An OpenCL Software Compilation Framework Targeting an SoC-FPGA VLIW Chip Multiprocessor

Samuel J. Parker, Vassilios A. Chouliaras
Wolfson School, Loughborough University,
Loughborough, LE11 3TU, UK

Abstract

Modern systems-on-chip augment their baseline CPU with coprocessors and accelerators to increase overall computational capability and power efficiency, and thus have evolved into heterogeneous multi-core systems. Several languages have been developed to enable this paradigm shift, including CUDA and OpenCL. This paper discusses a unified compilation environment to enable heterogeneous system design through the use of OpenCL and a highly configurable VLIW Chip Multiprocessor architecture known as the LE1. An LLVM compilation framework was researched and a prototype developed to enable the execution of OpenCL applications on a number of hardware configurations of the LE1 CMP. The presented OpenCL framework fully automates the compilation flow and supports work-item coalescing which better maps onto the ILP processor cores of the LE1 architecture. This paper discusses in detail both the software stack and target hardware architecture and evaluates the scalability of the proposed framework by running 12 industry-standard OpenCL benchmarks drawn from the AMD SDK and the Rodinia suites. The benchmarks are executed on 40 LE1 configurations with 10 implemented on an SoC-FPGA and the remaining on a cycle-accurate simulator. Across 12 OpenCL benchmarks results demonstrate near-linear wall-clock performance improvement of 1.8x (using 2 dual-issue cores), up to 5.2x (using 8 dual-issue cores) and on one case, super-linear improvement of 8.4x (FixOffset kernel, 8 dual-issue cores). The number of OpenCL benchmarks evaluated makes this study one of the most complete in the literature.

Index Terms

OpenCL; FPGA; Heterogeneous Computing; Multi-core; Compilation

Corresponding author: v.a.chouliaras@lboro.ac.uk. Tel:+44(0)1509 227113

I. Introduction

State-of-the-art silicon technology nodes empowered VLSI designers to integrate complex functionality on a single chip with such advanced Systems-on-Chip (SoC) incorporating multiple diverse (Heterogeneous) processing engines and connected via numerous, high bandwidth, point-to-point links. These engines are supplied with data by hundreds of local memory blocks under the control of Direct Memory Access (DMA) engines. On this bespoke computing substrate there is the implicit requirement that millions of lines of both legacy and new application code will run efficiently with both software and hardware components expected to be delivered to market under very tight deadlines. Further complications such as the substantial non-recurrent costs involved and verification closure at tape-out make state-of-the-art SoC design inaccessible to all but the largest of organizations.

In parallel, industry witnesses a revolution in performance and capability of Field-Programmable Gate Arrays (FPGAs) with the leading vendors (Xilinx and Altera, now Intel) consistently delivering high capacity programmable silicon incorporating hundreds of embedded (hard-wired) blocks. These include memory controllers, DSPs, clocking infrastructure, high-throughput interfaces (PCIe) and networking capability (Interlaken), supported by very high speed differential I/O (SERDES). The vendors supply a wealth of silicon intellectual property (IP) such as soft processors and more recently, high-value hardened IP (ARM A9 SMP subsystem in the Zynq [1] and Cyclone V SoC device families respectively), advanced interconnect (AXI4) and a number of other blocks covering every conceivable application. What is even more noteworthy is that this rich ecosystem, along with proprietary Electronic Design Automation (EDA) tools is provided for (nearly) free to the FPGA silicon customers. To address the design and verification bottleneck of very complex current (28 nm) and expected (16/14 nm) SoC-FPGAs vendors increasingly embrace a software-centric design approach based on Electronic System Level Methodologies (ESL). Potentially disruptive ESL technologies such as *Behavioural Synthesis* (AutoESL [2] from Xilinx and C2H from Altera and very recently tools such as SDSoC/SDAccel and AOCL [3] respectively) seem to be displacing established Register-Transfer-Level (RTL) methodologies when targeting these latest devices.

With the introduction of General-Purpose Graphics Programming Units (GPGPUs) and the release of the proprietary CUDA API [4] from NVIDIA, a trend towards the universal use of such devices in a number of market segments (spanning the continuum from High Performance Computing (HPC, [5]), Desktop and all the way to embedded and mobile computing) is emerging. The Open Compute Language (OpenCL, [6]) was proposed as an open standard API for general-purpose computing across CPUs, GPGPUs and other accelerators in response to CUDAs performance advantage on NVIDIA hardware. This was standardized by the Khronos Group and nowadays, OpenCL drivers are offered by all the major graphic processor designers such as AMD, Intel, and Qualcomm. [1] Unlike CUDA, OpenCL is target agnostic and this has enabled the emergence of an ecosystem around not only GPGPUs but also CPUs and FPGAs as will be discussed in Section II.

This research is motivated by the ever-increasing adoption of the Single Instruction Multiple Thread (SIMT) processing paradigm (via OpenCL) for advanced FPGA design and this paper presents an automated compilation framework that enables parallel computation, through the execution of OpenCL kernels[2] on a configurable VLIW Chip Multiprocessor (CMP) [7][8]. The LE1 architecture (Section III-A) is both configurable and extensible and is designed for embedded DSP applications on FPGA and standard-cell silicon. The researched software framework is in the form of a *user-space driver* which encompasses an LLVM-based compiler back-end as well as a source-to-source transformer that modifies the OpenCL kernels to execute more effectively on the LE1. A high-level view of the researched software/hardware framework is shown in Fig. 1. From the figure, inputs to the framework are the kernel and the machine description (machine.xml) which specifies micro-architectural parameters of the LE1 CMP instance. The kernel is transformed and compiled with a custom LLVM back-end developed for the LE1 resulting in a number of assembly (.s) files. These are combined into two binaries (iram.h, dram.h) with the instruction stream and the initialized data section loaded onto the processor via the API (executing on the ARM host). The final executable is loaded onto the FPGA target via the Xilinx Microprocessor Debugger (xmd) tool. At the same time, the tool-chain is used to validate the LE1 CMP at Register Transfer Level (RTL) using the flow depicted in the bottom half of Fig. 1.

The framework is capable of targeting many hardware configurations (as specified in the machine.xml) and executes OpenCL kernels both on the LE1 CMP, mapped onto a Zynq z7045 device (Xilinx zc706 development board), as well as on a highly cycle-accurate simulator. We evaluate the scalability of our approach using 12 OpenCL benchmarks from the AMD[3] and Rodinia [9] benchmark suites (Section IV-A1), across 40 machine configurations, making this the largest OpenCL study reported in academic literature to date.

The paper is organized as follows: Section II presents the background, state-of-the-art and motivation behind this research. The proposed software/hardware approach is introduced in Section III and the detailed methodologies are discussed in Section IV. Section V presents the execution results from

[1]Officially conformant devices:http://www.khronos.org/conformance/adopters/conformant-products#opencl

[2]An OpenCL kernel is a function executed by multiple processing elements on a 1D/2D/3D application space. Kernels are C-based and their arguments are augmented with memory space specifiers (private, local and global). OpenCL enables the execution of hundreds/thousands of such functions across multiple processing elements (PEs) resulting in substantial performance improvement compared to the sequential version of the application. Kernels are grouped into 'Work-groups' (WG) and multiple such work-groups constitute a Compute Unit (CU).

[3]http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk

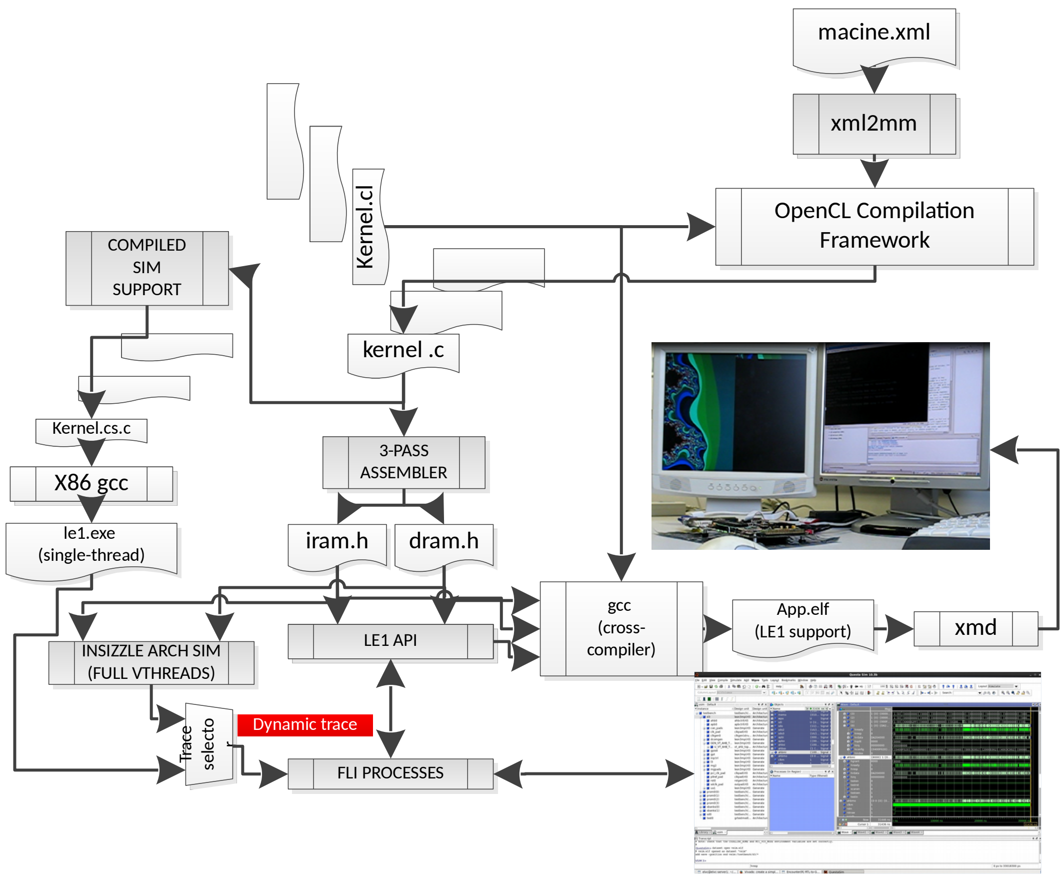

Fig. 1. High level view of proposed hardware and software framework for OpenCL compilation and execution integrated with the LE1 Tool-chain

applying our framework on the chosen OpenCL benchmarks and includes a thorough discussion of our findings. Section VI draws conclusions on the efficiency of our solution and this paper concludes with a number of suggestions for future software and hardware improvements in Section VII.

II. Motivation and Background

A. Motivation

The majority of accelerators currently used are deeply multi-threaded, many-core systems such as GPGPUs and Intel's MIC architecture. GPGPUs offer higher performance and energy-efficiency compared to commodity x86 CPUs. However, the US Department of Energy has identified custom designs and the use of co-design as very important in producing even more efficient computers [10]. Co-design can be used to create application specific instruction set extensions for deeply-embedded configurable processors and optimize the design of heterogeneous multi-core SoCs for more efficient computing [11]; a key reason for it's use is that the GPGPU execution model is not suitable for all types of problems. The latter relies on an implicit SIMD execution model (SIMT) where the concurrent execution of hundreds of threads is used to mask stalls and long latency operations.

GPGPUs achieve maximum throughput when the executing threads maintain the same program counter (PC), allowing the single-issued instruction to execute with different data across hundreds of data-paths. This also allows threads to issue memory operations with high spatial locality resulting in data traffic optimization in the memory hierarchy. These constraints have little effect on highly-regular graphic shader programs, but throughput can dramatically decrease in the presence of control-flow with bespoke solutions proposed to alleviate thread divergence [12][13]. System designers have looked into building systems with many cores that are not multi-threaded [14][15], but this approach still does not address the fact that not all problems can be solved effectively in the same manner.

FPGAs, by virtue of their user programmability and dense floating point performance (Altera *Arria10* and *Stratix10* families), are in a unique position of being adopted as universal OpenCL targets and previous generations of these devices have been shown to be faster than GPGPUs for some algorithms [16]. There are, however, major barriers in their widespread adoption as accelerators relating to the skill-set required to design, optimize and verify designs as well as long FPGA tools compilation times (hours to day/s). The latter makes them completely unsuitable for runtime OpenCL kernel compilation which is one of the cornerstones of the OpenCL API. High-level synthesis (HLS) has addressed these issues to a degree by offering higher levels of abstraction with more commonly used languages, such as C, SystemC and more recently, OpenCL. This does not address the issue of place and route time, and for absolute performance VLSI engineers still design at RT level.

The main motivation behind the researched software framework is the need to offer a fully programmable compute engine as a solution between fixed many-core systems such as the Intel MIC and the very fine-grained control when targeting SoC-FPGAs, while eliminating branch-divergence through source-transformation and ILP compilation and alleviating the substantial FPGA place-and-route runtimes. This is achieved through our core contributions which include the instantiation on the SoC-FPGA of the LE1 CMP and the subsequent on-line compilation of OpenCL kernels by our framework targeting the LE1 silicon. We also note that the LE1 is a capable MIMD accelerator, can easily accommodate shared-memory programming models such as OpenMP and POSIX Threads (PThreads) [17] and due to the proposed source transformation/compilation flow (Section IV), it does not suffer software-incurred performance inefficiencies due to thread divergence. The authors are unaware of any current heterogeneous systems that use fully configurable general-purpose, many-core, VLIW microprocessors as OpenCL accelerators on SoC-FPGAs.

B. Background

OpenCL [6] is a programming language and execution framework designed to allow programmers to offload compute intensive kernels to accelerators. OpenCL programs are split into two parts: host and device code. The host code can be written in a variety of languages (C, C++, with bindings for Python[4] and Java[5] amongst others) and runs on the host CPU. The purpose of the standard is to discover accelerator devices, submit work and manage data transfers to and from them. The host program is also free to make use of all the native parallelism, in the form of shared memory APIs such as OpenMP and PThreads and distributed memory (MPI), natively supported in the host environment (typically Linux). The next two sub-sections discuss OpenCL-related research targeting Multi-core CPUs/DSPs and FPGAs.

1) Multi-core CPUs and DSPs:

The MCUDA framework was developed to enable CUDA execution on multi-core CPUs [18]. As the overhead of managing and executing thousands of threads on a CPU can have a detrimental effect on performance, the unit of work was increased to the *thread block*. Loops were introduced to run the CUDA threads serially; *deep fission* being used to maintain synchronisation statement semantics within control structures. Such synchronisation points are control statements, such as `gotos` and `labels`, as they partition conditional regions that contain the `__syncthreads` command. For variables live past synchronisation points, the authors selectively replicated the necessary thread-dependent variables by expanding them into arrays and simply removing the `shared` keyword as these are not private to each thread any more. MCUDA uses PThreads to issue thread blocks across the CPU cores.

Twin Peaks is a software system designed to better utilise the system resources by executing kernels on CPUs as well as GPGPUs, taking into consideration the memory hierarchy of the former [19]. The aim is to use the CPU for smaller kernels as the communication overhead between the CPU and GPGPU address spaces is significant. This framework assigns a single CPU thread to a WG, utilising all the cores until all the WGs have completed. In the absence of any barriers each WI is completed and then another WI scheduled. Whereas in the presence of barriers the `setjmp` function, from the C standard library, is used to save the program state before executing the next WI. Once all the WIs have reached the synchronisation point, the `longjmp` function is used to restore the context of the WI to continue execution of the kernel.

A framework was devised to enable OpenCL capability on heterogeneous multi-core system with local memory, such as the IBM Cell BE processors [20]. The CBE contains one general-purpose processor core (GPC) and multiple accelerator processor cores (APCs), with the GPC generally performing system management tasks while the APCs are dedicated to compute-intensive workloads. The APCs are connected via a bi-directional interconnect (ring each way), utilize DMA-driven local memories with software managing data coherency amongst them. A similar technique to MCUDA

[4]PyOpenCL:http://mathema.tician.de/software/pyopencl

[5]JOCL:http://jogamp.org/jocl/www/

is used to transform the source code to embody the kernel within a triple nested loop, a technique known as *WI coalescing*. Private variables live beyond the scope of the nested regions are expanded into arrays and the authors also use a *web* of variable values to reduce memory usage.

The PACDSP is a five-way, dual-clustered VLIW DSP core with SIMD instructions and a distributed register file. Each cluster contains a load/store unit (LSU) and an ALU, with the fifth execution slot utilised by a shared scalar unit [21]. The PACDUO is platform with a dual-core PACDSP coupled to an ARM core and OpenCL is enabled on this device through source transformations. These serialise the WIs into a loop, vectorize the kernel using target intrinsics, employ *software thread integration* to merge conditional statements of concurrent threads and finally use intrinsics to assign work to the clusters. This is the closest architecture to ours however, it seems to lack the extensive multi-core scalability provided by the LE1 CMP architecture.

2) OpenCL on FPGAs:

In general, researchers have followed two routes to mapping OpenCL applications to FPGAs: A) ESL-based methodologies in which OpenCL is the input language to high-level synthesis tools and B) Using template architectures. There is an interesting distinction here between our proposed solution and the latter as will be discussed in the next Section; suffice to say that the LE1 target architecture is not a template but a fully programmable and highly configurable/extensible embedded VLIW CMP.

FCUDA was built upon the work of MCUDA, but instead of using CUDA to target CPUs, it uses HLS targeting FPGA silicon [22]. The framework uses the same methods of serialising kernels as MCUDA but also makes use of annotations (synthesis directives) on the kernel source to drive HLS. AutoPilot [23] was used as the HLS tool and the flow includes transformation of kernels into AutoPilot C; the latter is a subset of C designed for hardware synthesis. These source annotations enable the synthesis tool to generate appropriate circuitry for both data transfer and computation proper. Pragmas are also used to specify the multiplicity of the parallel processing cores.

MARC is a many-core architecture developed by researchers at Berkeley, comprising of a single control processor and a variable number of algorithmic processing cores [24]. The control processor is a simple RISC CPU while the algorithmic cores are simplified MIPS cores with fine-grained multi-threading and an extensible ISA. Barrier and atomic swap instructions are also included in the ISA for inter-kernel communication via shared memory. The private memory was implemented in distributed (LUT-based) RAMs with local and global memories residing in block RAMs. The global memory size be extended by using external memory and the compilation system is based on LLVM. The researchers designed application-specific processing cores by transforming the LLVM IR instructions to an optimised, predicated SSA form, directly mapping to pre-determined hardware primitives. The key differences between this architecture and the LE1 architecture is the use of a highly parameterizable VLIW (ILP) architecture for the compute units while being fully host agnostic (no need for a MIPS host).

POCL is a portable OpenCL implementation used within the TTA-based Co-design Environment (TCE) which targets a configurable TTA processor [25] in which both the host and device codes are merged into a single program [26]. The target architecture is configurable in the number and mix of functional units as well as having the capability of custom instructions to help accelerate the given algorithm. After the user has specified any custom operations, the system iteratively adds in functional units and register files, to satisfy the computational requirements and ILP of the kernel. A set of low-level LLVM passes operating on the IR are used to modify the kernel to chain several instances together, something analogous to loop unrolling. These passes also maintain the parallel semantics that OpenCL kernels explicitly provide, while code size is kept under control by setting an upper limit on the number of chained instances; any instances above the limit were rolled into a loop.

SOpenCL is an architectural synthesis tool that also maps OpenCL kernels to FPGA fabrics [27]. The tool uses an architectural (hardware) template that can be instantiated to match the target application data-flow using a network of FUs, stream units and distributed control logic to reconfigure the data-paths between producer and consumer units. The compilation front-end uses source transformations to convert the OpenCL kernel into a C function, while also coarsening it to represent a WG instead of a WI. The coarsened kernel is optimised and converted into a single basic block through if-conversion and the code is finally used to generate the streaming and compute engines.

A similar approach was taken to create an OpenCL compiler for a coarse grained reconfigurable architecture (CGRA) [28], specifically the SRP from Samsung, which is a VLIW architecture coupled with a CGRA. SRP has a simple memory hierarchy, using a scratch-pad memory instead of a data cache, similar to the LE1 Data Memory System. The CGRA is used to accelerate the kernel and consists of an array of PEs, such as FUs and register files, connected by dedicated buses. The compilation framework serialises the WIs (Kernel code) into loops via source transformations and in the process, it re-writes the source into standard C. The loops of that C application are then unrolled and modulo-scheduled to fully utilise the available functional units of the VLIW engine and the CGRA.

Altera has been the first adopter of OpenCL for their FPGA silicon [3] with the aims of reducing the very steep learning curve of high-throughput FPGA design while ensuring that algorithms are portable across different FPGA families. Altera's OpenCL compiler (AOCL [3]) transforms OpenCL kernels into deeply pipelined circuits to be mapped onto the FPGA fabric. The pipelined design allows for the data for each thread to be clocked in sequentially so that each stage of the pipeline can be used by different instances of the WI. Multiple pipelines can also be instantiated in parallel to further increase throughput. As well as the kernel data-path, the compiler also creates memory interfaces:

global loads and stores are performed using LSUs connected via a global interconnect to off-chip DDR whereas local accesses target on-chip static RAMs (Block-RAMs).

Prior to the very recent announcement for OpenCL support from Xilinx in the Vivado design suite 2014.2, research was undertaken to convert kernels into AutoESL C code and subsequently, synthesise them to silicon [29]. The Clang AST libraries were used alongside Graphtool to transform these kernels for processing by the synthesis tool; this involved converting barrier calls to *barrier_hit* and *barrier_done* signals as well creating interfaces to the block RAMs for the kernel arguments. The researchers used a Convey HC-1 hybrid system consisting of an Intel Xeon CPU and four Virtex-5 (XC5VLX330) FPGAs, the CPU acting as the host while each FPGA performed as a CU in the compute device. High-level and logic synthesis were performed by Xilinx AutoESL and ISE respectively with compilation performed offline due to excessive run-times; in this system, work size and dimensions are fixed at compile time.

Very recent research has been conducted on improving the compilation speed of OpenCL via HLS by using virtual coarse-grained reconfigurable contexts [30]. The authors use *intermediate fabrics* (IFs) which provide the virtual coarse-grained resources atop a physical FPGA. The IFs map behaviour onto application-specialised resource, such as floating-point units, instead of thousands of LUTs. To create an IF, the OpenCL kernels are compiled into LLVM IR and custom intrinsics are used for OpenCL built-in functions. The IR is then used to create a control data-flow graph (CDFG), mapping LLVM instructions to compatible cores provided by a user-specified library. The framework analyses the requirements of kernels and clusters the kernels into reconfiguration contexts, based on their functional similarity. Each context can implement one kernel at a time, time-multiplexing instances of it, with the WIs carefully pipelined to exploit data reuse. The clustering enables order-of-magnitude faster compilation and reconfiguration between kernels invocations. The authors report a compilation time speed-up of 4,211x while incurring 1.8x area overhead to implement a system of 20 kernels, compared to traditional synthesis techniques.

In summarising this Section, it is observed that multiple solutions exist for both ASIC processors and more recently, FPGAs. A major concern in practically all these solutions is the lack of detailed silicon execution statistics[6] for more elaborate OpenCL benchmarks. This observation was the primary motivating factor behind this research along with the need to elaborate further on the use of programmable architectures on SoC-FPGAs by making use of a highly configurable and extensible VLIW CMP. At the same time, the proposed research comprehensively addresses the issue of the simplistic benchmarking encountered in practically all previous studies by using OpenCL programs comprising

[6]Statistics such as those collected from either the instrumentation peripheral of the LE1 CMP or the Cycle-Accurate simulator. These include amongst others the total number of cycles the contexts were active for, stalls due to branch mis-predictions, memory bank congestion and execution due to LIW inter-packet dependencies.

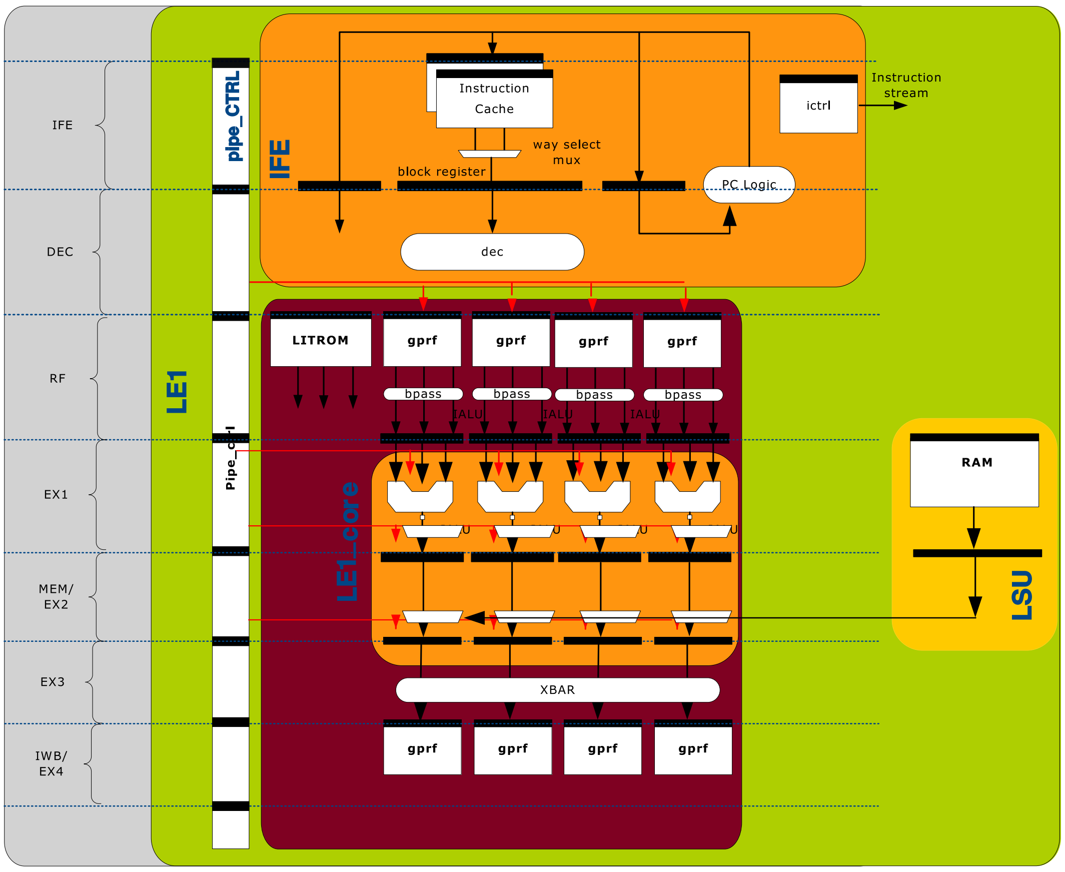

Fig. 2. Architecture overview of a 4-wide, LE1 core. The single core includes an Instruction Front End (IFE), the processing data-paths (LE1_CORE), and the Load-Store Unit (LSU). The pipeline stages are shown to the left.

of multiple kernels, heavy control-flow and executing on silicon (Xilinx zc706 development kit) for 10 machine configurations and on a highly-precise (near RT-level accuracy) cycle-accurate model of the LE1 platform for another 30 machine configurations.

III. Proposed Solution

A. The LE1 Architecture

The LE1 VLIW CMP is a configurable system-on-chip multiprocessor system, designed to accelerate signal and image processing algorithms on both FPGA and standard-cell silicon [31]. It is designed to be attached to a larger system which includes one or more scalar CPUs, running the OS, performing high-level data scheduling and interfacing. These external CPU(s) typically load the Instruction RAM (IRAM) of a particular context (processor core) with the program binary and the shared Data RAM (DRAM) with the initialised data section of that program. The host then initiates execution via issuing commands on a Cmd/Debug Interface and upon completion of the executing algorithm, the host processor extracts the results from the data memory of the LE1 system and allocates new tasks.

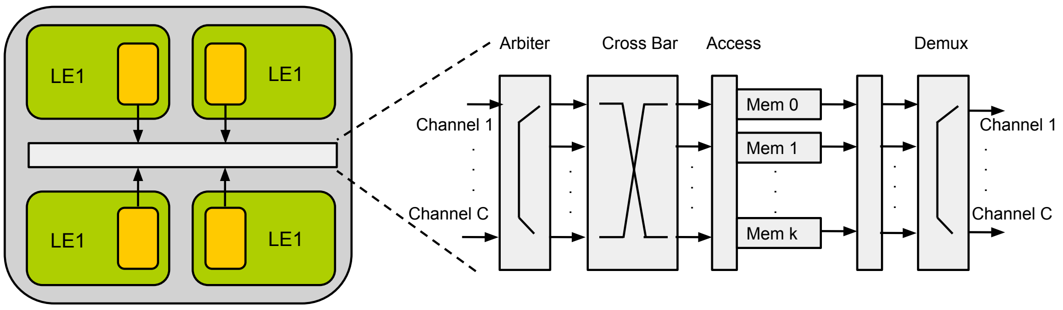

Fig. 3. High-level view of a Quad-Context (core) LE1 with the shared memory system. The latter includes a number of channels from the requesting contexts arbitrating for the use of up to K-banks.

A VLIW CMP was chosen as such architectures efficiently handle parallelism both at instruction (ILP) and thread (TLP) levels. ILP is exploited via the static (compile-time) specification of independent operations (referred to as ’syllables’ or RISCops) per VLIW instruction whereas TLP is exploited via the concurrent execution of coalesced WIs across multiple VLIW Cores (one WG/Core) thus achieving concurrent execution of multiple WGs.

A view of a 4-wide LE1 micro-architecture configuration is depicted in Fig. 2. The CPU consists of the Instruction Fetch Engine (IFE), the execution core, the pipeline controller (PIPE_CTRL) and the Load/Store Unit (LSU). The IFE can be configured with an instruction cache or alternatively, a closely-coupled instruction RAM (IRAM). These are accessed every cycle and return a long instruction word (LIW) consisting of multiple RISCops for decode and dispatch. The IFE controller handles interfacing to the external memory for ICache refills and provides debug capability into the ICache/IRAM. The PIPE_CTRL is a collection of interlocked, pipelined state machines, which schedule the execution data-paths, monitor the overall instruction flow down the processing and memory pipelines and maintain the decoding logic and control registers of the CPU. The LSU is the primary path of the core to the system memory and allows for up to the *issue_width*[7] memory operations per cycle and directly communicates with the shared data memory. The memory is a multi-banked, 2 or 3-stage pipelined cross-bar architecture and the number of channels and banks do not have to be equal. A quad-core LE1 system with the connecting memory system is shown in Fig. 3.

The LE1 implements the VT32PP ISA, loosely based upon the partially-predicated Multiflow architecture [32], which specifies a configurable (max. 16) number of clusters, each consisting of up to 64 static general purpose registers, 8 single-bit predicate registers (used for computing branch conditions and conditional selection), a PC and a Link register (LR). The architecture allows for each core to have the capacity for up to 16-way vertical multi-threading, custom instruction extensions and a

[7]This is the architectural width of the VLIW processor core and specifies the number of syllables that constitute an LIW packet. All syllables in the packet are statically scheduled by the compiler and execute concurrently under LEQ semantics.

floating-point pipeline. Note that multi-threading and FP support are not implemented in LE1 hardware used for this work. Up to 256 cores are enabled by the architecture and we used a maximum of 8 in our experiments due to FPGA capacity limitations. The micro-architecture is further configurable by: the issue width, number of ALUs, multipliers, LSUs and the number of memory banks. The results from the experiments presented in Section V utilise a single cluster but vary the other parameters and instantiate multiple cores in the system. The parametrisation is performed by the machine.xml file (Fig. 1) which describes the micro-architecture of the full system. This file can be used to either generate RTL or a model for the simulator. Substantial scripting automation is used throughout both hardware and software compilation flows such as to present the user with a unified, well-orchestrated research environment and abstract away a lot of the details.

B. Software: LLVM-based Driver

This Section elaborates on the LLVM-based OpenCL driver, which is used by the programmer utilising the LE1 as an OpenCL accelerator. As the LE1 is neither a GPGPU, nor multi-threaded (the version utilized in this work), execution of the kernels involves more than just compilation and kernel submission. The driver first transforms the input source into a WG so that WIs are serialised and synchronisation between the threads can be implemented (Figs. 20 and 21 depict the original input OpenCL kernel (WI) and the output C code (WG) respectively with the kernel barriers eliminated). The resulting WG code drives the LE1 compiler (discussed in Section III-B6) after the transformation processes are performed by our framework. The driver consists of the following parts: *front-end*, *source-to-source kernel transformer/compiler* and *runtime*. These will be described in detail in the following Sections, but they can be summarised as follows: the front-end client driver is the layer the programmer uses to interact with the system, while the transformer and compiler modify and translate the code to run upon the LE1; finally, the runtime support includes the means of data transfer, the execution of multiple WGs and the runtime library. This part is implemented on one of the two ARM A9 CPUs of the z7045 device using FreeRTOS as the executive.

1) System Overview:

The software driver provides a layer of abstraction for the LE1 hardware. Its task is to allow communication between them allowing the programmer to control the execution of the OpenCL program but also hiding the device-specific details. For this, the OpenCL 1.1 standard API has been implemented and the fact that the kernel is coarsened to a WG, statically linked and run on the hardware configurations of Table IV or the cycle accurate simulator (remaining configurations), is hidden from the application developer. The proposed software is in the form of the GNU/Linux shared object *(libOpenCL.so)* that can be used just as any other OpenCL driver. The driver is built around Clang/LLVM libraries which are statically linked into the driver; they allow the transformation and compilation of OpenCL kernels at runtime. The driver is composed of three main parts, depicted in Fig. 4:

- the driver front-end, which implements the OpenCL 1.1 API calls, allowing the user to control the rest of the driver,
- the source transformer, which converts (coarsens) the kernel from WI-based to WG-based taking into account barrier synchronisation, variable lifetime etc.,
- the back-end compiler, which links in the developed runtime library and produces the assembly code which executes unmodified on the LE1 CA simulator or the SoC-FPGA.

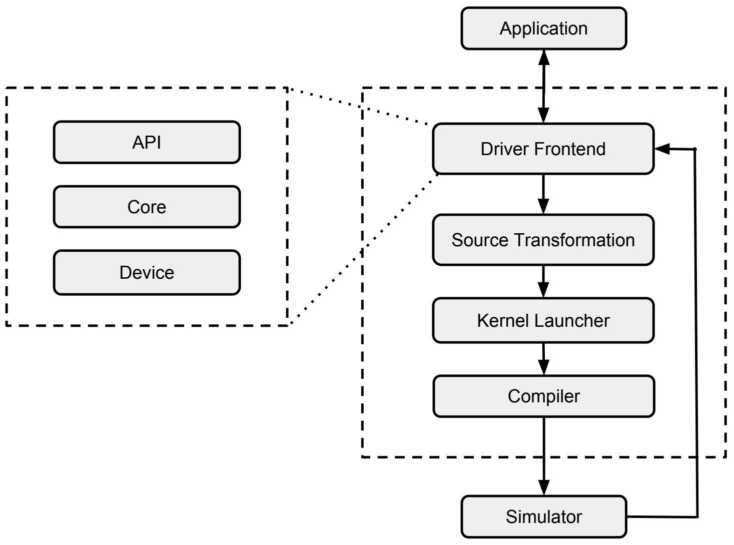

Fig. 4. Software system overview illustrating the various components of the proposed OpenCL framework.

2) Driver Front-end:

The front-end is based on the Clover project[8] which implements the OpenCL 1.1 API and supports the *OpenCL embedded profile* The client driver is split into three layers: the core, the API and the device. The main component of the core is the command queue which is used to transport commands, as well as their respective data structures, between the three layers of the driver. The core also contains the classes for OpenCL objects such as buffers, kernels and programs. The API layer implements the functions defined in the OpenCL standard, using the core to create the program objects and pushing events into the queue. The device layer takes the generic program objects, from the command queue, and specifies them for itself. The driver contains 240 statically instantiated devices (LE1 configurations) which the user can query/select for compilation using the standard API calls. The devices represent all the LE1 architecture variations that have been investigated (10

[8]http://cgit.freedesktop.org/mesa/clover/

hardware and 30 simulated), and which are described in full in the Section IV and Table IV. As well as the compiler target and simulator model, a device has a worker thread which queries its associated queue for events to act upon, such as buffer operations and running of kernels. Once the user calls `clEnqueueNDRangeKernel` all the necessary data is available for the kernel to be transformed.

3) Kernel Source Transformation:

OpenCL allows synchronisation between WIs in a WG and so expects many WIs to be executing concurrently. The LE1 runs a single thread on each core (with a run-to-completion model), utilizes a 1D Algorithm space and to execute multiple threads concurrently would require a software thread manager and many context switches. Instead of executing WIs in parallel, each core executes them sequentially and the code is transformed to explicitly handle the memory synchronisation as depicted in Figs. 20 and 21. The expert reader will notice in Fig. 21 a fixed local size of 64 whereas in reality the local size can change arbitrarily across kernel invocations. In it's first implementation, the driver stored the local size in memory and this value was recovered from that location with the use of the `get_local_size` API call. This was changed subsequently with the local size now being hard-wired in the kernel (as depicted in Fig. 21) which enables many subsequent loop optimizations, not possible with the previous method. To achieve this transformations are performed each time the user calls the `clEnqueueNDRange` API function resulting in the kernel being transformed (and on-line optimized) for the currently-used local size. Such transformed WGs can then be processed concurrently across multiple cores in the system. *The unit of work is enlarged from WI level to WG level through AST source-to-source transformations*, using Clangs libraries. Performing the transformation at a high-level allows the coarsening to only happen once, even in the presence of different multi-core accelerators in the heterogeneous system. The transformation takes place in three phases: A) code expansion and function in-lining, B) basic WG coarsening and C) barrier call and control-flow handling. Function in-lining happens at the source level as barriers can potentially live in them and macros are expanded to be able to successfully rewrite the coarsened source. In the absence of any barrier calls, the kernel body only needs to be enclosed in one or more for loops; one for each required dimension and any return statements are replaced with a `goto` effectively skipping the current WI. The transformation takes place once the programmer has requested the kernel execution, so the local size can be hard coded into the loop declaration in a hope to aid loop transformations (LLVM-driven loop unrolling presents many more opportunities to fill the ILP pipeline of an LE1 core). The kernel initialiser algorithm, shown in Fig. 17 is an implementation of the `RecursiveASTVisitor` class, that is part of Clang.

In the presence of barriers, the regions in the source in which the WIs would execute independently need to be found so that the kernel can be divided between those sections; for this loop fission is used [33]. Wherever there is a barrier, the WG loop is closed before and re-opened after the barrier with the barrier call removed. This guarantees that all the WIs have completed before continuing past the original barrier call as the OpenCL specification requires. If there are barriers located within nested regions, such as a for-loop (Note: not the outermost WG loops inserted by the transformation engine), those region boundaries are also used as fission points. This is necessary since a barrier within a loop would define that all WIs have to complete up to the barrier for the same iteration before any can pass it. Other statements, such as `break` or `continue`, complicate this situation further since they could skip WIs or the whole WG. If these statements exist within a cyclic region that also contains a barrier, the specification mandates that if one WI executes the statement, all of them will for that same iteration. To avoid the situation where some WIs would execute the barrier while others would not, (thus causing a live-lock) `continue` and `break` statements are also used as fission points.

Local variables are created for variables that are live past the chosen fission points and dependency analysis is applied to determine whether a variable is thread-dependent and thus needs to be expanded or not. As the kernel is explored depth-first from the outer thread loop, statements are checked to find whether their definition is ever dependent upon the WI ID - if not, the variable does not ever need to be expanded since the same value is computed for each WI. Thread dependent variables are ones that have a data dependency on either `get_local_id` or `get_global_id`, so any variables that are defined using those calls are added to a list of dependent variables. This list is then used to find further thread-dependent variables by examining if they refer to any members of the list. As the code is explored, and the algorithm enters a region that is executed conditionally upon a thread-dependent variable, any variables defined within that region are also added to the list. For thread-independent variables it would be possible to move them outside of the WG loop but for now it is left to the LLVM loop optimisations to do this. For scalar expanded values, all references of the original variable are visited and rewritten as array accesses using indices of the WG loop(s). The final algorithm is described in Fig. 18 with Figs. 20 and 21 depicting the pre and post-transform code respectively for the *permute* kernel.

4) Kernel Launcher:

The kernel is then linked with a small function which calls instances of the newly created WG. This launcher function uses the CPU (Core) ID and work dimension counters to calculate which WG the unit should be computing.[9] This value is used at the core level to determine the execution space; an intrinsic is used to read the CPUID, which is then used to offset the buffer address for each of the cores. Intrinsics are also used to keep count of the number of WGs completed. The launcher also checks whether the core is even supposed to operate as some data sets will not split over the whole algorithm NDR evenly, meaning that sometimes cores need to exit early and not perform the kernel

[9]The LE1 has an instruction which allows the user to query the CPUID (SYSTEM:CONTEXT:HC tuple), returning the value of the currently executing HC.

operation. An example is given in Fig. 19 which depicts such a situation.

5) Runtime Library:

The OpenCL standard defines certain runtime functions to be available to the programmer, and for this libclc[10] was used as a base. As the current LE1 ISA has no architected FP subset, it has been necessary to augment libclc with routines from compiler-rt[11] and soft-float[12] to enable floating-point emulation in software. This emulation is significantly slower than native operations; a kernel that performs array multiplication on integer data for 256 results takes 4416 cycles, whereas the soft-float calculation takes 32096 cycles. The runtime library is statically linked into the LLVM byte-code just before finally compiling the kernel to an assembly file. As the emulated FP routines are not evaluated and handled until the byte-code reaches the compiler back-end, the linker is unaware that external functions are needed. So, just before linking, the IR is iterated over once more and scanned for FP operations with the necessary functions being declared within the byte-code. Then *llvm-link* is used to create the final byte-code with all the needed FP-emulation functions included from the runtime library to create the final program.

6) Compiler:

An LLVM compiler back-end was developed for the LE1 whose ISA is loosely based on the VEX architecture [34]. LLVM has limited support for VLIW architectures with no region enlarging and scheduling techniques such as Trace scheduling [35]. However, multiple instructions were explicitly grouped together into LIW packets using the `DFAPacketizer` which operates on basic blocks. The flow has taken advantage of the wealth of optimisations and analyses passes included within LLVM and so all kernels are compiled with the `-O3` compilation flag. The back-end supports the generation of assembly code, which is used to drive the existing LE1 tool-chain of Fig. 1. For the SoC-FPGA implementation the tool-chain output results in two binary files, IRAM and DRAM for the kernel image and initialised data section. These are loaded onto the LE1 system from the ARM A9 host via the Debug I/F (CTRL_S_AXI port on the *vthreads_main_axi4_top_0* instance of Fig. 5, DBG_IF on Fig. 6).

The back-end operates on IR from the Clang front-end, which already supports OpenCL 1.1, so was modified with target-specific classes to add support for the LE1. These are the target address spaces, endianness and data sizes as well as handlers for the aforementioned intrinsics such as CPUID. The intrinsics are used to access reserved areas of memory to keep track of the WI execution. The key machine instruction for identifying the execution space is the `cpuid`, accessed using the `le1_read_cpuid` built-in instruction which returns a value from 0... n-1 where n is the

[10]http://libclc.llvm.org

[11]http://compiler-rt.llvm.org

[12]http://www.jhauser.us/arithmetic/SoftFloat.html

number of contexts (cores) in the device. This, combined with the `le1_set_group_id` and `le1_get_group_id` built-ins allows the device to statically iterate through the WGs.

7) Data Transfer:

As there is no integrated assembler within the developed LLVM back-end the existing LE1 tool-chain of Fig. 1 was used to automatically include data in the LLVM-produced assembly file. As well as writing buffer data, there is also a need to scan the final kernel and extract any data included from the runtime code including global data and static function variables which are treated as constant data. The kernel attributes, such as work sizes, are stored at location 0 of the shared DRAM of Fig. 2 with constant data after that and finally the buffers. For local buffers, enough DRAM space is allocated for each CU (LE1 context) and the `cpuid` is used to offset local buffer pointers for each such CU. The stack is used as an extension to the private memory space that is already held within the registers and is allocated per CU, growing from high-memory towards location 0. Data from kernel execution on silicon needs to be read back (via the *MEM* AXI4 Slave Interface of Fig. 5) after each run to update the global host.

IV. Methodology

A. Work-flow

Results for the compilation framework were collected by running the platform on a Xilinx zc706 development board. The 10 LE1 configurations are listed in Table IV and the whole design was implemented in the Vivado framework. The baseline system is based on the Xilinx Built-in-self-test (zc706_bist) design with modification for using PL-based DMA (AXIDMA) and one instance of LE1 (with between 1-8 contexts). The overall Vivado design is depicted in Fig. 5, with enlarged views of the LE1 and DMA hierarchies shown in Figs. 6 and 7 respectively.

Execution results presented in this report have been collected by running the benchmarks on the zc706 evaluation board using FreeRTOS executing on one of the two ARM A9 CPUs. As LLVM can't be compiled on the RTOS, a proxy-server/client architecture was devised where the LLVM calls were packetized (from the ARM9) and sent to an external host (Linux desktop PC) via Gigabit Ethernet. The latter included the full LLVM infrastructure including the framework presented in this paper, and served as the runtime compiler and as a debugging aid. Compiled byte-code was communicated back to the ARM9 via Ethernet where it was loaded onto the LE1 IRAM and DRAM. Very precise execution statistics are extracted from the LE1 after the execution of the kernels via the on-board Instrumentation IP including cycle counts, various stalls and other efficiency metrics.

As some configurations were not implemented on silicon, they were used on our study for reasons of completeness. For these simulated configurations, the final LE1 executable files are also passed to the simulator along with the appropriate machine model (auto-generated from machine.xml) to

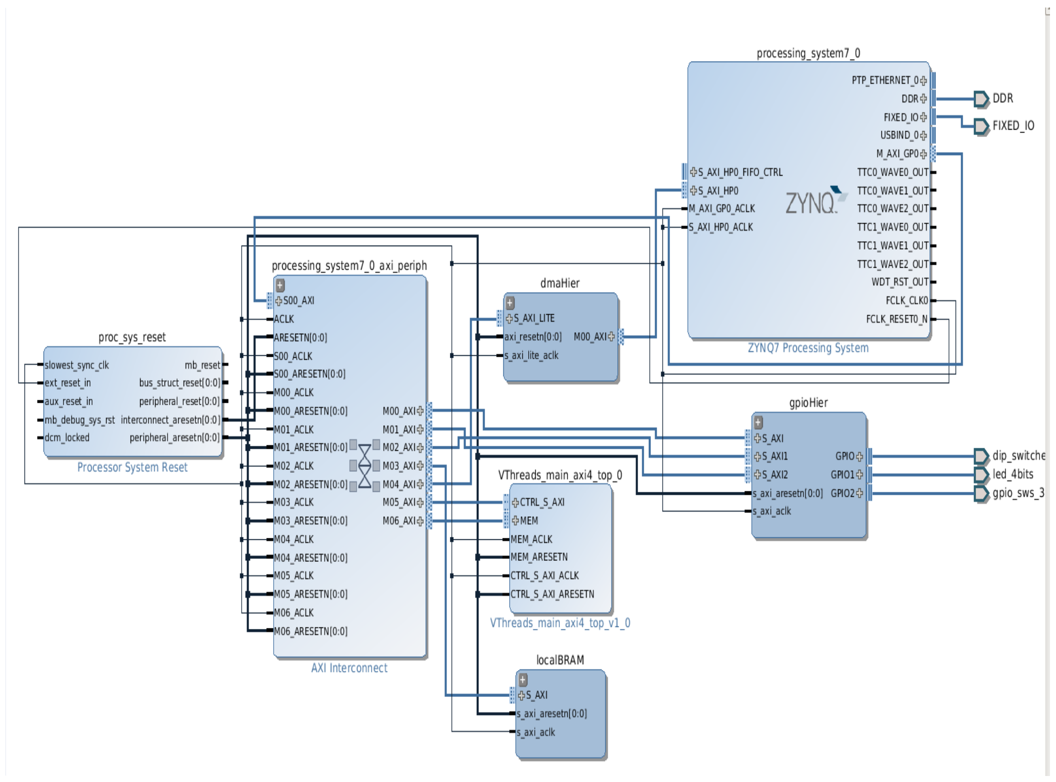

Fig. 5. Overall Vivado design view. The schematic includes the dual ARM A9 SMP system (processing_system_0), the DMA Hierarchy (dmaHier), the LE1 CMP (VThreads_main_axi4_top_0), a local RAM block (localRAM) and I/O peripherals (gpioHier). All components are connected via a single-tier AXI4 interconnect (processing_system7_0_axi_periph).

specify the micro-architecture of the simulated system. In the simulated case the cycle data (total cycles, pipeline stalls, memory stalls and NOPs) are recorded once the application completes and the mean average is taken over the number of iterations that the kernel was run for.

TABLE I
BENCHMARK NAMES AND SOURCES

Source	**Benchmark**
AMD	Binary Search, Bitonic Sort, Floyd Warshall, Fast Walsh Transform, Matrix Transpose, NBody Simulation, Radix Sort, Reduction
Rodinia	Breadth-First Search, Gaussian Elimination, Nearest Neighbour, Needleman-Wunsch

1) Benchmarks:

Part of the remit of this research was to execute realistic benchmarks on the LE1 VLIW CMP using an LLVM-based transformation and compilation flow. As such, a number of benchmarks, listed in Table I, have been selected with Tables II and III depicting the global and local dimension, the number of WGs and iterations each benchmark was run for. The benchmarks include codes from the Rodinia suite [9] and the AMD SDK v2.9. and represent a mix of real-world applications which tests the

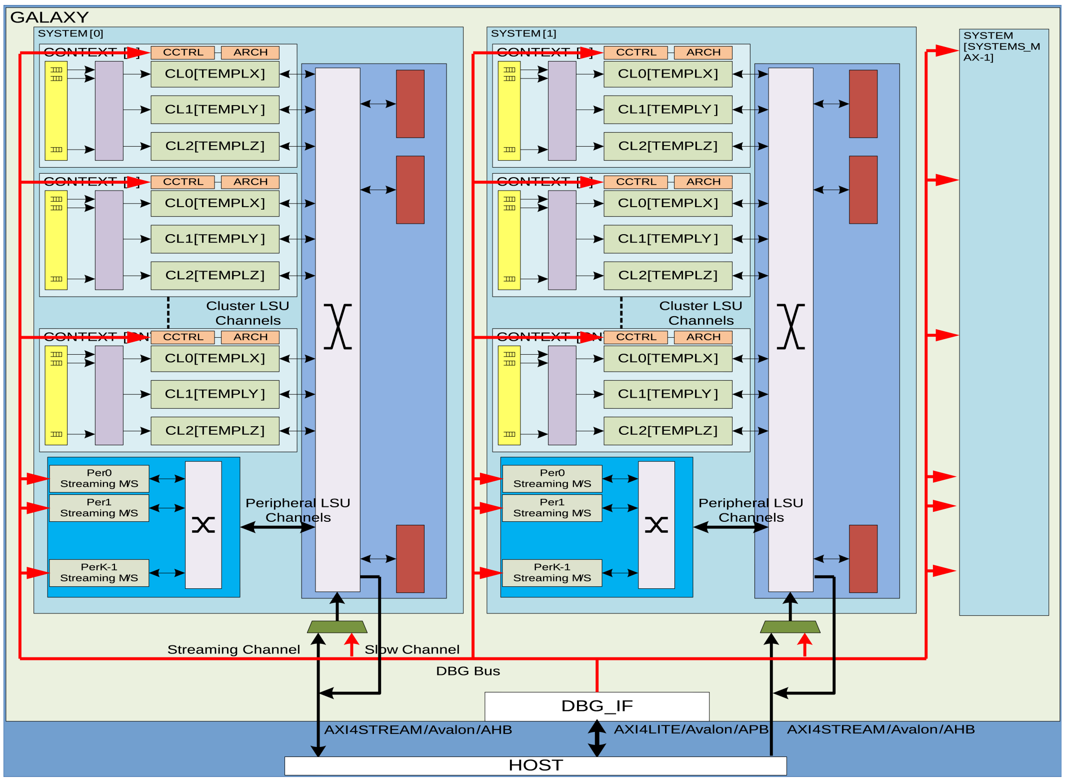

Fig. 6. Enlarged view of LE1 CMP. The GALAXY includes a collection of shared-memory systems (SYSTEM[]) with each system including an array of contexts (cores, CONTEXT[]) tapping into a multi-banked memory system via a pipelined XBar. A single Debug Interface is used to allow a host to control the processor whereas memory-mapped channels are used to insert/extract data (AXI4STREAM).

capabilities of both hardware and software. The selection of kernels includes *complex control-flow, barriers, vector data types*, both integer and *floating-point* based. Several benchmarks are comprised of multiple kernels which are also run for a number of iterations, requiring intra- and inter-kernel data transfer. The deliberate choice of bespoke benchmarks differentiates our work to prior research referenced in Section II.

2) *Machine Configurations:*

Each of the 12 benchmarks were run using the same data across varying LE1 configurations for the simulated OpenCL device. Devices with 1, 2, 4 and 8 CUs were used with varying issue widths (W), integer ALUs (A), integer multipliers (M), load/store units (LSUs) and memory banks (B). The results collected are from the Xilinx Zynq zc706 board with key LE1 micro-architectural parameters extracted from the processor through the Debug I/F and making use of the Instrumentation IP. Note that on the Fmax column of Table IV there are two frequencies; one is the max Frequency achieved by the LE1 block synthesized on it's own and the second (in brackets) is the actual system frequency (PL fabric clock making use of the ARM PLL). *The 2-wide machine configurations were chosen for the silicon implementation as it was generally found that the most substantial ILP gains were made*

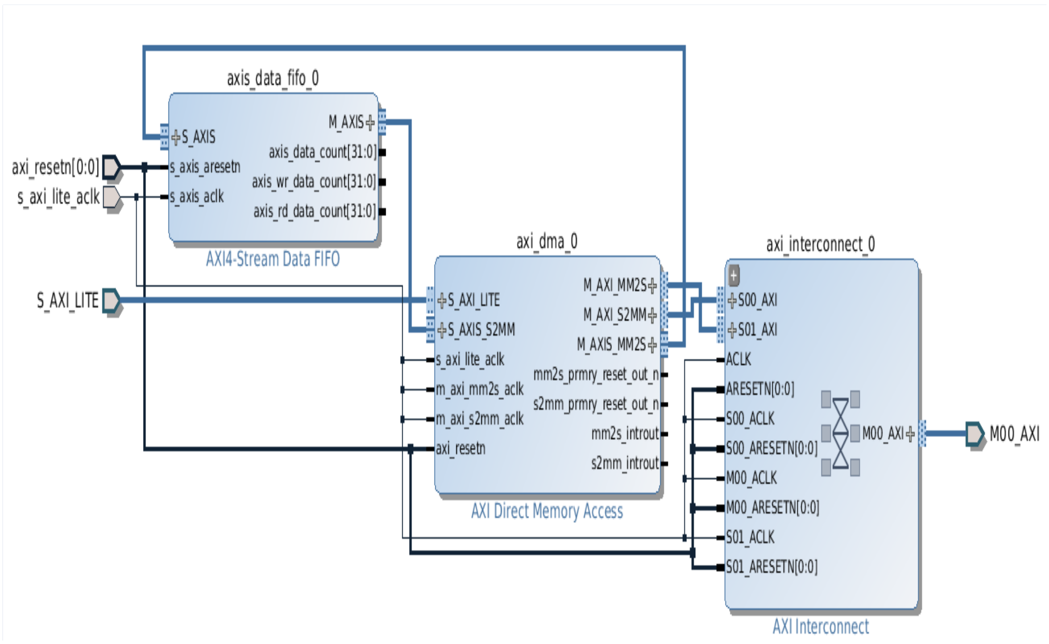

Fig. 7. Enlarged view of DMA hierarchy from Vivado. The axi_dma_0 instance drives (via a local interconnect) a single AXI4 master port used for data movement. The S_AXI_LITE port allows the host to control the DMA core.

TABLE II
KERNEL/BENCHMARKS WORK DIMENSIONS, SIZES AND EXECUTION ITERATIONS FOR AMD BENCHMARKS.

Kernel name	**Global sizes**	**Local sizes**	**Workgroups**	**Iterations**
BinarySearch	131072, -	128, -	1024	1
BitonicSort	16384, 1	256, -	64	120
FastWalshTransform	2048, -	256, -	8	12
FloydWarshall	128, 128	16, 16	64	128
MatrixTranspose	32, 32	16, 16	4	1
NBody	1024, -	256, -	4	1
Reduction	16384, -	256, -	64	7
		Radixsort		
Histogram	16384, -	256, -	256	4
ScanArraydims2	64, 256	64, 1	256	4
ScanArraydims1	256, -	256, -	1	4
Permute	64, -	64, -	1	4
FixOffset	64, 256	variable, variable	variable	4

in moving from a scalar configuration to a 2-wide. The Zynq was chosen as the target as it represents a low-power, cost effective, option for reconfigurable computing and includes a high-performance host (ARM A9 SMP system). The remaining configurations (4-wide) were studied on our highly-accurate simulator as the FPGA resources required were substantially higher compared to the 2-wide. As the LE1 is a statically-scheduled machine, has no cache hierarchy and it's memory subsystem is

TABLE III
KERNEL/BENCHMARKS WORK DIMENSIONS, SIZES AND EXECUTION ITERATIONS FOR RODINIA BENCHMARKS

Kernel Name	Global sizes	Local sizes	Workgroups	Iterations
Breadth-First Search				
BFS 1	4096, 1	256, 1	16	8
BFS 2	4096, 1	256, 1	16	8
Gaussian Elimination				
Fan1	16, 16	variable	variable	15
Fan2	16, -	variable	variable	15
Needleman Wunsch				
nw kernel1	variable (16-256), -	16, -	variable (1-16)	16
nw kernel2	variable (16-256), -	16, -	variable (1-16)	15
NN	42816, -	variable, -	variable	1

pipelined, banked with round-robin bank arbitration, it was modelled precisely on Insizzle. This is the key point which makes the simulator a highly-trusted source of information for the configurations that couldn't be synthesized due to FPGA capacity issues.

TABLE IV
FPGA RESOURCE UTILIZATION AND MAXIMUM CLOCK FREQUENCIES FOR A 2W-2A-1M-1L MICRO-ARCHITECTURE CONFIGURATION, WITH A 16KB IRAM AND 256 KB DRAM. RESULTS ARE FOR THE PLACED-AND-ROUTED LE1 DESIGN AND INDICATE (FMAX) THE RELEVANT Z7045 PLL USED TO ACHIEVE THE REQUIRED CLOCK.

CUs	Mem Banks	Slices	Slice Regs	Slice LUTs	RAMB36	DSP48	Fmax (MHz)
1	1	2753	2584	9070	32	3	159.2 (DDRPLL: 152.4)
2	1	5541	4956	18029	32	6	133.8 (ARMPLL: 133.3)
	2	5015	4932	16490	40	6	137.4 (DDRPLL: 133.3)
4	1	10311	9687	34160	32	12	121 (ARMPLL: 121.2)
	2	9751	964	32445	48	12	121.1 (DDRPLL:121.1)
	4	9979	9664	33646	48	12	119.5 (DDRPLL: 119.5)
8	1	20412	19173	68439	32	24	114.6 (IOPLL: 111.1)
	2	18926	19069	64934	64	24	119.0 (DDRPLL: 118.5)
	4	19834	19076	66674	64	24	113.9 (ARMPLL: 111.1)
	8	20550	19126	69054	64	24	92.7 (IOPLL: 90.9)

V. RESULTS

This Section presents the execution of the 12 OpenCL benchmarks on the LE1 hardware (Table IV configurations) and where a synthesized configuration was not available, from the Cycle-Accurate simulator. It is split into three sub-sections: A) Study of the benchmarks ILP by evaluating the performance increase as a function of the architecture width, functional unit mix and memory system configuration, compared to a scalar baseline architecture; B) Study of the multi-core system performance and the scalability of the solution; C) Study of application throughput on the silicon platform in terms of iterations/sec. The ILP investigation was performed using a single CU device with 31 different micro-architecture configurations: varying issue width (W), number of integer ALUs (A), number of integer multipliers (M), number of LSUs (L) and memory banks (B).

In the ILP study (sub-section V-A) we omit results for the benchmarks that did not utilise the full configurations, since performance data are essentially the same as the smaller devices. The results are here to depict the ILP scalability of our solution and the effectiveness of our experimental LLVM LE1 back-end, not as a complete performance metric based upon program completion time. Because of this, we have taken an average cycle count for benchmarks that run kernels for several iterations instead of presenting a total. However, in the TLP study (sub-section V-B) we include the execution time of the silicon (Tables VI and VII) as these are of interest to the reader. Finally, tables VIII and IX depict the actual benchmark throughput (iterations/sec). A final point to note is that the cycle data from the configurations executed solely on the simulator takes into consideration the number of clock cycles taken to transfer the data between the host and the device and vice-versa by assuming a 32-bit wide transfer channel (the LE1 hardware includes a configurable DMA engine based on the AXI DMA IP core as shown in Fig. 7) for maximum conformance to the hardware implementations.

A. ILP

For less computationally intensive kernels, such as *BFS_1, BFS_2* and *BinarySearch*, little performance is gained through ILP; as shown in Fig. 8. These kernels do not perform much computation relative to the amount of control-flow within them; none of them gain more than 1.08x speed-up from ILP. *BinarySearch* even suffers performance degradations in the 2-wide configurations due to an increase in the IF stalls. [13]

For the kernels of breadth-first search, the largest configurations only achieve 4.46% speed-up for *BFS_1* and 7.95% for *BFS_2* and this is only ∼1% more than the 2-wide configurations with single FUs.

Compiler optimisations and the partially predicted ISA allow the removal of control-flow statements within the small kernel of *Bitonic Sort*. This enables a small amount of ILP to be exploited in the 2-wide devices with singular FUs, with ∼2% performance increase. This is a modest increase due to the increase in NOPs, showing that there is not enough ILP to mask the pipeline latencies. An increase in IF stalls in the 2-wide devices with two LSUs cause these devices to perform little over 1% faster than the scalar devices, while the two machines with two ALUs but singular LSUs execute 10% faster. In the 4-wide configurations, the IF stalls decrease again enabling these configurations to perform better than the 2-wide machines. The increase in ALUs and MULs improves performance by

[13]These are stalls due to the instruction Front-end of the processor not producing a full LIW for execution by the LE1 back-end pipeline. These stalls are documented in our previous work [31] and are mostly eliminated when choosing a decoupled instruction front-end for the LE1, as this is a valid configuration option in a second generation micro-architecture. We have chosen to the demonstrate the performance of the LE1 with no decoupling as this is the worst-possible case. A detailed description of an improved LE1 micro-architecture which fully alleviates this issue has been submitted [17] and is briefly discussed in Section VII. We also note that simple re-ordering of the produced binary by our back-end can eliminate practically all these stalls on the existing LE1 however this hasn't been included in the current OpenCL framework.

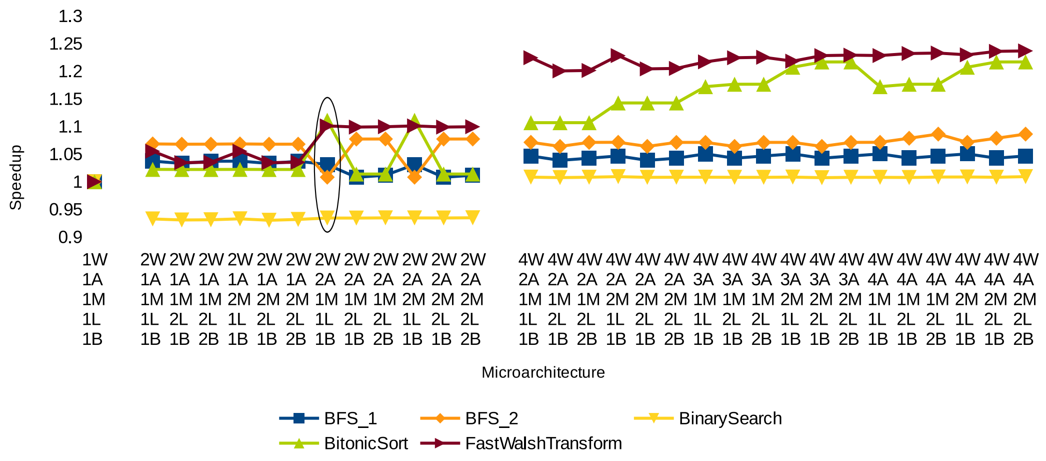

Fig. 8. Single core performance gains, via ILP, for Breadth-First Search, Binary Search, Bitonic Sort and Fast Walsh Transform benchmarks. Results from the zc706 FPGA implementation are highlighted

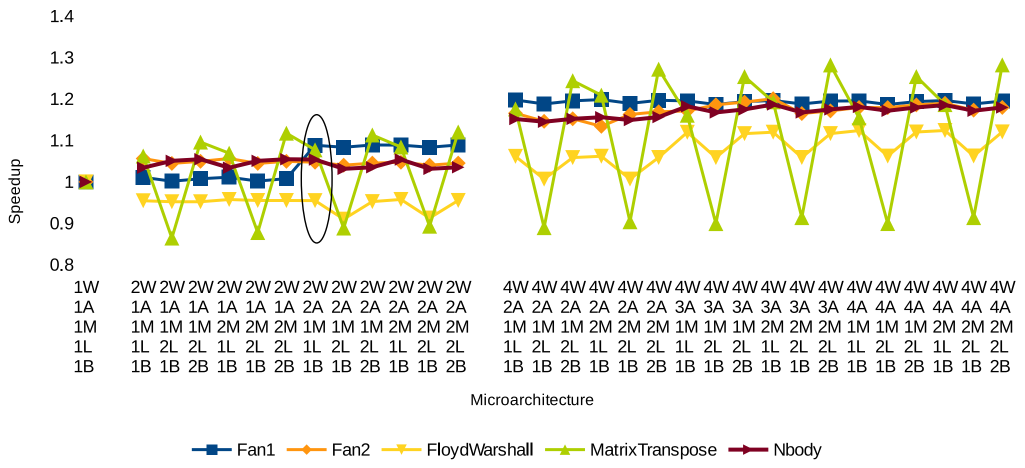

Fig. 9. Single core performance gains, via ILP, for Gaussian Elimination, Floyd-Warshall, Matrix Transpose and NBody Simulation benchmarks. Results from the zc706 FPGA implementation are highlighted

2-3% for each added unit but performance is capped at three ALUs and two MULs. The results of *FastWalshTransform* show that this kernel responds positively to increased issue widths and ALUs but is largely unaffected by other configuration variables; the largest configuration achieving a 19.14% reduction in cycles. The simpler 4-wide devices, each with one MUL, LSU and DRAM bank but with two and three ALUs, achieve 18.33% and 17.83% improvements respectively

Fig. 9 shows the results of the two kernels from Gaussian Elimination (*Fan1* and *Fan2*), *Floyd-Warshall*, *MatrixTranspose* and *NBody*. The micro-architecture only seems to affect the performance of *Fan1* in two changes to the configuration: A) increasing the number of ALUs to two in the 2-wide configuration and B) where the issue width increases from two to four with both enhancements to the architecture yielding the same performance increase of $\sim$8%. Again, as with other kernels, the

2-wide devices with two ALUs suffer an increase in IF stalls, the reduction in these stalls leads to the improvement seen in the 4-wide devices. The response of *Fan2* shows that the same improvement is observed when doubling the issue width from two to four and the small spikes are where there is a mismatch in the number of LSUs and DRAM banks. The 4-wide devices achieve ∼16% for *Fan1* whereas the same configurations vary between ∼12-16% for *Fan2* as performance improves up to three ALUs and IF stalls are more varied. The varying performance of the *FloydWarshall* benchmark is closely related to both the IF and memory stalls of the system, while the rest of the micro-architecture details have very little effect.[14] The increase in IF stalls for the 2-wide configurations, with a single ALU, are reflected in the total cycles which means that any ILP exploited is not enough to counter their effect. However the next increase, when using two ALUs, is not reflected in the total output which means that enough ILP is discovered to counter the detrimental effect, but still all the 2-wide devices perform worse than the scalar device by an average of 6.1%. The decreased IF stalls throughout the 4-wide configurations enable them to execute faster than the scalar, but only by ∼6-11% for the devices with just one LSU.

The response of *matrix transpose* is very volatile and highly dependent on the memory configuration, with the greatest dependence of all the benchmarks used. Results improve for each configuration where the number of LSUs are increased, along with the number of DRAM banks. Each of the predominant peaks represents where the number of banks does not match the number of LSUs and these configurations perform significantly worse than the scalar device by ∼9-16%. The IF stalls are also volatile: the reduction in IF stalls for the 4-wide machines occurs in this benchmark, but there are also general increases for the larger configurations which peak when two LSUs are combined with two DRAM banks. For *NBody*, the 2-wide devices show small improvements over the scalar machine with execution cycles decreased by ∼3-5%. The 2-wide devices with two ALUs show increases in both NOPs and IF stalls compared to the other 2-wide configurations, yet performance remains about the same; this suggests that enough ILP is discovered to counter both the NOPs and IF stalls. The number of NOPs remains relatively constant for the 4-wide machines, but the decrease in IF stalls enable these configurations to perform ∼10% better than their 2-wide counterparts.

Fig. 10 shows the single core results of *NearestNeighbour (NN)*, the two kernels from Needleman-Wunsch (*nw_kernel*) and *Reduction*. For *NearestNeighbour*, the 4-wide devices do not suffer the IF stalls observed in the 2-wide devices and so execute the fastest. Doubling the issue width, while maintaining the minimal mix of FUs, results in performance gains of ∼1.05x and ∼1.1x with 4-wide devices with three ALUs achieve ∼1.15x speed-up. The response of the Needleman-Wunsch kernels is quite flat in the 2-wide devices, but all of those configurations execute slower than the scalar device,

[14]The memory stalls are due to the number of LSUs being greater than the number of memory banks.

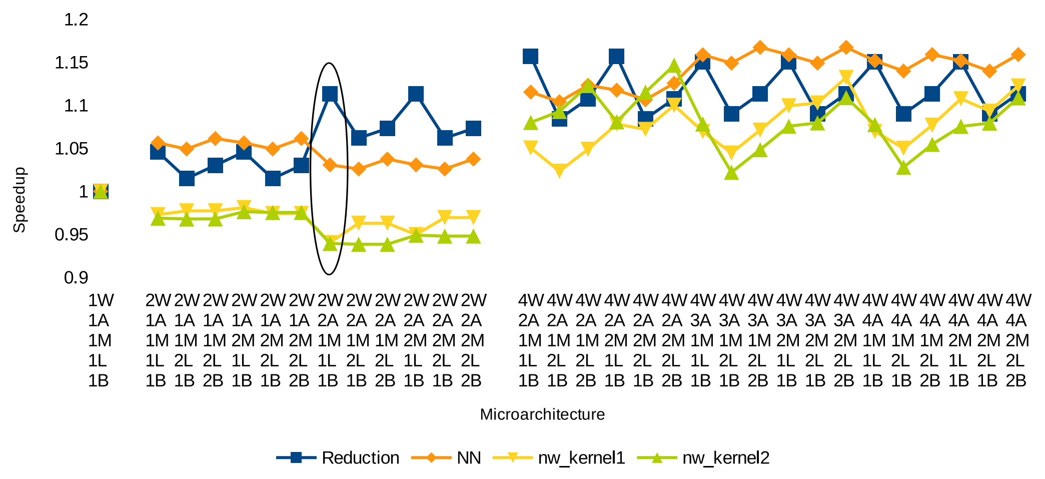

Fig. 10. Single core performance gains, via ILP, for Reduction, Nearest Neighbour and Needleman-Wunsch benchmarks. Results from the zc706 FPGA implementation are highlighted

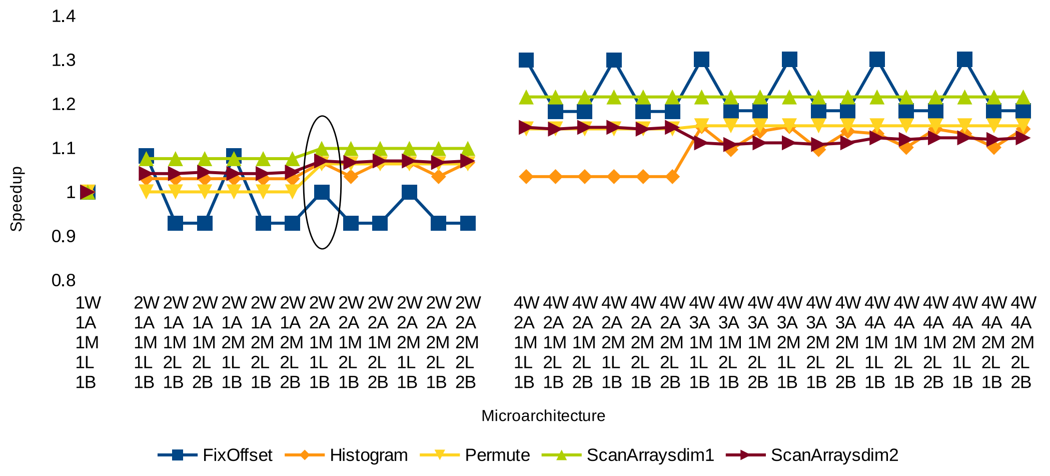

Fig. 11. Single core performance gains, via ILP, for the kernels that comprise the Radix Sort benchmark. Results from the zc706 FPGA implementation are highlighted

again because of IF stalls. The performance of the 4-wide machines varies with differences in the IF and memory access stalls, the results showing the best performance is with a matching number of memory banks and LSUs and two MULs. The results of Reduction show that it is largely variant to the micro-architecture configuration; this is due to memory and IF stalls. All the configurations that contain two LSUs suffer from significant memory stalls, even when there is a bank to support each LSU, and the stalls are higher in the 4-wide than the 2-wide devices. These stalls lead the performance of 4-wide devices to vary by $\sim$5-6%. In the 2-wide machines, the sharp reduction in IF stalls when using two ALUs leads to a performance increase.

The single core results of the kernels of *RadixSort* are depicted in Fig. 11, most of which are invariant to the varying micro-architectures. The variation in the performance of *FixOffset* is mainly

TABLE V
AVERAGE TOTAL CYCLES FOR KERNELS USING MULTI-CORE LE1 SYSTEMS USING THE 2W-2A-1M-1L MICRO-ARCHITECTURE AND VARYING (1/2/4/8) DRAM BANKS.

	2 CUs		**4 CUs**			**8 CUs**			
Kernel	**1B**	**2B**	**1B**	**2B**	**4B**	**1B**	**2B**	**4B**	**8B**
BFS_1	72456	69993	38161	36207	35721	21127	19121	18567	18303
BFS_2	39947	39916	20470	20192	20069	10794	10407	10250	10198
BinarySearch	7406	7411	3837	3825	3819	2379	2130	2122	2120
BitonicSort	259424	259420	132382	129729	129735	79498	67278	65015	65061
Fan1	2287	2277	1403	1380	1371	889	822	796	793
Fan2	18341	18326	10822	10742	10719	6994	6608	6491	6454
FastWalshTransform	365465	365377	183361	183188	183113	92533	92283	92205	92205
FloydWarshall	178734	176822	95374	90978	89401	56909	50310	47179	45618
MatrixTranspose	108293	91653	85443	57738	49065				
NBody	1629721K	1628620.2K	823244.3K	821336910	820594.4K				
NearestNeighbour	22868535	22859356	11471844	11451101	11440777	5777719	5745952	5730387	5723111
nw_kernel1	106029	100447	61006	59233	58769	40169	37081	35925	35485
nw_kernel2	97513	97127	57098	55405	54878	38144	35313	34165	33736
Reduction	1673049	1635090	929307	865574	835512	645214	488854	440184	419031
FixOffset	100483	100475	62706	59134	59129	49514	28103	21811	18295
Histogram	449799	449803	250631	230266	226426	165398	133974	121448	114814
ScanArraydims2	2512945	2493472	1405176	1335667	1287828	904315	744568	678261	649099

dominated by the effect of the memory stalls that occur whenever two LSUs are used. For the 2-wide devices with one ALU, these memory stalls also coincide with an increase in IF stalls resulting in only two of the 2-wide machines performing better than the scalar device.

B. Scalability

This Section presents the speed-ups achieved using a 2-wide LE1 configuration instantiating 2, 4, and 8 CUs with varying DRAM banks and comparing against a single, 2-wide CU. The cycle data from the simulator, Table V, shows that performance scales linearly, up to 4 CUs, for most of the kernels without the memory system having a significant impact on performance. This is particularly true for the benchmarks that use FP emulation (*FastWalshTransform, NBody* and *NearestNeighbour*) since it greatly increases the computational complexity. The configuration of the memory subsystem has a negligible effect on the performance of these kernels. The general linear speed-up across the benchmarks suggests that the static scheduling of WGs across the cores does not hinder the scalability.

TABLE VI
KERNEL EXECUTION TIME (μS) FOR THE ZC706 IMPLEMENTED CONFIGURATIONS (SET 1).

LE1 Config	BFS_1	BFS_2	B.Search	B.Sort	Fan1	Fan2	FastWalsh	FloydWarshall
2C/2W/2A/1M/1L/1B	526.8	300.9	55.4	1938.9	17.1	137.1	2717.3	1335.8
2C/2W/2A/1M/1L/2B	509.4	292.5	53.9	1888.1	16.6	133.4	2645.5	1286.9
4C/2W/2A/1M/1L/1B	306.4	169.6	31.7	1092.3	11.6	89.3	1505.4	786.9
4C/2W/2A/1M/1L/2B	299.0	167.8	31.6	1071.3	11.4	88.7	1505.1	751.3
4C/2W/2A/1M/1L/4B	298.9	168.9	32.0	1085.6	11.5	89.7	1524.6	748.1
8C/2W/2A/1M/1L/1B	186.5	97.2	21.4	715.6	8.0	63.0	830.8	512.2
8C/2W/2A/1M/1L/2B	161.4	87.8	18.0	567.7	6.9	55.8	777.2	424.6
8C/2W/2A/1M/1L/4B	167.1	92.3	19.1	585.2	7.2	58.4	828.2	424.7
8C/2W/2A/1M/1L/8B	201.4	112.2	23.3	715.7	8.7	71.0	1012.0	501.8

TABLE VII
KERNEL EXECUTION TIME (μS) FOR THE ZC706 IMPLEMENTED CONFIGURATIONS (SET 2).

LE1 Config	NBody	NN	nw_1	nw_2	Reduct.	FixOffs	Histo	ScnArdim2
2C/2W/1A/1M/1L/1B	12180276.3	170915.8	750.8	698.1	12504.1	751.0	3361.7	18781.4
2C/2W/1A/1M/1L/2B	11853131.1	166370.9	731.1	679.8	11900.2	731.3	3273.7	18147.5
4C/2W/1A/1M/1L/1B	6792444.9	94652.2	484.4	452.9	7667.5	517.4	2067.9	11593.9
4C/2W/1A/1M/1L/2B	6782303.1	94559.1	468.2	440.0	7147.6	488.3	1901.5	11029.5
4C/2W/1A/1M/1L/4B	6866898.6	95738.7	470.1	442.4	6966.6	494.8	1894.8	10776.8
8C/2W/1A/1M/1L/1B	0.0	0.0	351.2	334.8	5807.5	445.7	1488.7	8139.6
8C/2W/1A/1M/1L/2B	0.0	0.0	300.3	287.5	4125.4	237.2	1130.6	6283.3
8C/2W/1A/1M/1L/4B	0.0	0.0	309.2	296.4	3962.1	196.3	1093.1	6105.0
8C/2W/1A/1M/1L/8B	0.0	0.0	373.0	357.7	4609.8	201.3	1263.1	7140.8

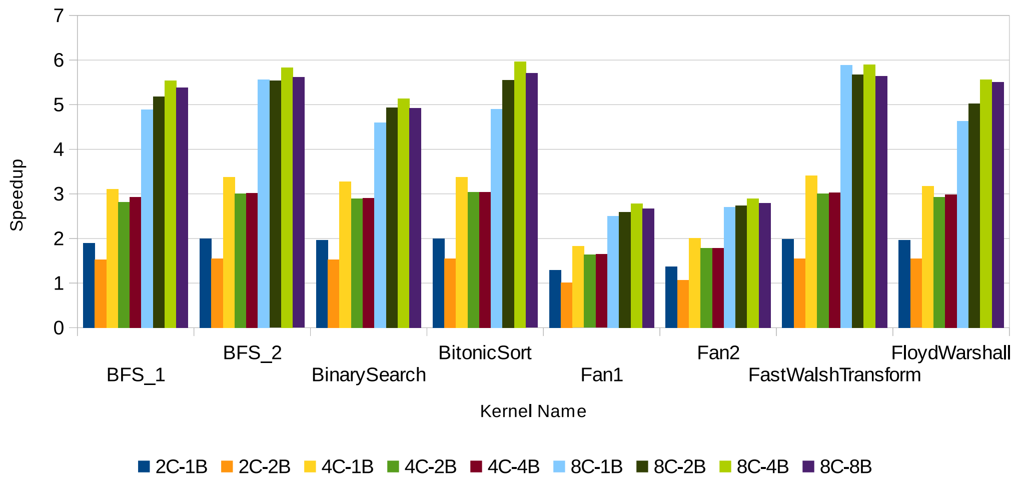

Fig. 12. Speed-up of kernels, relative to a 1CU-2W-1A-1M-1L device, across multi-core LE1 configurations. Results from the zc706 FPGA board

There are exceptions to the general scalability of our solution. Neither the kernels from Gaussian elimination (*Fan1* and *Fan2*) or Needleman-Wunsch (*nw_kernel*) scale very effectively due to the varying number of work-groups that are executed during each iteration. Although the Matrix Transpose kernel only scales to four work-groups, the 4 CU devices already exhibit a 15% difference between the systems. The multi-CU performance of the kernels of *Radix Sort* also varies with the memory configuration. The super-linear performance of the *FixOffset* kernel shows the 8 CU devices having a strong dependence on the memory subsystem. This performance gains in those devices are attributed to a super-linear reduction in memory stalls, which suggests that dividing the algorithm across eight cores and eight banks permits a very effective memory access pattern. The performance of the 8 CU devices varies by $\sim$35% and achieves an 11x speed-up over the single core machine. The performance of *Histogram* and *ScanArraydims2* in the 8 CU devices varies by $\sim$13% and $\sim$14% respectively.

Figs. 12 and 13 show the speed-up of all the executed OpenCL benchmarks on the zc706 platform for 1, 2, 4 and 8 CUs, 2W-2A-varying_B-1L LE1 configurations, collected from the LE1 instrumentation peripheral (silicon) and using the global timer of the Zynq ARM A9 PS under FreeRTOS. It

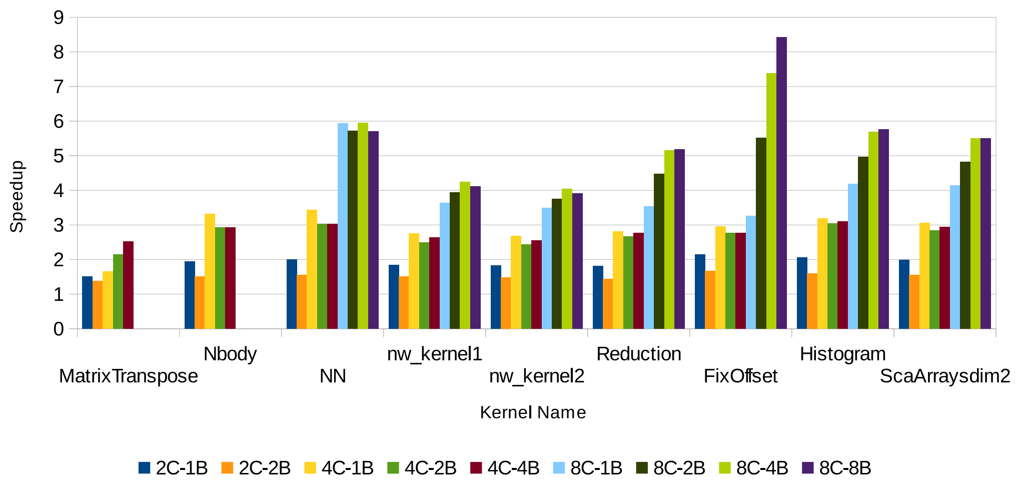

Fig. 13. Speed-up of kernels, relative to a 1CU-2W-1A-1M-1L device, across multi-core LE1 configurations. Results from the zc706 FPGA board

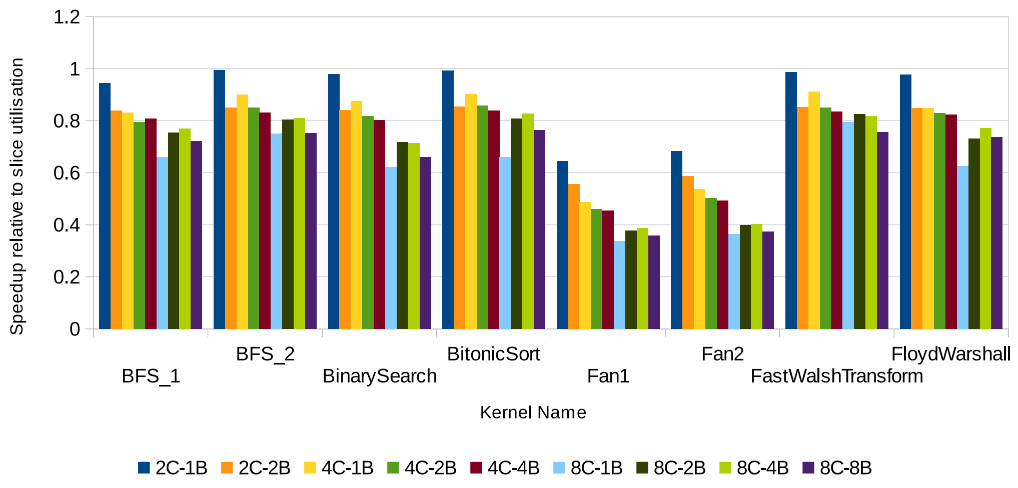

Fig. 14. Speed-up relative to slice utilisation of multi-core LE1 configurations.

is apparent that performance gains level-off beyond 8 CUs as the number of memory banks brings the achievable clock frequencies down (Table IV). The clock frequency reduction for 4 and 8 banks on the 8 CU devices results in those configurations rarely outperforming the same configuration but with just 2 banks. This particular point has demonstrated a key area where the LE1 CMP can be improved and that is the level of pipelining of the Core/Memory Xbar.[15] Finally, Tables VI and VII depict the real (wall-time) of the zc706-implemented benchmarks in μs.

The silicon implementation data is further used in Figs. 14 and 15 to depict the performance gains

[15]A design decision in the LE1 was to keep the Load/Use latencies to an absolute minimum in order to achieve maximum single-thread performance without the need to insert NOPs in the static instruction schedule. This kept the op_bpass/addr_calc stage adjacent to the Memory XBar which results in the Fmax reduction witnessed in this study. This is further compounded by targeting FPGAs which are less forgiving on mux-heavy designs such as a VLIW CMP. This key observation is elaborated in the companion paper [17] and will be mitigated in the next generation of the silicon using a further HDL parameter.

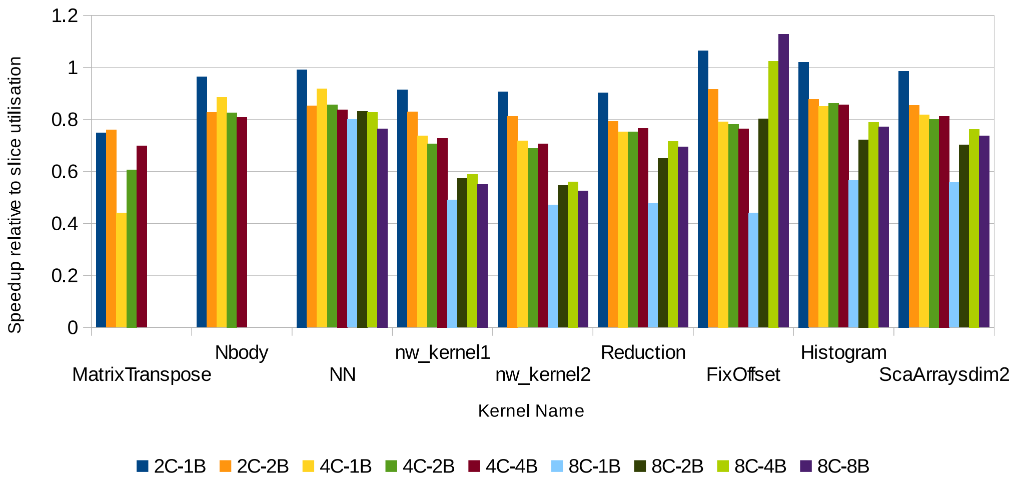

Fig. 15. Speed-up relative to slice utilisation of multi-core LE1 configurations.

as a ratio of the FPGA *slice utilisation*; with the single context configuration requiring 5% of the slices, whereas the 8C-8B requires 37.6%. The slice utilisation increases linearly with the increase in CUs, but again the reduction in Fmax means that the performance to area ratio is reduced significantly for the larger systems. The ratio of the 4C-4B and 8C-4B are very similar to one another, however the 8C-4B consistently outperforms the 4C-4B device, which suggests that overall the 8C-4B would be the best device for this selection of benchmarks when considering overall speed and size.

C. Application Throughput

This final Section considers the overall application throughput making use of the real-time kernel execution in Tables VI (set 1) and VII (set 2) and the actual number of WGs generated per application as shown on Tables II (AMD benchmarks) and III (Rodinia benchmarks). The throughput is calculated taking into account the actual kernel execution time, the number of iterations, the size of the Instruction and Data RAMs (and the time it takes for these to be DMA'ed-into the processor). The DMA transfers take place from the *localBRAM* block, driven by the *dmaHIer* block, through the AXI4 Matrix (*processing_system7_0_axi4_periph*) and into the AXI4MM port of the *vthreads_main_axi4_top_0* block. All blocks are depicted in the Vivado schematic of Fig. 5. Tables VIII and IX depict the AMD and Rodinia applications throughput in terms of iterations/s.

VI. Conclusions

It is clear from the results, that benchmarks generally fall into two categories: memory-bound and compute-bound. For the highly parallel benchmarks, the reduction in memory stalls achieved from increasing the number of memory banks, has to be very significant to account for the decrease in clock frequency. Only four kernels gained a greater speed-up when increasing the number of memory banks

TABLE VIII
AMD BENCHMARKS THROUGHPUT (ITERATIONS/SEC)

Config	BinarySearch	BitonicSort	FastWalshTransform	FloydWarshall	Matrix_Transpose	Nbody	RadixSort
2C/1B	251.58	3.81	30.27	4.94	1235.54	0.08	4.66
2C/2B	258.35	3.92	31.09	5.12	1499.13	0.08	4.84
4C/1B	229.43	6.11	53.94	7.38	1418.49	0.15	6.65
4C/2B	229.24	6.20	53.95	7.63	2097.41	0.15	6.95
4C/4B	226.22	6.12	53.26	7.63	2435.54	0.15	7.04
8C/1B	210.89	8.23	95.36	9.66			7.93
8C/2B	225.04	9.85	101.92	11.12			9.82
8C/4B	210.99	9.45	95.64	10.83			9.75
8C/8B	172.63	7.73	78.27	9.04			8.20

TABLE IX
RODINIA BENCHMARKS THROUGHPUT

Config	Breadth-search-first	Gaussian Elimination	Needlman Wunsch	NN
2C/1B	108.53	366.10	18.71	5.82
2C/2B	150.06	376.34	19.21	5.98
4C/1B	190.24	489.62	20.41	10.45
4C/2B	192.38	492.28	20.59	10.46
4C/4B	191.05	486.73	20.36	10.33
8C/1B	223.10	631.60	20.47	
8C/2B	244.38	700.74	22.26	
8C/4B	231.30	665.42	21.01	
8C/8B	190.09	546.40	17.23	

beyond 2 for the 8 CU device; these were reduction and three kernels from Radix Sort: *FixOffset, histogram* and *ScanArraydims2* and all systems with 8 banks performed slower.

Another hindrance is that insufficient ILP was exploited. The explicit nature of the OpenCL standard should mean that there is substantial parallelism to be harnessed, where we can convert explicit data level parallelism into ILP as well as having multiple threads combined into a single stream of instructions. The researched LLVM-based LE1 back-end can currently only schedule at the basic block level, and although threads (WIs) are being combined, the compiler is not aware that the resulting work-group loops are parallel. So as more loops are created in the kernel code to handle loop fission, more looping overhead is introduced causing the core to spend a lot of time stalling. Note that the current LE1 configuration includes a static branch predictor (PREDICT_NOT_TAKEN policy) which is being changed to a 2-bit saturating counter predictor for the next processor release. Finally, for all kernels, the mean average of total stalls across all single CU devices was evaluated. This resulted that 49.36% of all cycles were stalls clearly demonstrating substantial room for improvement which is addressed in a second generation micro-architecture (Section VII).

VII. FUTURE WORK

A number of improvements in both the software tool-chain and the hardware architecture were identified as a result of this study. In terms of the hardware, the removal of the artificial limitation of 256KB per shared-memory system, the decoupling of the instruction front end, the increase in the

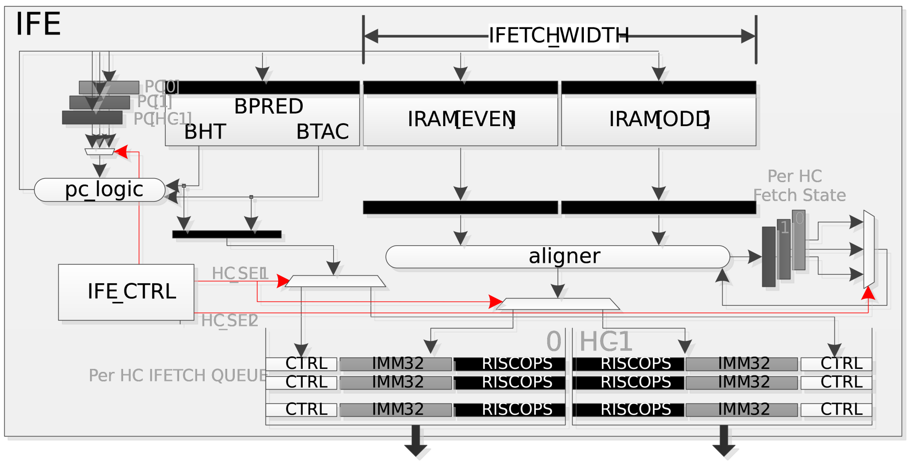

Fig. 16. Improved Instruction Fetch Engine

pipeline depth between address computation and Memory XBar stage and the use of a simple dynamic branch prediction scheme will eliminate practically all the stalls observed while pushing the Fmax to >200 MHz on the z7045 device resulting in substantial performance gains. These improvements have been implemented in a next-generation micro-architecture [17] and the new Instruction Fetch Engine (IFE) which fully eliminates the issue is depicted in Fig. 16. Unlike the single-stage buffers of the current IFE the new micro-architecture provides a configurable number of buffering across multiple HCs and fetches instructions ahead of their use, under the control of the branch predictor.

On the software side, further unrolling of the work-group loops will lead to a reduction in the number of branches and to an increase on the levels of ILP available to the compiler. Extended-Basic-Block and/or Super-block scheduling will then allow for the creation of wider static schedules thus allowing the use of 3 or 4-wide machines. It is also thought that the introduction of another pass that can analyse the computational requirements of the kernel and automatically choose a suitable machine configuration to maximise performance with the minimal amount of hardware will lead to better automation of the proposed HW/SW solution. It is noted that a final pass in the LLVM LE1 back-end can eliminate most stalls of the existing micro-architecture by ensuring that LIW packets don't span IRAM blocks. Finally, the use of a full ARM9 SMP Linux distribution with the developed compiler and LLVM cross-compiled for the ARM will eliminate the external host system and produce a stand-along integrated solution, further improving on our hybrid, PC Linux host/FreeRTOS runtime solution.

Samuel J. Parker was born in Suffolk, UK in 1988. He received a M.Eng in Electronic and Software Engineering from Loughborough University in 2011. He continued at Loughborough University as a research student and is currently a Ph.D candidate. His research interests include compilers for parallel computer architectures, heterogeneous computing, parallel languages and automated software systems.

Vassilios A. Chouliaras was born in Athens, Greece in 1969. He received a B.Sc. in Physics and Laser Science from Heriot-Watt University, Edinburgh in 1993 an M.Sc. in VLSI Systems Engineering from UMIST in 1995 and a Ph.D. from Loughborough University in 2005. He worked as an ASIC design engineer for a Telecoms OEM SA and as a senior R&D engineer/processor architect for a configurable, extensible embedded processor vendor. Currently, he is a senior lecturer in microelectronics in the Department of Electronic and Electrical Engineering at the University of Loughborough, UK where he's leading the research in CPU architecture and micro-architecture, SoC modelling and software parallelization. His research interests include CPU micro-architecture, high-performance embedded CPU implementations, OpenCL computing, custom instruction set design and Electronic System Level (ESL) design methodologies. He is the architect of the LE1 CMP and a founder of Axilica Ltd.

REFERENCES

[1] B. T. R. W. Sandeep Dutta, Vidya Rajagopolan, "Xilinx zynq embedded processing platform," ser. Hot Chips, no. 13. Stanford University, Aug.17-19 2013.

[2] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, pp. 473–491, April 2011. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5737854

[3] Altera Corp, "Implementing fpga design with the opencl standard," Tech. Rep., Nov. 2012.

[4] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*, June 2011.

[5] P. S. Paolucci, A. Biagioni, L. G. Murillo, F. Rousseau, L. Schor, L. Tosoratto, I. Bacivarov, R. L. Buecs, C. Deschamps, A. El-Antably, R. Ammendola, N. Fournel, O. Frezza, R. Leupers, F. L. Cicero, A. Lonardo, M. Martinelli, E. Pastorelli, D. Rai, D. Rossetti, F. Simula, L. Thiele, P. Vicini, and J. H. Weinstock, "Dynamic many-process applications on many-tile embedded systems and {HPC} clusters: The {EURETILE} programming environment and execution platforms," *Journal of Systems Architecture*, pp. –, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762115001423

[6] Khronos Group, *The OpenCL Specification*, Sep. 2010.

[7] D. Stevens and V. Chouliaras, "Le1: A parameterizable vliw chip-multiprocessor with hardware pthreads support," *2013 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, vol. 0, pp. 122–126, 2010.

[8] M. Milward, D. Stevens, and V. Chouliaras, "Embedded uml design flow to the configurable le1 multicore vliw processor," in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, July 2012, pp. 1–8.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, Oct 2009, pp. 44–54.

[10] J. Shalf, D. Quinlan, and C. Janssen, "Rethinking hardware-software codesign for exascale systems," *Computer*, vol. 44, no. 11, pp. 22–30, 2011.

[11] M. Wang and F. Bodin, "Compiler-directed memory management for heterogeneous {MPSoCs}," *Journal of Systems Architecture*, vol. 57, no. 1, pp. 134 – 145, 2011, special Issue On-Chip Parallel And Network-Based Systems. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762110001347

[12] N. Brunie, S. Collange, and G. Diamos, "Simultaneous branch and warp interweaving for sustained gpu performance," *SIGARCH Comput. Archit. News*, vol. 40, no. 3, pp. 49–60, Jun. 2012.

[13] L. Yu, X. Tang, M. Wu, and T. Chen, "Improving branch divergence performance on {GPGPU} with a new {PDOM} stack and multi-level warp scheduling," *Journal of Systems Architecture*, vol. 60, no. 5, pp. 420 – 430, 2014, embedded Systems Architecture and Applications. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762113002634

[14] N. P. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I. Ganev, R. A. Golliver, R. Knauerhase, R. Lethin, B. Meister, A. K. Mishra, W. R. Pinfold, J. Teller, J. Torrellas, N. Vasilache, G. Venkatesh, and J. Xu, "Runnemede: An architecture for ubiquitous high-performance computing," *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, vol. 0, pp. 198–209, 2013.

[15] N. Rajovic, P. M. Carpenter, I. Gelado, N. Puzovic, A. Ramirez, and M. Valero, "Supercomputing with commodity cpus: Are mobile socs ready for hpc?" in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 40:1–40:12.

[16] B. Cope, P. Cheung, W. Luk, and L. Howes, “Performance comparison of graphics processors to reconfigurable logic: A case study,” *Computers, IEEE Transactions on*, vol. 59, no. 4, pp. 433–448, April 2010.

[17] V.A.Chouliaras, D.Stevens and V.M, “VThreads: A novel VLIW Chip Multiprocessor with hardware-assisted PThreads,” *Submitted to Microprocessors and Microsystems*, June 2015.

[18] J. Stratton, S. Stone, and W.-m. Hwu, “Mcuda: An efficient implementation of cuda kernels for multi-core cpus,” in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, J. Amaral, Ed. Springer Berlin Heidelberg, 2008, vol. 5335, pp. 16–30. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-89740-8_2

[19] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng, “Twin peaks: A software platform for heterogeneous computing on general-purpose and graphics processors,” in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’10. New York, NY, USA: ACM, 2010, pp. 205–216. [Online]. Available: http://dx.doi.org/10.1145/1854273.1854302

[20] J. Lee, J. Kim, S. Seo, S. Kim, J. Park, H. Kim, T. T. Dao, Y. Cho, S. J. Seo, S. H. Lee, S. M. Cho, H. J. Song, S.-B. Suh, and J.-D. Choi, “An opencl framework for heterogeneous multicores with local memory,” in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’10. New York, NY, USA: ACM, 2010, pp. 193–204. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854301

[21] J.-J. Li, C.-B. Kuan, T.-Y. Wu, and J. K. Lee, “Enabling an opencl compiler for embedded multicore dsp systems,” in *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, Sept 2012, pp. 545–552.

[22] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W.-M. Hwu, “Fcuda: Enabling efficient compilation of cuda kernels onto fpgas,” in *Application Specific Processors, 2009. SASP ’09. IEEE 7th Symposium on*, July 2009, pp. 35–42.

[23] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, “Autopilot: A platform-based esl synthesis system,” in *High-Level Synthesis*, P. Coussy and A. Morawiec, Eds. Springer Netherlands, 2008, pp. 99–112. [Online]. Available: http://dx.doi.org/10.1007/978-1-4020-8588-8_6

[24] I. Lebedev, S. Cheng, A. Doupnik, J. Martin, C. Fletcher, D. Burke, M. Lin, and J. Wawrzynek, “Marc: A many-core approach to reconfigurable computing,” in *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, Dec 2010, pp. 7–12.

[25] O. Anjum, T. Ahonen, and J. Nurmi, “{MPSoC} based on transport triggered architecture for baseband processing of an {LTE} receiver,” *Journal of Systems Architecture*, vol. 60, no. 1, pp. 140 – 149, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762113001963

[26] P. JÃd’Ãd’skelÃd’inen, C. S. de La Lama, P. Huerta, and J. Takala, “Opencl-based design methodology for application-specific processors.” in *ICSAMOS*, F. J. Kurdahi and J. Takala, Eds. IEEE, 2010, pp. 223–230.

[27] M. Owaida, N. Bellas, K. Daloukas, and C. Antonopoulos, “Synthesis of platform architectures from opencl programs,” in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, May 2011, pp. 186–193.

[28] H.-S. Kim, M. Ahn, J. A. Stratton, and W. mei W. Hwu, “Design evaluation of opencl compiler framework for coarse-grained reconfigurable arrays.” in *FPT*. IEEE, 2012, pp. 313–320.

[29] K. Shagrithaya, K. Kepa, and P. Athanas, “Enabling development of opencl applications on fpga platforms,” in *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, June 2013, pp. 26–30.

[30] J. Coole and G. Stitt, “Fast, flexible high-level synthesis from opencl using reconfiguration contexts,” *Micro, IEEE*, vol. 34, no. 1, pp. 42–53, Jan 2014.

[31] C. V. A.-P. V. Z. J. E. A. Stevens, D. and S. Hu, “Biothreads: a novel vliw-based chip multiprocessor for accelerating biomedical image processing applications.” *IEEE Trans Biomed Circuits Syst*, vol. 6, no. 3, pp. 257–268, Jun 2012. [Online]. Available: http://dx.doi.org/10.1109/TBCAS.2011.2166962

[32] R. Colwell, R. Nix, J. O'Donnell, D. Papworth, and P. Rodman, "A vliw architecture for a trace scheduling compiler," *Computers, IEEE Transactions on*, vol. 37, no. 8, pp. 967–979, Aug 1988.

[33] K. Kennedy and J. R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[34] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded computing - a VLIW approach to architecture, compilers, and tools.* Morgan Kaufmann, 2005.

[35] J. Fisher, "Trace scheduling: A technique for global microcode compaction," *Computers, IEEE Transactions on*, vol. C-30, no. 7, pp. 478–490, July 1981.

```
$\forall$ Function $f \in$ Module
  do if isKernel($f$)
    then EncloseBodyWithNestedLoop($f$)
        InsertExitLabel()
        $\forall$ DeclStmt $ds \in f$
          do if NonSingleDeclStmt($ds$)
            then split($ds$)
        $\forall$ CallExpr $ce \in f$
          do if isOpenCLBuiltin($ce$)
            do if isIdCall($ce$)
              then replace $ce$
            do if isLocalSize($ce$)
              then replace $ce$ with immediate
            do if isBarrier($ce$)
              then $barrierList$.add($ce$)
        $\forall$ ReturnStmt $rs \in f$
          $returnList$.add($rs$)
        do if barrierList = $\emptyset$
          then $\forall$ ReturnStmt $rs \in returnList$
            replace $rs$ with a goto
```

Fig. 17. Simple kernel coarsening algorithm.

```
$\forall$ Function $f$ $\in$ Module
   do if isKernel($f$)
      then $\forall$ DeclRefExpr $dre$ $\in$ $f$
         declRefExprList.add($dre$)
      $\forall$ ForStmt $outer$ $\in$ $f$
         do if $outer$ = outerLoop
            then TraverseRegion($outer$) {
                  $regionMap$.add($outer$)
                  $\forall$ Stmt $s$ $\in$ $outer$
                     MapStmt($s$, $outer$)
                     FindThreadDeps($s$)
                  $\forall$ Stmt $inner$ $\in$ $outer$
                     TraverseRegion($inner$)
                 }
      SearchThroughRegions() {
         $\forall$ Stmt $region$ $\in$ $regionMap$
            do if isNotParallel($region$)
               then HandleNonParallelRegion($region$)
      }
      FindReferencesToExpand() {
         $\forall$ Stmt $region$ $\in$ $regionMap$
            $\forall$ DeclStmt $ds$ $\in$ $region$
               $\forall$ DeclRefExpr $dre$ $\in$ $declStmtMap$($ds$)
                  do if SeparatedByFissionPoint($ds$, $dre$)
                     then ScalarExpand($ds$)
      }
```

Fig. 18. Algorithm outline for the second, and final, stage of kernel coarsening.

```
extern int BufferArg_0;
extern int BufferArg_1;
extern int BufferArg_2;
extern int BufferArg_3;
int main(void) {
int id = 0;
int num_cores = 1;
int total_workgroups = 1;
int workgroupX = 1;
int workgroupY = 0;
int x = 0;
int y = 0;
id = __builtin_le1_read_cpuid();
while (id < total_workgroups) {
x = id;
if (x >= workgroupX) {
y = x
workgroupX;
x = x % workgroupX;
}
if (y > workgroupY)
return 0;
__builtin_le1_set_group_id_1(y);
__builtin_le1_set_group_id_0(x);
permute(&BufferArg_0, &BufferArg_1, 8, &BufferArg_2, &BufferArg_3);
id += num_cores;
}
return id;
}
```

Fig. 19. Permute transformed Kernel Launcher

```
__kernel
void permute(__global const uint* unsortedData,
             __global const uint* scanedBuckets,
             uint shiftCount,
             __local ushort* sharedBuckets,
             __global uint* sortedData)
{
    size_t groupId = get_group_id(0);
    size_t localId = get_local_id(0);
    size_t globalId = get_global_id(0);
    size_t groupSize = get_local_size(0);
    /* Copy prescaned thread histograms to corresponding thread shared block */
    for(int i = 0; i < RADICES; ++i)
    {
        uint bucketPos = groupId * RADICES * groupSize + localId * RADICES + i;
        sharedBuckets[localId * RADICES + i] = scanedBuckets[bucketPos];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    /* Premute elements to appropriate location */
    for(int i = 0; i < RADICES; ++i)
    {
        uint value = unsortedData[globalId * RADICES + i];
        value = (value >> shiftCount) & 0xFFU;
        uint index = sharedBuckets[localId * RADICES + value];
        sortedData[index] = unsortedData[globalId * RADICES + i];
        sharedBuckets[localId * RADICES + value] = index + 1;
barrier(CLK_LOCAL_MEM_FENCE);
    }
}
```

Fig. 20. Radix sort kernel (pre-transform).

```
__kernel void permute(__global const uint* unsortedData,
                      __global const uint* scanedBuckets,
                      uint shiftCount,
                      __local ushort* sharedBuckets,
                      __global uint* sortedData) {
 size_t localId[64], globalId[64];
 unsigned __esdg_idx = 0;
 for (__esdg_idx = 0; __esdg_idx < 64; ++__esdg_idx){
  size_t groupId = get_group_id(0);
  localId[__esdg_idx] = __esdg_idx;
  globalId[__esdg_idx] = get_group_id(0) * 64 + __esdg_idx;
  size_t groupSize = 64;
  for(int i = 0; i < ( 1 « 8 ); ++i) {
   uint bucketPos = groupId * ( 1 « 8 ) * groupSize + localId[__esdg_idx] * ( 1 « 8 ) + i;
   sharedBuckets[localId[__esdg_idx] * ( 1 « 8 ) + i] = scanedBuckets[bucketPos];
  }
  //barrier(1);
 }
 for (int i = 0; i < ( 1 « 8 ); ++i) {
  for (__esdg_idx = 0; __esdg_idx < 64; ++__esdg_idx) {
   uint value = unsortedData[globalId[__esdg_idx] * ( 1 « 8 ) + i];
   value = (value » shiftCount) & 0xFFU;
   uint index = sharedBuckets[localId[__esdg_idx] * ( 1 « 8 ) + value];
   sortedData[index] = unsortedData[globalId[__esdg_idx] * ( 1 « 8 ) + i];
   sharedBuckets[localId[__esdg_idx] * ( 1 « 8 ) + value] = index + 1;
   //barrier(1);
  }
 }
 for (__esdg_idx = 0; __esdg_idx < 64; ++__esdg_idx) __ESDG_END: ;
 }
}
```

Fig. 21. Permute source code after complete transformation demonstrating barrier removal.