values should enter the fabric, maintain memory correctness and support speculative computation necessary to increase performance of out-of-order processors. EDGE architectures provide a more advantageous substrate to integrate the fabric, because they execute atomic blocks of instructions. Block atomicity facilitates arranging speculative values for the fabric and enables to integrate the fabric with modest hardware additions, such as the fifo queues.

Each AIB that arranges the values sends all the inputs and receives all the outputs for one or more iterations of the configured computation. Due to atomicity, the AIB either 1) commits all its results to registers and/or memory or 2) squashes (discards) all the results. The fabric consumes a value from a fifo queue as soon as it is ready. Speculative values are computed by the fabric and results stored to the fifo queues, which hold outputs of the computation. The AIB may receive the results, but if the AIB squashes, all the results are discarded. If the block squashes before sending all the values required to perform the computation, the dynamically driven computation in the fabric could stall waiting for an input value that would never be sent. To avoid this, an AIB that squashes marks the allocated entries in the fifo queues to be invalid-but-ready. This way, the fabric can always finish its computation and eventually discard the results if the AIB is being squashed. Once the computation is finished and the AIB committed or squashed, the allocated entries in the fifo queues are released to be used for the next computation. This mechanism enables the accelerators of composable cores to perform speculative computation in the fabric, by simultaneously executing one or more speculative AIBs that control such computation.

A more complex hardware would be required to support the software pipelining optimization introduced in section 5.4 and speculative computation simultaneously. In software pipelining, the AIBs send values to be received by the following AIBs and receive values sent by previous AIBs. Speculative computation may provide values which need to be discarded, including the case in which the next AIB, that will use such values, has not started yet. Since in this case there is no AIB to mark the corresponding fifo queues entries as invalid, the fabric would require additional hardware to discard these results. Such design seems to be excessively complex and we do not incorporate it in this work. Instead, the workloads optimized with software pipelining do not leverage speculative computation. The fabric consumes the values when the AIBs commit. This makes all the computation in the fabric non-speculative and the results are never discarded. The results may be received by an AIB that squashes, but the fifo queue entries that hold the results are released only when the AIB commits. This guarantees the correctness and finishing of the software pipelined accelerations. In this case, the accelerators do not leverage speculative computations, but they avoid idling of AIBs that wait for the results with long delays. It is a trade-off between two optimizations and each one requires no complex hardware additions when they are applied separately.

## 5.6 Experimental Setup

### 5.6.1 Simulator

We evaluate the reconfiguration feature on a dynamic EDGE processor, by using a detailed, timing, in-house from Microsoft Research. The simulator is written in C++ and configured to model a composable 4-core EDGE CMP with the parameters shown in Table 5.1. To model the memory hierarchy properly, the simulator is coupled with the Ruby memory model (see http://www.m5sim.org/Ruby). The EDGE simulator dynamically composes cores through calls to the runtime

Table 5.1: Simulators configuration.

| | Component | Description |
|---|---|---|
| Per core | ALUs [FUs] | 2 [FMul, FAdd, IAdd, IMul, Logical] |
| | LSUs | 1 (Load/Store request per cycle) |
| | Register File | 64 entry |
| | Branch Predictor | tournament |
| | L/S Queue | 32 entry unordered LSQ |
| | L1 I/D-cache | 32 kB each, 1 cycle (hit), 8xMSHR |
| 4x CMP | L2 cache | 4 banks x 512 KB, 15 cycles (hit), 8xMSHR |
| | DRAM | latency: 256 cycles |
| | On-Chip-Network | 1 cycle/hop, Manhattan routing distance |
| Fabric | FUs | 4x10, (4 Cores x 2 ALUs [5FUs]) |
| | Switches | 5x11, 1 cycle/hop |
| | Fifos | 30, 8 entries/fifo, 1 bank/fifo |

system, via memory mapped system registers. To model the processor reconfiguration feature, we integrate an independent compute fabric simulator [24] that simulates a fabric with the parameters shown in Table 5.1. The fabric utilizes the existing processor FUs (4x10), by adding extra switches (5x11) and fifo queues (30), in a similar fashion to the design shown in Figure 5.2. The fabric simulator is based on a configurable switching network that models dynamically driven computation. The network connects and dynamically customizes the subset of available FUs for a particular computation. For some workloads that require more FUs than what the processor already provides, the fabric simulator is configured to model the amount of available and extra FUs added to enable acceleration. The processor and fabric simulators are connected by using input/output data buses between each core of the processor and the fifo queues of the fabric. The default configuration of the fabric utilizes one independent bank per each input or output fifo queue to allow for substantial bandwidth at the input and output of the fabric for each core of the CMP.

### 5.6.2 Benchmarks

To perform the evaluation of the reconfiguration feature, we select seven computing workloads shown in Table 5.2 from various benchmark suites [9, 60, 71, 78].

Table 5.2: Selected compute intensive workloads.

| Name | Description | Domain |
|---|---|---|
| fft | Fast Fourier Transform | Signal Processing |
| kmeans | K-Means Clustering | Data Mining |
| mm | Dense Matrix-Matrix Multiply | Scientific Computing |
| mriq | Magnetic Resonance Imaging - Q | Image Reconstruction |
| nbody | N-Body Simulation | Physics Simulation |
| spmv | Sparse-Matrix Vector Multiply | Scientific Computing |
| stencil | 3-D Stencil Operation | Fluid Dynamics |

Table 5.3: Accelerated portions of code and configuration overhead.

| Workload | % of code accelerated | Configuration overhead | | | |
|---|---|---|---|---|---|
| | | Instructions | AIBs | cycles | cycles-cached |
| fft | 76 | 277 | 3 | 3094 | 163 |
| kmeans | 19 | 357 | 4 | 3775 | 209 |
| mm | 82 | 269 | 3 | 3085 | 159 |
| mriq | 64 | 269 | 3 | 3098 | 159 |
| nbody | 81 | 281 | 3 | 3092 | 164 |
| spmv | 20 | 277 | 3 | 3082 | 163 |
| stencil | 85 | 317 | 3 | 3320 | 183 |

Each workload utilizes a different compute algorithm and each algorithm is typically found in emerging applications used in various segments of the computer industry, including in mobile devices.

### 5.6.3 Methodology

The code of each workload is inspected to select the most frequently executed compute regions to be executed in the accelerator. Table 5.3 shows the percentage of accelerated code in each workload and configuration overheads. The rest of the workload is executed on the general purpose cores . The selected regions are compiled with an in-house compiler [24] to provide the configuration bits for the fabric. By executing the AIBs that write the configuration bits, each workload configures the fabric to match the computation of its compute algorithm. The configuration overhead is shown in Table 5.3, and it includes its total number of

Table 5.4: Acceleration requirement and characteristics.

| Workload | FUs required | | | | FIFOs required | | Loop-unrolling |
|---|---|---|---|---|---|---|---|
| | FMul | FAdd | FSub | Or | Input | Output | $n$ iterations |
| fft | 4 | 3 | 3 | 0 | 6 | 4 | 2 |
| kmeans | 8 | 7 | 8 | 0 | 16 | 1 | 2 |
| mm | 8 | 7 | 0 | 0 | 16 | 1 | 2 |
| mriq | 6 | 4 | 0 | 1 | 7 | 2 | 4 |
| nbody | 7 | 5 | 3 | 0 | 7 | 3 | 4 |
| spmv | 8 | 6 | 0 | 0 | 16 | 2 | 1 |
| stencil | 2 | 10 | 2 | 0 | 14 | 2 | 1 |

instructions, AIBs and cycles required to execute these AIBs. The cycles differ whether the AIBs are cached or not (the configuration has been used already or not) and we include both cases.

Once the fabric is configured, it allocates various nodes of the network to perform an entire computation. The configuration requirements of each workload are presented in Table 5.4. Since the workloads perform floating-point computation, the fabric mostly allocates floating point FUs. The only exception is the Bitwise Or, used by workload *mriq*. The most commonly used FUs are multipliers and adders. A floating-point adder performs two operations: add and subtract. While performing one operation at most, the number of allocated adders is presented separately for each operation. The number of allocated multipliers goes up to 8, which is the amount provided by leveraging existing FUs on 4 dual-issue cores. On the other hand, the number of allocated adders goes over 8 in two cases: 15 in *kmeans*, 12 in *stencil*. To accelerate these two "adder-hungry" workloads it is necessary to incorporate more FUs than what is provided by reconfiguration of general purpose cores. Those workloads are examples where the fabric needs extra FUs to match a computation of a workload that intensively uses some operation. 2 of 7 selected workloads find it necessary to incorporate the modest set of extra FUs, while the others can accelerate their computations on the existing compute resources. To find out what may be the most preferable set of extra FUs, it is required to comprehensively analyze a broad range of mobile applications and their compute regions. To simplify our work, we avoid this step and when necessary we incorporate only the FUs that are required to accelerate the workload. The extended fabric requires extra switches to enable the configuration of the extra FUs. The number of configured switches varies per workload and it depends on the number of allocated FUs and their locations. The input and output switches of the configured fabric are connected to input and output fifo queues. The set of selected workloads use up to 16 (*kmeans, mm, spmv*) input and 4 output fifo queues (*fft*), which define the number of input and output values per iteration.

The compute regions of each workload are modified by hand to perform their computation in the configured fabric. We modify the code of the compute region by writing send and receive intrinsics similarly to the code transformations of the example shown in Figure 5.5. The workloads that use the fabric to perform a limited amount of computation per iteration are optimized by using loop unrolling as shown in Table 5.4. The AIBs of optimized workloads replicate send/receive intrinsics to send/receive the data of up to 4 compute iterations (*mriq, nbody*). The fabric does not need to be modified or reconfigured when changing the number of iterations unrolled per AIB.

We evaluate the speedup of the reconfiguration feature over general purpose computation on the composable cores while executing the entire workloads (including the non-accelerated parts). We evaluate the effect of composing cores to scale the resources of general purpose processors and accelerators. Each of the workloads uses static inputs, but repeats its computation many times to warm up the caches. Repeated computation minimizes the reconfiguration overhead, since the fabric is configured once per workload. We show the benefits of two optimization techniques: loop unrolling and software pipelining. Loop unrolling is included by default in the rest of the experiments, while software pipeline is used only in the experiment that shows the benefits of the software pipeline optimization. The general purpose computation always uses the best compiler optimization to make the comparisons fair.

## 5.7 Results

Figure 5.6 reports the performance improvements obtained by dynamically composing and reconfiguring cores. Label *Composing(kC)* indicates that *k* cores are

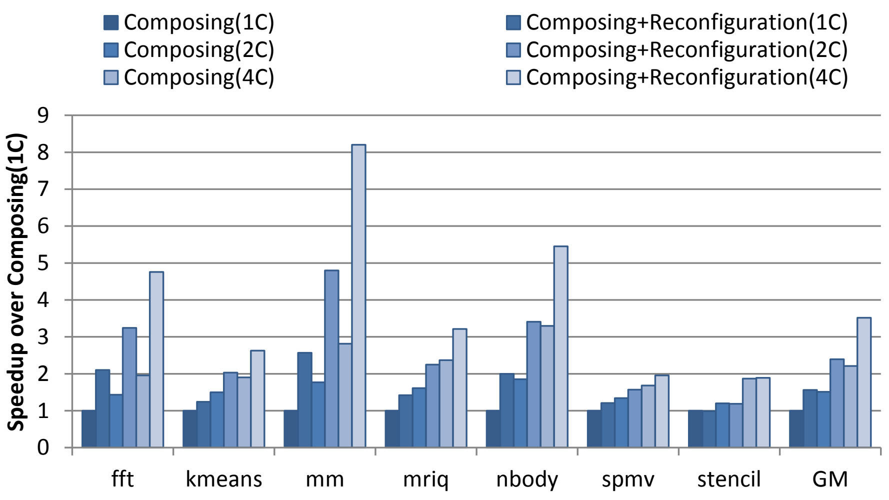


Figure 5.6: Reconfiguration speedup.

composed into one larger logical processor. Label *+Reconfiguration* indicates that the cores are reconfigured into an accelerator.

We first analyze the results of composing cores, without resorting to reconfiguration. The 2-core composed processor outperforms the 1-core processor by 1.51 times on average. The 4-core composed processor scales the performance even further by achieving over 2.2x of average speedup compared to the 1-core logical processor. Composing cores provides significant benefits for the selected workloads. This is due to their compute intensive algorithms, where abundant computation enables the enlarged substrate with more execution resources to simultaneously perform more computation and increase performance.

Reconfiguring the cores into accelerators increases performance even more by specializing the computing substrate. Each accelerator utilizes the fabric configured per workload and the pipeline of one or more composed cores to stream data through the fabric. Compared to the 1-core processor without reconfiguration, the accelerators increase the average performance by 1.56x, 2.39x and 3.51x, when 1, 2 or 4 cores are used in the accelerator's pipeline respectively. The results scale when the accelerators utilize more cores. More cores in the accelerator execute more speculative AIBs, which perform speculative computation and increase the utilization of the fabric, as explained in Section 5.5. The results of specialized computing on the accelerators scale in a similar way as the results of general purpose computing. With such similar scaling trends, the specialized computing always outperforms the general purpose computing with the same number of cores. The 1-, 2- and 4-core accelerator outperforms the 1- ,2 - and 4-core general purpose processor by 1.56x, 1.58x and 1.6x respectively.

The speedup of the accelerators over the general purpose processor varies per workload and goes from 1x (*stencil*) to 2.57x (*mm*), when using the 1-core configuration in each case. *Stencil* is a corner case that has a modest amount of computation over the stencil values, which are accessed following complex memory access patterns. Without extra memory optimizations [31], which are not analyzed in this work, the *stencil* workload does not have any benefits of specializing its relatively small computation. On the other hand, *mm* has intensive computation with sequentially accessed data of a dense matrix as input. The modified *mm* code for the accelerator removes most of the compute instructions. It enables more aggressive loop unrolling optimization by refilling the AIBs with extra memory instructions of the unrolled loop. The modified *mm* performs much faster on the accelerator. The accelerator uses the pipeline mostly for memory processing and leverages the configured fabric to simultaneously perform multiple operations in different FUs. Some workloads (*kmeans*) implement a compute intensive algorithm, but include non-compute instructions (e.g. library functions such as *malloc*) in the frequently executed code region. In such a case, the accelerator improves performance, but the non-compute instructions (81%) limit speedup. One of the workloads (*spmv*) implements an algorithm that has a few different computations in a single loop. Although one iteration of its loop combines various computations, the accelerator may be configured for only one of them, because of the configuration overhead. The rest of the computation (80%) is performed using the general purpose pipeline, which limits the performance.

Figure 5.7 shows the performance impact of two accelerator design decisions. It shows the impact of implemented wide memory execution in the general purpose pipeline (which is not default design) and the impact of banking the input/output fifo queues (default design). We separately apply these modifications to the 1-core and 4-core accelerators respectively with their default configurations (Table 5.1). The speedup is presented over the baseline 1-core processor. To avoid conflicts between memory accesses in multicore accelerator and isolate

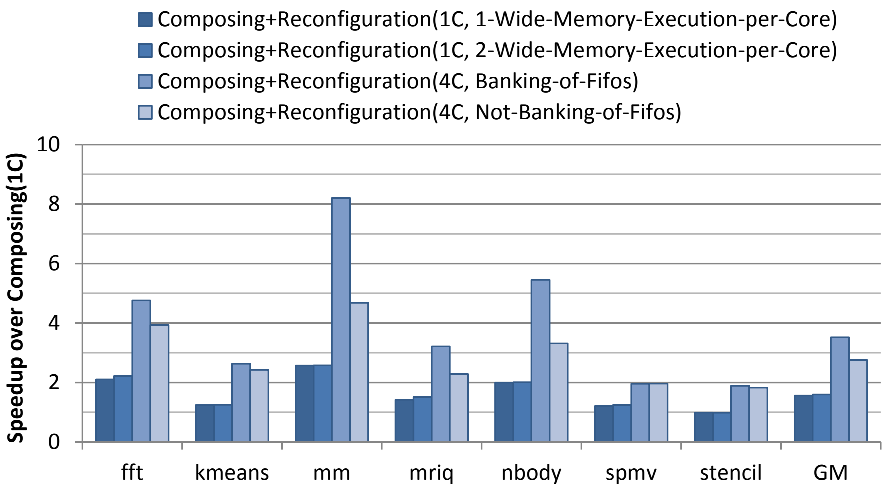


Figure 5.7: Performance impacts of two different designs of the accelerators.

the impact of wide memory execution in the core pipeline, we apply this modification onto the 1-core accelerator. We increase the number of load/store units (LSUs), as well as the number of ports in the LSQ and L1 data cache to enable wide memory execution through the entire pipeline. Instead of one LSU per core, which is the default design, we incorporate two LSUs. This modification enables the accelerator to issue up to 2 instead of 1 memory requests per cycle, but it increases the performance of the accelerator by only 1%. This happens because the AIB based execution of the selected compute intensive workloads provide the insufficient number of parallel memory instructions per AIB to saturate two LSUs. A possible way to increase the performance of the accelerator without having the parallel memory instructions is to execute SIMD load/store instructions. The SIMD instructions manage wide data accesses, when a workload exploits sequential data access patterns. Due to its associated additional complexity, this thesis does not investigate such SIMD optimization. While increasing the number of LSUs in the core pipeline has negligible impact on the performance, the data throughput between the fabric and composable pipeline significantly affects the performance of the accelerator. We study this impact on the 4-core accelerator, because it utilizes the 4-core pipeline to stream data through the fabric and requires the higher throughput of data between the pipeline and the fabric. If this accelerator accesses a fabric that has one bank of the input and one bank of

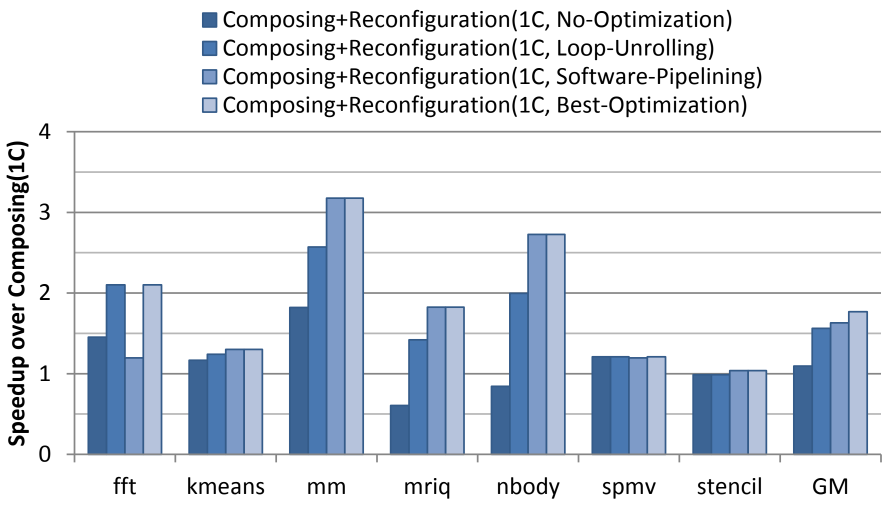


Figure 5.8: Performance benefits of various optimizations.

the output fifo queues, its performance degrades by 27% compared to by default designed accelerator that utilizes one dedicated bank per each input or output fifo queue. Although the banking of fifo queues increases the complexity of their implementation, our result show that such design decision greatly increases the performance of the 4-core accelerator.

Figure 5.8 shows the benefits of loop unrolling and software pipelining optimizations. We present the speedup of the 1-core accelerator over the 1-core general purpose processor. The general purpose execution in the processor uses loop unrolling, but not software pipelining. The results of the accelerator are presented by incrementally applying loop unrolling and software pipelining. Loop unrolling is applied to workloads that perform computation over a small amount of input values per iteration (see Table 5.4). Loop unrolling increases the size of the AIBs that arrange the compute values and reduces inter-block communication overhead. Loop unrolling increases performance by about 40% (*fft, mm, mriq, nbody* benefit from it). Since the general purpose execution uses loop unrolling by default, the reconfiguration feature sometimes does not yield any extra speedup without applying the loop unrolling optimization (*mriq, nbody* experience slowdown with "No-Optimization"). Software pipelining provides substantial additional performance when the workloads perform complex computations with long latencies (*mm, mriq, nbody*). The AIBs avoid this latency by receiv-

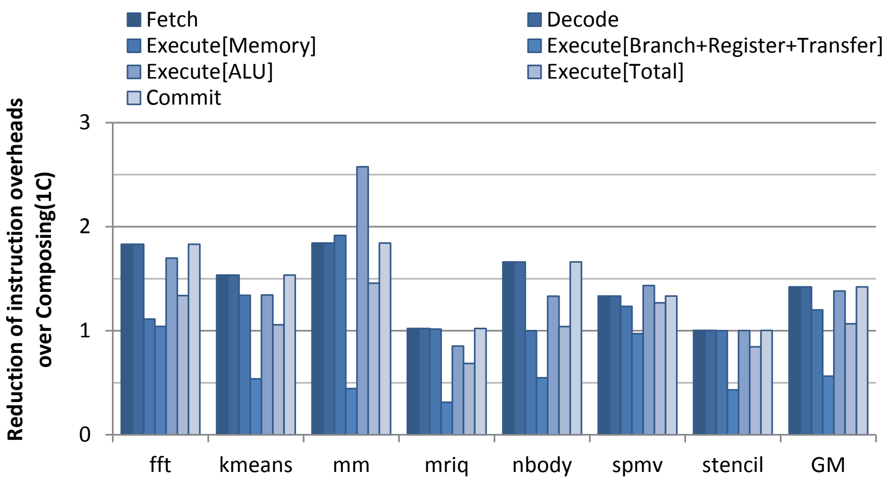


Figure 5.9: Reduction of instruction overheads in the general purpose pipeline when using reconfiguration.

ing the results of previous AIBs. Software pipeline is not applied to the general purpose execution, because the processor executes operations individually and all latencies are small, unlike the compute fabric which fuses multiple operations and produces results with long delays. Software pipelining may have a negative impact to the performance when computation is not extensively long or when an outer loop that repeats the computation is complex (*fft*). Since the software pipelining evaluated in this work does not require extra hardware support(only disables the speculative computation), it may be selectively applied only to workloads which find it beneficial. By choosing the best optimization, performance further increases over 10%.

Figure 5.9 shows the reduction of instruction fetch, decode, execute and commit overheads in the general purpose pipeline, while comparing the results in the pipeline of a 1-core accelerator over the pipeline of a 1-core processor. Additionally, the execution overhead is presented individually for: memory instructions; ALU instructions, including the instructions executed by the general purpose pipeline and the fabric; and other instructions, including branch, register read-/write and the later send/receive instructions. The results show that the accelerator moderately reduces the number of executed memory and ALU instructions, but increases the number of executed control-transfer instructions by 2x. This happens mainly because of the code modifications. The workloads are modified by mapping the floating-point compute instructions to the fabric and refilling the AIBs with extra memory and send/receive instructions to control the computation mapped onto the fabric. These modifications often diminish the number of temporary results stored in memory and bookkeeping complexity. In some workloads, the number of executed memory and ALU instructions is reduced when running on the accelerator (*kmeans, mm, spmv*), because these workloads perform memory accesses and address calculations for temporary results when running on the general purpose processor. Similarly in other workloads, the number of bookkeeping ALU instruction is reduced (*fft, nbody*) when running on the accelerator. As opposed to the number of ALU instructions, appended send/receive instructions in the modified AIBs increase the number of executed branch, register and transfer instructions. For some workloads, the accelerator executes over 3 times more of these instructions (*mriq*). In total, the accelerator does not notably reduce the average number of total executed instructions, which is the expected result. On the other hand, the accelerator reduces the instruction fetch, decode and commit overheads by over 40%. This happens because the accelerator maps the compute instructions to the fabric once, but executes them many times while avoiding fetch, decode and commit stages. The general purpose pipeline incorporated into the accelerator still performs the fetch and decode stages with the same amount of resources. But instead of processing the compute instructions, the pipeline fetches and decodes more memory and transfer instructions in the refilled AIBs. By including more memory instructions per AIB, the pipeline saturates the available memory bandwidth with more memory requests, tolerates memory latency and increases the utilization of the ALUs connected into the fabric. This way, the accelerator effectively leverages the existing resources of the CMP and improves their efficiency as well as performance.

## 5.8 Related Work

To increase flexibility, performance and efficiency, there is research that proposes dynamic CMPs [22, 32, 36, 41]. These processors compose one or more simple cores into a big wide-issue core for single-threaded execution. Through composability, they dynamically tune the allocation of execution resources to a particular workload phase. To further increase the performance and efficiency of dynamic CMPs, we extend the dynamic CMPs with reconfiguration feature. We build our design on composable cores based on the EDGE architecture [7, 69]. Reconfiguration of the composable cores is an advanced dynamic capability, which specializes one or more cores as an accelerator.

There is previous work that proposes to reconfigure existing resources of a processor into some kind of accelerator. In the context of EDGE architectures, vector execution on EDGE cores proposed in Chapters 3 and 4 dynamically repurposes the general purpose cores into a vector processor. The vector processor executes vector AIBs, which allocate the existing compute resources and repeat the computation by streaming the values of large vectors. A vector memory unit is incorporated to decouple vector memory accesses from computation and arrange the vector values in an efficient manner. The vector processor customizes the memory controller for vector access patterns, while the reconfigurable cores proposed in this chapter customize the compute resources for frequently executed computations. These two dynamic approaches have different trade-offs and it may be interesting to have both features on a single chip, applying them selectively depending on the workload or even combining them. For example, when the reconfigured cores compute values with regular access patterns, the vector memory unit may be used to efficiently arrange data and increase performance.

Various reconfigurable compute accelerators [11, 12, 23, 24, 26, 35, 49, 50, 57, 58] have been proposed to accelerate compute intensive code regions. They are added to the baseline processor to enable compute acceleration. The accelerators are based on coarse grained reconfigurable architectures, which incorporate the array of configurable data processing units. Each unit performs one compute task of the parallel workload (e.g. matrix arithmetic, signal or image processing), while data passes through the array. The processing units are either simple like general purpose FUs or more complex processing elements. The units are coupled by using a configurable interconnect in a grid-like compute fabric. Such fabric is integrated into a processor pipeline like a back-end processor. The pipeline arranges data, while the fabric computes the data. The fabric improves performance by overlapping the executions of multiple compute instructions in different stages of the fabric. It also improves efficiency by repeating many times a computation that is configured only once, which reduces per compute instruction fetch and decode overheads. Instead of integrating such fabric into a lightweight mobile processor, we propose to reconfigure the processor compute resources into a fabric. Our fabric is designed to resemble the previously proposed computing substrates, their performance and efficiency, but avoids as much as possible of their area overheads.

The fabric is configured by mapping frequently repeated computations to it. It may be configured statically (compile time) or dynamically (run time). The static configuration requires modification of the ISA; moreover an application compiled to utilize an accelerator cannot run on a processor without that accelerator. The dynamic configuration has a significant performance overhead. To avoid these drawbacks, a virtualized execution accelerator [11] has been proposed. That accelerator utilizes a hybrid static-dynamic configuration approach to map the compute regions to its substrate. To avoid hardware and performance penalties for the virtualized acceleration, in this work we statically configure the compute fabric for different computations.

To enhance the performance of workloads with pipeline compute parallelism in which streaming data is pipelined through various compute algorithms, a programmable multicore accelerator [58] has been proposed. Such accelerator exploits coarse-grained parallelism across different cores and fine-grained parallelism within a single core. One core performs one particular computation and multiple cores enable pipelining of different computations. In this work we do not investigate multicore accelerators, but we believe it is a very promising direction for future work.

## 5.9 Summary

In this chapter, we have presented a novel way to dynamically reconfigure the general purpose resources of a 4-core processor into accelerators. Reconfiguration maximizes the performance and efficiency of the processor yet incurring modest hardware additions. To facilitate reconfiguration, the layout of the processor with four identical cores is slightly modified by placing the execution resources of each core close to each other. Only a configurable circuit-switched network is added to connect the resources and enable their reconfiguration into a large compute fabric that performs computation in the accelerator. Additionally, the general purpose pipeline of one or more cores in the processor stream data through the configured fabric and along with the fabric composes the entire accelerator.

We have evaluated the proposed reconfiguration feature on a 4-core EDGE processor. The results show that reconfiguration yields 56% of an average speedup, while accelerating seven compute intensive workloads. Two of them require extra adders to enable the fabric to specialize its substrate for their computations. Software optimization techniques such as loop unrolling and software pipelining are necessary in some workloads to efficiently feed the accelerator and keep its resources busy. Beyond performance, reconfiguration improves the efficiency of the reconfigured processor by reducing the instruction fetch, decode and commit overheads over 40% on the accelerated workloads.

The promising results evaluated in this work open up new research directions, which involve reconfiguring the cores of mobile processors instead of adding a set of dedicated hardware accelerators. We think that future work in this direction should adapt the proposed design to more powerful 8- or 16- core reconfigurable processors, which would have 2 or 4 banks of compute fabrics. Beyond reconfiguration of the additional execution resources in many-core processors, we think it would be promising to research the additional memory optimizations that may further increase the utilization of the fabric and performance of the accelerators. For example, instead of composing cores to increase memory capabilities of the accelerators, a programmable memory unit could be used to achieve the same or even better results.

# Chapter 6

# Conclusion

This thesis shows us how innovative hardware techniques with minimal modifications of a general purpose EDGE processor greatly improve its performance and efficiency. We believe that these techniques would allow us to implement a low-cost, high-performance and power-efficient processor, which would be a competitive product to more complex heterogeneous processors typically found in nowadays devices.

We have proposed a novel specialization technique, *EVX*, that enables a low power EDGE core to execute sophisticated vector instructions. EVX utilizes the existing general purpose resources of the EDGE core to avoid area and complexity required when incorporating vector processors as accelerators. This way, the EVX enabled core provides the benefits of vector architectures with only modest hardware additions. Namely, EVX loads/stores configurable large vector operands by utilizing a dedicated hardware with sophisticated addressing modes that overlaps the latency of memory accesses with the computation of such configurable large operands. By utilizing the available execution resources more efficiently, EVX yields the high speedup compared to the baseline execution of DLP workloads.

The cost-efficient EVX design presented in this thesis led us to further investigate such approach. We have improved EVX to utilize the additional cores in CMPs. We name this advanced technique *DVX*, since it allows a CMP to dynamically tune the vector execution over the available cores. DVX on-the-fly configures the allocation of compute and memory resources, which are specialized to process vector elements in parallel. It minimizes the configuration overheads by utilizing hardware controlled threads to scale the execution of vector instructions. While dynamically configuring the appropriate number of cores for the vector execution, DVX efficiently accelerates workloads with abundant or insufficient DLP. The specialized execution on general purpose cores delivers energy and performance results competitive to a dedicated vector accelerator and yet avoids the high requirements of extra special-purpose resources for such accelerator.

The encouraging DVX results motivated us to explore another innovative technique that reconfigures cores of a CMP into accelerators. This technique imposes coarse-grained reconfiguration of the existing resources in a general purpose CMP. Reconfiguration of cores in the CMP composes accelerators of compute intensive DLP workloads without adding a special-function compute hardware to the chip. Only a circuit-switched network is added to connect the CMP execution resources and reconfigure them to perform frequently repeated computation in a DLP workload. The proposed reconfiguration technique avoids costs and complexity of dedicated hardware accelerators by delivering desired compute capabilities to diverse DLP workloads.

## 6.1 Moving Forward

We showed that the specialization and reconfiguration techniques improve the performance and efficiency of general purpose cores when running diverse DLP workloads. We see both of these techniques as competitive approaches to improve commercial processors in the domain of low-power mobile computers. We think that both techniques should be incorporated in such design of future mobile processors. They could be applied selectively as well as in synergistic interaction. The following paragraph discusses a few details of how we envision a system that implements both techniques should work.

When an application performs diverse compute algorithms over large vectors of data elements with regular access patterns, the specialization technique should be selected to accelerate the application. One or more cores specialized for vector execution would execute vector compute instructions of these algorithms over the partitioned vectors of data elements; and per core dedicated memory units would load/store the elements by utilizing their memory access patterns. When an application performs the same compute algorithm over large vectors of data elements with irregular access patterns, the reconfiguration technique should be selected to accelerate the application. In this case, one or more cores reconfigured into accelerator would perform statically configured computation of the algorithm over a large amount of data; and the execution substrate in the accelerator would be configured for such computation by utilizing a compute instruction pattern that repeat over different elements of the vectors. When an application performs the same compute algorithm over large vectors of data elements with regular access patterns, both techniques should be applied to accelerate the application in a synergistic manner. In this special case, the dedicated memory units would be configured to load/store the data elements and the execution substrate to perform computation over the elements.

We believe that the specialization and reconfiguration techniques along with their results discussed in this thesis encourage further research of the enhanced processor design based on these techniques. This research should incorporate the evaluation of one such processor for mobile devices when running a broad range of mobile workloads. The next step of future work should include the full system simulation with the interaction of multiple applications and operating systems. In this step, the optimal scheduling of multiple workloads onto the reconfigurable resources of the enhanced processor should be investigated. This would probably lead to proposing new hardware/software additions or modifications. The virtualization of reconfigurable resources would be challenging as well and seems like an interesting follow up research.

Apart from these simulated experiments, developing an RTL design of the proposed processor would allow for validating its results in performance, power and area domains. These results then could be compared against the results achieved with more complex state-of-the-art mobile processors based on heterogeneous system architecture. The experiments with the RTL processor design would show more advantages and/or limitations of our ideas. Although there are obvious challenges with these ideas, we still believe that the RTL design would prove their quality and hence we would like to have one such design in the future.

# Chapter 7

# Publications

The contents of this thesis led to the following publications:

- ReConFig 2013: ReCompAc: Reconfigurable compute accelerator, Milovan Duric, Oscar Palomar, Aaron Smith.
- DATE 2014: EVX: Vector execution on low power EDGE cores, Milovan Duric, Oscar Palomar, Aaron Smith, Osman S. Unsal, Adrin Cristal, Mateo Valero, Doug Burger.
- ICSAMOS 2014: Dynamic-vector execution on a general purpose EDGE chip multiprocessor, Milovan Duric, Oscar Palomar, Aaron Smith, Milan Stanic, Osman S. Unsal, Adrin Cristal, Mateo Valero, Doug Burger, Alexander V. Veidenbaum.
- ICSAMOS 2015: Imposing Coarse-Grained Reconfiguration to General Purpose Processors, Milovan Duric, Oscar Palomar, Milan Stanic, Osman S. Unsal, Adrin Cristal, Mateo Valero, Aaron Smith.
- In submission at IJPP: Dynamic Specialization of Low-Power Processors for Data-Parallel Applications, Milovan Duric, Oscar Palomar, Milan Stanic, Osman S. Unsal, Adrin Cristal, Mateo Valero, Aaron Smith.

The following papers were also published while on graduate studies but are not included in or directly related to this thesis:

- Conf. Computing Frontiers 2014: VALib and SimpleVector: tools for rapid initial research on vector architectures, Milan Stanic, Oscar Palomar, Ivan Ratkovic, Milovan Duric, Osman S. Unsal, Adrin Cristal.

- HPCS 2014: Evaluation of vectorization potential of Graph500 on Intel's Xeon Phi, Milan Stanic, Oscar Palomar, Ivan Ratkovic, Milovan Duric, Osman S. Unsal, Adrin Cristal, Mateo Valero.

- ISVLSI 2015: Joint Circuit-System Design Space Exploration of Multiplier Unit Structure for Energy-Efficient Vector Processors, Ivan Ratkovic, Oscar Palomar, Milan Stanic, Djordje Pesic, Milovan Duric, Osman Unsal, Adrian Cristal, Mateo Valero.

# References

[1] Dennis Abts, Abdulla Bataineh, Steve Scott, Greg Faanes, Jim Schwarzmeier, Eric Lundberg, Tim Johnson, Mike Bye, and Gerald Schwoerer. The cray blackwidow: A highly scalable vector multiprocessor. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 17:1–17:12, New York, NY, USA, 2007. ACM. 4, 8

[2] K. Asanovic and J. Beck. T0 engineering data. Technical Report CSD-97-931, 1997. 8

[3] Krste Asanovic. *Vector Microprocessors*. PhD thesis, University of California, Berkeley, 1998. 25, 72, 111

[4] Christopher Batten, Ronny Krashinsky, Steve Gerding, and Krste Asanovic. Cache refill/access decoupling for vector machines. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, pages 331–342. 111, 112

[5] Nathan Binkert et al. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011. 55

[6] N. Brookwood. AMD fusion family of APUs: enabling a superior, immersive PC experience. White Paper, 2010. 2, 15, 113, 115

[7] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, William Yoder, Xia Chen, Rajagopalan Desikan, Saurabh Drolia, Jon Gibson, Madhu Saravana Sibi Govindan, Paul Gratz, Heather Hanson, Changkyu Kim, Sundeep Kumar Kushwaha, Haiming Liu, Ramadass

Nagarajan, Nitya Ranganathan, Eric Reeber, Karthikeyan Sankaralingam, Simha Sethumadhavan, Premkishore Sivakumar, and Aaron Smith. Scaling to the End of Silicon with EDGE Architectures. *IEEE Computer*, 37(7):44–55, July 2004. 5, 17, 142

[8] M. Buxton, K. N. P. Jinbo, and N. Firasta. Intel avx: New frontiers in performance improvements and energy efficiency. White Paper, 2008. 4, 12, 74, 113

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC'09*, pages 44–54. 132

[10] Silviu Ciricescu et al. The reconfigurable streaming vector processor (RSVP$^{TM}$). In *Proc. of the 36th MICRO*, pages 141–150, 2003. 11, 12, 30, 38, 72

[11] N. Clark, A. Hormati, and S. Mahlke. Veal: Virtualized execution accelerator for loops. In *ISCA'08*, pages 389–400, 2008. 16, 142, 143

[12] N. Clark, M. Kudlur, Hyunchul Park, S. Mahlke, and K. Flautner. Application-specific processing on a general-purpose core via transparent instruction set customization. In *MICRO'04*, pages 30–40. 16, 142

[13] Convex Press. *CONVEX Architecture Reference Manual (C Series)*, 6th edition, April 1992. 8

[14] Henry Cook. Virtualizing local stores. Master's thesis, University of California, Berkeley, December 2009. 44

[15] Kenneth Czechowski, Victor W. Lee, Ed Grochowski, Ronny Ronen, Ronak Singhal, Richard Vuduc, and Pradeep Dubey. Improving the energy efficiency of big cores. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 493–504, Piscataway, NJ, USA, 2014. IEEE Press. 56

[16] R. Espasa and M. Valero. Decoupled Vector Architectures. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 281–290. 11, 38, 72, 80, 91, 111, 112

[17] Roger Espasa, Federico Ardanaz, Joel Emer, Stephen Felix, Julio Gago, Roger Gramunt, Isaac Hernandez, Toni Juan, Geoff Lowney, Matthew Mattina, and André Seznec. Tarantula: A Vector Extension to the Alpha Architecture. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 281–292. 8, 47, 111

[18] Roger Espasa, Mateo Valero, and James E. Smith. Out-of-order Vector Architectures. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 160–170. 11, 72, 111, 112

[19] Roger Espasa, Mateo Valero, and James E. Smith. Vector architectures: past, present and future. In *Proceedings of the 12th International Conference on Supercomputing*, pages 425–432, 1998. 4, 72

[20] S. Fuller. Motorola's AltiVec technology. White paper, 1998. 4, 12, 74, 113

[21] Mark Gebhart, Bertrand A. Maher, Katherine E. Coons, Jeff Diamond, Paul Gratz, Mario Marino, Nitya Ranganathan, Behnam Robatmili, Aaron Smith, James Burrill, Stephen W. Keckler, Doug Burger, and Kathryn S. McKinley. An evaluation of the trips computer system. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1–12, March 2009. 5

[22] Madhu Saravana Sibi Govindan, Behnam Robatmili, Dong Li, Bertrand Maher, Aaron Smith, Stephen W. Keckler, and Doug Burger. Scaling Power and Performance via Processor Composability. *IEEE Transactions on Computers*, March 2013. 4, 6, 141

[23] V. Govindaraju et al. DySER: unifying functionality and parallelism specialization for energy efficient computing. *IEEE Micro*, 32(5):38–51, 2012. 16, 142

[24] V. Govindaraju, Chen-Han Ho, and K. Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *HPCA'11*, pages 503–514. 17, 119, 122, 132, 133, 142

[25] P. Greenhalgh. Big.LITTLE processing with ARM Cortex$^{TM}$-A15 & Cortex-A7. White Paper, 2011. 2, 115

[26] Shantanu Gupta, Shuguang Feng, Amin Ansari, Scott Mahlke, and David August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 12–23, New York, NY, USA, 2011. ACM. 117, 142

[27] L. Gwennap. Sandy Bridge spans generations. *Microprocessor Report*, September 2010. 15, 113

[28] Timothy Hayes, Oscar Palomar, Osman Unsal, Adrian Cristal, and Mateo Valero. Vector extensions for decision support dbms acceleration. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pages 166–176. IEEE, 2012. 29

[29] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 5th edition, 2012. 27

[30] R. G. Hintz and D. P. Tate. Control data STAR-100 processor design. In *Compcon 72*, pages 1–4, 1972, IEEE. 4, 8

[31] Tassadaq Hussain, Oscar Palomar, Osman Unsal, Adrian Cristal, Eduard Ayguade, and Mateo Valero. Advanced pattern based memory controller for fpga based hpc applications. In *HPCS'14*, pages 287–294. 137

[32] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core fusion: Accommodating software diversity in chip multiprocessors. *SIGARCH Comput. Archit. News*, 35(2):186–197, June 2007. 4, 141

[33] Rabiul Islam, Anil Sabbavarapu, and Rajesh Patel. Power reduction schemes in next generation intel® atom processor based soc for handheld applications. In *VLSI Circuits (VLSIC), 2010 IEEE Symposium on*, pages 173–174. IEEE, 2010. 5

[34] Ujval Kapasi et al. The imagine stream processor. In *Proc. of the ICCD '02*. 12, 73

[35] S. Khawam, I Nousias, M. Milward, Ying Yi, M. Muir, and T. Arslan. The reconfigurable instruction cell array. *VLSI Systems, IEEE Trans.*, 16(1):75–85, 2008. 142

[36] Changkyu Kim, S. Sethumadhavan, D. Gulati, D. Burger, M.S. Govindan, N. Ranganathan, and S.W. Keckler. Composable Lightweight Processors. In *MICRO'07*. 4, 6, 19, 23, 141

[37] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. When polyhedral transformations meet SIMD code generation. In *Proceedings of the 34th ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 127–138, 2013. 31

[38] Christoforos Kozyrakis. Scalable vector media-processors for embedded systems. Technical report, Berkeley, CA, USA, 2002. 8, 11

[39] Christos Kozyrakis and David Patterson. Overcoming the Limitations of Conventional Vector Processors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 399–409. 8, 72, 111, 112

[40] Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, and Krste Asanovic. The vector-thread architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 52–63. 8, 11, 111, 112

[41] R. Kumar, N.P. Jouppi, and D.M. Tullsen. Conjoined-core chip multiprocessing. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pages 195–206, Dec 2004. 4, 141

[42] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *MICRO'03*, pages 81–92. 2, 115

[43] Rakesh Kumar, Dean M Tullsen, and Norman P Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 23–32. ACM, 2006. 2, 115

[44] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 129–140, 2011. 4, 5, 8

[45] S. Li, J.H. Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi. McPAT: an Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *42nd Annual International Symposium on Microarchitecture (MICRO)*, pages 469–480, 2009. 54, 99

[46] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, March 2008. 4, 8, 14, 113

[47] Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 149–158, New York, NY, USA, 2001. ACM. 14, 113

[48] F. M. McMahon. The Livermore FORTRAN Kernels: A Computer Test of Numerical Performance Range. Technical Report UCRL-55745, Lawrence Livermore National Laboratory, University of California. 55

[49] Mei et al. Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In *Field Programmable Logic and Application*, pages 61–70. Springer, 2003. 142

[50] Mahim Mishra, Timothy J. Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C. Goldstein, and Mihai Budiu. Tartan: evaluating spatial computation for whole program execution. *SIGOPS Oper. Syst. Rev.*, 40(5):163–174, October 2006. 17, 142

[51] John Montrym and Henry Moreton. The geforce 6800. *IEEE Micro*, 25(2):41–51, 2005. 14, 113

[52] Gordon E. Moore. Readings in computer architecture. chapter Cramming more components onto integrated circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000. 1

[53] John Nickolls and David Kirk. Graphics and computing gpus. 2003. 4

[54] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. White Paper, 2012. 4, 14, 15, 44, 85, 113

[55] S. Oberman et al. AMD 3DNow! technology: Architecture and implementations. *IEEE Micro*, 19:37–48, 1999. 4, 12, 74, 113

[56] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008. 4, 113

[57] Angshuman Parashar et al. Triggered instructions: a control paradigm for spatially-programmed architectures. In *ISCA'13*, pages 142–153. 17, 142

[58] Hyunchul Park, Yongjun Park, and Scott Mahlke. Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *MICRO'09*, pages 370–380. 17, 117, 142, 143

[59] A. Peleg and U. Weiser. MMX technology extension to the Intel architecture. *IEEE Micro*, 16:42–50, Aug. 1996. 4, 12, 74, 113

[60] Daniel P Playne, Mitchell Johnson, and Kenneth A Hawick. Benchmarking gpu devices with n-body simulations. In *CDES'09*. 132

[61] Francisca Quintana, Jesus Corbal, Roger Espasa, and Mateo Valero. Adding a Vector Unit to a Superscalar Processor. In *Proceedings of the 13th International Conference on Supercomputing*, pages 1–10, 1999. 8, 38, 72, 80, 91, 111

[62] Katie Roberts-Hoffman and Pawankumar Hegde. Arm cortex-a8 vs. intel atom: Architectural and benchmark comparisons. *Dallas: University of Texas at Dallas*, 2009. 5

[63] R. M. Russell. The CRAY-1 computer system. *Commun. ACM*, 21:63–72, Jan. 1978. 4, 8, 9, 72, 111

[64] Karthikeyan Sankaralingam et al. Universal mechanisms for data-parallel architectures. In *Proc. of the 36th MICRO*, pages 303–314, 2003. 73

[65] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 422–433, 2003. 73, 113

[66] Ruchira Sasanka, Man-Lap Li, Sarita V. Adve, Yen-Kuang Chen, and Eric Debes. Alp: Efficient support for all levels of parallelism for complex media applications. *ACM Trans. Archit. Code Optim.*, 4(1), March 2007. 38, 73, 113

[67] Larry Seiler et al. Larrabee: a many-core x86 architecture for visual computing. In *ACM Transactions on Graphics*, volume 27, August 2008. 12, 13

[68] N.T. Slingerland and A.J. Smith. Multimedia extensions for general purpose microprocessors: A survey. *Microprocessors and Microsystems*, 29(5):225–246, 2005. 4

[69] Aaron Smith, Jim Burrill, Jon Gibson, Bertrand Maher, Nick Nethercote, Bill Yoder, Doug Burger, and Kathryn S. McKinley. Compiling for EDGE architectures. In *CGO'06*, pages 185–195. 5, 17, 19, 31, 142

[70] Aaron Smith et al. Dataflow predication. In *Proc. of the 39th MICRO*, pages 89–102, 2006. 19

[71] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Tech. report impact-12-01, Center for Reliable and High-Performance Computing. 132

[72] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling preemptive multiprogramming on gpus. In *Proceeding of the 41st annual international symposium on Computer architecture*, pages 193–204. IEEE Press, 2014. 15

[73] S. Thakkar and T. Huff. Internet streaming SIMD extensions. *Computer*, 32:26–34, December 1999. 4, 12, 74, 113

[74] Sravanthi Kota Venkata et al. SD-VBS: The San Diego Vision Benchmark Suite. In *Proc. of the ISWC*, pages 55–64, 2009. 55

[75] W. Watson. The TI-ASC, a highly modular and exible super computer architecture. In *AFIPS '72 (Fall, part I)*, pages 221–228, 1972. 4, 8

[76] Nathan Whitehead and Alex Fit-Florea. Precision & performance: Floating point and ieee 754 compliance for nvidia gpus. *rn (A+ B)*, 21:1–1874919424, 2011. 15

[77] C.M. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi gf100 gpu architecture. *Micro, IEEE*, 31(2):50–59, March 2011. 4, 8, 14, 15, 113

[78] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 24–36, 1995. 132

[79] Thomas Zeiser, Georg Hager, and Gerhard Wellein. The world's fastest cpu and smp node: Some performance results from the nec sx-9. *Parallel and Distributed Processing Symposium, International*, 0:1–8, 2009. 8