J Sign Process Syst (2014) manuscript No.
(will be inserted by the editor)

OpenDwarfs: On the Characterization of Computation and Communication Patterns on Fixed and Reconfigurable Architectures

Konstantinos Krommydas · Wu-chun Feng · Christos D. Antonopoulos · Nikolaos Bellas

Received: date / Accepted: date

Abstract The proliferation of heterogeneous computing platforms presents the parallel computing community with new challenges. One such challenge entails evaluating the efficacy of such parallel architectures and identifying the architectural innovations that ultimately benefit applications. To address this challenge, we need benchmarks that capture the execution patterns (i.e., dwarfs or motifs) of applications, both present and future, in order to guide future hardware design. Furthermore, we desire a common programming model for the benchmarks that facilitates code portability across a wide variety of different processors (e.g., CPU, APU, GPU, FPGA, DSP) and computing environments (e.g., embedded, mobile, desktop, server).

As such, we present the latest release of OpenDwarfs, a benchmark suite that currently realizes the Berkeley dwarfs in OpenCL, a vendor-agnostic and open-standard computing language for parallel computing. Using OpenDwarfs, we characterize a diverse set of modern fixed and reconfigurable parallel platforms: multi-core CPUs, discrete and integrated GPUs, Intel Xeon Phi co-processor, as well as a FPGA. We describe the computation and communication patterns exposed by a representative set of dwarfs, obtain relevant profiling data and execution information, and draw conclusions that highlight the complex interplay between dwarfs' patterns and the underlying hardware architecture of modern parallel platforms.

Konstantinos Krommydas · Wu-chun Feng
Department of Computer Science, Virginia Tech
E-mail: {kokrommy, wfeng}@vt.edu

Christos D. Antonopoulos · Nikolaos Bellas
Department of Electrical and Computer Engineering, University of Thessaly
E-mail: {cda, nbellas}@uth.gr

Keywords OpenDwarfs · benchmarking · evaluation · dwarfs · performance · characterization

1 Introduction

Over the span of the last decade, the computing world has borne witness to a parallel computing revolution, which delivered parallel computing to the masses while doing so at low cost. The programmer has been presented with a myriad of new computing platforms promising ever-increasing performance. Programming these platforms entails familiarizing oneself with a wide gamut of programming environments, along with optimization strategies strongly tied to the underlying architecture. The aforementioned realizations present the parallel computing community with two challenging problems:

(a) The need of a common means of programming, and
(b) The need of a common means of evaluating this diverse set of parallel architectures.

The former problem was effectively solved through a concerted industry effort that led to a new parallel programming model, i.e., OpenCL. Other efforts, like SOpenCL [16] and Altera OpenCL [1] enable transforming OpenCL kernels to equivalent synthesizable hardware descriptions, thus facilitating exploitation of FPGAs as hardware accelerators, while obviating the overhead of additional development cost and expertise.

The latter problem cannot be sufficiently addressed by the existing benchmark suites. Such benchmarks suites (e.g., SPEC CPU [10], PARSEC [4]) are often written in a language tied to a particular architecture and porting the benchmarks to another platform would typically mandate re-writing them using the programming model suited for the platform under consideration. The additional caveat in simply re-casting these

benchmarks as OpenCL implementations is that existing benchmark suites represent collections of overly specific applications that do not address the question of what the best way of expressing a parallel computation is. This impedes innovations in hardware design, which will come as a quid pro quo, only when software idiosyncrasies are taken into account at design and evaluation stages. This is not going to happen unless software requirements are abstracted in a higher level and represented by a set of more meaningful benchmarks. To address all these issues, we proposed OpenDwarfs [9], a benchmark suite for heterogeneous computing in OpenCL, in which applications are selected based on the computation and communication patterns defined by Berkeley's Dwarfs [3].

Our contributions in this paper are two-fold:

(a) We present the latest OpenDwarfs release, in which we attempt to rectify prior release's shortcomings. We propose and implement all necessary changes towards a comprehensive benchmark suite that adheres both to the dwarfs' concept and established benchmark creation guidelines.
(b) We verify functional portability and characterize OpenDwarfs' performance on multi-core CPUs, discrete and integrated GPUs, the Intel Xeon Phi co-processor and even FPGAs, and relate our observations to the underlying computation and communication pattern of each dwarf.

The rest of the paper is organized as follows: in Section 2 we discuss related work and how our work differs and/or builds upon it. In Section 3 we provide a brief overview of OpenCL and the FPGA technology. Section 4 presents our latest contributions to the OpenDwarfs project and the rationale behind some of our design choices. Following this, in Section 5, we introduce SOpenCL, the tool we use for automatically converting OpenCL kernels to synthesizable Verilog for the FPGA. Section 6 outlines our experimental setup, followed by results and a detailed discussion for each one of the dwarfs under consideration in Section 7. Section 8 concludes the paper and discusses future work.

2 Related Work

HPC engineering and research have highlighted the importance of developing benchmarks that capture high-level computation and communication patterns. In [17] the authors emphasize the need for benchmarks to be related to scientific *paradigms*, where a paradigm defines what the important problems in a scientific domain are and what the set of accepted solutions is. This notion of paradigm parallels that of the *computational dwarf*. A dwarf is an algorithmic method that encapsulates a specific computation and communication pattern. The seven original dwarfs, attributed to P. Colella's unpublished work, became known as *Berkeley's dwarfs*, after Asanovic et al. [3] formalized the dwarf concept and complemented the original set of dwarfs with six more. Based in part on the dwarfs, Keutzer et al. later attempted to define a pattern language for parallel programming [11].

The combination of the aforementioned works sets a concrete theoretical basis for benchmark suites. Following this path and based on the very same nature of the dwarfs and the global acceptance of OpenCL, our work on extending OpenDwarfs attempts to present an all-encompassing benchmark suite for heterogeneous computing. Such a benchmark suite, whose application selection delineates modern parallel application requirements, can constitute the basis for comparing and guiding hardware and architectural design. On a parallel path with OpenDwarfs, which was based on OpenCL from the onset, many existing benchmark suites were re-implemented in OpenCL and new ones were released (e.g., Rodinia [5], SHOC [7], Parboil [19]). Most of them were originally developed as GPU benchmarks and as such still carry optimizations that favor GPU platforms. This violates the *portability* requirement for benchmarks that mandates a lack of bias for one platform over another [3,17] and prevents drawing broader conclusions with respect to hardware innovations. We attempt to address the above issues with our efforts in extending OpenDwarfs.

On the practical side of matters, benchmark suites are used for characterizing architectures. In [5] and [7] the authors discuss architectural differences between CPUs and GPUs on a higher level. Although not based on OpenCL kernels, a more detailed discussion on architectural features' implications with respect to algorithms and insight on future architectural design requirements is given in [15]. In this work, we complement prior research by characterizing OpenDwarfs on a diverse set of modern parallel architectures, including CPUs, APUs, discrete GPUs, the Intel Xeon Phi co-processor, as well as on FPGAs.

3 Background

3.1 OpenCL

OpenCL provides a parallel programming framework for a variety of devices, ranging from conventional Chip Multiprocessors (CMPs) to combinations of heterogeneous cores such as CMPs, GPUs, and FPGAs. Its plat-

form model comprises a *host* processor and a number of *compute devices*. Each device consists of a number of compute units, which are subsequently subdivided into a number of processing elements. An OpenCL application is organized as a *host program* and a number of *kernel functions*. The host part executes on the host processor and submits commands that refer to either the execution of a kernel function or the manipulation of memory objects. Kernel functions contain the computational part of an application and are executed on the compute devices. The work corresponding to a single invocation of a kernel is called a *work-item*. Multiple work-items are organized in a *work-group*.

OpenCL allows for geometrical partitioning of the grid of independent computations to an N-dimensional space of work-groups, with each work-group being subsequently partitioned to an N-dimensional space of work-items, where $1 \leq N \leq 3$. Once a command that refers to the execution of a kernel function is submitted, the host part of the application defines an abstract index space, and each work-item executes for a single point in the index space. A work-item is identified by a tuple of IDs, defining its position within the work-group, as well as the position of the work-group within the computation grid. Based on these IDs, a work-item is able to access different data (SIMD style) or follow a different path of execution.

Data transfers between host and device occur via the PCIe bus in the cases of discrete GPUs and other types of co-processors like Intel Xeon Phi. In such cases, the large gap between the (high) computation capability of the device and the (comparatively low) PCIe bandwidth may incur significant overall performance deterioration. The problem is aggravated when an algorithmic pattern demands multiple kernel launches between costly host-to-device and device-to-host data transfers. Daga et al. [6] re-visit Amdahl's law to account for the parallel overhead incurred by data transfers in accelerators like discrete or fused GPUs. Similar behavior, with respect to restricting available parallelism is observed in CPUs and APUs, too, when no special considerations are taken during OpenCL memory buffer creation and manipulation. In generic OpenCL implementations, if the CPU-as-device scenario is not taken into account, unnecessary buffers are allocated and unnecessary data transfers take place within the *common* memory space. The data transfer part on the CPU cases can be practically eliminated. This requires use of the *CL_MEM_USE_HOST_POINTER* flag and passing the host-side pointer to the CPU allocated memory location as a parameter at OpenCL buffer creation time. The OpenCL data transfer commands are consequently rendered useless. In APUs, due to the tight coupling of the CPU and GPU core on the same die, and depending on the exact architecture, more data transfer options are available for faster data transfers between the CPU and GPU side. Lee et al. [14] and Spafford et al. [18] have studied the tradeoffs of fused memory hierarchies. We leave a detailed study of the dwarfs with respect to data transfers on APUs for future research.

3.2 FPGA Technology

Compared to the fixed hardware of the CPU and GPU architectures, FPGAs (*field-programmable gate arrays*) are configured post-fabrication through configuration bits that specify the functionality of the configurable high-density arrays of uncommitted logic blocks and the routing channels between them. They offer the highest degree of flexibility in tailoring the architecture to match the application, since they essentially emulate the functionality of an ASIC (Application Specific Integrated Circuit). FPGAs avoid the overheads of the traditional ISA-based von Neumann architecture followed by CPUs and GPUs and can trade-off computing resources and performance by selecting the appropriate level of parallelism to implement an algorithm. Since reconfigurable logic is more efficient in implementing specific applications than multicore CPUs, it enjoys higher power efficiency than any general-purpose computing substrate.

The main drawbacks of FPGAs are two-fold:

(a) They are traditionally programmed using Hardware Description Languages (VHDL or Verilog), a time-consuming and labor-intensive task, which requires deep knowledge of low-level hardware details. Using SOpenCL, we alleviate the burden of implementing accelerators in FPGAs by utilizing the same OpenCL code-base used for CPU and GPU programming.
(b) The achievable clock frequency in reconfigurable devices is lower (by almost an order of magnitude) compared to high-performance processors. In fact, most FPGA designs operate in a clock frequency less than 200 MHz, despite aggressive technology scaling.

4 OpenDwarfs Benchmark Suite

OpenDwarfs is a benchmark suite that comprises 13 of the dwarfs, as defined in [3]. The dwarfs and their corresponding instantiations (i.e., applications) are shown in Table 1. This OpenDwarfs release provides full coverage of the dwarfs, including more stable implementations of

Table 1 Dwarf Instantiations in OpenDwarfs

Dwarf	Dwarf Instantiation
Dense Linear Algebra	LUD (LU Decomposition)
Sparse Matrix-Vector Matrix Multiplication	CSR (Compressed Sparse-Row Vector Multiplication)
Graph Traversal	BFS (Breadth-First Search)
Spectral Methods	FFT (Fast Fourier Transform)
N-body Methods	GEM (Electrostatic Surface Potential Calculation)
Structured Grid	SRAD (Speckle Reducing Anisotropic Diffusion)
Unstructured Grid	CFD (Computational Fluid Dynamics)
Combinational Logic	CRC (Cyclic Redundancy Check)
Dynamic Programming	NW (Needleman-Wunsch)
Backtrack & Branch and Bound	NQ (N-Queens Solver)
Finite State Machine	TDM (Temporal Data Mining)
Graphical Models	HMM (Hidden Markov Model)
MapReduce	StreamMR

the *Finite State Machine* and *Backtrack & Branch and Bound* dwarfs. CSR (*Sparse Linear Algebra* dwarf) and CRC (*Combinational Logic* dwarf) have been extended to allow for a wider range of options, including running with varying work-group sizes or running the main kernel multiple times. We plan to propagate these changes to the rest of the dwarfs, as they can uncover potential performance issues for each of the dwarfs on devices of different capabilities.

One of the most important changes in this implementation of OpenDwarfs is related to the *uniformity* of optimization level across all dwarfs. More precisely, none of the dwarfs contains optimizations that would make a specific architecture more favorable than another. Use of shared memory, for instance, in many of the dwarfs in the previous OpenDwarfs release favored GPU architectures. Such favoritism limits the scope of a benchmark suite, as we discuss in Section 2, in that it takes away from the general suitability of an architecture with respect to the *computation and communication pattern* intrinsic to a dwarf and rather focuses attention into very architecture-specific and often exotic software optimizations. We claim that architectural design should be guided by the dwarfs on the premise that they form the basic, recurring, patterns of computation and communication, and that the ensuing architectures following this design approach would be efficient without the need for the aforementioned optimizations (or at least the most complex - and painfull for programmers - ones).

Of course, the above point does not detract from the usefulness of optimized dwarf implementations for specific architectures that may employ each and every software technique available to get the most of the *current* underlying architecture. In fact, we have ourselves been working on providing such optimized implementations for dwarfs on a wide array of CPUs, GPUs and MIC (e.g., N-body methods [12]) and plan to enrich the OpenDwarfs repository with such implementations as a next step. The open source nature of OpenDwarfs actively encourages the developers' community to embrace and contribute to this goal, as well.

In the end, optimized and unoptimized implementations of *dwarf* benchmarks are complementary and one would argue essential constituent parts of a complete benchmark suite. We identify three cases that exemplify why the above is a practical reality:

(a) Hardware (CPU, GPU, etc.) vendors are mostly interested in the most optimized implementation for their device, in order to stress their current device's capabilities. When designing a new architecture, however, they need a basic, unoptimized implementation *based on the dwarfs' concept*, so that the workloads are *representative* of broad categories, on which they can subsequently build and develop their design in a hardware-software synergistic approach.
(b) Compiler writers also employ both types of implementations: the unoptimized ones to test their compiler back-end optimizations on and the (manually) optimized ones to compare the efficacy of such compiler optimizations. Once more, the generality of the benchmarks, being based on the dwarfs concept, is of fundamental importance in the generality (and hence success) of new compiler techniques.
(c) Independent parts/organizations (e.g., lists ranking hardware, IT magazines) want a set of benchmarks that is *portable* across devices and in which *all* devices start from the same starting point (i.e., unoptimized implementations) for fairness in comparisons/rankings.

In order to enhance code uniformity, readability and usability for our benchmark suite, we have augmented the OpenDwarfs library of common functions. For ex-

ample, we introduce more uniform error checking functionality and messages, while a set of common options can be used to select and initialize the desired OpenCL device type at run-time. CPU, GPU, Intel Xeon Phi and FPGA are the currently available choices. Finally, it retains the previous version's timing infrastructure. The latter offers custom macro definitions, which record, categorize and print timing information of the following types: *data transfer time* (host to device and device to host), *kernel execution time*, and *total execution time*. The former two are reported both as an aggregate, and in its constituent parts (e.g., total kernel execution time, and time per kernel- for multi-kernel dwarf implementations).

The build system remains largely the same, except for changes allowing selecting the Altera OpenCL SDK for FPGA execution, while a test-run *make* target allows verifying correct installation and execution of the dwarfs using default small test datasets. FPGA support for Altera FPGAs is offered, but currently limited to two of the dwarfs, due to lack of complete support of the OpenCL standard by the Altera OpenCL SDK, which requires certain alterations to the code for successful compilation and full FPGA compatibility [2]. We plan to provide full coverage in upcoming releases, but for completeness in the context of this work we use SOpenCL that enables full Xilinx FPGA OpenCL support.

5 SOpenCL (Silicon OpenCL) Tool

We use the SOpenCL tool [16] to automatically generate hardware accelerators for the OpenDwarf kernels, thus dramatically minimizing development time and increasing productivity. SOpenCL enables quick exploration of different architectural scenarios and evaluation of the quality of the design in terms of computational bandwidth, clock frequency, and size. The final output of this procedure is synthesizable Verilog, functionally equivalent to the original OpenCL kernels, which can in turn be used to configure an FPGA. This section outlines some of the basic concepts of the SOpenCL compilation tool-flow and template architecture.

5.1 Front-end

The SOpenCL front end is a source-to-source compiler that adjusts the parallelism granularity of an OpenCL kernel to better match the hardware capabilities of the FPGA. OpenCL kernel code specifies computation at a work-item granularity. A straightforward approach would map a work-item to an invocation of the hardware accelerator. This approach is suboptimal for FPGAs which incur heavy overhead to initiate thousands of work-items of fine granularity. SOpenCL, instead, applies source-to-source transformations that collectively aim to coarsen the granularity of a kernel function at a work-group level. The main step in this series of transformations is logical thread serialization. Work-items inside a work-group can be executed in any sequence, provided that no synchronization operation is present inside a kernel function. Based on this observation, we serialize the execution of work-items by enclosing the instructions in the body of a kernel function into a triple nested loop, given that the maximum number of dimensions in the abstract index space within a work-group is three. Each loop nest enumerates the work-items in the corresponding dimension, thus serializing their execution. The output of this stage is a semantically equivalent C code at the work-group granularity.

5.2 Back-end

SOpenCL back-end flow is based on the LLVM compiler infrastructure [13] and generates the synthesizable Verilog for synthesizing the final hardware modules of the accelerator. The functionality of the back-end supports *bitwidth optimization*, *predication*, and *swing modulo scheduling* (SMS) as separate LLVM compilation passes:

(a) *Bitwidth optimization* is used to minimize the width of functional units and wiring connecting them, to the maximum expected width of operands at each level of the circuit, based on the expected range of input data and the type of operations performed on input and intermediate data. Experimental evaluation on several integer benchmarks shows significant area and performance improvement due to bitwidth optimizations.

(b) *Predication* converts control dependencies to data dependences in the inner loop, transforming its body to a single basic block. This is a prerequisite in order to apply modulo scheduling in the subsequent step.

(c) *Swing modulo scheduling* is used to generate a schedule for the inner loops. The scheduler identifies an iterative pattern of instructions and their assignment to functional units (FUs), so that each iteration can be initiated before the previous ones terminates. SMS creates software pipelines under the criterion of minimizing the Initiation Interval (II), which is the constant interval between launches of successive work-items. Lower values of Initiation In-

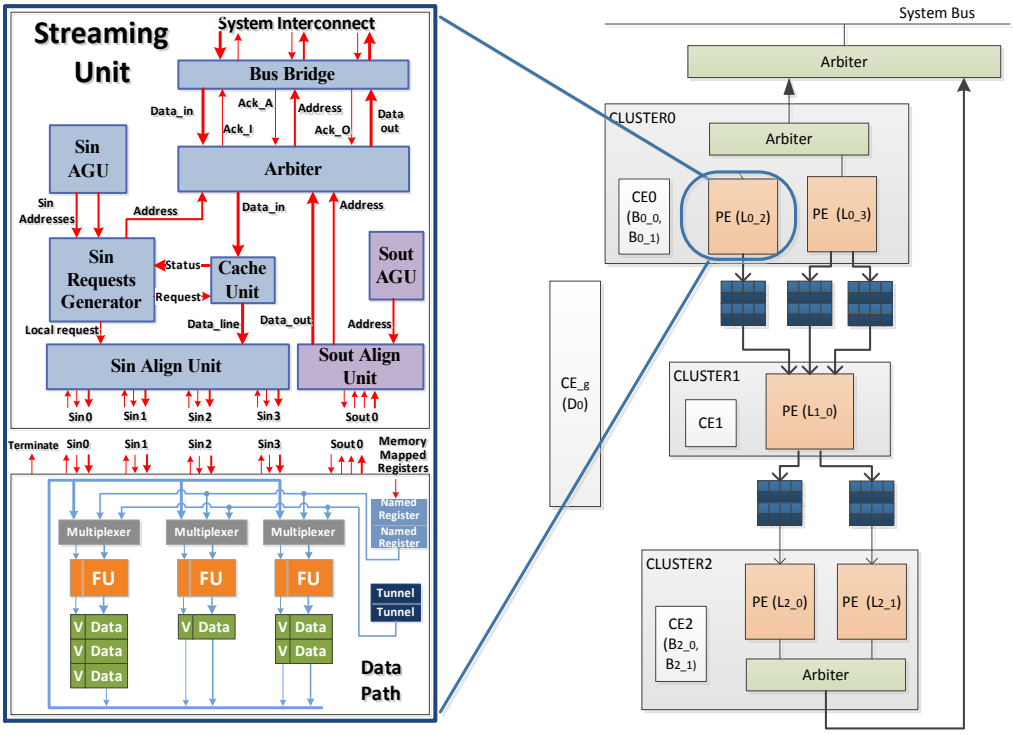


Fig. 1 Architectural template of a Processing Element (PE) module (left). An example block diagram of an automatically generated hardware accelerator (right) instantiates multiple PEs, although only the PEs with external access are equipped with a Streaming Unit. OpenCL arrays are implemented as internal FPGA SRAMs.

terval correspond to higher throughput since more work-items are initiated and, therefore, more results are produced per cycle. That makes the Initiation Interval the main factor affecting computational bandwidth in modulo scheduled loop code.

5.3 Accelerator Architecture

Figure 1 outlines the architectural template of a Processing Element (PE), which consists of the data path and the streaming unit. The Data Path implements the modulo-scheduled computations of an innermost loop in the OpenCL kernel. It consists of a network of functional units (FUs) that produce and consume data elements using explicit input and output FIFO channels to the streaming units. The customizable parameters of the data path are the type and bitwidth of functional units (ALUs for arithmetic and logical instructions, shifters, etc.), the custom operation performed within a generic functional unit (e.g., only addition or subtraction for an ALU), the number and size of registers in the queues between functional units, and the bandwidth to and from the streaming unit. For example, when II=1, one FU will be generated for each LLVM instruction in the inner loop. The data path supports both standard and complex data types and all standard arithmetic operations, including integer and IEEE-754 compliant single- and double-precision floating point. At compile time, the system selects and integrates the appropriate implementation according to precision requirements and the target initiation interval. We use floating-point (FP) units generated by the FloPoCo [8] arithmetic unit generator.

In case the kernel consists of a single inner loop, the streaming unit handles all issues regarding data transfers between the main memory, and the data path. These include address calculation, data alignment, data ordering, and bus arbitration and interfacing. The streaming unit consists of one or more input and output stream modules. It is generated to match the memory access pattern of the specific application, the characteristics of the interconnect to main memory, and the bandwidth requirements of the data path.

SOpenCL infrastructure supports arbitrary loop nests and shapes. Different loops at the same level of a loop nest are implemented as distinct PEs data paths, which communicate and synchronize through local memory buffers (Figure 1). Similarly, SOpenCL supports barrier synchronization constructs within a computational kernel.

Finally, Control Elements (CEs) are used to control and execute code of outer loops in a multilevel loop nest. CEs have a simpler, less optimized architecture, since outer loop code does not execute as frequently as inner loop code.

Table 2 Configuration of the Target Fixed Architectures

Model	AMD Opteron 6272	AMD Llano A8-3850	AMD Radeon HD 6550D	AMD A10-5800K	AMD Radeon HD 7660D	AMD Radeon HD 7970	Intel Xeon Phi P1750
Type	CPU	CPU*	Integr. GPU*	CPU*	Integr. GPU*	Discrete GPU	Co-processor
Frequency	2.1 GHz	2.9 GHz	600 MHz	3.8 GHz	800 MHz	925 MHz	1.09 GHz
Cores	16	4	5†	4	6†	32†	61
Threads/core	1	1	5	1	4	4	4
L1/L2/L3 Cache (KB)	16/2048/ 8192‡	64/1024/- (per core)	8/128/- (L1 per CU)	64/2048/- (per 2 cores)	8/128/- (L1 per CU)	16/768/- (L1 per CU)	32/512/- (per core)
SIMD (SP)	4-way	4-way	16-way	8-way	16-way	16-way	16-way
Process	32nm	32nm	32nm	32nm	32nm	32nm	22nm
TDP	115W	100W*	100W*	100W*	100W*	210W	300W
GFLOPS (SP)	134.4	46.4	480	121.6	614.4	3790	2092.8

† Compute Units (CU) ‡ L1: 16KBx16 data shared, L2: 2MBx8 shared, L3: 8MBx2 shared * CPU and GPU fused on the same die, total TDP

6 Experimental Setup

This section presents our experimental setup. First, we present the software setup and methodology used for collecting the results and discuss the hardware used in our experiments.

6.1 Software and experimental methodology

For benchmarking our target architectures we use OpenDwarfs (as discussed in Section 4), available for download at *https://github.com/opendwarfs/OpenDwarfs*.

The CPU/GPU/APU software environment consists of 64-bit Debian Linux 7.0 with kernel version 2.6.37, GCC 4.7.2 and AMD APP SDK 2.8. AMD GPU/APU drivers are AMD Catalyst 13.1. Intel Xeon Phi is hosted on a CentOS 6.3 environment with the Intel SDK for OpenCL applications XE 2013. For profiling we use AMD CodeXL 1.3 and Intel Vtune Amplifier XE 2013 for the CPU/GPU/APU and Intel Xeon Phi, respectively. In Table 3 we provide details about the subset of dwarf applications used and their input datasets and/or parameters. Kernel execution time and data transfer times are accounted for and measured by use of the corresponding OpenDwarfs timing infrastructure. In turn, the aforementioned infrastructure lies on the OpenCL events (which return timing information as a *cl_ulong* type) to provide accurate timing in nanosecond resolution.

6.2 Hardware

In order to capture a wide range of parallel architectures, we pick a set of representative device types: a high-end multi-core CPU (AMD Opteron 6272) and a high-performance discrete GPU (AMD Radeon HD 7970). An integrated GPU (AMD Radeon HD 6550D) and a low-powered low-end CPU (A8-3850), both part of a heterogeneous Llano APU system (i.e., CPU and GPU fused on the same die), as well as a newer generation APU system (Trinity) comprising an A10-5800K and an AMD Radeon HD 7660D integrated GPU. Finally, an Intel Xeon Phi co-processor. Details for each of the aforementioned architectures are given in Table 2. To evaluate OpenDwarfs on FPGAs, we use the Xilinx Virtex-6 LX760 FPGA on a PCIe v2.1 board, which consumes approximately 50 W and contains 118560 logic slices. Each slice includes 4 LUTs and 8 flip-flops. FPGA clock frequency ranges from 150 to 200 MHz for all designs.

Table 3 OpenDwarfs Benchmark Test Parameters/Inputs

Benchmark	**Problem Size**
GEM	Input file & parameters: nucleosome 80 1 0.
NW	Two protein sequences of 4096 letters each.
SRAD	2048x2048 FP matrix, 128 iterations.
BFS	Graph: 248,730 nodes and 893,003 edges.
CRC	Input data-stream: 100MB.
CSR	2048^2 x 2048^2 sparse matrix.

7 Results

Here we present our results of running a representative subset of the dwarfs on a wide array of parallel architectures. After we verify functional portability across all platforms, including the FPGA, we characterize the dwarfs and illustrate their utility in guiding architectural innovation, which is one of the main premises of the OpenDwarfs benchmark suite.

7.1 N-body Methods: GEM

The n-body class of algorithms refers to those algorithms that are characterized by all-to-all computations within a set of particles (bodies). In the case of GEM, our n-body application, the electrostatic surface potential of a biomolecule is calculated as the sum of

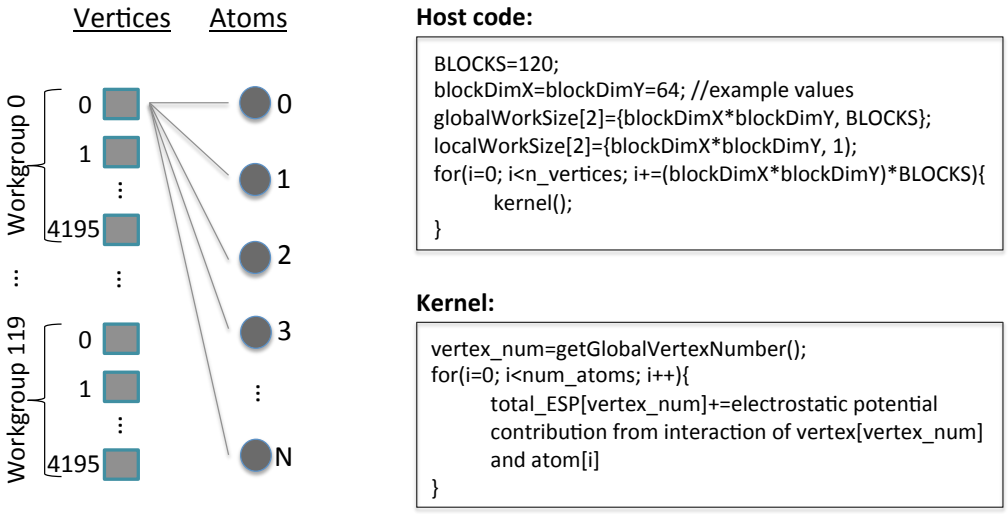


Fig. 2 GEM

charges contributed by all atoms in the biomolecule due to their interaction with a specific surface vertex (two sets of bodies). In Figure 2 we illustrate the computation pattern of GEM and present the pseudocode running on the OpenCL host and device. Each work-item accumulates the potential at a single vertex due to every atom in the biomolecule. A number of work-groups (*BLOCKS=120* in our example) each having *blockDimX*blockDimY* work-items (*4096* in our example) is launched, until all vertices' potential has been calculated.

GEM's computation pattern is regular, in that the same amount of computation is performed by each work-item in a work-group and no dependencies hinder computation continuity. Total execution time mainly depends on the maximum computation throughput. Computation itself is characterized by FP arithmetic, including (typically expensive) division and square root operations that constitute one of the main bottlenecks. Special hardware can provide low latency alternatives of these operations, albeit at the cost of minor accuracy loss that may or may not be acceptable for certain types of applications. Such fast math implementations are featured in many architectures and typically utilize look-up tables for fast calculations.

With respect to data accesses, atom data is accessed in a serial pattern, simultaneously by all work-items. This facilitates efficient utilization of cache memories available in each architecture. Figure 3 and Table 2 can assist in pinpointing which architectural features are important for satisfactory GEM performance: good FP performance and sufficient first level cache. With respect to the former, Opteron 6272 and A10-5800K CPUs reach about 130 GFLOPS and A8-3850 falls behind by a factor of 2.9, as defined by their number of cores, SIMD capability and core frequency. However, the cache hierarchy between the three CPU architectures is fundamentally different. Opteron 6272 has 16K of L1 cache per core, which is *shared* among all 16 cores. Given the computation and communication pattern of n-body dwarfs, such types of caches may be an efficient choice. Cache miss rates at this level (L1), are also indicative of the fact: A8-3850 with 64KB of dedicated L1 cache per core is characterized by a 0.55% L1 cache miss rate, with Opteron 6272 at 10.2% and A10-5800K a higher 24.25%. Those data accesses that result in L1 cache misses are mostly served by L2 cache and rarely require expensive RAM memory accesses. Measured L2 cache miss rates are 4.5%, 0.18% and 0%, respectively, reflecting the L2 cache capability of the respective platforms (Table 2). Of course, the absolute number of accesses to L2 cache, depend on the previous level's cache misses, so a smaller percentage on a platform, tells only part of the story if we plan to compare different platforms to each other. In cases where data accesses follow a predictable pattern, like in GEM, specialized hardware can predict what data is going to be needed and fetch it ahead of time. Such *hardware prefetch* units are available - and of advanced maturity - in multi-core CPUs. This proactive loading of data can take place between the main memory and last level cache (LLC) or between different cache levels. In all three CPU platforms, a large number of prefetch instructions is emitted, as seen through profiling the appropriate counter, which, together with the regular data access patterns, verify the overall low L1 cache miss rates mentioned earlier.

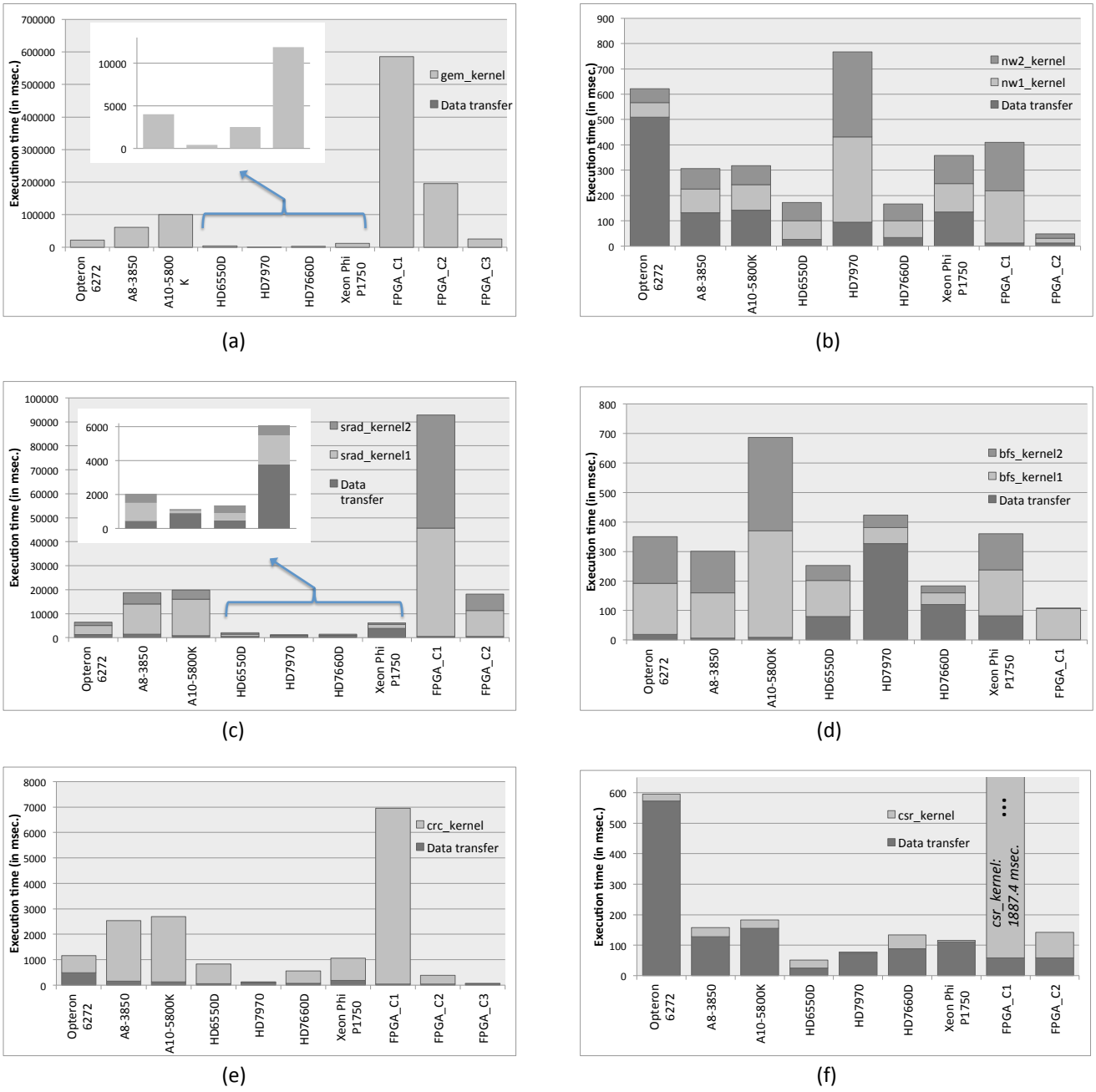


Fig. 3 Results: (a) GEM, (b) NW, (c) SRAD, (d) BFS, (e) CRC, (f) CSR

Xeon Phi's execution is characterized by high vectorization intensity (12.84, the ideal being 16), which results from regular data access patterns and implies efficient auto-vectorization on behalf of the Intel OpenCL compiler and its *implicit vectorization module*. However, profiling reveals that the estimated latency impact is high indicating that the majority of L1 misses result in misses in L2 cache, too. This signifies the need for optimizations such as data reorganization and blocking for L2 cache, or the introduction of a more advanced hardware prefetch unit in future Xeon Phi editions - currently there is lack of automatic (i.e., hardware) prefetching to L1 cache (only main memory to L2 cache prefetching is supported). Further enhancement of the ring interconnect that allows efficient sharing of the dedicated (per core) L2 cache contents across cores would also assist in attaining better performance for the n-body dwarf. While Xeon Phi, lying between the multi-core CPU and many-core GPU paradigms, achieves good overall performance for this - unopti-

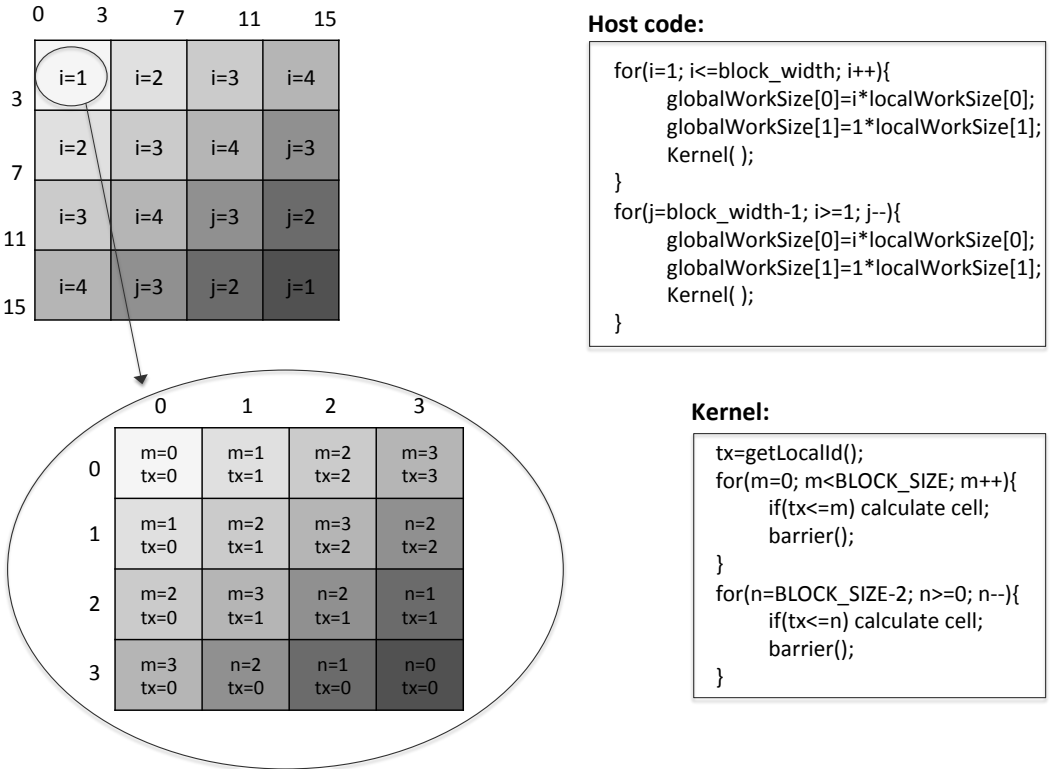


Fig. 4 Needleman-Wunsch

mized, architecture agnostic - code implementation, it falls behind its theoretical maximum performance of nearly 2 TFLOPS.

With respect to GPU performance, raw FP performance is one of the deciding factors, as well. As a result HD 7970 performs the best and is characterized by the best occupancy (70%), compared to 57.14% and 37.5% for HD 7660D and HD 6550D, respectively. In all three cases, cache hit rates are over 97% (reaching 99.96% for HD 7970, corroborating that our conclusions for the CPU cache architectures hold for GPUs, too, for this class of applications (i.e, n-body dwarf). Correspondingly, the measured percentage of memory unit stalls is held at low levels. In fact, the memory unit is kept busy for over 76% of the time for all three GPU architectures, including all extra fetches and writes and taking any cache or memory effects into account.

Although FPGAs are not made for FP performance, SOpenCL produces accelerators whose performance lies between that of CPUs and GPUs. SOpenCL instantiates modules for single-precision FP operations, such as division and square root. Partially unrolling the outer loop executed by each thread four times results in nearly 4-fold speedup (FPGA_C2) compared to the base accelerator configuration (FPGA_C1). Multiple accelerators can be instantiated and process in parallel different vertices on the grid, thus providing even higher speedup (FPGA_C3).

7.2 Dynamic Programming: Needleman-Wunsch (NW)

Dynamic programming is a programming method in which a complex problem is solved by decomposition into smaller subproblems. Combining the solutions to the subproblems provides the solution to the original problem. Our dynamic programming dwarf, Needleman-Wunsch, performs protein sequence alignment, i.e., attempts to identify the similarity level between two given strings of aminoacids. Figure 4 illustrates its computation pattern and two levels of parallelism. Each element of the 2D matrix depends on the values of its west, north and north-west neighbors. This set of dependencies limits available parallelism and enforces a wavefront computation pattern. On the first level blocks of computation (i.e., OpenCL work-groups) are launched across the anti-diagonal and on the second level, each of the work-group's work-items works on cells on each anti-diagonal. Available parallelism at each stage is variable, starting with a single work-group, increasing as we reach the main anti-diagonal and decreasing again as we reach the bottom right. Parallelism varies within each work-group in a similar way, as shown in the re-

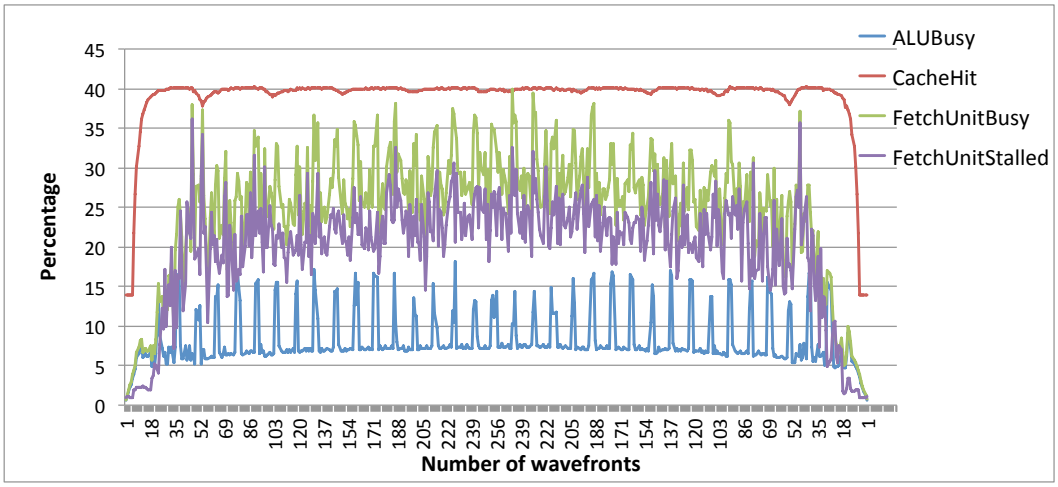


Fig. 5 NW profiling on HD 7660D.

spective figure, where a variable number of work-items work independently in parallel at each anti-diagonal's level. Needleman-Wunsch algorithm imposes significant synchronization overhead (repetitive barrier invocation within the kernel) and requires modest integer performance. Computations for each 2D matrix cell entail calculating an alignment score that depends on the three neighboring entries (west, north, northwest) and a max operation (i.e., nested if statements).

In algorithms like NW that are characterized by inter- and intra-work-group dependencies there are two big considerations. First, the overhead for repetitively launching a kernel (corresponding to inter-work-group synchronization), and second, the cost of the intra- work-group synchronization via *barrier()* or any other synchronization primitives. Introducing system-wide (hardware) barriers would help to solve the former of the problems, while optimization of already existing intra-work-group synchronization primitives would be beneficial for this kind of applications for the latter case.

Memory accesses follow the same pattern as computation, i.e., for each element the west, north and northwest elements are loaded from the reference matrix. For each anti-diagonal m within a work-group (Figure 4) the updated data from anti-diagonal *m-1* is used.

As we can observe, GPUs do not perform considerably better than the CPUs. In fact, Opteron 6272 surpasses all GPUs (and even Xeon Phi), when we only take kernel execution time into account. What needs to be emphasized in the case of algorithms, such as NW, is the variability in the characteristics of each kernel iteration. In Figure 5 we observe such variability for metrics like the percentage of the time the ALU is busy, the cache hit rate, the fetch unit is busy or stalled, on the HD 7660D. Similar behavior is observed in the case of HD 6550D. Most of these metrics can be observed to be a function of the number of active wavefronts in every kernel launch. For instance, cache hit follows an inverse-U-shaped curve, as do most of the aforementioned metrics. In both cases, occupancy is below 40% (25% for HD 6550D) and ALU packing efficiency barely reaches 50%, which indicates a mediocre job on behalf of the shader compiler in packing scalar and vector instructions as VLIW instructions of the Llano and Trinity integrated GPUs (i.e., HD 6550D and HD 7660D).

As expected, the FPGA performs the best when it comes to integer code, in which case, its performance lies closer to GPUs than to CPUs. Multiple accelerators (5 pairs) and fully unrolling the innermost loop deliver higher performance (FPGA_C2) than a single pair (FPGA_C1) and render the FPGA implementation the fastest choice for the dynamic programming dwarf. In the FPGA implementation of NW, the data fetches' pattern favors decoupling of the compute path from the *data fetch & fetch address generation unit*, as well as from the *data store & store address generation unit*. This allows aggressive data prefetching in buffers ahead of time of the actual data requests.

7.3 Structured Grids: Speckle Reducing Anisotropic Diffusion (SRAD)

Structured grids refers to those algorithms in which computation proceeds as a series of grid update steps. It constitutes a separate class of algorithms from unstructured grids, in that the data is arranged in a regular grid of two or more dimensions (typically 2D or 3D). SRAD is a structured grids application that attempts to eliminate speckles (i.e., locally correlated noise) from images, following a partial differential equation approach. Figure 6 presents a high-level overview of the SRAD

Host code:

```
Loop for iter number of iterations{
    calculate statistics for the region of interest
    blockX=columns/BLOCK_SIZE;
    blockY=rows/BLOCK_SIZE;
    localWorkSize[2]={BLOCK_SIZE, BLOCK_SIZE};
    globalWorkSize[2]={blockX*localWorkSize[0],
                       blockY*localWorkSize[1]};
    kernel1();
    kernel2();
}
```

Kernel1:

```
(Each work-item (i,j) works on a 2D table element)
dN[i][j]=J[north][j]-J[i][j];
dS[i][j]=J[south][j]-J[i][j];
dW[i][j]=J[i][west]-J[i][j];
dE[i][j]=J[i][east]-J[i][j];
Calculate various parameters based above
values & initial J[i][[j] value;
Using the above value, calculate diffusion
coefficient c[i][j];
```

Kernel2:

```
(Each work-item (i,j) works on a 2D table element)
cN=c[i][j];
cS=c[north][j];
cW=c[i][j];
cE=c[i][east];
D=cN*dN[i][j]+cS*dS[i][j]+cW*dW[i][j]+cE*dE[i][j];
J[i][j]=J[i][j]+0.25*lambda*D;
```

Fig. 6 SRAD

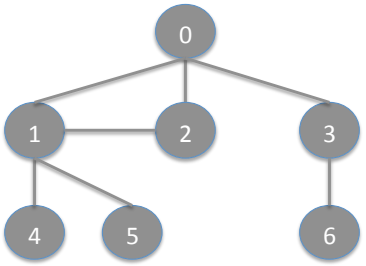
algorithm, without getting into the specific details (parameters, etc.) of the method. Performance is determined by FP compute power. The computational pattern is characterized by a mix of FP calculations including divisions, additions and multiplications. Many of the computations in both SRAD kernels are in the form: $x = a * b + c * d + e * f + g * e$. These computations can easily be transformed by the compiler to multiply-and-add operations. In such cases, special *fused multiply-and-add* units can offer a faster alternative to the typical series of separate multiplication and addition. While such units are already existent, more instances can be beneficial for the structured grids dwarf.

A series of *if* statements (simple in *kernel1*, nested in *kernel2*) handles boundary conditions and different branches are taken by different work-items, potentially within the same work-group. Since boundaries constitute only a small part of the execution profile, especially for large datasets, these branches do not introduce significant divergence. In the case of CPU and Xeon Phi execution, branch misprediction rate never exceeded 1%, while on the GPUs *VALUUtilization* remained above 86% indicating a high number of active vector ALU threads in a wave and consequently minimal branch divergence and code serialization.

Following its computational pattern, memory access patterns in SRAD, as in all kinds of stencil computation, are localized and statically determined, an attribute that favors data parallelism. Although the data access pattern is a priori known, non-consecutive data accesses, prohibit ideal caching. As in the NW case, where data is accessed in a non-linear pattern, data locality is an issue here, too. Cache hit rates, especially for the GPUs, remain low (e.g., 33% for HD 7970). This leads to the memory unit being stalled for a large percentage of the execution time (e.g., 45% and 29% on average for HD 7970, for the two OpenCL kernels). Correspondingly, the vector and scalar ALU instruction units are busy for a small percentage of the total GPU execution time (about 21% and 5.6% for our example, on the two kernels on HD 7970). All this is highlighted by comparing performance across the three GPUs, and once more, indicates the need for advancements in the memory technology that would make fast, large caches more affordable for computer architects.

On the CPU and Xeon Phi side, large cache lines can afford to host more than one row of the 2D input data (depending on the input sequences' sizes). The huge L3 cache of Opteron 6272, along with its high core count, make it very efficient in executing this structured grid dwarf. In such algorithms, it is a balance between cache and compute power that distinguishes a good target architecture. Of course, depending on the input data set there are obvious trade-offs, as in the case of GPUs, which despite their poor cache performance are able to hide the latency by performing more computation simultaneously while waiting for the data to be available.

An FPGA implementation with a single pair of accelerators (one accelerator for each OpenCL kernel) offers performance worse even than that of the single-threaded Opteron 6272 execution (FPGA_C1). This is attributed mainly to the complex FP operations FPGAs are notoriously inefficient at. Multiple instances of these pairs of accelerators (five pairs in FPGA_C2) can process parts of the grid independently, bringing FPGA performance close to that of multicore CPUs. Different work-groups access separate portions of memory, hence multiple accelerators instances access different on-chip memories, keeping accelerators isolated and self-contained.



Iter.	Kernel	Thread id (tid)						
		0	1	2	3	4	5	6
1	kernel1	✔	✗	✗	✗	✗	✗	✗
1	kernel2	✗	✔	✔	✔	✗	✗	✗
2	kernel1	✗	✔	✔	✔	✗	✗	✗
2	kernel2	✗	✗	✗	✗	✔	✔	✔
3	kernel1	✗	✗	✗	✗	✔	✔	✔
4	kernel2	✗	✗	✗	✗	✗	✗	✗

Host code:

```
maxThreads=numNodes < maxThreads?numNodes : maxThreads;
globalWorkSize=(numNodes/maxThreads)*maxThreads+
                ((numNodes%maxThreads)==0?0:maxThreads);
localWorkSize=max_threads;
node_as_source[]={1,0,0,0,0,0,0};
node_visited[]={1,0,0,0,0,0,0};
update_node_info[]={0,0,0,0,0,0,0};
cost[]={0,0,0,0,0,0,0};
```

```
do{
    stop=1;
    kernel1();
    kernel2();
}while(stop==0);
```

Kernel1:

```
tid=getGlobalId();
if(tid<numNodes and node_as_source[tid]==1){
    node_as_source[tid]=0;;
    for (all neighbors neighb[i] of current node)
        if(!node_visited[neighb[i]]){
            cost[neighb[i]]=cost[tid]+1;
            update_node_info[neighb[i]]=1;
        }
}
```

Kernel2:

```
tid=getGlobalId();
if(tid<numNodes and
update_node_info==1){
    node_as_source[tid]=1;
    mark node_visited[tid]=1;
    update_node_info[tid]=0;
    stop=0;
}
```

Fig. 7 BFS

7.4 Graph Traversal: Breadth-First Search (BFS)

Graph traversal algorithms entail traversing a number of graph nodes and examining their characteristics. As a graph traversal application, we select a BFS implementation. BFS algorithms start from the root node and visit all the immediate neighbors. Subsequently, for each of these neighbors the corresponding (unvisited) neighbors are inspected, eventually leading to the traversal of the whole graph. BFS's computation pattern can be observed through a simple example (Figure 7), as well as by its host and device side pseudocode. The BFS algorithm's computation pattern is characterized by an *imbalanced* workload per kernel launch that depends on the sum of the degrees $\deg(v_i)$ of the nodes at each level. For example (Figure 7), $\deg(v_0)$=3, so only three work-items perform *actual* work in the first invocation of *kernel2*. Subsequently, *kernel1* has three work-items, as well. Second invocation of *kernel2* performs work on three nodes again ($\deg(v_1)$ + $\deg(v_2)$ + $\deg(v_3)$ = 8, but nodes v_0, v_1, v_2 have already been visited, so effective $\deg(v_1)$ + $\deg(v_2)$ + $\deg(v_3)$ = 3). Computation itself is negligible, being reduced to a simple addition with respect to each node's cost.

The way the algorithm works might lead to erroneous conclusions, if only occupancy and ALU utilization is taken into account, as in all three GPU cases it is over 95% and 88%, respectively (for both kernels). The problem lies in the fact that *not* all work-items perform useful work, and the fact that the kernels are characterized by reduced compute intensity (Figure 7). In such cases, up to a certain degree of problem size or for certain problem shapes, the number of compute units or frequency are not of paramount importance and high-end cards, like HD 7970 are about as fast as an integrated GPU (e.g., HD 7660D). The above is highlighted by the hardware performance counters that indicate poor ALU packing (e.g., 36.1% and 38.9% for the two BFS OpenCL kernels, on HD 7660D). Similarly, for HD 7970, the vector ALU is busy only for 5% (approximate value across kernel iterations) of the GPU execution time, even if the number of active vector ALU threads in the wave is high (*VALUUtilization*: 88.8%).

For similar reasons, CPU execution performance is capped on Opteron 6272, which performs only marginally better than A8-3850. It is interesting to see that A10-5800K and even Xeon Phi, with 8- and 16-way SIMD are characterized by lack of performance scalability. Why performance of A10-5800K is not *at least* similar to that of A8-3850 could not be pinpointed during profiling. However, in both A10-5800K and Xeon Phi cases, we found that the OpenCL compiler could not take advantage of the 256- and 512-bit wide vector unit, because of the very nature of graph traversal.

With respect to data accesses, BFS exhibits irregular access patterns. Each work-item accesses discontiguous memory locations, depending on the connectivity properties of the graph, i.e, how nodes of the current level being inspected are being connected to other nodes in the graph. Figure 7 is not only indicative of the resource utilization (work-items doing useful work), but of the inherent irregularity of memory accesses that

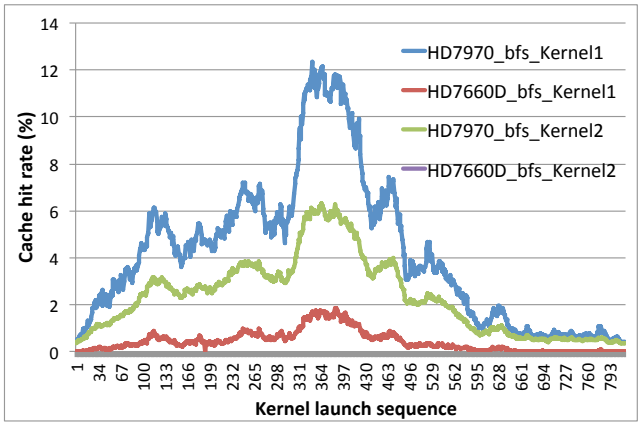


Fig. 8 BFS cache performance comparison between HD 7970 and HD 7660D.

depend on run-time assessed multiple levels of indirection, as well. Available caches' size define the cache hit rate, even in these cases, so HD 7970, which provides larger amounts of cache memory provides higher cache hit rates compared to the HD 7660D (varying for each kernel iteration, Figure 8).

The FPGA implementation of BFS (FPGA_C1) is the fastest across all tested platforms. While *kernel1* is not as fast as in the fastest of our GPU platforms, minimal execution time for *kernel2* and data transfer time render it the ideal platform for graph traversal, despite the dynamic memory access pattern causing the input streaming unit to be merged with the data path, eliminating the possibility of aggressive data prefetching.

7.5 Combinational Logic - Cyclic Redundancy Check (CRC)

Cyclic Redundancy Check (CRC) is an error-detecting code designed to detect errors caused by network transmission (or any other accidental error on the data). On a higher level, a polynomial division by a predetermined CRC polynomial is performed on the input data stream S and the remainder from this division constitutes the stream's CRC value. This value is typically added to the end of the data stream as it is transmitted. At the receiver end, a division of the augmented data stream with the (same, pre-determined) polynomial, will yield zero remainder on successful transmission. CRC algorithms that perform at the bit level are rather inefficient and many optimizations have been proposed that operate in larger units, namely 8, 16 or 32 bits. The implementation in OpenDwarfs follows a byte-based table-driven approach, where the values of the look-up table can be computed ahead of time and reused for CRC computations. The algorithm we use exploits a multi-level look-up table structure that eliminates the existence of an additional loop, thereby trading-off on-the-fly computation with the need for pre-computation and additional storage. Figure 9 shows the pseudocode of this implementation and provides a small, yet illustrative example of how the algorithm is implemented in parallel in OpenCL: the input data stream is split in byte-chunked sizes and each OpenCL work-item in a work-group is responsible for performing computation on this particular byte. The final CRC value is computed on the host once all partial results have been computed in the device. Figure 10 supplements Figure 9 by illustrating how multi-level look-up tables used in the kernel work and their specific values for the example at hand.

CRC, being a representative application of combinational logic algorithms is characterized by abundance of simple logic operations and data parallelism at the byte granularity. Such operations are fast in most architectures, and can be typically implemented as minimal-latency instructions, in comparison to complex instructions (like floating point division) that are split across multiple stages in modern superscalar architectures and introduce a slew of complex dependencies. Given the computational pattern of the CRC algorithm at hand, which is highly parallel, we are not surprized to observe high speedups for multi-threaded execution, in all platforms. For instance, in the Opteron 6272 CPU case, we observe a 12.2-fold speedup over the single-threaded execution. Similarly, Xeon Phi execution for the OpenCL kernel reaches maximum hardware thread utilization, according to our profiling results. The integrated GPUs in our experiments, which belong to the same architecture family, exhibit performance that is analogous to their number of *cores* and *threads per core* (as defined in Table 2). HD 7970, is a representative GPU of the AMD GCN (Graphics Core Next) architecture and bears fundamental differences to its predecessors, which may affect performance, as we see below.

With respect to the algorithm's underlying communication patterns, memory accesses in CRC are affine functions of a dynamically computed quantity (*tmp*). Specifically, as we see in Figure 9, inner-loop, cross-iteration dependencies due to stored state in variable *tmp*, cause input data addresses to the multi-level look-up table to be runtime-dependent. Obviously, this implies lack of cache locality, is detrimental to any prefetching hardware utilization and hence results to poor overall cache behavior. The effect of such cache behavior is highlighted by our findings in profiling runs across our test architectures. All three GPUs suffer from cache hit rates that range from 5.48% to 7.13%. Depending on the CRC size, such precomputed tables may be able to

Host code:

```
localWorkSize=getMaxWorkitemsPerWorkgroup();
globalWorkSize=N_bytes/localWorkSize-N_bytes%localWorkSize;
Kernel();
for(i=0; i<N_bytes;i++){
      crc=crc^crc_loc[i]; //crc_loc[] contains crc for byte i,
                          //calculated on the device.
}
```

Kernel:

```
tid=getGlobalId();
If(tid<N_bytes){
      tmp=in_stream_byte[tid];
      val=N_bytes-tid;
      for(i=0; i<numTables; i++){
            if( (val>>i)%2==1){
                  tmp=table[i][tmp]
            }
      }
      crc_loc[tid]=tmp;
}
```

Example:

S = 0000101100000011
in_stream_byte[] =
{00001011, 00000011}

Work-item 0

tid=0
tmp=00001011(=11)
val=2-0=2
i=0: (condition false)
i=1: tmp=table[1][11]
crc_loc[0]=tmp;

Work-item 1

tid=1
tmp=00000011(=3)
val=2-1=1
i=0: tmp=table[0][3]
i=1: (condition false)
crc_loc[1]=tmp;

Fig. 9 CRC pseudocode and a simple traceable example.

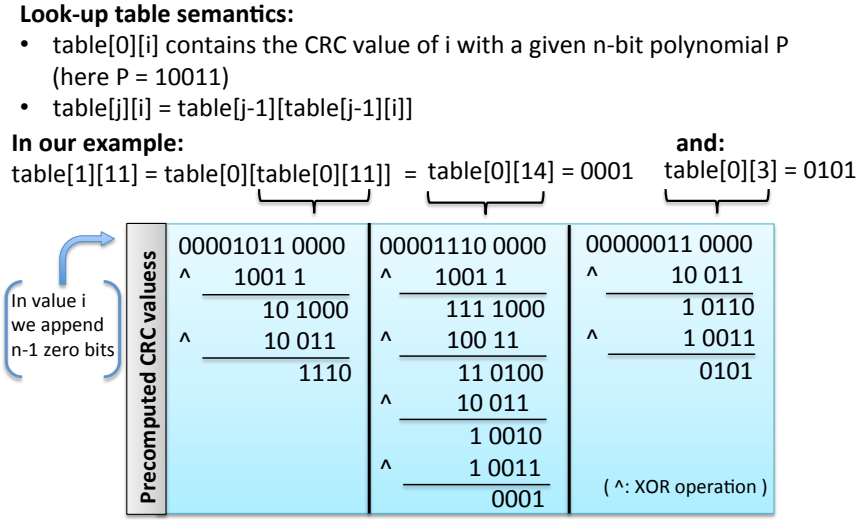


Fig. 10 CRC look-up table semantics.

fit into lower level caches. In such cases, more efficient data communication may be achieved, even in the adverse, highly probable case of consecutive data accesses spanning multiple cache lines. CRC is yet another dwarf that benefits from fast cache hierarchies.

Of course, in algorithms like this where operations take place on the byte-level the existence of efficient methods for accessing such data sizes and operating on them is imperative, if one is to fully utilize wider than 8-bit data-path, bus widths, etc. Such an example is SIMD architectures that allow packed operations on collections of different data sizes/types (such as bytes, single or double precision floating point elements). CPU and GPU architectures follow a semantically similar approach.

Profiling for Xeon Phi corroborates a combination of the above claims. For instance, *vector intensity* is 14.4 close to the ideal value (16). This metric portrays the ratio between the total number of data elements processed by vector instructions and the *total* number of vector instructions. It highlights the vectorizability opportunities of the CRC OpenCL kernel, and helps

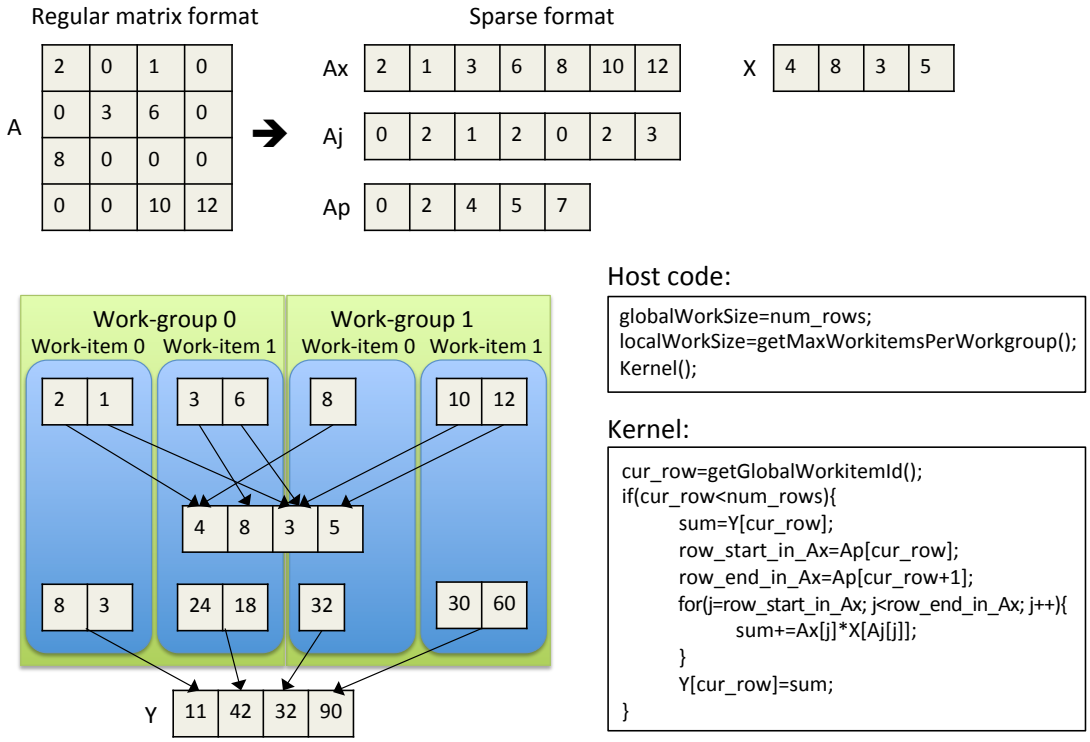


Fig. 11 CSR representation and algorithm.

quantify the success of the Intel OpenCL compiler's vectorization module in producing efficient vector code for the MIC architecture.

L1 compute to data access ratio is a mere 2.45. The ideal value would be close to the calculated vector intensity (14.4). This metric portrays the average number of vector operations per L1 cache access and its low value highlights the irregular, dynamic memory access pattern's toll in caching. In this case vector operations, even on high-width vector registers will not benefit performance being bounded by the time needed to serve consecutive L1 cache misses.

On the FPGA, the SOpenCL implementation cannot disassociate the module that fetches data (input streaming unit) from the module that performs computations (data path), hence, reducing the opportunity for aggressive prefetching. A Processing Element (PE) is generated for the inner for-loop (FPGA_C1). This corresponds to a "single-threaded" FPGA implementation. If multiple FPGA accelerators are instantiated and operate in parallel, the execution time is better than that of the lower-end HD 6550D GPU. The number of accelerators that can "fit" in an FPGA is a direct function of available resources. In our case, up to 20 accelerators can be instantiated in a Virtex-6 LX760 FPGA, each reading one byte per cycle from on-chip BRAM (FPGA_C2). The area of accelerator can be reduced after bitwidth optimization. Utilization of fully customized bitwidths results to higher effective bandwidth between BRAM memory and the accelerators, which in turn translates to performance similar to that of HD 7970, with a more favorable performance-per-power ratio (FPGA_C3).

7.6 Sparse Linear Algebra - Compressed Sparse Row Matrix-Vector Multiplication (CSR)

CSR in OpenDwarfs calculates the sum of each of a matrix's rows' elements, after it is multiplied by a given vector. The matrix is not stored in its entirety but rather in a compressed form, known as compressed row storage sparse matrix format. This matrix representation is very efficient in terms of storage when the number of non-zero elements is much smaller than the zero elements.

Figure 11 provides an example of how a "regular" matrix corresponds to a sparse matrix representation. Specifically, only non-zero values are stored in Ax (thus

saving space from having to store a large number of zero elements). Alongside, Aj[i] stores the column that corresponds to the same position i of Ax. Ap is of size num_rows+1 and each pair of positions i, i+1 denote the range of values for j where Ax[j] belongs to that row. The pseudocode of CSR and a small, traceable example is depicted in Figure 11.

In this particular implementation of sparse matrix-vector multiplication, a reduction is performed across each row, in which the results of the multiplication of that row's non-zero elements are summed with the corresponding vector's elements. Such operations' combinations, which are typical in many domains, such as digital signal processing, can benefit from specialized *Fused multiply-add* (FMADD) instructions and hardware implementations thereof. This is yet another example where a typical, recurring combination of operations in a domain is realized in a fast, efficient way in architecture itself. FMADD instructions are available in CPUs, GPUs, and Intel Xeon Phi alike. OpenDwarfs, based on the dwarfs concept that *emphasizes* such recurring patterns, seeks to aid computer architects in this direction.

CSR is memory-latency limited and its speedup by activating multiple threads on the two CPUs is low (5-fold and 1.8-fold for 16 and 4 threads on the Opteron and Llano CPUs, respectively). While performance in absolute terms is better in HD 7970 and Xeon Phi, its bad scalability is obvious and speedups compared to the CPU multithreaded execution are mediocre. As we can see in Figure 11, data parallelism in accessing vector x is based on indexed reads, which limits memory-level parallelism. As with other dwarfs, such runtime-dependent data accesses limit the efficiency of mechanisms like prefetching. Indeed, in contrast to dwarfs like *n-body* the number of prefetch instructions emitted in all three CPUs, as well as in Xeon Phi are very low. Gather-scatter mechanisms, on the other side, are an important architectural addition that alleviates the effects of indirect addressing that are typical in sparse linear algebra. Especially in sparse linear algebra applications, the problem is aggravated from the large distance between consecutive elements within a row's operations (due to the high number of - conceptual - zero elements in the sparse matrix) and elements across rows (depending on the parallelization level/approach, e.g., multithreading, vectorization). In these cases, cache locality is barely existent and larger caches may only prove of limited value. Overall cache misses are less in Opteron 6272 that employs a larger L2 cache and an L3 cache, compared to the rest of the CPUs. On the GPU side, we have similar observations: HD 7970 13.27% cache hit rate, followed by 4.3% and 3.93% in HD 7660D and HD 6550D, respectively. The memory unit is busy (*MemUnitBusy* and *FetchUnitBusy* counters for HD 7990 and HD 6550D/HD 7660D) for most of the kernel execution time (reaching 99% in all the GPU cases). Any cache or memory effects are taken into account and the above indicates the algorithm in GPUs is fetch-bound.

VALUBusy and *ALUBusy* counters indicate a reciprocal trend of low ALU utilization, ranging from 3-6%. Even during this time, ALU vector packing efficiency, especially n Llano/Trinity is in the low 30%, which indicates ALU dependency chains prevent full utilization. The case is not much different in Xeon Phi, where the ring interconnect traffic becomes a serious bottleneck, as L1 and L2 caches are shared across the 61 cores.

In the FPGA implementation of sparse matrix-vector multiplication, cross-iteration dependence due to $y[row]$ causes tunnel buffers to be used to store $y[row]$ values. Tunnels are generated wherever a load instruction has a read-after-write dependency with another store instruction with constant cross-iteration distance larger than or equal to one (FPGA_C1). Allowing OpenCL to fully unroll the inner loop dramatically improves FPGA performance by almost 23-fold because it reduces iteration interval (II) from 8 down to 2 (FPGA_C2).

8 Conclusions and Future Work

In this paper we presented the latest release of OpenDwarfs, which provides enhancements upon the original OpenDwarfs benchmark suite. We verified functional portability of dwarfs across a multitude of parallel architectures and characterized a subset's performance with respect to specific architectural features. Computation and communication patterns of these dwarfs lead to diversified execution behaviors, thus corroborating the suitability of the dwarf concept as a means to characterize computer architectures. Based on dwarfs' underlying patterns and profiling we provided insights tying specific architectural features of different parallel architectures to such patterns exposed by the dwarfs.

Future work with respect to the OpenDwarfs is multifaceted. We plan to:

(a) Further enhance the OpenDwarfs benchmark suite by providing features such as input dataset generation, automated result verification and OpenACC implementations. More importantly, we plan to *genericize* each of the dwarfs, i.e., attempt to abstract them on a higher level, since some dwarf applications may be considered too application-specific.
(b) Characterize more architectures including Altera FPGAs by using Altera OpenCL SDK, evaluate different vendors' OpenCL runtimes and experiment with varying size and/or shape of input datasets.

(c) Provide architecture-aware optimizations for dwarfs, based on existing implementations. Such optimizations could be eventually integrated as compiler back-end optimizations after some form of application signature (i.e., dwarf) is extracted by code inspection, user-supplied hints, or profile-run data.

Acknowledgements This work was supported in part by the Institute for Critical Technology and Applied Science (ICTAS) at Virginia Tech, NSF I/UCRC IIP-1266245 via the NSF Center for High-Performance Reconfigurable Computing (CHREC), and EU (European Social Fund ESF) and Greek funds through the operational program Education and Lifelong Learning of the National Strategic Reference Framework (NSRF) - Research Funding Program: THALIS. The authors thank Muhsen Owaida for his contribution to prior versions of this work.

References

1. Altera Corporation (2012). *Implementing FPGA Design with the OpenCL Standard*, 2.0 edn.
2. Altera Corporation (2013). *Altera SDK for OpenCL: Programming Guide*. URL `http://www.altera.com/literature/hb/opencl-sdk/aocl_programming_guide.pdf`
3. Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K. (2006). *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. Rep. UCB/EECS-2006-183, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley
4. Bienia, C., Kumar, S., Singh, J.P., Li, K. (2008). The PARSEC Benchmark Suite: Characterization and Architectural Implications. In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. New York, NY
5. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K. (2009). Rodinia: A Benchmark Suite for Heterogeneous Computing. In: *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC '09)*. Austin, TX
6. Daga, M., Aji, A.M., Feng, W.c. (2011). On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing. In: *Proceedings of the Symposium on Application Accelerators in High-Performance Computing (SAAHPC '11)*. Washington, DC
7. Danalis, A., Marin, G., McCurdy, C., Meredith, J.S., Roth, P.C., Spafford, K., Tipparaju, V., Vetter, J.S. (2010). The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In: *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU '10)*. Pittsburgh, PA
8. de Dinechin, F., Pasca, B., Normale, E. (2011). Custom Arithmetic, Datapath Design for FPGAs using the FloPoCo Core Generator. *IEEE Design & Test of Computers* **28**(4)
9. Feng, W.C., Lin, H., Scogland, T., Zhang, J. (2012). OpenCL and the 13 Dwarfs: A Work In Progress. In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE '12)*. Boston, MA
10. Henning, J.L. (2006). SPEC CPU2006 Benchmark Descriptions. *SIGARCH Computer Architecture News* **34**(4)
11. Keutzer, K., Massingill, B.L., Mattson, T.G., Sanders, B.A. (2010). A Design Pattern Language for Engineering (Parallel) Software: Merging the PLPP and OPL Projects. In: *Proceedings of the Workshop on Parallel Programming Patterns (ParaPLoP '10)*. New York, NY
12. Krommydas, K., Scogland, T., Feng, W.c. (2013). On the Programmability and Performance of Heterogeneous Platforms. In: *2013 International Conference on Parallel and Distributed Systems (ICPADS)*
13. Lattner, C., Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis Transformation. In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, CA
14. Lee, K., Lin, H., Feng, W.c. (2013). Performance Characterization of Data-intensive Kernels on AMD Fusion Architectures. *Computer Science - Research and Development* **28**(2-3)
15. Lee, V.W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A.D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., Dubey, P. (2010). Debunking the 100X GPU vs. CPU Myth: an Evaluation of Throughput Computing on CPU and GPU. In: *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. New York, NY
16. Owaida, M., Bellas, N., Daloukas, K., Antonopoulos, C.D. (2011). Synthesis of Platform Architectures from OpenCL Programs. In: *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '11)*. Salt Lake City, UT
17. Sim, S.E., Easterbrook, S., Holt, R.C. (2003). Using Benchmarking to Advance Research: a Challenge to Software Engineering. In: *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*. Washington, DC
18. Spafford, K.L., Meredith, J.S., Lee, S., Li, D., Roth, P.C., Vetter, J.S. (2012). The tradeoffs of fused memory hierarchies in heterogeneous computing architectures. In: *Proceedings of the 9th Conference on Computing Frontiers (CF '12)*. New York, NY
19. Stratton, J.A., Rodrigues, C., Sung, I.J., Obeid, N., Chang, L.W., Anssari, N., Liu, G.D., Hwu, W.m.W. (2012). *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Tech. Rep. IMPACT-12-01, University of Illinois at Urbana-Champaign

Konstantinos Krommydas is currently a Ph.D. student at the Computer Science Department at Virginia Tech. He received a B.Eng. from the Department of Electrical and Computer Engineering at University of Thessaly in Volos, Greece and his MSc from the Department of Computer Science at Virginia Tech. He is a member of the Synergy Lab under the supervision of Dr. Wu Feng. His research interests lie in the broader area of parallel computing with an emphasis on evaluation, analysis and optimization for parallel heterogeneous architectures.

Dr. Wu-chun (Wu) Feng is the Elizabeth & James E. Turner Fellow and Professor in the Department of Computer Science. He also holds adjunct faculty positions with the Department of Electrical & Computer Engineering and the Virginia Bioinformatics Institute at Virginia Tech. Previous professional stints include Los Alamos National Laboratory, The Ohio State University, Purdue University, Orion Multisystems, Vosaic, NASA Ames Research Center, and IBM T.J. Watson Research Center. His research interests encompass high-performance computing, green supercomputing, accelerator and hybrid-based computing, and bioinformatics. Dr. Feng received a B.S. in Electrical & Computer Engineering and in Music in 1988 and an M.S. in Computer Engineering from Penn State University in 1990. He earned a Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 1996.

Christos Antonopoulos is an Assistant Professor at the Department of Electrical and Computer Engineering (ECE) of the University of Thessaly in Volos, Greece. His research interests span the areas of system and applications software for high performance computing, and application-driven redefinition of the hardware/software boundary on accelerator-based systems. He received his Ph.D., M.S. and Diploma from the Computer Engineering and Informatics Department of the University of Patras, Greece. He has co-authored more than 45 research papers in international, peer-reviewed scientific journals and conference proceedings.

Nikos Bellas received his Diploma in Computer Engineering and Informatics from the University of Patras in 1992, and the M.S. and Ph.D. in ECE from the University of Illinois at Urbana-Champaign in 1995 and 1998, respectively. From 1998 to 2007 he was a Principal Staff Engineer at Motorola Labs, Chicago. Since 2007 he has been a faculty member at the ECE Department of the University of Thessaly. His research interests is in embedded systems, computer architecture, approximate computing and low-power design. He holds 10 US patents.